

# TypeScript

# Quickly

YAKOV FAIN  
ANTON MOISEEV



MEAP



MANNING



**MEAP Edition  
Manning Early Access Program  
TypeScript Quickly  
Version 9**

Copyright 2019 Manning Publications

For more information on this and other Manning titles go to  
[manning.com](https://manning.com)

# welcome

---

Thank you for purchasing the first MEAP for *TypeScript Quickly!*

According to the latest issue of the reputable ThoughtWork's Technology Radar (see <https://goo.gl/sYHY6r>), "TypeScript is a carefully considered language and its consistently improving tools and IDE support continues to impress us. With a good repository of TypeScript-type definitions, we benefit from all the rich JavaScript libraries while gaining type safety." Surveys on Stackoverflow consistently show that TypeScript is one of the most loved languages, and this book is about TypeScript.

We use this language at work and write this book for developers who want to become more productive with developing apps that will run in the browsers' or standalone JavaScript engines. We expect the reader to know the basics of HTML, CSS, and JavaScript (based on ECMAScript 5 spec).

We really like TypeScript for allowing us to produce executable JavaScript code faster. And the chances of getting runtime errors are substantially lower compared to the code that was originally written in JavaScript.

This book is divided into two parts. In Part 1, we cover various syntax elements of TypeScript using small code snippets for illustration. Part 2 (starts from chapter 8) is a collection of various apps written in TypeScript.

At some point, we started thinking of the app for showcasing how TypeScript could be applied in real-world projects, and we didn't want it to be yet another ToDo app. Long story short, each chapter in Part 2 is a small app that implements hot blockchain technology. In one chapter it's just a standalone app running under Node.js illustrating the basics of the blockchain technology. In another – it's a blockchain built as a web app. We'll have a chapter where we use TypeScript with WebSockets. We'll describe a blockchain using the web client in Angular, React, and Vue. While reviewing the code in all these apps, you'll see TypeScript applied.

The editorial board became a little nervous after learning that we want to use a not trivial blockchain technology for TypeScript demo apps. We insisted because we believe in our ability to explain complex concepts in a simple language, and we believe in you, our reader. Having said that, we would really appreciate your feedback on all the content of this MEAP.

This MEAP includes fifteen chapters and appendix. Chapters 15 and 16 were added to this edition. The repository of the book code samples is located on GitHub at <https://github.com/yfain/getts>.

Chapter 1 will get you started with TypeScript development. You'll compile and run the very basic programs so you understand the workflow from writing the program in TypeScript to compiling it into runnable JavaScript. We'll also cover the benefits of programming in TypeScript over JavaScript and will introduce the Visual Studio Code editor.

Chapter 2 explains how to declare variables and functions with types. You'll also learn how to declare type aliases with the type keyword. You'll see how to declare custom types with classes and interfaces and will understand the difference between nominal and structural type systems.

Chapter 3 explains how class inheritance works, and when to use abstract classes. You'll also see how TypeScript interfaces can force a class to have methods with known signatures without worrying about the implementation details. We'll also explain what "programming to interfaces" means.

Chapter 4 is dedicated to enums and generics. This chapter covers the benefits of using enums, the syntax for numeric and string enums, what generic types are for, and how to write classes, interfaces, and functions that support generics

Chapter 5 This chapter covers decorators, mapped and conditional types. It's about advanced TypeScript types and you should be familiar with the syntax of generics to understand the materials of this chapter.

Chapter 6 This chapter is about tooling. We'll explain the use of source maps and TSLinter. Then we'll show you how to compile and bundle TypeScript apps with Webpack. You'll also see how and why to compile TypeScript with Babel.

Chapter 7 In this chapter we show you how to use JavaScript libraries in your TypeScript app. We start by explaining the role of the type definition files. Then, we show a small app that uses the JavaScript library in the TypeScript app. Finally, we go over the process of the gradual upgrade of an existing JavaScript project to TypeScript.

Chapter 8 opens Part 2 of the book by introducing the principles of blockchain apps. You'll learn what the hashing functions are for, what the block mining means, and why the proof of work is required to add a new block to a blockchain. After learning the blockchain basics, we'll create a project and explain the code that creates a primitive blockchain app. Each chapter in Part 2 has a runnable project with detailed explanations of how it was written and how to run it.

Chapter 9 describes how to create a web client for a blockchain. This app will not use any web frameworks; we'll just use HTML, CSS, and TypeScript. We'll also create a small library for hash generation that can be used in both web and standalone clients. You'll also see how to debug the TypeScript code in the browser.

Chapter 10 represents a code review of the blockchain app that uses a messaging server for communications between the members of the blockchain. We're going to create a Node.js and WebSocket server in TypeScript, and will show you how our blockchain uses the longest chain rule to achieving consensus. You'll find practical illustrations of using TypeScript interfaces, abstract classes, access qualifiers, enums, and generics.

Chapter 11 is a brief introduction to development of the web apps in Angular with TypeScript.

Chapter 12 is a code review of the blockchain web client developed using Angular framework and TypeScript.

Chapter 13 is a brief introduction to development of the web apps in React.js with TypeScript.

In chapter 14, is a code review the blockchain client developed in React.js and TypeScript.

Chapter 15 (new) is a brief introduction to development of the web apps in Vue.js with TypeScript.

Chapter 16 (new) is a code review the blockchain client developed in Vue.js and TypeScript.

Appendix A contains an overview of the syntax introduced in ECMAScript 6, 7, and 8. You'll learn how to use classes, fat arrow functions, spread and rest operators, what destructuring is, and how to write asynchronous code as if it's synchronous with the help of `async-await` keywords. We'd like to make a clear distinction between the syntax described in the latest ECMAScript specifications, and the syntax that's unique to TypeScript. That's why we recommend you to read the Appendix A first, so you know where ECMAScript ends and TypeScript begins.

Enjoy the reading!

—Yakov Fain and Anton Moiseev.

# *brief contents*

---

## **PART 1: MASTERING THE TYPESCRIPT SYNTAX**

- 1 *Getting familiar with TypeScript*
- 2 *Basic and custom types*
- 3 *Object-oriented programming with classes and interfaces*
- 4 *Enums and Generics*
- 5 *Decorators and advanced types*
- 6 *Tooling*
- 7 *Using TypeScript and JavaScript in the same project*

## **PART 2: APPLYING TYPESCRIPT IN A BLOCKCHAIN APP**

- 8 *Developing your own blockchain app*
- 9 *Developing a browser-based blockchain node*
- 10 *Client-server communications using Node.js, TypeScript, and WebSockets*
- 11 *Developing Angular apps with TypeScript*
- 12 *Developing the blockchain client in Angular*
- 13 *Developing React.js apps with TypeScript*
- 14 *Developing a blockchain client in React.js*
- 15 *Developing Vue.js apps with TypeScript*
- 16 *Developing the blockchain app in Vue.js*

*Appendix A: Modern JavaScript*

# *Part 1: Mastering the TypeScript syntax*

P1



# *Getting familiar with TypeScript*

***The goal of this chapter is to get you started with the TypeScript development. We'll start with paying respect to JavaScript, and then will share with you our own opinion on why you should be programming in TypeScript. Then we'll compile and run the very basic programs so you understand the workflow from writing the program in TypeScript to compiling it into runnable JavaScript. This chapter covers:***

- The benefits of programming in TypeScript over JavaScript
- How to compile the TypeScript code into JavaScript
- How to work the Visual Studio Code editor

In this book, we'll be programming in TypeScript, but to run the code, we'll need to compile the code to JavaScript, so let's go back in time and do a quick review of this language.

## ***1.1 A little bit of a JavaScript history***

In May of 1995, after 10 days of hard work, Brendan Eich had created the JavaScript programming language. This scripting language didn't need a compiler and was meant to be used in the web browser Netscape Navigator.

No compilers were needed for deploying a program written in JavaScript in the browser. Adding a `<script>` tag with the JavaScript sources (or a reference to the file with sources) would instruct the browser to load and parse the code and execute it in the browser's JavaScript engine. People enjoyed the simplicity of the language - no need to declare types of variables, no need to use any tools - just write your code in a plain text editor and use it in a web page.

If you just started learning JavaScript, you can see your first program running in 15 minutes. There is nothing to install or configure, and there's not need to compile the program as

JavaScript is an interpreted language.

JavaScript is also a dynamically-typed language, which would give additional freedom for software developers. No need to declare upfront the object's properties as the JavaScript engine would create the property during the runtime if the object didn't already have it.

Actually, there's no way to declare the type of a variable in JavaScript. The JavaScript engine will guess the type based on the assigned value (e.g. `var x = 123` would mean that `x` is a number). If later on, the script would have an assignment `x = "678"`, the type of `x` would automatically change from a number to string. Did you really want to change the type of `x` or it's a mistake? You'll know it only during the runtime as there is no compiler to warn you about it.

JavaScript is a very forgiving language, which is not a shortcoming if the codebase is small and you're the only person working on this project. Most likely, you remember that `x` is supposed to be a number, and you don't need any help with this. And of course, you'll work for your current employer forever so the variable `x` is never forgotten.

Over the years, JavaScript became super popular and the de facto standard programming language of the web. But if 20 years ago we just used Javascript to display web pages with some interactive content, today we develop complex web apps that contain thousands of lines of code developed by teams of developers. And you know what? Not everyone in your team remembers that `x` supposed to be a number. To minimize the number of runtime errors, JavaScript developers write unit tests and perform code review.

To be more productive, software developers could use help from the tooling like IDE with auto-complete features, easy refactoring et al. But how an IDE can help you with refactoring if the language allows complete freedom in adding properties to objects and changing types on the fly?

We needed a better language, but replacing JavaScript with another one that would be supported by all different browsers was not a mission, so new compile-to-JavaScript languages were created. They were more tool-friendly, but the program would still have to be converted to JavaScript before deployment so every browser would support them. TypeScript is one of such languages, and let's see what makes it stand out.

## 1.2 Why program in TypeScript

TypeScript is a compile-to-JavaScript language, which was released as an open source project by Microsoft in 2012. A program written in TypeScript has to be *transpiled* into JavaScript first, and then it can be executed in the browser or a standalone JavaScript engine.

The difference between transpiling and compiling is that the latter turns the source code of a program into bytecode or machine code, whereas the former converts the program from one

language to another, e.g. from TypeScript to JavaScript. But in TypeScript community, the word *compile* is more popular, and we'll use it in this book to describe the process of conversion of the TypeScript code into JavaScript.

You may wonder, why go through a hassle of writing a program in TypeScript and then compiling it into JavaScript, if you could write this program in JavaScript in the first place? To answer this question, let's look at the TypeScript from a very high-level perspective.

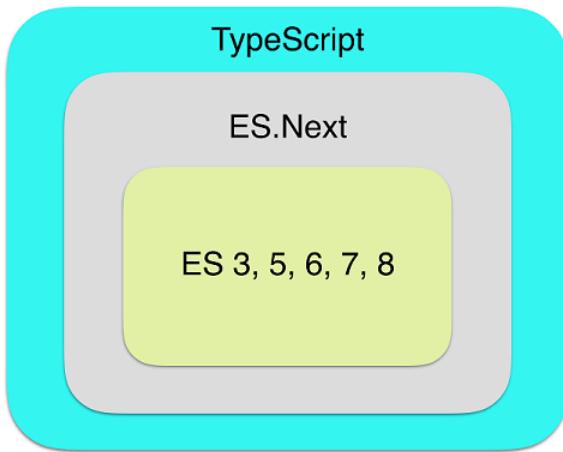
TypeScript is a superset of JavaScript, and you can take any JavaScript file, e.g. `myProgram.js`, change its name extension from `.js` to `.ts`, and most likely, the file `myProgram.ts` becomes a valid TypeScript program. We say most likely, because your JavaScript code may have hidden type-related bugs, you may dynamically change the types of object properties or add new ones after the object was declared and some other things that can be revealed only after your JavaScript code is compiled.

**TIP**

In section 7.3 in chapter 7, we'll provide some tips on migrating your JavaScript code to TypeScript.

In general, the word *superset* means that it contains everything that the *set* has plus something else. The main addition to the "JavaScript set" is that TypeScript also supports *static typing* whereas JavaScript supports only *dynamic typing*. Here, the word typing refers to assigning types to program variables.

Figure 1.1 illustrates TypeScript as a superset of ECMAScript, which is a spec for all versions of JavaScript. ES.Next represents the very latest additions to ECMAScript that are still in the works.



**Figure 1.1 TypeScript as a superset**

In programming languages with static typing, a type must be assigned to a variable before you can use it. In TypeScript, you can declare a variable of a certain type, and any attempt to assign

to it a value of a different type results in a compilation error. This is not the case in JavaScript, which doesn't know about the types of your program variables until the runtime. Even in the running program, you can change the type of a variable just by assigning to it a value of different type.

For example, if you declare a variable as a `string`, trying to assign a numeric value to it will result in the *compile-time* error.

```
let customerId: string;
customerId = 123; // compile-time error
```

JavaScript decides on the variable type during the runtime, and the type can be dynamically changed as in the following example:

```
let customerId = "A15BN"; // OK, customerId is a string
customerId = 123; // OK, from now on it's a number
```

Now let's consider a JavaScript function that applies a discount to a price. It has two arguments and both must be numbers.

```
function getFinalPrice(price, discount) {
  return price - price / discount;
}
```

How do you know that the arguments must be numbers? First of all, you authored this function some time ago, and having an exceptional memory, you may just remember all types of all functions arguments. Secondly, you used descriptive names of the arguments that hint their types. Thirdly, you could guess the types by reading the function code.

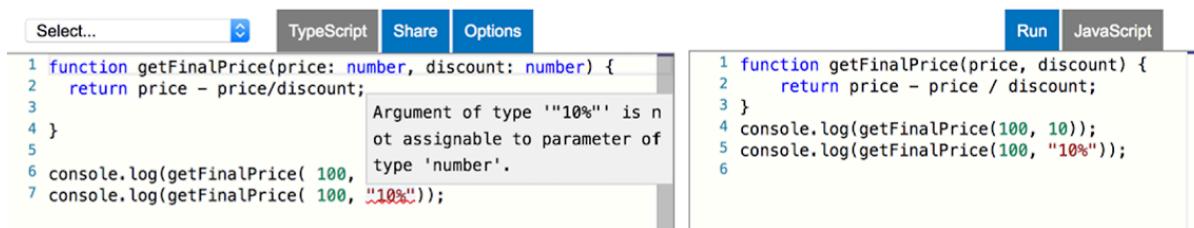
This is a pretty simple function, but let's say someone would invoke this function by providing a discount as a string, this function would print `Nan` during *runtime*.

```
console.log(getFinalPrice( 100, "10%")); // prints NaN
```

This is an example of a runtime error caused by the wrong use of a function. In TypeScript, you could provide the types for the function arguments, and such a runtime error would never happen. If someone would try to invoke the function with a wrong type of an argument, this error would be caught as you were typing. Let's see it in action.

The official TypeScript web page is located at [www.typescriptlang.org](http://www.typescriptlang.org). It offers the language documentation and a playground, where you could enter the code snippets in TypeScript, which would be immediately compiled into JavaScript.

Follow this link [bit.ly/2IyVNlj](http://bit.ly/2IyVNlj), and you'll see our code snippet in the TypeScript playground, with the squiggly red line under the `"10%"`. If you hover the mouse over the erroneous code, you'll see a prompt explaining the error as shown in figure 1.2.



**Figure 1.2 Using TypeScript playground**

This error is caught by the TypeScript static code analyzer as you type even before you compile this code with the TypeScript compiler tsc. Moreover, if you specify the variable types, your editor or IDE would offer the auto-complete feature suggesting you the argument names and types of the `getFinalPrice()` function.

Isn't it nice that errors are caught before runtime? We think so. The vast majority of the developers with the background in such languages as Java, C++, C# take it for granted that the errors are caught during compile time, and this is one of the main reasons why they like TypeScript.

**NOTE**

There are two types of programming errors - those that are immediately reported by the tools as you type, and those that are reported by the users of your program. Programming in TypeScript substantially decreases the number of the latter.

**TIP**

The site [typescriptlang.org](https://typescriptlang.org) has a section called Quick Start. There you may find some useful tips for configuring TypeScript in a specific environment like Angular, React, Node.js at al.

Still, some of the hard-core JavaScript developers say that TypeScript slows them down by requiring to declare types, and in JavaScript, they'd be more productive. Remember that types in TypeScript are optional and you can continue writing in JavaScript but introduce the TypeScript compiler in your workflow. Why? Because you'll be able to use the latest ECMAScript syntax (e.g. `async` and `await`) and compile your JavaScript down to ES5 so your code can run in older browsers.

But the majority of the web developers are not JavaScript ninjas and can appreciate a helping hand offered by TypeScript. As a matter of fact, all strongly typed languages provide better tool support and thus increase productivity (even for ninjas). Having said that, we'd like to stress that TypeScript gives you the benefits of statically typed languages when and where you want it without stopping you from using the good old dynamic JavaScript objects when you want them.

There are more than a hundred programming languages that are compiled to JavaScript, and you can see them at [github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js](https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js). But

what makes TypeScript stand out is that its creators follow the ECMAScript standards and implement the upcoming JavaScript features a lot faster than Web browsers vendors.

You can find the current proposals of the new ECMAScript features at [github.com/tc39/proposals](https://github.com/tc39/proposals). A proposal has to go through several stages to be included into the final version of the next ECMAScript spec. If a proposal makes it to stage 3, most likely it's included in the latest version of TypeScript.

In the Summer of 2017, the `async` and `await` keywords (see section A.10.4 in Appendix A) were included into the ECMAScript specification ES2017, a.k.a. ES8. It took more than a year for major browsers to start supporting these keywords. But TypeScript supported them since November of 2015. This means that TypeScript developers could start using these keywords about three years earlier than those who waited for the browsers' support.

And the best part is that you can use the future JavaScript syntax in today's TypeScript code and compile it down to the older JavaScript syntax (e.g. ES5) supported by all browsers!

Having said that, we'd like to make a clear distinction between the syntax described in the latest ECMAScript specifications, and the syntax that's unique to TypeScript. That's why we recommend you to read the Appendix A first, so you know where ECMAScript ends and TypeScript begins.

Although JavaScript engines do a decent job of guessing the types of variables by their values, development tools have a limited ability to help you without knowing the types. In mid- and large-size applications, this JavaScript shortcoming lowers the productivity of software developers.

TypeScript follows the latest specifications of ECMAScript and adds to them types, interfaces, decorators, class member variables (fields), generics, enums, the keywords `public`, `protected`, and `private` and more. Check the TypeScript roadmap at [github.com/Microsoft/TypeScript/wiki/Roadmap](https://github.com/Microsoft/TypeScript/wiki/Roadmap) to see what's available and what's coming in the future releases of TypeScript. And one more thing: The generated JavaScript code is easy to read, and it looks like hand-written code.

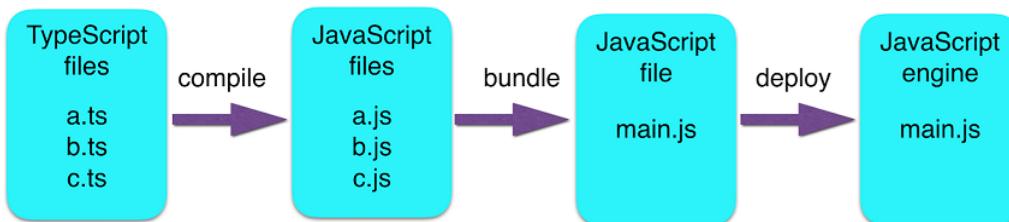
**SIDE BAR**    **Five facts about TypeScript**

1. The core developer of TypeScript is Andres Hejlsberg, who also designed Turbo Pascal and Delphi, and is the lead architect of C# at Microsoft.
2. At the end of 2014, Google approached Microsoft asking if they could introduce decorators in TypeScript so this language could be used for developing of the Angular 2 framework. Microsoft agreed, and this gave a tremendous boost to the TypeScript popularity given the fact that hundreds of thousands of developers use Angular.
3. As of May of 2019, the TypeScript compiler had **more than five million** downloads per week from [npmjs.org](https://www.npmjs.org/package/typescript), and this is not the only TypeScript repository. For current statistics, see [www.npmjs.com/package/typescript](https://www.npmjs.com/package/typescript).
4. As per Redmonk, a respectable software analytics firm, TypeScript came 12th in the programming language rankings chart in January of 2019 (see [redmonk.com/sogrady/2019/03/20/language-rankings-1-19](https://redmonk.com/sogrady/2019/03/20/language-rankings-1-19)).
5. According to the StackOverflow Survey 2019, TypeScript is the third most loved language, see [insights.stackoverflow.com/survey/2019](https://insights.stackoverflow.com/survey/2019).

Now, we'll introduce the process of configuring and using the TypeScript compiler on your computer.

### **1.3 Typical TypeScript workflows**

Let's get familiar with the workflow from writing a TypeScript code to deploying your app. Figure 1.3 shows such a workflow assuming that the entire source code of the app is written in TypeScript.



**Figure 1.3 Deploying an app written in TypeScript**

This diagram shows the project that consists of three TypeScript files: a.ts, b.ts, and c.ts. These files have to be compiled to JavaScript by the TypeScript compiler, which will generate three files: a.js, b.js, and c.js. Later in this section, we'll show you how to tell the compiler to generate JavaScript of specific version.

After looking at figure 1.3, some JavaScript developers will say, “TypeScript forces me to introduce an additional compilation step between writing code and seeing it running.” We’ll respond with a question, “Do you really want to stick to the ES5 version of JavaScript ignoring all the latest syntax introduced by ES6, 7, 8, and 9? If not, then you’ll have the compilation step in your workflow anyway – compile the sources written in a newer JavaScript into the well-supported ES5 syntax.”

Figure 1.3 shows just three files, but real-world projects may have hundreds or even thousands files. We don’t want to deploy so many files in our web server or as a standalone JavaScript app. Hence, we usually bundle these files (think concatenate) together.

JavaScript developers use different bundlers like Webpack or Rollup, which not only concatenate multiple JavaScript files, but can optimize the code and remove unused code (a.k.a. perform tree-shaking). If your app consists of several modules, each module can be deployed as a separate bundle.

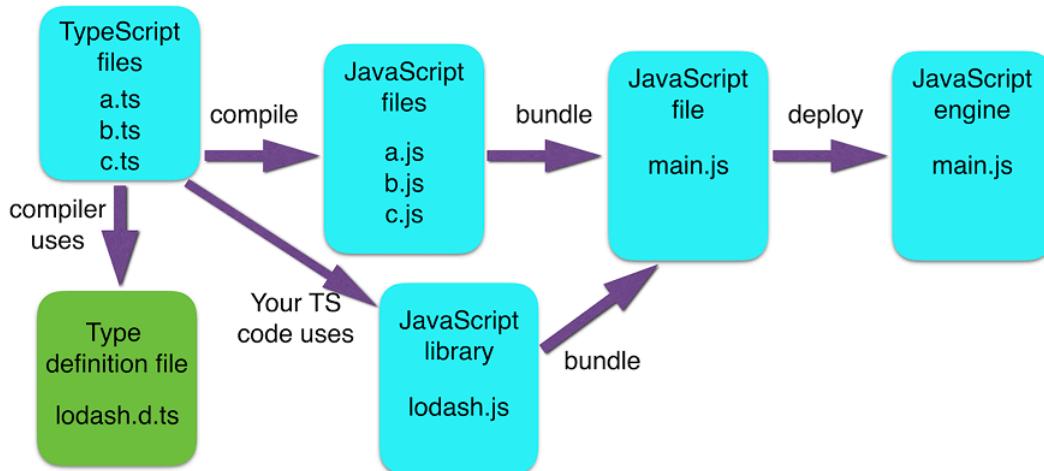
Figure 1.3 shows just one bundle main.js. If this is a web app, there will be some HTML file that will have a `<script src='main.js'>` tag. If the app should run in a standalone JavaScript engine like Node.js, you can start it by running the following command (assuming that Node.js is installed):

```
node main.js
```

The JavaScript ecosystem includes thousands of libraries, which are not going to be re-written in TypeScript. The good news is that your app doesn’t have to be TypeScript-only, and it can use any of the existing JavaScript libraries.

If you’ll just add the JavaScript library to your app, the TypeScript compiler won’t help with auto-complete or error messages when you use the API of these libraries. But there are special type definition files with the name extension .d.ts (covered in chapter 6), and if they are present, the TypeScript compiler will show you the errors and offer context-sensitive help for this library.

Figure 1.4 shows a sample workflow for an app that uses popular JavaScript library lodash.



**Figure 1.4 Deploying an app written in TypeScript and JavaScript**

This diagram includes the type definition file `lodash.d.ts`, which is used by the TypeScript compiler during development. It also includes the actual JavaScript library `lodash.js` that will be bundled with the rest of your app during deployment. The term *bundle* means a process of combining several script files into one.

## 1.4 Using TypeScript compiler

In this section, you learn how to compile a basic TypeScript program into its JavaScript version. TypeScript compiler can be bundled with your IDE of choice or can be installed as an IDE plugin, but we prefer to install it independently of an IDE using the npm package manager that comes with Node.js.

Node.js (or simply *Node*) isn't just a framework or a library, but it's a JavaScript runtime environment as well. We use the Node runtime for running various utilities like npm or launching the JavaScript code without the browser. So to get started, you need to download and install the current version of Node.js from [nodejs.org](https://nodejs.org). After installation is complete, you can open your Terminal or Command window and check the versions of Node and npm by entering the following commands:

```
node -v
npm -v
```

These commands will print the versions of Node and npm, e.g. 10.10.0. and 6.4.1. We'll use npm to install the TypeScript compiler and other packages from the npm repository located at [www.npmjs.com](https://www.npmjs.com), which hosts more than half a million packages.

Using npm, you can install software either locally inside your project directory, or globally so it can be used across projects. Let's install the TypeScript compiler globally by running the following command in your Terminal window:

```
npm install -g typescript
```

The `-g` option installs the TypeScript compiler globally on your computer. In all code samples of this book, we used TypeScript version 3 or newer, and to check the version of your TypeScript compiler, run the following command from the Terminal window:

```
tsc -v
```

### NOTE

For simplicity, in the first part of this book we'll use the globally-installed TypeScript compiler. In the real-world projects, we prefer to install `tsc` locally in the project directory by adding it in the `devDependencies` section in the project's `package.json`. You'll see how we do it in listing 8.4 in chapter 8 where we start working on the sample blockchain project.

Code written in TypeScript has to be compiled into JavaScript so web browsers can execute it. Let's see how to compile a simple program from TypeScript to JavaScript. In any directory, create a new file `main.ts` with the following content:

### Listing 1.1 main.ts

```
function getFinalPrice(price: number, discount: number) { ①
    return price - price/discount;
}

console.log(getFinalPrice(100, 10)); ②
console.log(getFinalPrice(100, "10%")); ③
```

- ① Function arguments have types
- ② Correct function invocation
- ③ Wrong function invocation

The following command will compile `main.ts` into `main.js`.

```
tsc main
```

It'll print the error message *argument of type "10%" is not assignable to parameter of type 'number'*, but will generate the file `main.js` with the following content anyway:

### Listing 1.2 main.js

```
function getFinalPrice(price, discount) { ①
    return price - price/discount;
}

console.log(getFinalPrice(100, 10)); ②
console.log(getFinalPrice(100, "10%")); ③
```

- ① Arguments have no types

- ② Correct function invocation
- ③ Wrong function invocation, but the error will be shown during runtime only

You may say, "What's the point of producing the JavaScript file if there is a compilation error?". Well, from JavaScript perspective, the content of the file main.js is valid. But in the real-world TypeScript projects we don't want to allow code generation for erroneous files.

The TypeScript compiler offers about dozens of compilation options described at [mng.bz/rf14](#), and one of them is `noEmitOnError`. Delete the file main.js and try to compile main.ts as follows:

```
tsc main --noEmitOnError true
```

Now the file main.js won't be generated until the error is fixed in main.ts.

The compiler's option `--t` allows you to specify the target JavaScript syntax. For example, you can use the same source file and generate its JavaScript peer compliant with ES5, ES6 or newer syntax. Here's how to compile the code to ES5-compatible syntax:

```
tsc --t ES5 main
```

The TypeScript compiler allows you to pre-configure the process of compilation (specifying the source and destination directories, target, and so on). The presence of the `tsconfig.json` file in the project directory means you can just enter `tsc` on the command line, and the compiler will read all the options from `tsconfig.json`. A sample `tsconfig.json` file is shown here.

### **Listing 1.3 tsconfig.json**

```
{
  "compilerOptions": {
    "baseUrl": "src",          ①
    "outDir": "./dist",        ②
    "noEmitOnError": true,     ③
    "target": "es5"            ④
  }
}
```

- ① Transpile .ts files located in the src directory
- ② Save the generated .js files in the the dist directory
- ③ If any of the files has compilation errors, don't generate JavaScript files
- ④ Transpile TypeScript files into the ES5 syntax

**TIP**

The compiler's option `target` is also used for syntax checking. For example, if you specify `es3` as the compilation target, TypeScript will complain about the getter methods in your code. It simply doesn't know how to compile getters into the ECMAScript 3 version of the language.

Let's see if you can do it yourself by following the instructions listed below.

1. Create the file named `tsconfig.json` in the folder where the `main.ts` is located (its content is listed earlier in this section). Add the following content to `tsconfig.json`:

```
{
  "compilerOptions": {
    "noEmitOnError": true,
    "target": "es5",
    "watch": true
  }
}
```

Note the last option `watch`. The compiler will watch your typescript files and as soon as they change, `tsc` will recompile them.

2. In the Terminal window, go to this directory and run the following command:

```
tsc
```

You'll see the error message described earlier in this section, but the compiler won't exit because it's running in the watch mode. The file `main.js` won't be created.

3. Fix the error and the code will be automatically recompiled. Check to see that the file `main.js` was created this time.

If you want to get out of the watch mode, just hit `CTRL-C` on your keyboard in the Terminal window.

#### TIP

To start a new TypeScript project, run the command `tsc --init` in any directory. It'll create the file `tsconfig.json` for you. This file will have all compiler's options and most of them will be commented out. Uncomment them as needed.

#### NOTE

A `tsconfig.json` file can inherit configurations from another file using the `extends` property. In chapter 10, we'll describe a sample project that has three config files: the first one with common `tsc` compiler options for the entire project, the second for the client and the third for the server portion of the project. See section 10.4.1 for details.

#### SIDEBAR

#### REPL

REPL stands for Read-Evaluate-Print-Loop. This term is used to refer to a simple interactive language shell that allows you to quickly execute a code fragment. The TypeScript Playground at [www.typescriptlang.org/play](http://www.typescriptlang.org/play) is an example of the REPL that allows you to write, compile, and execute a code snippet in a browser. Figure 1.5 shows how a simple TypeScript class is compiled into the ES5 version of JavaScript

```

1  class Person {
2    name = '';
3  }
4
5
6
7 
```

```

1  "use strict";
2  var Person = /** @class */ (function () {
3    function Person() {
4      this.name = '';
5    }
6    return Person;
7 })(); 
```

**Figure 1.5 Transpiling TypeScript to ES5**

Figure 1.6. shows how the same code was compiled into the ES6 version of JavaScript.

```

1  class Person {
2    name = '';
3  }
4
5
6 
```

```

1  "use strict";
2  class Person {
3    constructor() {
4      this.name = '';
5    }
6 } 
```

**Figure 1.6 Transpiling TypeScript to ES6**

**SIDE BAR** The Playground has a menu Options, where you can select compiler's options. In particular, you can select the compilation target, e.g. ES2018 or ES5.

If you'd like to run code snippets from the command line without the browser, install the TypeScript Node REPL available at [github.com/TypeStrong/ts-node](https://github.com/TypeStrong/ts-node).

## 1.5 Getting familiar with Visual Studio Code

Integrated Development Environments (IDE) and code editors increase developers productivity, and TypeScript is well supported by such tools, e.g. Visual Studio Code, WebStorm, Eclipse, Sublime Text, Atom, Emacs, and Vim. For this book, we picked the open-source and free editor Visual Studio Code (a.k.a VS Code) created by Microsoft, but you can use any other editor or IDE to work with TypeScript.

**NOTE** According to the Stack Overflow Developer Survey 2019 (see [insights.stackoverflow.com/survey/2019](https://insights.stackoverflow.com/survey/2019)), VS Code is the most popular developer environment, and more than 50% of *all* respondents use it. By the way, VS Code is written in TypeScript.

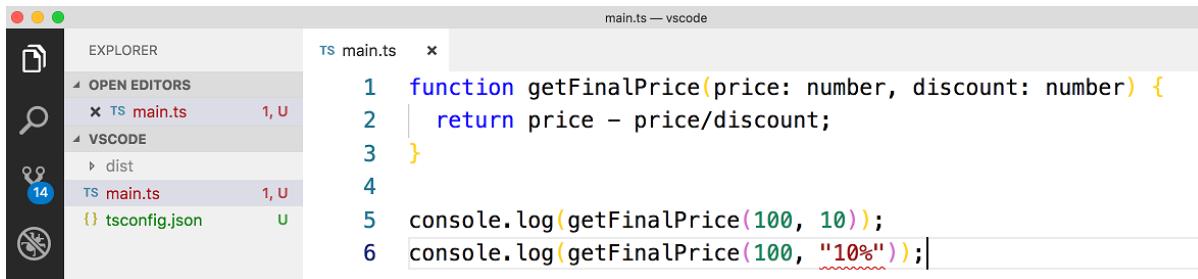
On real-world projects, good context-sensitive help and support for refactoring are very important. Renaming all occurrences of a TypeScript variable or function name in statically typed languages is done by IDEs in a split second, but this isn't the case in JavaScript, which doesn't support types.

If you make a mistake in a function, class, or a variable name, it's marked in red.

You can download VS Code from [code.visualstudio.com](https://code.visualstudio.com). The installation process depends on

the OS installed on your computer, and it's explained in the Setup section of the VS Code documentation (see [code.visualstudio.com/docs](https://code.visualstudio.com/docs)).

Start VS Code, and using the menu File | Open, open the directory chapter1/vscode included with this book's code samples. It contains the file main.ts from the previous section and a simple file tsconfig.json. Figure 1.7 shows the "10%" underlined with a red squiggly line indicating an error. If you'd hover the mouse pointer over the underlined code, it'd show the same error message as in figure 1.2.



**Figure 1.7 Highlighting errors in VS Code**

**NOTE**

VS Code supports two modes for the TypeScript code: *file scope* and *explicit project*. The file scope is pretty limited as it doesn't allow a script in file use the variables declared in another. The explicit project mode requires you to have the tsconfig.json file in the project directory. The following listing shows the tsconfig.json file that comes with this section:

**Listing 1.4 vscode/tsconfig.json**

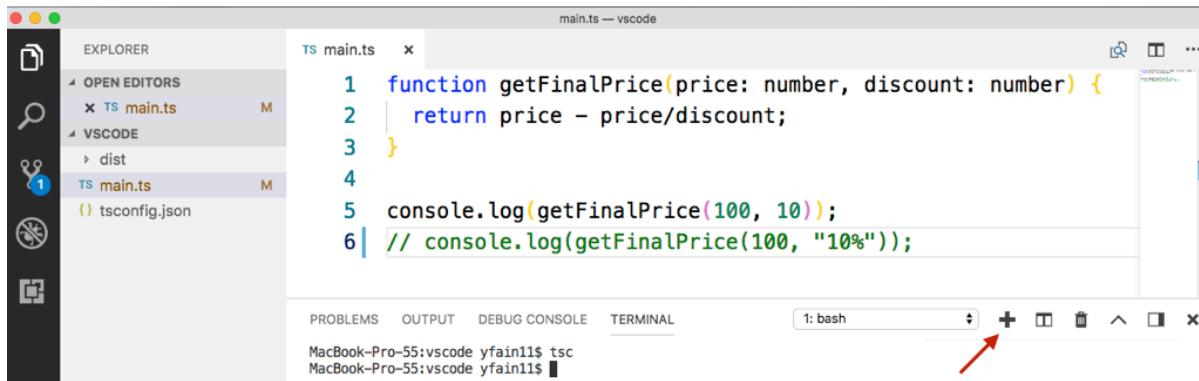
```
{
  "compilerOptions": {
    "outDir": "./dist",           ①
    "noEmitOnError": true,        ②
    "lib": ["dom", "es2015"]      ③
  }
}
```

- ① Save generated JavaScript files in the dist directory
- ② Don't generate JavaScript until all errors are fixed
- ③ The libraries were added so tsc won't complain about the unknown APIs, e.g. console()

If you'd like to be able to open VS Code from the command prompt, its executable has to be added to the environment variable PATH of your computer. In Windows, the setup process should do it automatically. In Mac OS, start VS Code, go to the menu View | Command Palette, type shell command, and pick the option "Shell command: Install 'code' command in PATH". Then restart your Terminal window and enter `code .` from any directory. VS Code will start, and you'll be able to work with the files from the directory you're in.

In the previous section, we compiled the code in a separate Terminal window, but if you use VS Code, it comes with the integrated terminal, so you don't need to leave the editor window right inside VS Code by using the menu View | Terminal or Terminal | New Terminal.

Figure 1.8 shows the integrated terminal view right after we executed the tsc command. The arrow on the right points at the plus icon that allows you to open as many terminal views as needed. Note the compiled file main.js in the dist directory on the left.



**Figure 1.8 Running tsc command in VS Code**

**TIP**

VS Code picks the tsc compiler that's included with Node.js installed on your computer. Open any TypeScript file and you'll see the tsc version at the bottom toolbar on the right. If you prefer to use tsc that you've installed globally on your computer, just click on the version number at the bottom right corner and select the tsc compiler of your choice.

At the bottom left you see a square icon that is used for finding and installing extensions from the VS Code marketplace. The following list shows some of the extensions that will make your TypeScript programming in VS Code more enjoyable:

- TSLint - integrated the TypeScript linter
- Prettier - code formatter
- Path Intellisense - auto-completes file paths

For more details about using VS Code for TypeScript programming, please visit the product documentation at [code.visualstudio.com/docs/languages/typescript](https://code.visualstudio.com/docs/languages/typescript).

**TIP**

There is an excellent online IDE called StackBlitz (see [stackblitz.com](https://stackblitz.com)). It's powered by VS Code, but you don't need to install it on your computer.

## 1.6 Summary

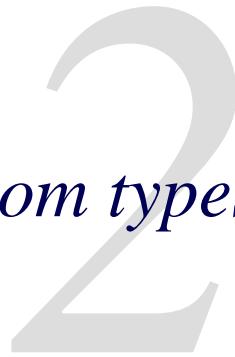
In this chapter, we presented the benefits of developing apps in TypeScript and introduced the tooling required to write, compile and deployed applications. We went through the installation process of the Node.js runtime, TypeScript compiler, and introduced the Visual Studio Code IDE.

In chapter 6, we'll show you more tools that are used in the real-world apps, but let's start learning the TypeScript syntax first.

**NOTE**

Part 2 of this book contains various versions of a sample blockchain app. While reading Part 2 is optional, we recommend you to read at least chapter 8 and 9. If you're eager to do more hands-on work, jump to the Part 2 of this book and get familiar with the sample Blockchain app described in chapter 8 and proceed to learning and running the code from this chapter. If you prefer to get a better understanding of the syntax of the language before developing applications, just continue reading chapters from Part 1.

# *Basic and custom types*



## **This chapter covers:**

- How to declare variables with types and use types in function declarations
- How to declare type aliases with the type keyword
- How to declare custom types with classes and interfaces

You can think of TypeScript as JavaScript with types. It's an over-simplified statement because TypeScript has some syntax elements that JavaScript doesn't (e.g. interfaces, generics, et al.). Still, the main power of TypeScript is types.

While declaring types of identifiers before their use is highly recommended, it's still optional. In this chapter, you'll start getting familiar with different ways of using the built-in and custom types.

In particular, you'll see how to use classes and interfaces for declaring custom types, but the coverage of classes and interfaces will continue in chapter 3.

### **NOTE**

If you're not familiar with the syntax of the modern JavaScript, it may be beneficial if you read appendix A before proceeding with learning TypeScript. This will also help you in understanding which syntax elements exist in JavaScript and which were added in TypeScript.

## **2.1 Declaring variables with types**

Why declare variable types if in JavaScript, you can just declare a variable name and store the data of any type there? Writing code in JavaScript is easier than in other languages mainly because you don't have to specify types of identifiers, isn't it?

Moreover, in JavaScript, you can assign a numeric value to a variable, and later on, assign a text value to this variable. This is not the case in TypeScript, where after the type is assigned to a variable you can't change its type as shown in figure 2.1.

The screenshot shows the TypeScript playground interface. At the top, there are tabs for 'Select...', 'TypeScript' (which is selected), 'Share', 'Options', 'Run', and 'JavaScript'. Below the tabs, there are two panes. The left pane contains TypeScript code:

```

1 let taxCode = 1;
2 taxCode = 'lowIncome';
3

```

The second line, 'taxCode = 'lowIncome';', has a red squiggly underline underneath 'lowIncome'. The right pane shows the compiled JavaScript code:

```

1 var taxCode = 1;
2 taxCode = 'lowIncome';
3

```

**Figure 2.1 Attempting to change the variable type**

On the left side, you see the TypeScript code entered in the Playground section of [typescriptlang.org](https://typescriptlang.org). But where did we declared the type of the variable `taxCode`? We didn't do it explicitly, but since we initialized it with a numeric value, TypeScript assigns the type `number` to `taxCode`.

The second line is marked with a squiggly line indicating an error. If you hover over this squiggly line, the error message will read "an errorType 'lowIncome' is not assignable to type 'number'." In the TypeScript world, this means that if you declared the variable to store numeric values, don't try to assign string values to it.

The right side in figure 2.1 (the compiled JavaScript code) doesn't show any errors as JavaScript allows changing a variable type during runtime.

Although declaring variable types forces developers write more code, the productivity of developers increases in the long run because more often than not, if a developer tries to assign a string value to the variable that stores a number it's a mistake. It helps that the compiler can catch this human error during development rather than runtime.

The type can be assigned to a variable either explicitly by a software developer or implicitly (a.k.a. *inferred* types) by the TypeScript compiler. Let's get familiar with both methods.

### 2.1.1 Basic type annotations

When you declare a variable, you can add a colon and a *type annotation* to specify the variable type, for example:

```
let firstName: string;
let age: number;
```

TypeScript offers the following type annotations:

- `string` - for textual data
- `boolean` - for true/false values
- `number` - for numeric values
- `symbol` - unique value created by calling the `Symbol` constructor

- `any` - for variables that can hold values of various types, which may be unknown when you write code
- `unknown` - a counterpart of `any` but no operations are permitted on an `unknown` without first asserting or narrowing to a more specific type.
- `never` - for representing an unreachable code (we'll provide an example shortly)
- `void` - an absence of a value

Most of the basic types are self-descriptive, and we just comment some of them.

Starting with ECMAScript 2015, `symbol` is a primitive data type, which is always unique and immutable. In the following code snippet, `sym1` is not equal to `sym2`.

```
const sym1 = Symbol("orderID");
const sym2 = Symbol("orderID");
```

When you create a new symbol (note the absence of the `new` keyword), you can optionally provide its description, e.g. `orderID`. Symbols are typically used to create unique keys for object properties:

### **Listing 2.1 Symbols as object properties**

```
const ord = Symbol('orderID');      ①

const myOrder = {
    ord: "123"                   ②
};

console.log(myOrder['ord']);        ③
```

- ① Creating a new symbol
- ② Using the symbols as the object property
- ③ This line prints 123

Being a superset of JavaScript, TypeScript also has two special values `null` and `undefined`. A variable that has not been assigned a value has a value of `undefined`. The function that doesn't return a value still has a value of `undefined`. The value of `null` represents an intentional absence of value, e.g. `let someVar = null;`.

You can assign `null` and `undefined` values to a variable of any type, but more often we use them in a combination with values of other types. The following code snippet shows how you can declare a function that returns either a `string` or a `null` value (the vertical bar represents a *union type* introduced later in this chapter):

```
function getName(): string | null {
    ...
}
```

As in most programming languages, if you declare a function that returns `string`, you can still

`return null`, but being explicit about what a function can return increases code readability.

If you declare a variable of type `any`, you can assign any value to it being that numeric, textual, boolean, or a custom type like `Customer`. You should avoid using the type `any` as you're loosing the benefits of type checking and the readability of your code suffers.

The `never` type is either assigned to a function that never returns, e.g. either keeps running forever or just throws an error. The following arrow function never returns and the type checker will *infer* (guess) its return type as `never`:

### Listing 2.2 An arrow function that returns the never type

```
const logger = () => {
    while (true) { ①
        console.log("The server is up and running");
    }
};
```

- ① This function never ends

When you assign a function expression to a variable, its type is the same as the return type of the expression. In the above code snippet, the type assigned to `logger` is `never`. In listing 2.9, you'll see an another example where the `never` type is assigned.

The type `void` is not something that you'd use in a variable declaration, but it's used to declare a function that doesn't return a value:

```
function logError(errorMessage: string): void {
    console.error(errorMessage);
}
```

Unlike with the `never` type, the `void` function does complete its execution, but returns no value.

#### TIP

During runtime, when a function body doesn't have a `return` statement, it still returns the value of type `undefined`. However the `void` type annotation can be used to prevent programmers from accidentally returning any explicit value from the function.

The fact that any JavaScript program is a valid TypeScript program means that using type annotations is optional in TypeScript. If some variables don't have explicit type annotations, TypeScript's type checker will try to infer the types.

The following two lines are valid TypeScript syntax:

```
let name1 = 'John Smith'; ①
let name2: string = 'John Smith'; ②
```

- ① Declaring and initializing a variable without a type annotation
- ② Declaring and initializing a variable with a type annotation

The first line declares and initializes a variable `name1` in a JavaScript style, and we can say that the inferred type of `name1` is `string`. Do you think that the second line is a good example of declaring and initializing the variable `name2` in TypeScript? From the code style point of view, specifying types is redundant here.

Although the second line is correct from the TypeScript syntax perspective, specifying the type `string` is unnecessary because the variable is initialized with the string and TypeScript will infer that the type of `name2` is `string`.

You should avoid explicit type annotations where the TypeScript compiler can infer them. The following code snippet declares variables `age` and `yourTax` without specifying their types. There's no need to specify the types of these variables because the TypeScript compiler will infer the types.

### **Listing 2.3 Identifiers with inferred types**

```
const age = 25;          ①

function getTax(income: number): number {
    return income * 0.15;
}

let yourTax = getTax(50000);  ②
```

- ① The constant `age` doesn't declare its type
- ② The variable `yourTax` doesn't declare its type

TypeScript also allows you to use literals as types. The following line declares a variable of *type* John Smith.

```
let name3: 'John Smith';
```

We can say that the variable `name3` has a literal type `John Smith`. The variable `name3` will only allow one value: `John Smith` and an attempt to assign another value to a variable of a literal type would result in type checker's error:

```
let name3: 'John Smith';

name3 = 'Mary Lou'; // error: Type '"Mary Lou"' is not assignable to type '"John Smith"'
```

While it's not likely that you'll be using string literals for declaring a type as shown in the variable `name3`, you may use string literals as types in unions (explained in section 2.1.3) and enums (explained in chapter 4).

**SIDE BAR** **Type Widening**

If you declare a variable without initializing it with a value, the TypeScript compiler will infer its type as `any`. In general, the compiler's attempt to infer the type of the variable is called *type widening*. In JavaScript, the value of the following variable would be `undefined`.

```
let productId;
```

The TypeScript compiler applies type widening and assigns the type `any` to `null` and `undefined` values. Hence the type of the variable `productId` is `any`.

Here are some examples of variables declared with explicit types:

```
let salary: number;
let isValid: boolean;
let customerName: string = null;
```

**TIP**

Add explicit type annotations for function or methods signatures and public class members.

**NOTE**

TypeScript includes other types that are used in interactions with the web browser, such as `HTMLElement` and `Document`. Also, you can use the keywords `type`, `class`, `interface` to declare your own types, e.g. `Customer` or `Person`, and we'll show how to do it in the next section. You'll also see how you can combine types using *unions*.

Type annotations are used not only for declaring variable types, but also for declaring types of function arguments and their return values, which we'll discuss next.

### **2.1.2 Types in function declarations**

TypeScript functions and function expressions are similar to JavaScript functions, but you can explicitly declare the types of arguments and return values. Let's start with writing a JavaScript function (no type annotations) that calculates tax. It'll have three parameters and will calculate tax based on the state, income, and number of dependents. For each dependent, the person is entitled to a \$500 or \$300 tax deduction, depending on the state the person lives in. This function is shown in listing 2.4.

## Listing 2.4 Calculating tax in JavaScript

```
function calcTax(state, income, dependents) {      ①
  if (state === 'NY') {
    return income * 0.06 - dependents * 500;      ②
  } else if (state === 'NJ') {
    return income * 0.05 - dependents * 300;      ③
  }
}
```

- ① The function arguments have no type annotations
- ② Calculate the New York tax
- ③ Calculate the New Jersey tax

Say a person with an income of \$50,000 lives in the state of New Jersey and has two dependents. Let's invoke `calcTax()`:

```
let tax = calcTax('NJ', 50000, 2);
```

The `tax` variable gets the value of 1,900, which is correct. Even though `calcTax()` didn't declare any types for the function parameters, we guessed how to call this function based on the parameter names. What if we didn't guess it right? Let's invoke it the wrong way, passing a `string` value for a number of dependents:

```
let tax = calcTax('NJ', 50000, 'two');
```

You won't know there's a problem until you invoke this function. The `tax` variable will have a `NaN` value (not a number). A bug sneaked in just because you couldn't explicitly specify the types of the parameters and the compiler couldn't infer the types of the function arguments. Listing 2.5 shows another version of this function, using type annotations for the function arguments and return value.

## Listing 2.5 Calculating tax in TypeScript

```
function calcTax(state: string, income: number, dependents: number) : number {      ①

  if (state === 'NY'){
    return income * 0.06 - dependents * 500;
  } else if (state ==='NJ'){
    return income * 0.05 - dependents * 300;
  }
}
```

- ① The function arguments and its return value have type annotations

Now there's no way to make the same mistake and pass a `string` value for the number of dependents:

```
let tax: number = calcTax('NJ', 50000, 'two');
```

The TypeScript compiler will display an error “Argument of type `string` is not assignable to parameter of type `number`”. Moreover, the return value of the function is declared as `number`, which stops you from making another mistake and assigning the result of the tax calculations to a non-numeric variable:

```
let tax: string = calcTax('NJ', 50000, 'two');
```

The compiler will catch this, producing the error “The type ‘number’ is not assignable to type ‘string’: var tax: string.” This kind of type-checking during compilation can save you a lot of time on any project.

#### SIDE BAR    Fixing the function `calcTax()`

This section has JavaScript and TypeScript versions of the function `calcTax()`. These functions process only two states: NY and NJ. Invoking any of these functions for any other state will return `undefined` during runtime. TypeScript compiler won’t give you a warning that the function from listing 2.5 is poorly written and may return `undefined`. But the TypeScript syntax allows you to warn the person who’ll read this code that the function shown in listing 2.5 may return not only the number but also an `undefined` value. You should change this function signature to declare such a use case(s) as follows:

```
function calcTax(state: string, income: number, dependents: number) : number | undefined
```

### 2.1.3 The union type

Unions allow you to express a value that can be one of several types. You can declare a new type based on two or more existing types. For example, you can declare a variable that can accept either a `string` value or a `number` (the vertical bar means union):

```
let padding: string | number;
```

While the variable `padding` can store the value of any of the above two types, at any given time it can be of only one type - either a `string` or a `number`. Although TypeScript supports the type `any`, the preceding declaration provides some benefits compared to the declaration `let padding: any`. Listing 2.6 shows one of the code samples from the TypeScript documentation (see [mng.bz/5742](#)). This function can add the left padding to the provided string. The padding can be specified either as a string that has to prepend the provided value or as a number of spaces that should prepend the provided string.

## Listing 2.6 padleft with the type any

```
function padLeft(value: string, padding: any ): string {           ①
    if (typeof padding === "number") {                            ②
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {                           ③
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'.`); ④
}
```

- ① Provide the string and the padding of type any.
- ② For a numeric argument, generate spaces.
- ③ For a string, use concatenation.
- ④ If the second argument is neither a string nor a number, throw an error.

Listing 2.7 illustrate the use of `padLeft()`:

## Listing 2.7 Invoking the padleft function

```
console.log( padLeft("Hello world", 4));          ①
console.log( padLeft("Hello world", "John says ")); ②
console.log( padLeft("Hello world", true));         ③
```

- ① Returns " Hello world"
- ② Returns "John says Hello world"
- ③ Runtime error

### SIDEBAR Type guards `typeof` and `instanceof`

An attempt to apply conditional statements to refine the variable type is called *type narrowing*, and in the if-statement in listing 2.6, we used the *type guard* `typeof` for narrowing the type of a variable that could store a value of more than one of the TypeScript types. We used `typeof` to find out the actual type of `padding` during runtime.

Similarly, the type guard `instanceof` is used with custom types (with constructors) explained in section "Defining custom types". The `instanceof` guard allows you to check the actual object type during runtime, for example:

```
if (person instanceof Person) {...}
```

The difference between `typeof` and `instanceof` is that the former is used with the built-in TypeScript types while the latter with the custom ones.

But if we change the type of the `padding` to the union of a `string` and `number`, the compiler

will report an error if you try to invoke `padLeft()` providing anything other than `string` or `number`. This will also eliminate the need to throw an exception. Listing 2.8 shows a more bulletproof version of the function `padLeft()`.

### Listing 2.8 padleft with the union type

```
function padLeft(value: string, padding: string | number ): string { ①
    if (typeof padding === "number") {
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
        return padding + value;
    }
}
```

- ① Allow only a string or a number as a second argument.

Now invoking `padLeft()` with the wrong type (for example, `true`) of the second argument returns a compilation error:

```
console.log( padLeft("Hello world", true)); // compilation error
```

#### TIP

If you need to declare a variable that can hold the values of more than one type, don't use the type `any`; use the union, e.g. `let padding: string | number`. Another choice is to declare two separate variables: `let paddingStr: string; let paddingNum: number;`

Let's modify the code in listing 2.8 to illustrate the type `never` that was introduced earlier. This type we'll show how the type checked will infer the `never` type for an impossible value. We'll just add an `else` clause to the `if`-statement from listing 2.8 as shown in listing 2.9:

### Listing 2.9 The never type of an impossible value

```
function padLeft(value: string, padding: string | number ): string {
    if (typeof padding === "number") {
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
        return padding + value;
    }
    else {
        return padding; ①
    }
}
```

- ① This `else` block is never executed

Since we declared in the function signature that the `padding` argument can be either `string` or `number`, any other value for `padding` is impossible. In other words, there is no `else` case here, and the type checked will infer the type `never` for the `padding` variable in the `else` clause. You

can see it for yourself by copying the code from listing 2.7 into the TypeScript playground and hovering the mouse over the `padding` variable.

**NOTE**

Another benefit of using the union type is that IDEs have an auto-complete feature that will prompt you with allowed argument types, so you won't even have a chance to make such a mistake.

Here, we used a union of primitive types (`string` and `number`), but in the next section, you'll see how to declare unions of custom types.

Compare the code of the functions `padLeft()` shown in listing 2.4 and 2.7. What are the main benefits of using the union `string | number` vs the type `any` for the second argument? Using the union will have the TypeScript compiler to eliminate the possibility of the wrong invocation of `padLeft()` by reporting an error during compile time.

## 2.2 Defining custom types

TypeScript allows you to create custom types by using the keyword `type`, by declaring a class or an interface, or by declaring an `enum` (covered in chapter 4). Let's get familiar with the `type` keyword first.

### 2.2.1 Using the `type` keyword

The `type` keyword allows you to declare a new type or a type alias for the existing one. Let's say your app deals with patients that are represented by the name, height, and weight. Both height and weight are numbers, but to improve the readability of your code you can create aliases hinting the units in which the height and weight are measured:

#### **Listing 2.10 Declaring alias types Foot and Pound**

```
type Foot = number;
type Pound = number;
```

You can create a new type `Patient` and use the above aliases in its declaration:

#### **Listing 2.11 Declaring a new type that uses aliases**

```
type Patient = {
  name: string;
  height: Foot;          ①
  weight: Pound;         ②
}
```

- ① Declaring the type `Patient`
- ② Using the type alias `Foot`
- ③ Using the type alias `Pound`

Declarations of type aliases don't generate code in the compiled JavaScript. Declaring and initializing a variable of type `Patient` can look as follows:

### **Listing 2.12 Declaring and initializing type's properties**

```
let patient: Patient = {    ①
  name: 'Joe Smith',
  height: 5,
  weight: 100
}
```

- ① We create an instance using the object literal notation

What if while initializing the variable `patient` you forgot to specify the value of one of the properties, e.g. `weight`?

### **Listing 2.13 Forgetting to add the property weight**

```
let patient: Patient = {
  name: 'Joe Smith',
  height: 5
}
```

TypeScript will complain:

```
"Type '{ name: string; height: number; }' is not assignable to type 'Patient'.
  Property 'weight' is missing in type '{ name: string; height: number; }'."
```

If you want to declare some of the properties as optional, you have to add a question mark to their names. In the following type declaration providing the value for the property `weight` is optional and there won't be any errors:

### **Listing 2.14 Declaring optional properties**

```
type Patient = {
  name: string;
  height: Height;
  weight?: Weight;    ①
}

let patient: Patient = {    ②
  name: 'Joe Smith',
  height: 5
}
```

- ① The property `weight` is optional
- ② The variable `patient` is initialized without the `weight`

**TIP**

You can use the question mark to define optional properties in classes or interfaces as well. You'll get familiar with TypeScript classes and interfaces later in this section.

You can also use the `type` keyword for declaring a type alias to a function signature. Imagine, you're writing a framework that should allow to create form controls and assign the validator function(s) to them. The validator function must have a specific signature - accept the object of type `FormControl` and return either an object describing the errors of the form control value, or `null` if the value is valid. You can declare a new type `ValidatorFn` as follows:

```
type ValidatorFn =
  (c: FormControl) => { [key: string]: any } | null
```

Here, `{ [key: string]: any }` means an object that can have properties of any type.

The constructor of the `FormControl` can have a parameter for the validator function, and it can use the custom type `ValidatorFn` as follows:

```
class FormControl {

  constructor (initialValue: string, validator: ValidatorFn | null) {...};
```

**TIP**

In appendix A, you can see the syntax for declaring optional function parameters in JavaScript. The above code snippet shows you another way of declaring an optional parameter using the TypeScript union type.

## 2.2.2 Using classes as custom types

We assume you're familiar with JavaScript classes covered in appendix A. In this section, we'll start showing you additional features that TypeScript brings to JavaScript classes. In chapter 3, we'll continue class coverage in more detail.

JavaScript doesn't offer syntax for declaring class properties, but TypeScript does. In figure 2.2 on the left, you can see how we declared and instantiated the class `Person` that has three properties. The right side of figure 2.2 shows the ES6 version of this code produced by the TypeScript compiler:

TypeScript	JavaScript (ES6)
<pre>1  class Person { 2    firstName: string; 3    lastName: string; 4    age: number; 5  } 6 7  const p = new Person(); 8  p.firstName = "John"; 9  p.lastName = "Smith"; 10 p.age = 25;</pre>	<pre>1  "use strict"; 2  class Person { 3  } 4  const p = new Person(); 5  p.firstName = "John"; 6  p.lastName = "Smith"; 7  p.age = 25;</pre>

Figure 2.2 The class `Person` compiled into JavaScript (ES6)

As you see, there are no properties in the JavaScript version of the class `Person`. Also, since the

class `Person` didn't declare a constructor, we had to initialize its properties after instantiating. A constructor is a special function that's executed once when the instance of a class is created.

Declaring a constructor with three arguments would allow you to instantiate the class `Person` and initialize its properties in one line. In TypeScript, you can provide type annotation for constructor's arguments, but there's more.

TypeScript offers access level qualifiers `public`, `private`, and `protected` (covered in chapter 3), and if you use any of them with the constructor arguments, the TypeScript will generate the code for adding these arguments as properties in the generated JavaScript object as shown in figure 2.3.

TypeScript	JavaScript (ES6)
<pre> 1  class Person { 2    constructor(public firstName: string, 3              public lastName: string, public age: number) {} 4  } 5 6  const p = new Person("John", "Smith", 25); 7 8 9 10 </pre>	<pre> 1  "use strict"; 2  class Person { 3    constructor(firstName, lastName, age) { 4      this.firstName = firstName; 5      this.lastName = lastName; 6      this.age = age; 7    } 8    ; 9  } 10 const p = new Person("John", "Smith", 25); </pre>

**Figure 2.3 The class `Person` with constructor**

Now the code of the TypeScript class (on the left) is more concise and the generated JavaScript code creates three properties in the constructor. We'd like to bring your attention to line 6 in figure 2.3 on the left. We declared the constant without specifying its type, but we could have re-written this line explicitly specifying the type of `p` as follows:

```
const p: Person = new Person("John", "Smith", 25);
```

This illustrates an unnecessary use of an explicit type annotation. Since you declare a constant and immediately initialize it with an object of a known type (i.e. `Person`), the TypeScript type checker can easily infer and assign the type to the constant `p`. The generated JavaScript code will look the same with or without specifying the type of `p`. To see it in action, follow this link: [bit.ly/2tahDTp](http://bit.ly/2tahDTp).

**TIP**

We use the `public` access level with each constructor argument in the TypeScript class, which simply means that the generated corresponding properties can be accessed from any code located both inside and outside of the class.

When you declare properties of a class, you can also mark them as `readonly`. Such properties can be initialized either at the declaration point or in the class constructor, and their values can't be changed afterwards. The `readonly` qualifier is similar to the `const` keyword, but the latter

can't be used with class properties.

In chapter 8, we'll start developing a blockchain app, and any blockchain consists of blocks with immutable properties. That app will include a class `Block` and its fragment is shown in listing 2.15.

### **Listing 2.15 The properties of the Block class**

```
class Block {
  readonly nonce: number;      ①
  readonly hash: string;       ①

  constructor (
    readonly index: number,      ②
    readonly previousHash: string, ②
    readonly timestamp: number,   ②
    readonly data: string        ②
  ) {
    const { nonce, hash } = this.mine();  ③
    this.nonce = nonce;
    this.hash = hash;
  }
  // The rest of the code is omitted for brevity
}
```

- ① This property is initialized in the constructor
- ② The value for this property is provided to the constructor during instantiation
- ③ Using destructuring to extract the values from the object returned by the method `mine()`

The class `Block` includes six `readonly` properties. We'd like to reiterate that we don't need to explicitly declare class properties for the constructor arguments that have `readonly`, `private`, `protected`, or `public` qualifiers like we do in other object-oriented languages. In listing 2.15, two class properties are declared explicitly, and four - implicitly.

### **2.2.3 Using interfaces as custom types**

Many object-oriented languages include a syntax construct called `interface`, which is used to enforce the implementation of specified properties or methods on an object. JavaScript doesn't support interfaces but TypeScript does.

In this section we'll show you how to use interfaces for declaring a custom type, and in chapter 3, you'll see how to use interfaces to ensure that a class implements the specified members.

TypeScript includes the keywords `interface` and `implements` to support interfaces, but interfaces aren't compiled into JavaScript code. They just help you to avoid using wrong types during development. Let's get familiar with the use of the `interface` keyword for declaring a custom type.

Let's say we want to write a function that can save the information about people in some storage. This function should take an object that represents a person, and we want to ensure that this object has specific properties of specific types. You can declare an interface `Person` as follows:

### **Listing 2.16 Declaring a custom type using an interface**

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

The script on left side of figure 2.2 declares a similar custom type `Person` but using the `class` keyword. What's the difference? If you declare a custom type as `class`, you can use it as a value, i.e. you can instantiate it using the `new` keyword as shown in figures 2.2 and 2.3.

Also, if you use the `class` keyword in the TypeScript code, it'll have the corresponding code in generated JavaScript (e.g. a function in ES5 and class in ES6). If you use the keyword `interface`, there won't be any corresponding code in JavaScript as seen in figure 2.4.

<pre>1  interface Person { 2    firstName: string; 3    lastName: string; 4    age: number; 5  } 6 7  function savePerson (person: Person): void { 8    console.log('Saving ', person); 9  } 10 11 const p: Person = { 12   firstName: "John", 13   lastName: "Smith", 14   age: 25 }; 15 16 savePerson(p);</pre>	<pre>1  "use strict"; 2  function savePerson(person) { 3    console.log('Saving ', person); 4  } 5  const p = { 6    firstName: "John", 7    lastName: "Smith", 8    age: 25 9  }; 10 savePerson(p); 11</pre>
---	---

**Figure 2.4 The custom type Person as an interface**

Note that there is no mentioning of the interface on the right side, and the JavaScript is more concise, which is good for any deployable code. But during development, the compiler will check that the object given as an argument to the function `savePerson()` includes all the properties declared in the interface `Person`.

We encourage you to experiment with the code snippet from figure 2.4 by visiting the following link: [bit.ly/2MBwzSf](https://bit.ly/2MBwzSf). For example, remove the `lastName` property defined in line 13. The TypeScript type checker will immediately underline the variable `p` with a red line. Hover the mouse pointer over the variable `p`, and you'll see the following error message:

```
Type '{ firstName: string; age: number; }' is not assignable to type 'Person'.
  Property 'lastName' is missing in type '{ firstName: string; age: number; }'.
```

Keep experimenting: Try to access `person.lastName()` inside `savePerson()`. If the interface `Person` doesn't declare the property `lastName`, TypeScript would give you a compiler error, but

the JavaScript code would just crash during runtime.

Try another experiment: remove the type annotation `Person` in line 11. The code is still valid and no errors are reported in line 16. Why TypeScript allows you to invoke the function `savePerson()` with an argument that was not explicitly assigned the type `Person`? The reason is that TypeScript has a structural type system, which means if two different types include the same members, the types are considered compatible. We'll discuss structural type system in the next section in more detail.

#### SIDE BAR    Which keyword to use: type, interface, or class?

We've shown you that a custom type could be declared using the keywords `type`, `class`, or `interface`. Which of these keyword should you use for declaring a custom type like `Person`?

If the custom type doesn't need to be used for instantiating objects during runtime, use `interface` or `type`; otherwise use `class`. In other words, use `class` for creating a custom type if it should be used to represent a *value*.

If you declare a custom type just for additional safety offered by the TypeScript's type checker, use `type` or `interface`. Neither interfaces nor types declared with the `type` keyword have representations in the emitted JavaScript code, which makes the runtime code smaller (byte wise). If you use classes for declaring types, they will have a footprint in the generated JavaScript.

Defining a custom type with the `type` keyword offers the same features as `interface` plus some extras. For example, you can't use types declared as interfaces in unions or intersections. Also, in chapter 5, you'll learn about conditional types, which can't be declared using interfaces.

### 2.2.4 Structural vs nominal type systems

A primitive type has just a name (e.g. `number`) while a more complex type like an object or class has a name and some structure represented by properties (e.g. a class `Customer` has properties `name` and `address`).

How would you know if two types are the same or not? In some languages (e.g. Java) two types are the same if they have the same names, which represents a *nominal type system*. In Java, the last line wouldn't compile because the names of the classes are not the same even though they have the same structure:

### Listing 2.17 A Java code snippet

```
class Person {    ①
    String name;
}

class Customer {    ②
    String name;
}

Customer cust = new Person();    ③
```

- ① Declare the class (think type) Person
- ② Declare the class Customer
- ③ Syntax error: the names of the classes on the left and right are not the same

But TypeScript and some other languages use the *structural type system*. In listing 2.18, we re-wrote the above code snippet in TypeScript:

### Listing 2.18 A TypeScript code snippet

```
class Person {    ①
    name: string;
}

class Customer {    ②
    name: string;
}

const cust: Customer = new Person();    ③
```

- ① Declare the class (think type) Person
- ② Declare the class Customer
- ③ No errors: the type structures are the same

This code doesn't report any errors because TypeScript uses a structural type system, and since both classes `Person` and `Customer` have the same structure, it's OK to assign an instance of one class to a variable of another.

Moreover, you can use object literals to create objects and assign them to class-typed variables or constants as long as the shape of the object literal is the same. The following code snippet will compile without errors:

## Listing 2.19 Compatible types

```
class Person {
    name: String;
}

class Customer {
    name: String;
}

const cust: Customer = { name: 'Mary' };
const pers: Person = { name: 'John' };
```

**TIP**

The access level modifiers affect type compatibility. For example, if we'd declare the `name` property of the class `Person` as `private`, the code in listing 2.19 wouldn't compile.

Our classes didn't define any methods, but if both of them would define a method(s) that has the same signature (name, arguments, and the return type) they would also be compatible.

What if the structure of `Person` and `Customer` are not exactly the same? Let's add a property `age` to the class `Person` as in listing 2.20:

## Listing 2.20 When classes are not the same

```
class Person {
    name: String;
    age: number; ①
}

class Customer {
    name: String;
}

const cust: Customer = new Person(); // still no errors
```

- ① We've added this property

Still no errors! TypeScript sees that `Person` and `Customer` have *the same shape* (something in common). We just wanted to use the constant of type `Customer` (it has the property `name`) to point at the object of type `Person`, which also has the property `name`.

What can you do with the object represented by the variable `cust`? You can write something like `cust.name='John'`. The instance of `Person` has the property `name` so compiler doesn't complain.

**NOTE**

Since we can assign an object of type `Person` to variable of type `Customer`, we can say that the type `Person` is **assignable** to type `Customer`.

Follow the link [bit.ly/2MbHvpH](https://bit.ly/2MbHvpH) and you'll see this code in TypeScript playground. Click Ctrl-Space after the dot in `cust.` and you'll see that only the `name` property is available even though the class `Person` has also the property `age`.

In listing 2.20, the class `Person` had more properties than `Customer`, and the code compiled without errors. Would the code compile if the class `Customer` had more properties than `Person` as shown below?

### **Listing 2.21 The instance has more properties than a reference variable**

```
class Person {
    name: string;
}

class Customer {
    name: string;
    age: number;
}

const cust: Customer = new Person(); ❶
```

- ❶ The types don't match

The code from listing 2.21 wouldn't compile because the reference variable `cust` would point to an object `Person` that wouldn't even allocate memory for the property `age`, and the assignment like `cust.age = 29` wouldn't be possible. This time, the type `Person` is *not assignable* to type `Customer`.

**TIP**

We'll come back to the TypeScript structural typing in section 4.2 in chapter 4 while discussing generics.

#### **2.2.5 Unions of custom types**

In the previous section, we introduced union types that allow you to declare that a variable can have one of the listed types. For example, in listing 2.8 we specified that the function argument `padding` can be *either* a `string` *or* `number`. This is an example of the union that included primitive types. Obviously, a variable can't be a `string` and a `number` at the same time, but it can be one of them.

Let's see how you can declare a union of custom types. Imagine an app that can dispatch various actions as responses to the user's activity. Each action is represented by a class with a different name. Each action must have a type and optionally may carry a payload, e.g. a search query. Listing 2.22 includes the declarations of three action classes and a union type `SearchAction`.

## Listing 2.22 actions.ts

```
export class SearchAction {    ①
  actionTypes = "SEARCH";

  constructor(readonly payload: {searchQuery: string}) {}
}

export class SearchSuccessAction {  ①
  actionTypes = "SEARCH_SUCCESS";

  constructor(public payload: {searchResults: string[]}) {}
}

export class SearchFailedAction {  ②
  actionTypes = "SEARCH_FAILED";
}

export type SearchActions = SearchAction | SearchSuccessAction | SearchFailedAction;  ③
```

- ① A class with the action type and payload
- ② A class with the action type but without payload
- ③ The union type declaration

**TIP**

The code in listing 2.22 needs an improvement because just stating that each action must have a property describing its type is more of a JavaScript style of programming. In TypeScript, such a statement can be enforced programmatically, and we'll do it in section "Interfaces as contracts" in chapter 3.

*Discriminated unions* include type members that have a common property - the discriminant. Depending on the value of the discriminant, you may want to perform different actions.

The union shown in listing 2.22 is an example of a discriminated union because each member has a discriminant `actionType`. Let's create another discriminated union of two types: `Rectangle` and `Circle` as shown in listing 2.23

## Listing 2.23 A union with the discriminant

```
interface Rectangle {
  kind: "rectangle";  ①
  width: number;
  height: number;
}
interface Circle {
  kind: "circle";  ①
  radius: number;
}

type Shape = Rectangle | Circle;  ②
```

- ① The discriminant

## ② The union

The type `Shape` is a discriminated union, and both `Rectange` and `Circle` have a common property `kind`. Depending on the value in the `kind` property, we can calculate the area of the `Shape` differently as seen in listing 2.24.

### **Listing 2.24 Using the discriminated union**

```
function area(shape: Shape): number {
  switch (shape.kind) { ①
    case "rectangle": return shape.height * shape.width; ②
    case "circle": return Math.PI * shape.radius ** 2; ③
  }
}

const myRectangle: Rectangle = { kind: "rectangle", width: 10, height: 20 };
console.log(`Rectangle's area is ${area(myRectangle)}`);

const myCircle: Circle = { kind: "circle", radius: 10 };
console.log(`Circle's area is ${area(myCircle)}');
```

- ① Switching on the discriminator's value
- ② Applying the formula for rectangles
- ③ Applying the formula for circles

You can run this code sample in the Playground at [bit.ly/2P05VEx](https://bit.ly/2P05VEx).

**SIDE BAR** **The `in` type guard**

The type guard `in` acts as a narrowing expression for types. For example, if you have a function that can take an argument of a union type, you can check the actual type given during the function invocation.

Listing 2.25 shows two interfaces with different properties. The function `foo()` can take either an object `A` or `B` as an argument. Using the `in` type guard, the function `foo()` can check if the provided object has a specific property before using it.

**Listing 2.25 Using the type guard `in`**

```
interface A { a: number };
interface B { b: string };

function foo(x: A | B) {
    if ("a" in x) { ①
        return x.a;
    }
    return x.b;
}
```

- ① Check for a specific property using `in`

The property that you check has to be a string, e.g. "a".

## 2.3 Types `any`, `unknown`, and user-defined type guards

In the beginning of this chapter, we mentioned the types `any` and `unknown`. In this section, we'll show you the difference between them. You'll also see how to write a custom type guard in addition to `typeof`, `instanceof`, and `in`.

Declaring a variable of type `any` allows you to assign to it a value of any type. It's like writing in JavaScript where you don't specify a type. Similarly, trying to access a non-existing property on an object of type `any`, may give unexpected result during runtime.

The type `unknown` was introduced in TypeScript 3.0. If you declare a variable of type `unknown`, the compiler will force you to narrow its type down before accessing its properties, sparing you from potential runtime surprises.

To illustrate the difference between `any` and `unknown`, let's assume that we declared the type `Person` on the front end, and it's populated with the data coming from the back end in the JSON format. To turn the JSON string into an object, we'll use the method `JSON.parse()` that returns `any`.

## Listing 2.26 Using the type any

```
type Person = {  
    address: string;  
}  
  
let person1: any;      ②  
  
person1 = JSON.parse('{"address": "25 Broadway"}');  ③  
  
console.log(person1.address);  ④
```

- ① Declaring a type alias
- ② Declaring a variable of type any
- ③ Parsing the JSON string
- ④ This prints undefined

The last line will print `undefined` because we misspelled the word `address` in the JSON string. The method `parse` returns the JavaScript object that has a property `adress`, but not `address` on `person1`. To experience this issue, you need to run this code.

Now let's see how the same use case works with the variable of type `unknown`.

## Listing 2.27 The compiler's error with the type unknown

```
let person2: unknown;  ①  
  
person2 = JSON.parse('{"address": "25 Broadway"}');  
  
console.log(person2.address);  ②
```

- ① Declaring a variable of type `unknown`
- ② Attempting to use a variable of the `unknown` type results in compilation error

This time the last line won't even compile because we tried to use the variable `person2` of type `unknown` without narrowing its type down.

TypeScript allows you to write user-defined type guards that could check if an object is of a particular type. This would be a function that returns something like "thisFunctionArg is `SomeType`". Let's write a type guard `isPerson()` assuming that if the object under test has the property `address`, it's a person.

## Listing 2.28 The first version of the type guard `isPerson`

```
const isPerson = (object: any): object is Person => "address" in object;
```

This type guard returns `true` if the given object has the `address` property. Now you can apply this guard as shown in listing 2.29.

### Listing 2.29 Applying the type guard `isPerson`

```
if (isPerson(person2)) {    ①
  console.log(person2.address);  ②
} else {
  console.log("person2 is not a Person");
}
```

- ① Applying the type guard
- ② Safely accessing the property address

This code has no compilation errors, and it works as expected unless the guard `isPerson()` gets a falsy object as an argument. For example, passing `null` to `isPerson()` will result in a runtime error in the expression `"address"` in object.

Listing 2.30 shows a safer version of the `isPerson()` guard. The double bang operator `!!` will ensure that the given object is truthy.

### Listing 2.30 The type guard `isPerson`

```
const isPerson = (object: any): object is Person => !!object && "address" in object;
```

You can try this code in the TypeScript playground at [bit.ly/2Vs2dp6](https://bit.ly/2Vs2dp6).

In this example, we assumed that the existence of the property `address` is enough to identify the `Person` type. In some cases, checking just one property is not enough, e.g. the classes `Organization` or `Pet` may also have the property `address`. In other words, you'd need to check several properties to determine if the object matched a specific type.

A simpler solution is to declare your own discriminator property that would identify this type as a person:

```
type Person = {
  discriminator: 'person';
  address: string;
}
```

Then your custom type guard could look like this:

```
const isPerson = (object: any): object is Person => !!object && object.discriminator === 'person';
```

OK, we've covered just enough of the TypeScript syntax related to types, and it's time to apply the theory in practice.

## 2.4 A mini project

If you're a type of person who prefers learning by doing, we can offer you a little assignment, which will be followed by a solution. We won't provide detailed explanations to the offered solution - just see if you can understand it or offer a better one.

Write a program with two custom types `Dog` and `Fish`, which are declared using classes. Each of these types has to have a property `name`. The class `Dog` should have a method `sayHello(): string`, and the class `Fish` should have the method `dive(howDeep: number): string`.

Declare a new type `Pet` as a union of `Dog` and `Fish`. Write a function `talkToPet(pet: Pet): string` that will use type guards and will invoke on the pet either the method `sayHello()` or `dive()`.

Invoke `talkToPet()` three times providing the object `Dog` first, then `Fish`, and finally, an object, which is neither `Dog` nor `Fish`.

The solution is shown in listing 2.31.

### Listing 2.31 The solution

```
class Dog {    ①
  constructor(readonly name: string) { };

  sayHello(): string {
    return 'Dog says hello!';
  }
}

class Fish {    ②
  constructor(readonly name: string) { };

  dive(howDeep: number): string {
    return `Diving ${howDeep} feet`;
  }
}

type Pet = Dog | Fish;    ③

function talkToPet(pet: Pet): string {

  if (pet instanceof Dog) {    ④
    return pet.sayHello();
  } else if (pet instanceof Fish) {
    return 'Fish cannot talk, sorry.';
  }
}

const myDog = new Dog('Sammy');    ⑤
const myFish = new Fish('Marry');  ⑥

console.log(talkToPet(myDog));    ⑦
console.log(talkToPet(myFish));   ⑦
talkToPet({ name: 'John' });      ⑧
```

<sup>①</sup> Declaring a custom type `Dog`

- ② Declaring a custom type Fish
- ③ Creating a union of Dog and Fish
- ④ Using a type guard
- ⑤ Creating an instance of a Dog
- ⑥ Creating an instance of a Fish
- ⑦ Invoking talkToPet() passing a Pet
- ⑧ This won't compile - wrong parameter type

You can see this script in action at [codepen.io/yfain/pen/yqZxdL?editors=1011](https://codepen.io/yfain/pen/yqZxdL?editors=1011)

## 2.5 Summary

This chapter illustrated multiple ways of using types. First, we used built-in TypeScript primitive types. Then, we explained three different methods for creating custom types:

- Using the keyword `type`
- Using the keyword `class`
- Using the keyword `interface`

We also showed you how to combine two or more types into a new one using the keyword `union`. In chapter 3, we'll continue learning about TypeScript classes and interfaces.



# *Object-oriented programming with classes and interfaces*

## **This chapter covers:**

- How class inheritance works
- Why and when to use abstract classes
- How interfaces can force a class to have methods with known signatures without worrying about implementation details
- What "programming to interfaces" means

In chapter 2, we introduced classes and interfaces for creating custom types. In this chapter, we'll continue learning classes and interfaces from the object-oriented programming (OOP) perspective. OOP is a programming style where your programs are about handling objects rather than being a composition of actions (think functions). Of course, some of these functions would create objects as well, but in OOP, objects are the center of the universe.

Developers who work with object-oriented languages use interfaces as a way to enforce certain API on classes. Also, you can often hear the phrase "program to interfaces" in conversations of programmers. In this chapter, we'll explain what it means. In short, this chapter is a whirlwind tour of OOP using TypeScript.

## **3.1 Working with classes**

Let's recap what we know about TypeScript classes from chapter 2:

- You can declare classes with properties, which in other object-oriented languages are called member variables.
- As in JavaScript, classes may declare constructors, which are invoked once during instantiation.
- The TypeScript compiler converts classes into JavaScript constructor functions if the ESS

is specified as a compilation target syntax. If ES6 or later is specified as a target, TypeScript classes will be compiled into JavaScript classes.

- If a class constructor defines arguments that use such keywords as `readonly`, `public`, `protected`, and `private`, TypeScript creates class properties for each of such arguments.

But there's more to classes, and in this chapter, we'll cover class inheritance, what the abstract classes are for, and what's the use of the access modifiers `public`, `protected`, and `private`.

### 3.1.1 Getting familiar with class inheritance

In real life, every person inherits some features from his or her parents. Similarly, in the TypeScript world, you can create a new class, based on an existing one. For example, you can create a class `Person` with some properties and the class `Employee` that will *inherit* all the properties of `Person` as well as declare additional ones.

Inheritance is one of the main features of any object-oriented language, and the keyword `extends` is used to declare that one class inherits from another.

The line 7 in figure 3.1 on the left, shows how to declare an `Employee` class that extends the class `Person` and declares an additional property `department`. In line 11 on the left, we create an instance of the class `Employee`.

```

1 class Person {
2   firstName: string;
3   lastName: string;
4   age: number;
5 }
6
7 class Employee extends Person {
8   department: string;
9 }
10
11 const empl = new Employee();
12
13 empl.
14   ↗ age          (property) Person.age: number
15   ↗ department
16   ↗ firstName
17   ↗ lastName

```

**Figure 3.1 Class inheritance in TypeScript**

This screenshot was taken from the playground at [typescriptlang.org](https://typescriptlang.org) after we entered `empl`, followed by CTRL-Space on line 13. The TypeScript's static analyzer recognizes that the type `Employee` is inherited from `Person` so it suggests the properties defined in both classes `Person` and `Employee`.

In our example, the class `Employee` is a *subclass* of `Person`. Accordingly, the class `Person` is a *superclass* of `Employee`. You can also say that the class `Person` is an *ancestor* and `Employee` is a descendant of `Person`.

**NOTE**

Under the hood, JavaScript supports prototypal *object-based* inheritance, where one object can be assigned to another object as its prototype, and this happens during the runtime. During transpiling TypeScript to JavaScript, the generated code uses the syntax of the prototypal inheritance.

In addition to properties, a class can include *methods* - this is how we call functions declared inside the classes. And if a method(s) is declared in a superclass, it will be inherited by the subclass unless the method was declared with the access qualified `private` which we'll discuss a bit later.

The next version of the class `Person` is shown in figure 3.2, and it includes the method `sayHello()`. As you can see in line 17, the TypeScript's static analyzer included this method in the auto-complete dropdown.

The screenshot shows a code editor interface with tabs for 'Select...', 'TypeScript' (which is selected), 'Share', and 'Options'. The code itself is as follows:

```

1 class Person {
2     firstName: string;
3     lastName: string;
4     age: number;
5
6     sayHello(): string {
7         return `My name is ${this.firstName} ${this.lastName}`;
8     }
9 }
10
11 class Employee extends Person {
12     department: string;
13 }
14
15 const empl = new Employee();
16
17 empl.

```

At line 17, the cursor is at the end of `empl.` and an auto-complete dropdown is open, listing the following properties:

- age (property) `Person.age: number`
- department
- firstName
- lastName
- sayHello

**Figure 3.2 A method `sayHello()` from a superclass is visible**

You may be wondering, "Is there any way to control which properties and methods of a class are accessible from other scripts?" The answer is yes - this is what the `private`, `protected`, and `public` keywords are for.

### 3.1.2 Access modifiers `public`, `private`, `protected`

TypeScript includes the keywords `public`, `protected`, and `private` to control access to the members of a class i.e. properties or methods. Class members marked as `public` can be accessed from the internal class methods as well as from external scripts. This is a default access, so if you place the keyword `public` in front of each property or the `sayHello()` method of the class `Person` shown in figure 3.2, nothing will change in terms of accessibility of these class members.

Class members marked as `protected` can be accessed either from the internal class code or from class descendants.

The `private` class members are visible only within the class.

**NOTE**

If you know languages like Java or C#, you may already know the concept of restricting the access level with `private` and `protected` keywords. But TypeScript is a superset of JavaScript, which doesn't support the `private` keyword, so the keywords `private` and `protected` (as well as `public`) are removed during the code compilation. The resulting JavaScript won't include these keywords and you can consider them just as a convenience during development.

Figure 3.3 illustrates the `protected` and `private` access level modifiers. In line 15, we can access the `protected` ancestor's method `sayHello()`, because we do this from the descendant. But when we clicked Ctrl-Space after `this.` in line 21, the variable `age` is not shown in the auto-complete list because it's declared as `private` and can be accessed only within the class `Person`.

The screenshot shows a code editor with the following TypeScript code:

```

1 class Person {
2   public firstName: string;
3   public lastName: string;
4   private age: number;
5
6   protected sayHello(): string {
7     return `My name is ${this.firstName} ${this.lastName}`;
8   }
9 }
10
11 class Employee extends Person {
12   department: string;
13
14   reviewPerformance(): void{
15     this.sayHello();
16
17     this.increasePay(5);
18   }
19
20   increasePay(percent: number): void {
21     this.
22   }
23 }
24
25
26

```

A tooltip is displayed over the line `this.` in the `increasePay` method, listing the following properties:

- department (property) Employee.department: string
- firstName
- increasePay
- lastName
- reviewPerformance
- sayHello

**Figure 3.3 The private property age is not visible**

This code sample shows that the subclass can't access the `private` member of the superclass. In general, only a method from the class `Person` can access `private` members from this class.

To try this code on your own, visit the TypeScript playground at [goo.gl/wFfqYk](http://goo.gl/wFfqYk).

While `protected` class members are accessible from the descendant's code, they are not accessible on the class instance. For example, the following code won't compile and will give you the error "*Property 'sayHello' is protected and only accessible within class 'Person' and its subclasses*":

```
const empl = new Employee();
empl.sayHello(); // error
```

Let's look at another example of the class `Person`, which has a constructor, two `public` and one `private` property as seen in figure 3.4. This is a verbose version of the class declaration because we explicitly declared three properties of the class. The constructor in the class `Person` performs a tedious job of assigning the values from its argument to the respective properties of this class.

The screenshot shows the TypeScript playground interface with the following code:

On the left (TypeScript tab):

```

Select... TypeScript Share Options
1 class Person {
2   public firstName: string;
3   public lastName: string;
4   private age: number;
5
6   constructor(firstName:string, lastName: string, age: number)
7     this.firstName = firstName;
8     this.lastName;
9     this.age = age;
10 }
11

```

On the right (JavaScript tab):

```

Run JavaScript
1 var Person = /** @class */ (function () {
2   function Person(firstName, lastName, age) {
3     this.firstName = firstName;
4     this.lastName;
5     this.age = age;
6   }
7   return Person;
8 })();
9

```

**Figure 3.4 A verbose version of the class**

Now let's declare a more concise version the class `Person` as shown in figure 3.5 on the left. By using access qualifiers with the constructor's arguments, we instruct the TypeScript compiler to create class properties having the same names as constructor's arguments. The compiler will auto-generate the JavaScript code to assign the values given to the constructor to class properties.

The screenshot shows the TypeScript playground interface. On the left, the TypeScript code is as follows:

```

1 class Person {
2     constructor(public firstName: string,
3                 public lastName: string,
4                 public age: number) { }
5 }
6
7 const pers = new Person('John', 'Smith', 29);
8
9 console.log(` ${pers.firstName} ${pers.age}`)
10

```

On the right, the generated JavaScript code is:

```

1 var Person = /** @class */ (function () {
2     function Person(firstName, lastName, age) {
3         this.firstName = firstName;
4         this.lastName = lastName;
5         this.age = age;
6     }
7     return Person;
8 })();
9 var pers = new Person('John', 'Smith', 29);
10 console.log(" " + pers.firstName + " " + pers.age);

```

**Figure 3.5 Using access qualifiers with the constructor's arguments**

In line 8 on the left, we create an instance of the class `Person` passing the initial property values to its constructor, which will assign these values to the respective object's properties. In line 10, we wanted to print the values of the object's properties `firstName` and `age`, but the latter is marked with a red squiggly line. If you hover the mouse over the erroneous fragment, you'll see that the TypeScript's static analyzer (it runs even before the compiler) properly reports an error:

*Property 'age' is private and only accessible within class 'Person'.*

In the TypeScript playground, the JavaScript code is generated anyway because from the JavaScript perspective, the code in line 10 (on the right) is perfectly fine. But in your projects, you can use the compiler's option `noEmitOnError` to prevent the generation of JavaScript until all TypeScript syntax errors are fixed.

Compare the left sides of figures 3.4. and 3.5. In figure 3.4, the class `Person` explicitly declares three properties, which we initialize in the constructor. In figure 3.5, the class `Person` has no explicit declarations of properties and no explicit initialization in the constructor.

Now compare the generated JavaScript versions of the class declaration (lines 1-8) on the right sides in figures 3.4 and 3.5 - they're literally the same.

So what's better - explicit or implicit declaration of class properties? There are arguments for both programming styles. Explicit declaration and initialization of the class properties may increase the readability of the code while implicit declaration makes the code of the TypeScript class more concise. But it doesn't have to be an either-or decision. For example, you can declare `public` properties explicitly and `private` and `protected` - implicitly. Most of the times, we use implicit declarations unless the property initialization involves some logic.

### 3.1.3 Static variables and a singleton example

In JavaScript, starting from ES6, when a some property has to be shared by each instance of this class, we declare it as `static` (see section A.9.3 in appendix A). Being a superset of JavaScript, TypeScript supports the `static` keyword as well. In this section we'll start with a basic example, and then will implement a Singleton design pattern with the help of a `static` property and `private` constructor.

So a group of gangsters is on a mission (no worries - it's just a game). We need to monitor the total number of bullets left. Every time any gangster shoots, this value has to be decreased by one. The total number of bullets should be known to each gangster.

#### **Listing 3.1 A gangster with a static property**

```
class Gangsta {
    static totalBullets = 100;      ①

    shoot(){
        Gangsta.totalBullets--;   ②
        console.log(`Bullets left: ${Gangsta.totalBullets}`);
    }
}

const g1 = new Gangsta();      ③
g1.shoot();                  ④

const g2 = new Gangsta();      ③
g2.shoot();                  ④
```

- ① Declare and initialize a static variable
- ② Update the number of bullets after each shot
- ③ Create a new instance of the Gangsta
- ④ This gangster shoots once

After running the code shown in listing 3.1 (you can see it in the playground at [bit.ly/2OCK587](https://bit.ly/2OCK587)), the browser console will print the following:

```
Bullets left: 99
Bullets left: 98
```

Both instances of the class `Gangsta` share the same variable `totalBullets`. That's why no matter which gangster shoots, the shared variable `totalBullets` is updated. Note that in the method `shoot()` we didn't write `this.totalBullets` because this is not an instance variable. Hence you access static class members by prepending their names with the class name, e.g. `Gangsta.totalBullets`.

**NOTE**

Static class members are not shared by subclasses. If you create the `SuperGangsta` class that subclasses `Gangsta`, it will get its own copy of the property `totalBullets`. We provided an example at [goo.gl/3BSnjZ](http://goo.gl/3BSnjZ).

Now let's consider another example. Imagine, you need to create a single place that serves as a storage of important data in memory representing the current state of the app. Various scripts can have an access to this storage but you want to make sure that only one such object can be created for the entire app, also known as *a single source of truth*.

*Singleton* is a popular design pattern that restricts the instantiation of a class to only one object. How do you create a class that you can instantiate only once? It's a rather trivial task in any object-oriented language that supports the `private` access qualifier.

Basically, you need to write a class that won't allow using the `new` keyword, because with the `new`, you can create as many instances as you want. The idea is simple - if a class has a `private` constructor, the operator `new` will fail.

Then, how to create even a single instance of the class? The thing is that if the class constructor is `private`, you can access it only within the class, and as the author of this class, you'll responsibly create it only once by invoking that same `new` operator from the class method.

But can you invoke a method on a class that was not instantiated? You can do it by making the class method `static`, which doesn't belong to any particular object instance and belongs to the class.

Listing 3.2 shows our implementation of the singleton design pattern in a class `AppState`, which has the property `counter`. Let's assume that the `counter` represents our app state, which may be updated from multiple scripts in the app, and there should be the only place that stores the value of the `counter`, which is the single instance of `AppState`. Any script that needs to know the latest value of the `counter` will also get it from this `AppState` instance.

## Listing 3.2 A singleton class

```
class AppState {
    counter = 0;      ①
    private static instanceRef: AppState;   ②

    private constructor() { }      ③

    static getInstance(): AppState { ④
        if (AppState.instanceRef === undefined) {
            AppState.instanceRef = new AppState(); ⑤
        }
        return AppState.instanceRef;
    }
}

// const appState = new AppState(); // error because of the private constructor

const appState1 = AppState.getInstance(); ⑥
const appState2 = AppState.getInstance(); ⑥

appState1.counter++; ⑦
appState1.counter++; ⑦
appState2.counter++; ⑦
appState2.counter++; ⑦

console.log(appState1.counter); ⑧
console.log(appState2.counter); ⑧
```

- ① This property represents our app state
- ② This property stores the reference to the single instance of AppState
- ③ Private constructor prevents using the new operator with AppState
- ④ The only method to get an instance of AppState
- ⑤ Instantiate the object AppState if it doesn't exist yet
- ⑥ This variable gets a reference to the AppState instance
- ⑦ Modify the counter (we use two reference variables)
- ⑧ Print the value of the counter (we use two reference variables)

The class `AppState` has a `private` constructor, which means that no other script can instantiate it using the statement `new`. It's perfectly fine to invoke such a constructor from within the `AppState` class, and we do this in the static method `getInstance()`.

The method `getInstance()` is `static`, and this is the only way we can invoke a method in the absence of the class instance.

Both `console.log()` invocations will print 4 as there is only one instance of `AppState`.

To see this code sample in Playground, visit [bit.ly/2ELs1GO](https://bit.ly/2ELs1GO).

**TIP**

Often, we have several methods that perform similar actions. For example, we need to write a dozen of functions validating user's input in different fields on the UI. Instead of having separate functions, you can group them in a class with a dozen of static methods.

### 3.1.4 The method `super()` and the keyword `super`

Let's continue looking into class inheritance. In figure 3.3 in line 15, we simply invoked the method `sayHello()` that was declared in the superclass. What if both the super and subclass have methods with the same name? What if both have constructors? Can we control which method is executed?

If both the superclass and the subclass have constructors, the one from the subclass must invoke the constructor of the superclass using the method `super()` as seen in listing 3.3.

**NOTE**

We already discussed using `super()` and `super` in the section A9.2 in appendix A. In this section we'll provide a similar example but in TypeScript.

#### Listing 3.3 Invoking the constructor of the super class

```
class Person {
    constructor(public firstName: string,
                public lastName: string,
                private age: number) {} ①
}

class Employee extends Person { ②

    constructor (firstName: string, lastName: string,
                age: number, public department: string) { ③
        super(firstName, lastName, age); ④
    }
}

const empl = new Employee('Joe', 'Smith', 29, 'Accounting'); ⑤
```

- ① The constructor of the superclass Person
- ② The subclass Employee
- ③ The constructor of the subclass Employee
- ④ Invoking the constructor of the superclass
- ⑤ Instantiating the subclass

Since both classes define their constructors, we must ensure that each of them is invoked with the required parameters. If the constructor of the class `Employee` is automatically invoked when

we use the operator `new`, we have to manually invoke the constructor of the superclass `Person`. The class `Employee` has a constructor with four arguments, but only one of them - `department` - is needed for constructing the object of type `Employee`. The other three parameters are needed for constructing the object `Person` and we pass them over to `Person` by invoking the method `super()` with three arguments. You can play with this code in the TypeScript playground at [goo.gl/E6qkUm](http://goo.gl/E6qkUm).

Now let's consider a situation when both the superclass and subclass have methods with the same names. If a method in a subclass wants to invoke a method with the same name defined in the superclass, it needs to use keyword `super` instead of `this` when referencing the superclass method.

Let's say a class `Person` has a method `sellStock()` that connects to a stock exchange and sells the specified amount of shares of the given stock. In the class `Employee`, we'd like to reuse this functionality, but every time an employee sells stocks, she must report it to a compliance department of the firm.

We can declare a method `sellStock()` on the class `Employee`, and this method will call `sellStock()` on `Person` and then its own method `reportToCompliance()` as shown in listing 3.4.

### **Listing 3.4 Using the keyword super**

```
class Person {
    constructor(public firstName: string,
                public lastName: string,
                private age: number) { }

    sellStock(symbol: string, numberofShares: number) { ❶
        console.log(`Selling ${numberofShares} of ${symbol}`);
    }
}

class Employee extends Person {
    constructor (firstName: string, lastName: string,
                age: number, public department: string) {
        super(firstName, lastName, age); ❷
    }

    sellStock(symbol: string, shares: number) { ❸
        super.sellStock(symbol, shares); ❹

        this.reportToCompliance(symbol, shares);
    }

    private reportToCompliance(symbol: string, shares: number) { ❺
        console.log(`${this.lastName} from ${this.department} sold ${shares} shares of ${symbol}`);
    }
}

const empl = new Employee('Joe', 'Smith', 29, 'Accounting');
empl.sellStock('IBM', 100); ❻
```

- ① The sellStock() method in the ancestor
- ② Invoking the constructor of the ancestor
- ③ The sellStock() method in the descendant
- ④ Invoking the sellStock() on the ancestor
- ⑤ A private method reportToCompliance()
- ⑥ Invoking the sellStock() on the Employee object

Note that we declared the method `reportToCompliance()` as `private` as we want it to be called only by the internal method of the class `Employee` and never from an external script. You can run this program at [goo.gl/68HtTC](http://goo.gl/68HtTC) and the browser will print the following on its console:

```
Selling 100 of IBM
Smith from Accounting sold 100 shares of IBM
```

With the help of the `super` keyword, we reused the functionality from the method declared in the superclass and added new functionality as well.

### 3.1.5 Abstract classes

If you add the `abstract` keyword to the class declaration, it can't be instantiated. An abstract class may include methods that are implemented as well as the abstract ones that are *only declared*.

And why would you even want to create a class that can't be instantiated? The reason is that you may want to delegate implementation of some methods to its subclasses, and want to make sure these methods will have specific signatures when implemented. Let's consider a coding assignment and see how abstract classes can be used.

## SIDE BAR An assignment with abstract classes

A company has employees and contractors. Design the classes to represent workers of this company. Any worker's object should support the following methods:

- constructor(name: string)
- changeAddress(newAddress: string)
- giveDayOff()
- promote(percent: number)
- increasePay(percent: number)

In our scenario, to promote means giving one day off and raising the salary by the specified percent. The method increasePay() should raise the yearly salary for employees but increase the hourly rate for contractors.

How do you implement methods is irrelevant; a method can just have one `console.log()` statement.

Let's work on this assignment. We'll need to create the classes `Employee` and `Contractor`, which should have some common functionality. For example, changing address and giving a day off should work the same way for contractors and employees, but increasing pay requires different implementation for these categories of workers.

Here's the plan: we'll create the abstract class `Person` with two descendants: `Employee` and `Contractor`. The class `Person` will implement methods `changeAddress()`, `giveDayOff()`, and `promote()`. This class will also include a declaration of the abstract method `increasePay()`, which will be implemented (differently!) in the subclasses of `Person` shown in listing 3.5.

### **Listing 3.5 The abstract class Person**

```
abstract class Person {    ①

    constructor(public name: string) { };

    changeAddress(newAddress: string) {    ②
        console.log(`Changing address to ${newAddress}`);
    }

    giveDayOff() {    ②
        console.log(`Giving a day off to ${this.name}`);
    }

    promote(percent: number) {    ②
        this.giveDayOff();
        this.increasePay(percent);    ③
    }

    abstract increasePay(percent: number);    ④
}
```

- ① Declaring an abstract class
- ② Declaring and implementing a method
- ③ "Invoking" the abstract method
- ④ Declaring an abstract method

**TIP**

If you don't want to allow invoking the method `giveDayOff()` from external scripts, add `private` to its declaration. If you want to allow invoking `giveDayOff()` only from the class `Person` and its descendants, make this method `protected`.

Note that you are allowed to write a statement that "invokes" the abstract method. Since the class is abstract, it can't be instantiated, and there is no way that the abstract (unimplemented method) will be actually invoked. If you want to create a descendant of the abstract class that can be instantiated, you must implement all abstract methods of the ancestor. The code in listing 3.6 shows how we implemented the classes `Employee` and `Contractor`.

### **Listing 3.6 Descendants of the class Person**

```
class Employee extends Person {
    increasePay(percent: number) { ①
        console.log(`Increasing the salary of ${this.name} by ${percent}%`);
    }
}

class Contractor extends Person {
    increasePay(percent: number) { ②
        console.log(`Increasing the hourly rate of ${this.name} by ${percent}%`);
    }
}
```

- ① Implementing the method `increasePay()` for employees
- ② Implementing the method `increasePay()` for contractors

In the section 2.2.4 in chapter 2, we used the term *assignable* while discussing listing 2.20. When we have a `class A extends class B`, this means that the `class B` is more general, and `class A` is more specific (e.g. adds more properties).

A more specific type is assignable to a more general one. That's why you can declare a variable of type `Person` and assign to it an object `Employee` or `Contractor`, and you'll see this in listing 3.7.

In Listing 3.7, We'll create an array of workers with one employee and one contractor, and then iterate through this array invoking the method `promote()` on each object.

## Listing 3.7 Running the promotion campaign

```
const workers: Person[] = [];      ①

workers[0] = new Employee('John');
workers[1] = new Contractor('Mary');

workers.forEach(worker => worker.promote(5));    ②
```

- ① Declaring an array of the superclass type
- ② Invoking `promote()` on each object

The `workers` array is of type `Person`, which allows us to store there the instances of descendant objects as well.

**TIP**

Since the descendants of `Person` don't declare their own constructors, the constructor of the ancestor will be invoked automatically when we instantiate `Employee` and `Contractor`. If any of the descendants declared its own constructor, we'd have to use `super()` to ensure that the constructor of the `Person` is invoked.

You can run this code sample in the TypeScript playground at [goo.gl/nRuzVL](http://goo.gl/nRuzVL), and the browser console will show the following output:

```
Giving a day off to John
Increasing the salary of John by 5%
Giving a day off to Mary
Increasing the hourly rate of Mary by 5%
```

The code in listing 3.7 gives an impression that we iterate through the objects of type `Person` invoking `Person.promote()`. But realistically, some of the objects can be of type `Employee` while others are instances of `Contractor`. The actual type of the object is evaluated only during runtime, which explains why the correct implementation of the `increasePay()` is invoked on each object. This is an example of *polymorphism* - a feature that each object-oriented language supports.

**SIDE BAR    Protected constructors**

In section 3.1.3, we declared a `private` constructor to create a singleton. There's some use for `protected` constructors as well. Say you need to declare a class that can't be instantiated, but its subclasses can. Then, you could declare a `protected` constructor in the superclass and invoke it using `super()` from the subclass constructor.

This mimics one of the features of abstract classes. But a class with `protected` constructor wouldn't let you declare abstract methods unless the class itself is declared as abstract.

In section 10.6.1 in chapter 10, you'll see another example of using an abstract class for handling WebSocket messages in a blockchain app.

### **3.1.6 Method overloading**

Object-oriented programming languages like Java or C# support method overloading, which means that a class can *declare* more than one method with the same name but with different arguments. For example, you can write two versions of the method `calculateTax()` - one with two arguments the person's income and number of dependents, and the other with one argument of type `Customer` that has all the required data about the person.

In strongly-typed languages, the ability to overload methods with specifying types of the arguments and a return value is important, because you can't just invoke a class method providing an argument of an arbitrary type. But TypeScript is a sugar coating for JavaScript, which allows invoking a function passing more or less arguments than the function signature declares. JavaScript won't complain, and it doesn't need to support function overloading. Of course, you may get a runtime error if inside the method you didn't properly handle the provided object, but this would happen later at runtime.

Anyway, let's see if the code shown in listing 3.8 works:

### Listing 3.8 An erroneous attempt of method overloading

```
class ProductService {

    getProducts() { ①
        console.log(`Getting all products`);
    }

    getProducts(id: number) { // error ②
        console.log(`Getting the product info for ${id}`);
    }
}

const prodService = new ProductService();

prodService.getProducts(123);

prodService.getProducts();
```

- ①** The method `getProducts()` without arguments
- ②** The method `getProducts()` with one argument

The TypeScript compiler will give you an error *Duplicate function implementation* for the second `getProduct()` declaration as shown in figure 3.6.

The screenshot shows a TypeScript playground interface. On the left, the TypeScript code has several syntax errors underlined in red, such as 'getProducts()' and 'getProducts(id: number)'. On the right, the generated JavaScript code runs without errors. The JavaScript code defines a `ProductService` class with two methods: one for getting all products and another for getting a specific product by ID. Both methods log their respective messages to the console.

```
1 var ProductService = /** @class */ (function () {
2     function ProductService() {
3     }
4     ProductService.prototype.getProducts = function () {
5         console.log("Getting all products");
6     };
7     ProductService.prototype.getProducts = function (id) {
8         console.log("Getting the product info for " + id);
9     };
10    return ProductService;
11 })();
12 var prodService = new ProductService();
13 prodService.getProducts(123);
14 prodService.getProducts();
```

Figure 3.6 Erroneous TypeScript but valid JavaScript

The syntax in the TypeScript code (on the left) is wrong, but the JavaScript syntax on the right is perfectly fine. The first version of the method `getProducts` (line 4) was replaced with the second one (line 7) so during runtime, the JavaScript version of this script has only one version of `getProducts(id)`.

Let's ignore the compiler's errors and try to run the generated JavaScript in the TypeScript playground. The browser console will print the messages from the only method `getProducts(id)` even though we wanted to invoke different versions of this method:

```
Getting the product info for 123
Getting the product info for undefined
```

In the compiled JavaScript, the method (or a function) can have only one body, which can

account for all allowed method parameters. Still, TypeScript offers the syntax to specify method overloading, which comes down to declaring all allowed method signatures without implementing these methods followed by one implemented method as shown in listing 3.9.

### **Listing 3.9 Correct syntax for method overloading**

```
class ProductService {

    getProducts();          ①
    getProducts(id: number); ①
    getProducts(id?: number) { ②
        if (typeof id === 'number') {
            console.log(`Getting the product info for ${id}`);
        } else {
            console.log(`Getting all products`);
        }
    }
}

const prodService = new ProductService();

prodService.getProducts(123);
prodService.getProducts();
```

- ① Declaring the allowed method signature
- ② Implementing the method

Note the question mark after the argument `id` in the implemented method. This question mark declares this argument as optional. Without making this argument optional, the compiler would give you the error *Overload signature is not compatible with function implementation*. In our code sample, this means that if you declared a no-argument method signature `getProducts()`, the method implementation should allow invoking this function without arguments.

**TIP**

Omitting the first two declarations in listing 3.9 wouldn't change this program behavior. These lines just help IDEs provide better auto-complete options for the function `getProducts()`.

Try this code sample in the TypeScript playground at [bit.ly/2PRzieZ](https://bit.ly/2PRzieZ). Note that the generated JavaScript has just one function `getProducts()` as shown in figure 3.7.

```

Select... TypeScript Share Options Run JavaScript
1 class ProductService {
2     getProducts();
3     getProducts(id: number);
4     getProducts(id?: number) {
5         if (!!id) {
6             console.log(`Getting the product info for ${id}`);
7         } else {
8             console.log('Getting all products');
9         }
10    }
11 }
12 }
13
14 const prodService = new ProductService();
15
16 prodService.getProducts(123);
17
18 prodService.getProducts();

```

```

1 var ProductService = /** @class */ (function () {
2     function ProductService() {
3     }
4     ProductService.prototype.getProducts = function (id) {
5         if (!!id) {
6             console.log("Getting the product info for " + id);
7         } else {
8             console.log("Getting all products");
9         }
10    };
11    return ProductService;
12 })();
13
14 var prodService = new ProductService();
15 prodService.getProducts(123);
16 prodService.getProducts();
17

```

**Figure 3.7 Proper syntax for overloading the method**

Similarly, you can overload a method signature to indicate that it can not only have different arguments but return values of different types. Listing 3.10 shows the script with the overloaded method `getProducts()`, which can be invoked in two ways:

1. Providing the product description and returning an array of type `Product`
2. Providing the product id and returning a single object of type `Product`

### **Listing 3.10 Different arguments and return types**

```

interface Product { ①
  id: number;
  description: string;
}

class ProductService {

  getProducts(description: string): Product[]; ②
  getProducts(id: number): Product; ③
  getProducts(product: number | string): Product[] | Product{ ④
    if (typeof product === "number") { ⑤
      console.log(`Getting the product info for id ${product}`);
      return { id: product, description: 'great product' };
    } else if (typeof product === "string") { ⑥
      console.log(`Getting product with description ${product}`);
      return [{ id: 123, description: 'blue jeans' },
              { id: 789, description: 'blue jeans' }];
    } else {
      return null;
    }
  }
}

const prodService = new ProductService();

console.log(prodService.getProducts(123));

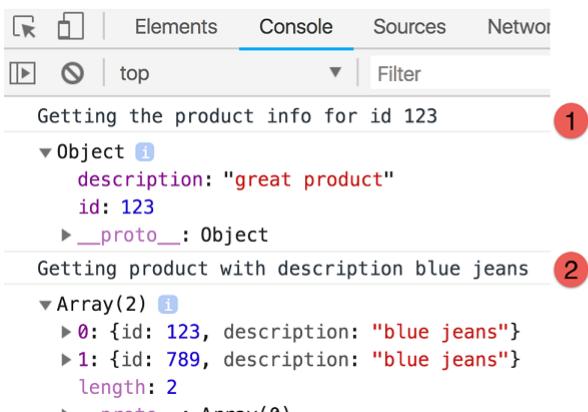
console.log(prodService.getProducts('blue jeans'));

```

- ① Define the type `Product`
- ② The first overloaded signature of `getProducts()`
- ③ The second overloaded signature of `getProducts()`
- ④ Implementation of `getProducts()`

- ⑤ See if the method was invoked with product id
- ⑥ See if the method was invoked with product description

You can see and run the code sample from listing 3.10 in the TypeScript playground at [goo.gl/rq3GJ1](https://goo.gl/rq3GJ1). The output on the browser console will look as shown in figure 3.9.



**Figure 3.8 Overloading with different returns**

1. `getProducts(123)` returns a single object
2. `getProducts('blue jeans')` returns an array

Now, let's experiment with the code in listing 3.10. If you comment out two lines that declare signatures of `getProducts()`, the program still works and you can invoke this method providing either the number or string as an argument and this method will return either one `Product` or an array of them.

The question is, "Why even declare overloaded signatures if you can simply implement a single method using unions in argument types and return values?" Method overloading helps TypeScript compiler to properly map provided argument types to the types of the return values. When the overloaded method signatures are declared, TypeScript static analyzer will properly suggest possible ways of invoking the overloaded method, and figure 3.9 shows you the first prompt (note 1/2) for invoking the method `getProducts()`.

```

24
25 const prodService = new ProductService();
26
27 const product: Product = prodService.getProducts()
28

```

getProducts(description: string):  
1/2 Product[]  
▼

**Figure 3.9 Prompting the first method signature**

Figure 3.10 shows you the second prompt (note 2/2) for invoking the method `getProducts()` with a different parameter type and return.

```

25 const prodService = new ProductService();
26
27 const product: Product = prodService.getProducts()
28

```

**Figure 3.10 Prompting the second method signature**

If we would comment out the declarations of the signatures of `getProducts()`, the prompt wouldn't be that easy to reason about and understand which argument type results in returning a value of which type as shown in figure 3.11.

```

24
25 const prodService = new ProductService();
26
27 const product: Product = prodService.getProducts()
28

```

**Figure 3.11 The prompt without overloading**

You may argue that the benefits offered by TypeScript method overloading are not too convincing and we could agree that it may be easier to just declare two methods with different names, e.g. `getProduct()` and `getProducts()` without unionizing their argument and return types. This is true except one use case: overloading constructors. In TypeScript classes, there's only one name you can give to a constructor, which is `constructor`. So if you want to create a class with several constructors having different signatures, you may want to use the syntax for overloading as shown in listing 3.11.

### Listing 3.11 Overloading constructors

```

class Product {
  id: number;
  description: string;

  constructor();           ①
  constructor(id: number); ②
  constructor(id: number, description: string); ③
  constructor(id?: number, description?: string) {
    // Constructor implementation goes here ④
  }
}

```

- ① A no-argument constructor declaration
- ② A one-argument constructor declaration
- ③ A two-argument constructor declaration
- ④ An implementation of constructor handling all possible arguments

**NOTE** Since we wanted to allow a no-argument constructor, we made all arguments optional in the constructor implementation .

But again, overloading constructors is not the only way to initialize the object properties. For example, you may declare a single interface to represent all possible parameters for a constructor. Listing 3.12 declares an interface with all optional properties and the class with one

constructor that takes one optional argument.

### **Listing 3.12 A single constructor with an optional argument**

```
interface ProductProperties { ①
  id?: number;
  description?: string;
}

class Product {
  id: number;
  description: string;

  constructor(properties?: ProductProperties) { ②
    // Constructor implementation goes here
  }
}
```

- ① A ProductProperties interface with two optional properties
- ② A class constructor with an optional argument of type ProductProperties

To summarize, use common sense when overloading a method or a constructor in TypeScript. While overloading provides multiple ways of invoking a method, its logic may quickly become difficult to reason about. In our daily TypeScript work, we rarely use overloading.

## **3.2 Working with interfaces**

In chapter 2, we used TypeScript interfaces only for declaring custom types, and came up with a general rule: If you need a custom type that includes a constructor use a class, otherwise use an interface. In this section, we'll show you how to use TypeScript interfaces to ensure that a class implements specific API.

### **3.2.1 Enforcing the contract**

An interface can declare not only properties, but it can also include method declarations (no implementations though). A class declaration can include the keyword `implements` followed by the name of the interface. In other words, while an interface just contains method signatures, a class can contain their implementations.

Say you own a car Toyota Camry. It has thousands of parts that perform various actions, but as a driver, you just need to know how to use a handful of controls to drive a car, e.g. how to start and stop the engine, how to accelerate and brake, how to turn on the radio and so on. All these actions can be considered *a public interface* offered to you by the designers of Toyota Camry.

Now imagine that you had to rent a car, and they gave you Ford Taurus that you've never driven before. Will you know how to drive it? Yes, because it has a familiar *interface*: a key for starting the engine, acceleration and brake pedals et al. When you rent a car, you can even request a car that has a specific interface, e.g. an automatic transmission.

Let's model some of the car interfaces using the TypeScript syntax. The code in listing 3.13 shows an interface `MotorVehicle` that declares five methods.

### **Listing 3.13 The interface MotorVehicle**

```
interface MotorVehicle {
    startEngine(): boolean;      ①
    stopEngine(): boolean;       ①
    brake(): boolean;           ①
    accelerate(speed: number);  ①
    honk(howLong: number): void; ①
}
```

- ① Declaring a method signature that should be implemented by a class

Note that none of the methods of `MotorVehicle` is implemented. Now we can declare a class `Car` that will implement all methods declared in the interface `MotorVehicle`. By using the keyword `implements` we declare that a class implements a certain interface(s), for example:

```
class Car implements MotorVehicle {
}
```

This simple class declaration won't compile giving an error *Class Car incorrectly implements interface MotorVehicle*. When you declare that a class implements some interface, you must implement each and every method declared in the interface. In other words, the above code snippet states "I swear that the class `Car` will implement the API declared in the interface `MotorVehicle`". Listing 3.14 shows a simplified implementation of this interface by the class `Car`.

### **Listing 3.14 A class that implements MotorVehicle**

```
class Car implements MotorVehicle {
    startEngine(): boolean { ①
        return true;
    }
    stopEngine(): boolean{ ①
        return true;
    }
    brake(): boolean { ①
        return true;
    }
    accelerate(speed: number) { ①
        console.log(`Driving faster`);
    }

    honk(howLong: number): void { ①
        console.log(`Beep beep yeah!`);
    }
}

const car = new Car(); ②
car.startEngine(); ③
```

- ① Implementing the methods from the interface

- ② Instantiating the class Car
- ③ Using the Car's API to start the engine

Note that we didn't explicitly declare the type of the constant `car` - this is an example of type inference. We could explicitly declare the type of the `car` type as follows (although it's not required):

```
const car: Car = new Car();
```

We could also declare the constant `car` of type `MotorVehicle` because our class `Car` implements this custom type:

```
const car: MotorVehicle = new Car();
```

What's the difference between these two declarations of the constant `car`? Let's say a class `Car` implements eight methods: five of them come from the interface `MotorVehicle` and the others are some arbitrary methods. If the constant `car` is of type `Car`, you can invoke all eight methods on the instance of the object represented by `car`. But if the type of `car` is `MotorVehicle`, only the five methods declared in this interface can be invoked using the constant `car`.

We can say that *an interface enforces a specific contract*. In our code sample, this means that we force the class `Car` to implement each of the five methods declared in the interface `MotorVehicle` or else the code won't compile.

Now let's design an interface for the James Bond's car. Yes, for the agent 007. This special car should be able to fly and swim as well. Not a problem. Let's declare a couple of interfaces first.

### **Listing 3.15 Flyable and Swimmable interfaces**

```
interface Flyable {
  fly(howHigh: number);
  land();
}

interface Swimmable {
  swim(howFar: number);
}
```

A class can implement more than one interface, so let's make sure that our class implements these two interfaces as well:

### **Listing 3.16 A car with three interfaces**

```
class Car implements MotorVehicle, Flyable, Swimmable {
  // Implement all the methods from three
  // interfaces here
}
```

Actually, making every car flyable and swimmable is not a good idea, so let's not modify the class `Car` shown earlier in listing 3.14, but use class inheritance and create the class `SecretServiceCar` that extends `Car` and adds more features:

### **Listing 3.17 A class that extends and implements**

```
class SecretServiceCar extends Car implements Flyable, Swimmable {
    // Implement all the methods from two
    // interfaces here
}
```

After implementing all the methods declared in `Flyable` and `Swimmable` our `SecretServiceCar` turns a regular motor vehicle into a flyable and swimmable object. The class `Car` continues representing a regular auto with the functionality defined in the interface `MotorVehicle`.

### **3.2.2 Extending interfaces**

As you saw in the previous section, combining classes and interfaces brings flexibility to designing your code. Let's consider yet another option - extending an interface.

In the previous section, when the requirement to develop the secret service car came in, we had the interface `MotorVehicle` and the class `Car` that implemented this interface. Listing 3.17 shows the class `SecretServiceCar` that's inherited from `Car` and implements two additional interfaces.

But when you design a special car for secret service, you may want to implement all the methods listed in the `MotorVehicle` interface differently as well, which means that you may want to declare the class `SecretServiceCar` as shown in listing 3.18.

### **Listing 3.18 A class that implements three interfaces**

```
class SecretServiceCar implements MotorVehicle, Flyable, Swimmable {
    // Implement all the methods from three interfaces here
}
```

On the other hand, our flyable object is a motor vehicle as well, so we can declare the `Flyable` interface as follows:

### **Listing 3.19 Extending an interface**

```
interface Flyable extends MotorVehicle{ ①
    fly(howHigh: number); ②
    land(); ②
}
```

- ① One interface extends another

② Declaring a method signature to be implemented in a class

Now if a class includes `implements Flyable` in its declaration, it must implement five methods declared in the `MotorVehicle` interface (see listing 3.13) as well as two methods from `Flyable` (listing 3.19) - seven methods total. Our class `SecretServiceCar` must implement these seven methods plus one from `Swimmable` as shown in listing 3.20.

### Listing 3.20 A class that implements Flyable and Swimmable

```
class SecretServiceCar implements Flyable, Swimmable {

    startEngine(): boolean { ①
        return true;
    };
    stopEngine(): boolean{ ①
        return true;
    };
    brake(): boolean { ①
        return true;
    };
    accelerate(speed: number) { ①
        console.log(`Driving faster`);
    }

    honk(howLong: number): void { ①
        console.log(`Beep beep yeah!`);
    }

    fly(howHigh: number) { ②
        console.log(`Flying ${howHigh} feet high`);
    }

    land() { ②
        console.log(`Landing. Fasten your belts.`);
    }

    swim(howFar: number) { ③
        console.log(`Swimming ${howFar} feet`);
    }
}
```

- ① Implementing the method from `MotorVehicle`
- ② Implementing the method from `Flyable`
- ③ Implementing the method from `Swimmable`

**TIP**

Even if `Swimmable` would also extend `MotorVehicle`, the TypeScript compiler wouldn't complain.

Declaring interfaces that include method signatures improve code readability as any interface clearly describes a well-defined set of features, which can be implemented by one or more concrete classes. By looking at the interface you don't know how some class will implement it, and developers who practice object-oriented approach in programming have a mantra "Program to interfaces, not implementations". In the next section we'll see what this means.

### 3.2.3 Programming to interfaces

To understand the meaning and benefits of programming to interfaces, let's consider a use case where this technique was not used. Imagine that you had to write code that would allow you to get the information about all or one product from some data source.

You know how to write classes and will start implementing them right away. You could define a custom type `Product` and a class `ProductService` with two methods as follows:

#### Listing 3.21 Programming to implementations

```
class Product { ①
  id: number;
  description: string;
}

class ProductService { ②

  getProducts(): Product[] { ③
    // the code for getting products
    // from a real data source should go here

    return [];
  }

  getProductById(id: number): Product {
    // the code for getting products
    // from a real data source should go here

    return { id: 123, description: 'Good product' };
  }
}
```

- ① A custom type `Product`
- ② A concrete implementation of `ProductService`
- ③ An implemented method

Then, in multiple places in your app, you instantiated the `ProductService` and used its methods, for example:

```
const productService = new ProductService();
const products = productService.getProducts();
```

That was easy, wasn't it? You proudly commit this code to the source code repository, but your manager says that the backend guys are delaying the implementation of the server that was supposed to provide data for our `ProductService`. He asks you to create another class `MockProductService` with *the same API* that would return hard-coded product data. No problem, and you write another implementation of the product service:

## Listing 3.22 Another implementation of the product service

```
class MockProductService {    ①
    getProducts(): Product[] {    ②
        // the code for getting hard-coded
        // products goes here

        return [];
    }

    getProductById(id: number): Product {
        ②
        return { id: 456, description: 'Not a real product' };
    }
}
```

- ① A concrete implementation of `MockProductService`
- ② An implemented method

**TIP**

You may need to create the `MockProductService` not only because the backend guys are running late, but also if you need to write unit tests, in which we don't use real services.

So you created two concrete implementations of the product service. Hopefully, you didn't make mistakes while declaring methods in the class `MockProductService` and they are exactly the same as in `ProductService`. Otherwise you may break the code that uses `MockProductService`.

We don't like the word *hopefully* in the previous sentence. We already know that an interface allows to enforce a contract upon a class, i.e. upon the `MockProductService`. But we didn't declare any interface here! It sounds weird, but in TypeScript, you can declare a class that implements another class, so the better (not the best) approach would be to start writing `MockProductService` as follows:

```
class MockProductService implements ProductService {
    // implementation goes here
}
```

TypeScript is smart enough to understand that if you use the class name after the word `implements`, you want to use it as interface and enforce the implementation of all public methods of `ProductService`. This way there is no chance that you forget to implement or make a mistake in the signature of `getProducts()` or `getProductById()`. Your code won't compile until you properly include the implementation of these methods to the class `MockProductService`.

But the best way would be if to program to interfaces from the very beginning. When you got the requirement to write `ProductService` with two methods, you should have started with declaring

an interface with these methods without worrying about their implementation.

Let's call this interface `IProductService` and declare two method signatures there. Then declare the class `ProductService` that implements this interface as shown in listing 3.23.

### Listing 3.23 Program to interface

```
interface Product { ①
  id: number;
  description: string;
}

interface IProductService { ②
  getProducts(): Product[];
  getProductById(id: number): Product
}

class ProductService implements IProductService { ③

  getProducts(): Product[]{
    // the code for getting products
    // from a real data source goes here

    return [];
  }

  getProductById(id: number): Product {
    // the code for getting a product by id goes here
    return { id: 123, description: 'Good product' };
  }
}
```

- ① Declaring a custom type using an interface
- ② Declaring API as an interface
- ③ Implementing the interface

Declaring an API as an interface shows that you spent time thinking about the required functionality and only after, you took care of concrete implementation. Now, if a new class `MockProductService` has to be implemented, you'll start it like this:

```
class MockProductService implements IProductService {
  // Another concrete implementation of the
  // interface methods goes here
}
```

Have you noticed that the custom type `Product` is implemented differently in listings 3.23 and 3.21? Use the keyword `interface` instead of `class` if you don't need to instantiate this custom type (e.g. `Product`), and the JavaScript footprint will be smaller. Try the code from these two listings in the TypeScript playground and compare the generated JavaScript. The version where `Product` was an interface is shorter.

**TIP**

We named the interface `IProductService` starting with the capital `I`, while the class name was `ProductService`. Some people prefer using the suffix `Impl` with concrete implementations, e.g. `ProductServiceImpl` and the interface would be simply named `ProductService`.

Another good example of programming to interfaces is factory functions, which implement some business logic and then return the proper instance of the object. If we had to write a factory function that would return either `ProductService` or `MockProductService`, we'd use the interface as its return type:

**Listing 3.24 A factory function**

```
function get ProductService(isProduction: boolean): IProductService {    ①
  if (isProduction) {
    return new ProductService();
  } else {
    return new MockProductService();
  }
}

const productService: IProductService;    ②

...
const isProd = true;    ③
productService = get ProductService(isProd);    ④
const products[] = productService.getProducts();    ⑤
```

- ① A factory function that uses the interface as a return type
- ② A constant of the interface type
- ③ In real world get its value from the environment variable
- ④ Get the proper instance of the product service
- ⑤ Invoke the method on the product service

In this example, we used the constant `isProd` with the hard-coded value `true`. In real-world apps, its value would be obtained from some property file or environment variable. By changing this property from `true` to `false` we change the behavior of our app during runtime. We achieved this by using the properly written factory function that returns an object of type `IProductService`, which can represent several concrete types. Program to interfaces!

**NOTE**

In section 11.1.4 in chapter 11, while explaining specifics of using dependency injection the Angular framework, we'll come back to the idea of programming to abstractions. There, you'll see why TypeScript interfaces can't be used, but you could use abstract classes instead.

### 3.3 Summary

In this chapter, we continued learning classes and interfaces. We focused on the object-oriented style of programming, and the main takeaways are:

- You can create a class using another one as a base. We call it class inheritance.
- A subclass can use `public` or `protected` properties of a superclass.
- If a class property is declared as `private`, it can be used only within this class.
- You can create a class that can only instantiated once by using a `private` constructor.
- If a method with the same signature exists in the superclass and a subclass, we call it method overriding. Class constructors can be overridden as well. The keyword `super` and the method `super()` allow a subclass invoke the class members of a superclass.
- You can declare several signatures for a method, and this is called method overloading.
- Interfaces can declare method signatures, but can't contain their implementations.
- You can inherit one interface from another.
- While implementing a class, see if there are certain methods that can be declared in a separate interface. Then your class has to implement that interface. This approach provide a clean way to separate declaring the functionality from implementing it.

# *Enums and Generics*



## **This chapter covers:**

- The benefits of using enums
- The syntax for numeric and string enums
- What generic types are for
- How to write classes, interfaces, and functions that support generics

## **4.1 Enums**

Enumerations (a.k.a. enums) allow you to create limited sets of named constants that have something in common. For example, a week has seven days, and you can assign numbers from 1 to 7 to represent them. But what's the first day of the week?

According to ISO 8601, the standard on Data elements and interchange formats, Monday is the first day of the week, which doesn't stop such countries as USA, Canada, and Australia consider Sunday as the first day of the week. Hence using just numbers from 1 to 7 for representing days may not be a good idea. Also, what if someone will assign the number 8 to the variable that store the day? We don't want this to happen and using day names instead of the numbers makes our code more readable.

On the other hand, using numbers for storing days is more efficient than their names. So we want to have readability, the ability to restrict values to a limited set, and efficiency in storing data. This is where enums can help.

TypeScript has the `enum` keyword that can define a limited set of constants, and we can declare the new type for weekdays as follows:

## Listing 4.1 Defining weekdays using enum

```
enum Weekdays {
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
    Sunday = 7
}
```

Listing 4.1 defines a new type `Weekdays` that has a limited number of values. We initialized each enum member with a numeric value, and a day of the week can be referred using the dot notation:

```
let dayOff = Weekdays.Tuesday;
```

The value of the variable `dayOff` is 2, but if you'd be typing the above line in your IDE or in TypeScript Playground, you'd be prompted with the possible values as shown in figure 4.1.



**Figure 4.1 Autocomplete with enums**

Using the members of the enum `Weekdays` stops you from making a mistake and assigning a wrong value (e.g. 8) to the variable `dayOff`. Well, strictly speaking, nothing stops you from ignoring this `enum` and write `dayOff = 8` but this would be a misdemeanor.

In listing 4.1, we could initialize only Monday with 1, and the rest of the days values will be assigned using auto-increment, e.g. Tuesday will be initialized with 2, Wednesday with 3 and so on.

## Listing 4.2 The enum with auto-increment values

```
enum Weekdays {
    Monday = 1,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

By default, enums are zero-based, and if we wouldn't initialize the `Monday` member with one, its value would be zero.

**SIDE BAR** **Reversing numeric enums**

If you know the value of the numeric enum, you can find the name of the corresponding enum member. For example, you may have a function that returns the weekday number and you'd like to print its name. By using this value as index, you can retrieve the name of the day.

**Listing 4.3 Finding the name of the enum member**

```
enum Weekdays {  
    Monday = 1,  
    Tuesday,  
    Wednesday,  
    Thursday,  
    Friday,  
    Saturday,  
    Sunday  
}  
  
console.log(Weekdays[3]);
```

- ① Declaring a numeric enum
- ② Getting the name on the member that's equal to 3

In the last line in listing 4.3, we retrieve the name of the day 3, and it'll print Wednesday on the console.

In some cases, you don't even care which numeric values are assigned to the `enum` members, and the following function `convertTemperature()` illustrates this. It converts the temperature from Fahrenheit to Celsius or vice versa. In this version of `convertTemperature()` we won't use enums, but then will re-write it with them.

**Listing 4.4 Converting temperature without enums**

```
function convertTemperature(temp: number, fromTo: string): number {  
    ①  
  
    return ('FtoC' === fromTo) ?  
        (temp - 32) * 5.0/9.0: ②  
        temp * 9.0 / 5.0 + 32; ③  
    }  
  
    console.log(`70F is ${convertTemperature(70, 'FtoC')}`); ④  
    console.log(`21C is ${convertTemperature(21, 'CtoF')}`); ⑤  
    console.log(`35C is ${convertTemperature(35, 'ABCD')}`); ⑥
```

- ① This function takes two parameters: temperature and conversion direction
- ② Convert from Fahrenheit to Celsius
- ③ Convert from Celsius to Fahrenheit
- ④ Convert 70 degrees Fahrenheit

- ⑤ Convert 21 degrees Celsius
- ⑥ Invoking function with the meaningless fromTo

The function in listing 4.4 converts the value from Celsius to Fahrenheit if you pass any value as a `fromTo` parameter except `FtoC`. In the last line, we purposely provided the erroneous value `ABCD` as a `fromTo` parameter, and this function still converts the temperature from Celsius to Fahrenheit. The attempts to invoke a function with the erroneous values should be caught by the compiler and this is what TypeScript enums are for. You can see it in action at [codepen.io/yfain/pen/XBGMzB/?editors=0011](https://codepen.io/yfain/pen/XBGMzB/?editors=0011)

In listing 4.5, we declare the `enum Direction` that restricts the allowed constants to either `FtoC` or `CtoF` and nothing else. We also changed the type of the `fromTo` parameter from `string` to `Direction`.

### Listing 4.5 Converting temperature with enums

```
enum Direction { ①
  FtoC,
  CtoF
}

function convertTemperature(temp: number, fromTo: Direction): number { ②
  return (Direction.FtoC === fromTo) ?
    (temp - 32) * 5.0/9.0:
    temp * 9.0 / 5.0 + 32;
}

console.log(`70F is ${convertTemperature(70, Direction.FtoC)}C`); ③
console.log(`21C is ${convertTemperature(21, Direction.CtoF)}F`); ③
```

- ① Declaring the enum `Direction`
- ② The type of the second parameter is `Direction`
- ③ Invoking the function using the enum members

Since the type of the second parameter of the function is `Direction`, it tells us to invoke this function providing one of this enum's member, e.g. `Direction.CtoF`. We're not interested in what is the numeric value of this member. The purpose of this enum is just to provide a limited set of constants: `CtoF` and `FtoC`. The IDE will prompt you with two possible values for the second parameter, and you won't make a mistake providing a random value.

**TIP** Using the type `Direction` for the second argument doesn't stop you from another misdemeanor and invoking this function like `convertTemperature(50.0, 99)`.

Enum members are initialized with values (either explicitly or implicitly). All examples included

in this section had enum members initialized with numbers, but TypeScript allows you to create enums with string values, and we'll see such examples next.

### 4.1.1 String enums

In some cases, you may want to declare a limited set of string constants, and you can use string enums for this, i.e. enums that have their members initialized with string values. Say you're programming a computer game where the player can move in the following directions:

#### **Listing 4.6 Declaring a string enum**

```
enum Direction {
    Up = "UP",      ①
    Down = "DOWN",  ①
    Left = "LEFT",  ①
    Right = "RIGHT", ①
}
```

- ① Initializing the enum member with a string value

When you declare a string enum, you must initialize each member. You may ask, "Why not just use a numeric enum here so TypeScript would automatically initialize its members with any numbers?" The reason is that in some cases you want to give meaningful values to the enum members. For example, you need to debug the program and instead of seeing that the last move was 0, you'll see that the last move was UP.

And the next question you may ask, "Why declare the enum `Direction` if I can just declare four string constants with the values UP, DOWN, LEFT, and RIGHT?" You can, but let's say we have a function with the following signature:

```
move(where: string)
```

A developer can make a mistake (or a typo) and invoke this function as `move( "North" )`. But North is not a valid direction, and it's safer to declare this function using the enum `Direction`:

```
move(where: Direction)
```

The screenshot shows a code editor with the following TypeScript code:

```

1 enum Direction {
2     Up = "UP",
3     Down = "DOWN",
4     Left = "LEFT",
5     Right = "RIGHT",
6 }
7
8 function move(where: Direction) {
9
10    if (where === Direction.Up) {
11        // Do something
12    }
13 }
14 move("North");
15
16
17 move(Direction.);
18 
```

A red box highlights the error at line 14: `move("North");`. A tooltip says: "Wrong argument type is caught by compiler". Another red box highlights the auto-complete suggestion at line 17: `move(Direction.)`. A tooltip says: "Auto-complete prevents mistakes". A dropdown menu shows the enum members: Down, Left, Right, Up. Below the menu, a note says: "(enum member) Direction.Down = "DOWN"".

**Figure 4.2 Catching erroneous function invocations**

We made a mistake and provided a string "North" in line 15, and the compile-time error would read "Argument of type ""North"" is not assignable to the parameter of type 'Direction'." In line 18, the IDE offers you a selection of valid enum members so there's no way you provide the wrong argument.

Now, let's imagine that you need to keep track of the app state changes. The user can initiate a limited number of actions in each of the views of your app. Say, you want to log the actions taken in the view Products. Initially, the app tries to load products, and this action can either succeed or fail. The user can also search for products. To represent the states of the view Products you may declare a string enum as follows:

#### Listing 4.7 Declaring the string enum for monitoring actions

```

enum ProductsActionTypes {
    Search = 'Products Search', ①
    Load = 'Products Load All', ②
    LoadFailure = 'Products Load All Failure', ③
    LoadSuccess = 'Products Load All Success' ④
}

// If the function that loads products fails... ⑤
console.log(ProductsActionTypes.LoadFailure);

```

- ① Initializing the member Search
- ② Initializing the member Load
- ③ Initializing the member LoadFailure
- ④ Initializing the member LoadSuccess
- ⑤ Prints "Products Load All Failure"

When the user clicks on the button to load products, you can log the value of the member

`ProductsActionTypes.Load`, which will log the text 'Products Load All'. If the products were not loaded successfully, log the value of `ProductsActionTypes.LoadFailure`, which will log the text 'Products Load All Failure'.

**NOTE**

Some state management frameworks (e.g. Redux) require the app to emit actions when the app state changes. If we'd declare a string enum like in listing 4.7, we'd be emitting actions `ProductsActionTypes.Load`, `ProductsActionTypes.LoadSuccess` et al.

String enums are easily mapped to string values coming from a server or database (order status, user role, etc.) with no additional coding providing you all the benefits of the strong typing. We'll illustrate this at the very end of this chapter in listing 4.18.

**NOTE**

String enums are not reversible, and you can't find the member's name if you know its value.

### 4.1.2 `const enums`

If you use the keyword `const` while declaring an `enum`, its values will be inlined and no JavaScript will be generated. Let's compare the generated JavaScript of `enum` vs `const enum`. The left side of figure 4.3 shows the `enum` declared without the `const`, and the right side shows the generated JavaScript. For illustration purposes, in the last line, we just print the next move.

```
1 enum Direction {
2     Up = "UP",
3     Down = "DOWN",
4     Left = "LEFT",
5     Right = "RIGHT",
6 }
7
8 const theNextMove = Direction.Down;
```

```
1 var Direction;
2 if(function (Direction) {
3     Direction["Up"] = "UP";
4     Direction["Down"] = "DOWN";
5     Direction["Left"] = "LEFT";
6     Direction["Right"] = "RIGHT";
7 })(Direction || (Direction = {}));
8 var theNextMove = Direction.Down;
```

Figure 4.3 The enum without the `const` keyword

Now let's just add the keyword `const` on the first line before the `enum`, and compare the generated JavaScript (on the right) with the one from figure 4.3.

```
1 const enum Direction {
2     Up = "UP",
3     Down = "DOWN",
4     Left = "LEFT",
5     Right = "RIGHT",
6 }
7
8 const theNextMove = Direction.Down;
```

```
1 var theNextMove = "DOWN" /* Down */;
```

Figure 4.4 The enum with the `const` keyword

As you see, the JavaScript code for the `enum` `Direction` was not generated - it was erased. But the values of the enum member that was actually used in the code (i.e. `Direction.Down`) was

inlined in JavaScript.

**TIP**

In listing 6.3, we reversed the third member of the enum: Weekdays[3]). This wouldn't be possible with `const enum` as they are not represented in the generated JavaScript code.

Using `const` with `enum` results in more concise JavaScript, but keep in mind that since there is no JavaScript code that would represent your `enum`, you may run into some limitations, e.g. you won't be able to reverse the numeric enum member name by its value.

Overall, with enums, the readability of your programs increases.

## 4.2 Generics

We know that TypeScript has built-in types and you can create the custom ones as well. But there's more to it. Strange as it may sound, types can be *parameterized*, i.e. you can provide a type (not the value) as a parameter.

It's easy to declare a function that takes parameters of specific concrete types e.g. a number and a string:

```
function calcTax(income: number, state: string){...}
```

But TypeScript generics allow you to write a function that can work with a variety of types. In other words, you can declare a function that works with a generic type(s), and the concrete type(s) can be specified later by the caller of this function.

In TypeScript, you can write generic functions, classes, or interfaces. A generic type can be represented by an arbitrary letter(s), e.g. T in `Array<T>`, and when you declare a specific array, you provide a concrete type in angle brackets, e.g. `number`:

```
let lotteryNumbers: Array<number>;
```

In this section, you'll learn how to use generic code written by someone else as well as how to create your own classes, interfaces, and functions that can work with generic types.

### 4.2.1 Understanding generics

A generic is a piece of code that can handle values of multiple types that are specified at the moment of using this piece of code (e.g. function invocation or class instantiation). Let's consider TypeScript arrays, which can be declared as follows:

1. Specify the type of the array element followed by [ ]:

```
const someValues: number[];
```

2. Use a generic Array followed by the type parameter in angle brackets:

```
const someValues: Array<number>;
```

If all elements of the array have the same type, both ways of declarations are equivalent but the first way is simpler to read. A bit later, we'll show you when declaring an array with generic type is better.

With the second syntax, the angle brackets represent a type parameter. You can instantiate this the `Array` like any other while restricting the type of allowed values, which is `number` in our example.

The next code snippet creates an array that will initially have 10 objects of type `Person`, and the inferred type of the variable `people` is `Person[]`.

```
class Person{ }

const people = new Array<Person>(10);
```

TypeScript arrays can hold objects of any type, but if you decide to use the generic type `Array`, you must specify which particular value types are allowed in the array, e.g. `Array<Person>`. By doing this, you place a constraint on this instance of the array. If you were to try to add an object of a different type to this array, the TypeScript compiler would generate an error. In another piece of code, you can use an array with a different type parameter, e.g. `Array<Customer>`.

The code in listing 4.8 declares a class `Person`, its descendant `Employee`, and a class `Animal`. Then it instantiates each class and tries to store all these objects in the `workers` array using the generic array notation with the type parameter `Array<Person>`.

### **Listing 4.8 Using a generic type**

```
class Person {    ①
  name: string;
}

class Employee extends Person {    ②
  department: number;
}

class Animal {    ③
  breed: string;
}

const workers: Array<Person> = [];    ④

workers[0] = new Person();    ⑤
workers[1] = new Employee();  ⑤
workers[2] = new Animal(); // compile-time error ⑤
```

- ① Declaring the class `Person`
- ② Declaring a subclass of the `Person`
- ③ Declaring the class `Animal`

- ④ Declaring and initializing a generic array with a concrete parameter
- ⑤ Adding objects to the array

In listing 4.8, the last line won't compile because the array `workers` was declared with a type parameter `Person`, and our `Animal` is not a `Person`. But the class `Employee` extends `Person` and is considered a *subtype* of a `Person`; you can use the subtype `Employee` anywhere where the supertype `Person` is allowed.

So, by using a generic array `workers` with the parameter `<Person>`, we announce our plans to store only instances of the class `Person` or compatible types there. An attempt to store an instance of the class `Animal` (as it was defined in listing 4.8) in the same array will result in the following compile-time error: "Type `Animal` is not assignable to type `Person`. Property name is missing in type `Animal`." In other words, using TypeScript generics helps you to avoid errors related to using the wrong types.

**NOTE** The term *generic variance* is about the rules for using subtypes and supertypes in any particular place of your program. For example, in Java, arrays are *covariant*, which means that you can use `Employee[]` (the subtype) where the array `Person[]` (the supertype) is allowed. Since TypeScript supports structural typing, you can use either an `Employee` or any other object literal that's compatible with the type `Person` where the `Person` type is allowed. In other words, generic variance applies to objects that are structurally the same. Given the importance of anonymous types in JavaScript, an understanding of this is important for the optimal use of generics in Typescript. To see if type A can be used where type B is expected, read about structural subtyping at [www.typescriptlang.org/docs/handbook/type-compatibility.html](http://www.typescriptlang.org/docs/handbook/type-compatibility.html).

**TIP** We used `const` (and not `let`) to declare the identifier `workers` because its value never changes in the above listing. Adding new objects to the array `workers` doesn't change the address of the array in memory hence the value of the identifier `workers` remains the same.

If you're familiar with generics in Java or C#, you may get a feeling that you understand TypeScript generics as well. There is a caveat, though. While Java and C# use the *nominal* type system, TypeScript uses the *structural* one as explained in section 2.2.4 in chapter 2.

In the nominal system, types are checked against their names, but in a structural system, by their structure. In languages with the nominal type system, the following line would *always* result in an error:

```
let person: Person = new Animal();
```

With a structural type system, as long as the structures of the type are similar, you may get away with assigning an object of one type to a variable of another. Let's add the property `name` to the class `Animal`, as seen in listing 4.9.

### **Listing 4.9 Generics and the structural type system**

```
class Person {
    name: string;
}

class Employee extends Person {
    department: number;
}

class Animal {
    name: string; ①
    breed: string;
}

const workers: Array<Person> = [];

workers[0] = new Person();
workers[1] = new Employee();
workers[2] = new Animal(); // no errors
```

- ① The only additional line compared to listing 4.8

Now the TypeScript compiler doesn't complain about assigning an `Animal` object to the variable of type `Person`. The variable of type `Person` expects an object that has a property `name`, and the `Animal` object has it! This is not to say that `Person` and `Animal` represent the same types, but these types are compatible.

Moreover, you don't even have to create a new instance of `Person`, `Employee`, or `Animal` classes but use the syntax of object literals instead. Adding the following line to the listing 4.9 is perfectly fine because the structure of the object literal is compatible with the structure of type `Person`:

```
workers[3] = { name: "Mary" };
```

On the other hand, trying to assign the `Person` object to a variable of type `Animal` will result in the compilation error:

```
const worker: Animal = new Person(); // compilation error
```

The error message would read "Property `breed` is missing in type `Person`", and it makes sense because if you declare a variable `worker` of type `Animal` but create an instance of the object `Person` that has no property `breed`, you wouldn't be able to write `worker.breed` hence the compile-time error.

**NOTE**

The previous sentence may irritate savvy JavaScript developers who're accustomed to adding object properties like `worker.breed` without thinking twice. If the property `breed` doesn't exist on the object `worker`, the JavaScript engine would simply create it, right? This works in the dynamically typed code, but if you decided to use the benefits of the static typing, you have to play by the rules. When in Rome, do as the Romans do.

Generics can be used in a variety of scenarios. For example, you can create a function that takes values of various types, but during its invocation, you must explicitly specify a concrete type. To be able to use generic types with a class, interface, or a function, the creator of this class, interface, or function has to write them in a special way to support generic types.

**SIDE BAR****When use generic arrays**

We started this section by showing two different ways of declaring an array of numbers. Let's take another example:

```
const values1: string[] = ["Mary", "Joe"];
const values2: Array<string> = ["Mary", "Joe"];
```

When all elements of the array have the same type, use the syntax as in declaration of `values1` - it's just easier to read and write. But if an array can store elements of different types, you can use generics to restrict the types allowed in the array. For example, you may declare an array that allows only strings and numbers. In the following code snippet, the line that declares `values3` will result in compilation error because boolean values are not allowed in this array.

```
const values3: Array<string | number> = ["Mary", 123, true]; // error
const values4: Array<string | number> = ["Joe", 123, 567]; // no errors
```

<Note to PROD: Shorten URL> Open TypeScript's type definition file (`lib.d.ts`) from the [TypeScript GitHub repository](#) at [github.com/Microsoft/TypeScript/blob/1344b14bd54854fd8d8993b625d9aca6368d1835/bin/lib.d.ts#L](https://github.com/Microsoft/TypeScript/blob/1344b14bd54854fd8d8993b625d9aca6368d1835/bin/lib.d.ts#L) and you'll see the declaration of the interface `Array`, as shown in figure 4.5.

```

1008 interface Array<T> {
1009   /**
1010    * Gets or sets the length of the array. This is a number one higher than the length.
1011    */
1012   length: number;
1013   /**
1014    * Returns a string representation of an array.
1015    */
1016   toString(): string;
1017   toLocaleString(): string;
1018   /**
1019    * Appends new elements to an array, and returns the new length of the array.
1020    * @param items New elements of the Array.
1021    */
1022   push(...items: T[]): number;
1023   /**
1024    * Removes the last element from an array and returns it.
1025    */
1026   pop(): T;

```

**Figure 4.5 The fragment of lib.d.ts describing the Array API**

The `<T>` in line 1008 is a placeholder for a concrete type that must be provided by the application developer during the declaration of the array like we did in listing 4.9. TypeScript requires you to declare a type parameter with `Array`, and whenever you'll be adding new elements to this array, the compiler will check that their type matches the type used in the declaration.

In listing 4.9, we used the concrete type `<Person>` as a replacement of the generic parameter represented by the letter `<T>`:

```
const workers: Array<Person>;
```

But because generics aren't supported in JavaScript, you won't see them in the code generated by the transpiler - generics (as any other types) are erased. Using type parameters is just an additional safety net for developers at compile time.

You can see more generic types `T` in lines 1022 and 1026 in figure 4.5. When generic types are specified with the function arguments, no angle brackets are needed, and you'll see this syntax in listing 4.10. There's no `T` type in TypeScript. The `T` here means the `push()` and `pop()` methods let you push or pop objects of the type provided during the array declaration. For example, in the following code snippet, we declared an array using the type `Person` as a replacement of `T` and that's why we can use the instance of `Person` as the argument of the method `push()`:

```
const workers: Array<Person>;
workers.push(new Person());
```

#### NOTE

The letter `T` stands for type, which is intuitive, but any letter or word can be used for declaring a generic type. In a map, developers often use the letter `K` for key and `V` for value.

Seeing the type `T` in the API of the interface `Array` tells us that its creator enabled support of generics. Even if you're not planning to create your own generic types, it's really important that you understand the syntax of generics while reading someone else's code or TypeScript documentation.

### 4.2.2 Creating your own generic types

You can create your own generic classes, interfaces or functions. In this section, we'll create a generic interface, but the explanations are applicable to creating generic classes as well.

Let's say you have a class `Rectangle` and need to add the capability of comparing the sizes of two rectangles. If you didn't adopt the concept of programming to interfaces (introduced in section 3.2.3 in chapter 3), you'd simply add the method `compareRectangles()` to the class `Rectangle`.

But armed with the concept of programming to interfaces you think differently: *"Today, I need to compare rectangles, and tomorrow they'll ask me to compare other objects. Let me play smart and declare an interface with a function `compare()`. Then the class `Rectangle` as well as any other class in the future can implement this interface. The algorithms for comparing rectangles will be different than comparing, say, triangles, but at least they'll have something in common and the method signature of `compareTo()` will look the same".*

Say there is an object of some type and it should have the method `compareTo()` to compare this object with another object of the same type. If this object is bigger than the other, the method `compareTo()` returns a positive number; if smaller, returns negative; if they are equal, it returns 0.

If you wouldn't be familiar with generic types, you'd define this interface as follows:

```
interface Comparator {
  compareTo(value: any): number;
}
```

The method `compareTo()` can take any object as its argument, and the class that implements `Comparator` must include the appropriate comparison algorithm there (e.g. comparing the sizes of rectangles).

In listing 4.10, we show you a partial implementation of classes `Rectangle` and `Triangle` that use the interface `Comparator`.

## Listing 4.10 Using the interface without a generic type

```
interface Comparator {
    compareTo(value: any): number; ①
}

class Rectangle implements Comparator {

    compareTo(value: any): number { ②
        // the algorithm of comparing rectangles goes here
    }
}

class Triangle implements Comparator {

    compareTo(value: any): number { ③
        // the algorithm of comparing triangles goes here
    }
}
```

- ① The method `compareTo()` takes one parameter of type `any`
- ② Implementation of `compareTo()` in `Rectangle`
- ③ Implementation of `compareTo()` in `Triangle`

If a developer had a good night sleep and a cup of double espresso, he'd create instances of two rectangles and then passed one of them to the method `compare()` of another:

```
rectangle1.compareTo(rectangle2);
```

But what if the coffee machine wasn't working that morning? Our developer can make a mistake and try to compare the rectangle and triangle.

```
rectangle1.compareTo(triangle1);
```

`Triangle` fits the `any` type of the parameter of `compareTo()`, and the above code can result in the runtime error. To catch such an error during the compile time, we could use a generic type (instead of `any`) to place a constraint of which types can be given to the method `compareTo()`:

```
interface Comparator<T> {
    compareTo(value: T): number;
}
```

It's important that both the interface and the method use the generic type represented by the same letter `T`. Now, the classes `Rectangle` and `Triangle` can implement `Comparator` specifying concrete types in angle brackets:

## Listing 4.11 Using the interface with a generic type

```

interface Comparator<T> {      ①
    compareTo(value: T): number;  ②
}

class Rectangle implements Comparator<Rectangle> {

    compareTo(value: Rectangle): number { ③
        // the algorithm of comparing rectangles goes here
    }
}

class Triangle implements Comparator<Triangle> {

    compareTo(value: Triangle): number { ④
        // the algorithm of comparing triangles goes here
    }
}

```

- ① Declaring a generic interface the takes one type as parameter
- ② The method `compareTo()` takes one parameter of the generic type
- ③ In class `Rectangle`, `compareTo()` has a parameter of type `Rectangle`
- ④ In class `Triangle`, `compareTo()` has a parameter of type `Triangle`

Let's say our developer tries to make the same mistake:

```
rectangle1.compareTo(triangle1);
```

The TypeScript code analyzer will underline `triangle1` with a red squiggly line reporting an error "Argument of type 'Triangle' is not assignable to parameter of type 'Rectangle'." As you see, using generic types lowers dependency of your code quality from the coffee machine!

Listing 4.12 shows a working example that declares the interface `Comparator<T>` that declares a method `compareTo()`. This code shows you how this interface can be used for comparing rectangles as well as programmers. Our algorithms are simple:

- If the area of the first rectangle (width multiplied by height) is bigger than of the second one, the first rectangle is bigger. Two rectangles may have the same areas.
- If the salary of the first programmer is higher than of the second one, the first one is richer. Two programmers may have the same salaries.

Running the script from listing 4.12 prints the following:

```
rect1 is bigger
John is poorer
```

## Listing 4.12 A working example that uses a generic interface

```

interface Comparator<T> {      ①
    compareTo(value: T): number;
}

class Rectangle implements Comparator<Rectangle> {      ②

    constructor(private width: number, private height: number){};

    compareTo(value: Rectangle): number {      ③
        return this.width * this.height - value.width * value.height;
    }
}

const rect1:Rectangle = new Rectangle(2,5);
const rect2: Rectangle = new Rectangle(2,3);

rect1.compareTo(rect2) > 0 ? console.log("rect1 is bigger"):
    rect1.compareTo(rect2) == 0 ? console.log("rectangles are equal") : console.log("rect1 is smaller");
④

class Programmer implements Comparator<Programmer> {      ⑤

    constructor(public name: string, private salary: number){};

    compareTo(value: Programmer): number{      ⑥
        return this.salary - value.salary;
    }
}

const prog1:Programmer = new Programmer("John",20000);
const prog2: Programmer = new Programmer("Alex",30000);

prog1.compareTo(prog2) > 0 ? console.log(` ${prog1.name} is richer`):
    prog1.compareTo(prog2) == 0?
        console.log(` ${prog1.name} and ${prog1.name} earn the same amounts`):
        console.log(` ${prog1.name} is poorer`);      ⑦

```

- ① Declare a generic interface Comparator
- ② Create a class that implements Comparator for the type Rectangle
- ③ Implement the method for comparing rectangles
- ④ Compare rectangles (the type T is erased and replaced with Rectangle)
- ⑤ Create a class that implements Comparator for the type Programmer
- ⑥ Implement the method for comparing programmers
- ⑦ Compare programmers (the type T is erased and replaced with Programmer)

You can see this program in action at [codepen.io/yfain/pen/GPRMpg?editors=0011](https://codepen.io/yfain/pen/GPRMpg?editors=0011).

## SIDE BAR Default values of generic types

To use a generic type, you have to provide a concrete type. The following code won't compile because we didn't specify a concrete type parameter while using the type `A`.

```
class A <T> {
    value: T;
}

class B extends A { // Compile error

}
```

Adding the type `any` while *using* the class `A` would fix this error:

```
class B extends A <any> {

}
```

Another way to fix that error would be to specify the default parameter while *declaring* the generic type. The following code snippet will compile without any errors:

```
class A < T = any > { // declaring default parameter type
    value: T;
}

class B extends A { // No errors

}
```

Instead of `any`, you can specify another dummy type:

```
class A < T = {} >
```

With this technique, you won't need to specify a generic parameter when you use generic classes. In section 12.1.5 in chapter 12, you'll see how the class `Component` from the React library uses these default types.

Of course, if you create your own generic types and can provide the parameter defaults that make some business sense (not just `any`), do it by all means.

In this section, we've created a generic interface `Comparator<T>`. Now let's see how to create a generic function.

### 4.2.3 Creating generic functions

We all know how to write a function that can take arguments of concrete types and return a value of a concrete type. This time around, we'll write a generic function that can take parameters of multiple types. But first, let's consider a not-so-good solution where a function can take a parameter of the type `any` and return the value of the same type as shown in listing 4.13. This function can log the objects of different type and return the data that was logged.

#### Listing 4.13 A function with the type any

```
function printMe(content: any): any { ①
    console.log(content);
    return content;
}

const a = printMe("Hello"); ②

class Person{ ③
    constructor(public name: string) { }
}

const b = printMe(new Person("Joe")); ④
```

- ① Declaring a function expression with any
- ② Invoking `printMe()` with the string argument
- ③ Declaring a custom type `Person`
- ④ Invoking `printMe()` with the argument of type `Person`

This function works for various types of arguments, but TypeScript doesn't remember the argument type the function `printMe()` was invoked with. If you hover the mouse over the variables `a` and `b` in your IDE, the TypeScript static analyzer will report the types of both variables as `any`.

If we care to know which types of arguments were used in the invocation of `printMe()`, we need to re-write it as a generic function. Listing 4.14 shows the syntax for providing the generic type `<T>` for the function, its parameter, and the return value.

#### Listing 4.14 A generic function

```
function printMe<T> (content: T): T { ①
    console.log(content);
    return content;
}

const a = printMe("Hello"); ②

class Person{
    constructor(public name: string) { }
}

const b = printMe(new Person("Joe")); ③
```

- ① Using the type T for the function, param, and return value
- ② Invoking printMe() with the string argument
- ③ Invoking printMe() with the argument of type Person

In this version of the function, we declare the generic type for the function as `<T>`, the type of the parameter as well as the return value as `T`. Now the types are preserved, and the type of the constant `a` is `string` and the type of `b` is `Person`. If later on in the script, you'll need to use `a` and `b`, the TypeScript static analyzer (and compiler) will perform the proper type checking.

**NOTE**

By using the same letter `T` for the function argument type and the return type, we place a constraint to ensure that no matter what concrete type is used during invocation, the return type of this function will be the same.

Similarly, you could use generics in fat arrow function expressions. The code sample from listing 4.14 could be re-written as follows:

### **Listing 4.15 Using generic type in fat arrow functions**

```
const printMe = <T> (content: T): T => {    ①
  console.log(content);
  return content;
}

const a = printMe("Hello");

class Person{
  constructor(public name: string) { }
}

const b = printMe(new Person("Joe"));
```

- ① The signature of this fat arrow function starts with `<T>`

You could also invoke these functions specifying the types explicitly in angel brackets:

```
const a = printMe<string>("Hello");

const b = printMe<Person>(new Person("Joe"));
```

But using explicit types is not required here as the TypeScript compiler will infer the type of `a` as `string` and `b` as `Person`.

The above code snippet may not look too convincing - it seems redundant to use `<string>` if it's clear that "Hello" is a string. But this may not always be the case, and you'll see it in listing 4.18.

Let's write yet another little script that will provide more illustrations of using generics in a class and function. We'll declare a class that can represent a pair: a key and a value. Both key and

value can be represented by multiple types so we smell generics here.

### **Listing 4.16 The generic class Pair**

```
class Pair<K, V> { ①
    key: K; ②
    value: V; ③
}
```

- ① Declaring a class with two parameterized types
- ② Declaring a property of generic type K
- ③ Declaring a property of generic type V

When you write a piece of code introducing generic type parameters represented by some letters (e.g. `K` and `V`), you can declare variables using these letters as if they are built-in TypeScript types. When you'll declare (and compile) a concrete `Pair` with specific types for `K` and `V`, these `K` and `V` will be erased and replaced with the declared types.

Actually, let's write a more concise version of the class `Pair`, which has a constructor and automatically-created properties `key` and `value`:

```
class Pair<K, V> {
    constructor(public key: K, public value: V) {}
}
```

Now we'll write a generic function that can compare generic pairs. Is your head spinning yet? The function in listing 4.16 declares two generic types that are also represented by `K` and `V`.

### **Listing 4.17 A generic compare function**

```
function compare <K,V> (pair1: Pair<K,V>, pair2: Pair<K,V>): boolean { ①
    return pair1.key === pair2.key && ②
        pair1.value === pair2.value;
}
```

- ① Declaring the generic function
- ② Compare keys and values of the pairs

During the invocation of the function `compare()` you're allowed to specify two concrete types, which should be the same as the types provided for its parameters - the `Pair` objects.

Listing 4.18 shows the working script that uses the generic class `Pair` as well as the generic function `compare()`. First, we'll create and compare two `Pair` instances that use the type `number` for key and `string` for values. Then we'll compare another `Pair` instances that use the type `string` for both key and value.

## Listing 4.18 Using compare() and Pair

```

class Pair<K, V> {
    constructor(public key: K, public value: V) {}
}

function compare <K,V> (pair1: Pair<K,V>, pair2: Pair<K,V>): boolean {
    return pair1.key === pair2.key &&
           pair1.value === pair2.value;
}

let p1: Pair<number, string> = new Pair(1, "Apple");      ①

let p2 = new Pair(1, "Orange");      ②

// Comparing apples to oranges
console.log(compare<number, string>(p1, p2));      ③

let p3 = new Pair("first", "Apple");      ④

let p4 = new Pair("first", "Apple");      ⑤

// Comparing apples to apples
console.log(compare(p3, p4));      ⑥

```

- ① Creating the first <number, string> pair
- ② Creating the second <number, string> using type inference
- ③ Comparing the pairs (prints false)
- ④ Creating the first <string, string> pair
- ⑤ Creating the second <string, string> pair
- ⑥ Comparing the pairs (prints true)

Please note that in the first invocation of `compare()`, we explicitly specified the concrete parameters, and in the second invocation, we didn't:

```

compare<number, string>(p1, p2)

compare(p3, p4)

```

The first line is easier to reason about as we see what kinds of pairs are `p1` and `p2`. Besides, if you make a mistake and specify the wrong types, the compiler will catch it right away:

```
compare<string, string>(p1, p2) //compile error
```

You can see this code in action at [bit.ly/2HWYZpR](https://bit.ly/2HWYZpR)

In the next script, we'll show you another example of the generic function where we map the members of a string enumeration to the user roles returned by a function.

Imagine, that we have an authorization mechanism that returns one of the following user's roles: admin and manager. We want to use string enums and map the users roles to the corresponding

members of this enum. First, we'll declare a custom type `User` as follows:

```
interface User {
    name: string;
    role: UserRole;
}
```

Then, we'll create a string enum that lists a limited set of constants that can be used as the user's roles.

```
enum UserRole {
    Administrator = 'admin',
    Manager = 'manager'
}
```

For simplicity, we'll create a function that loads the object with hard-coded user's name and its role:

```
function loadUser<T>(): T {
    return JSON.parse('{ "name": "john", "role": "admin" }');
```

In the real-world apps, we'd make a request to some authorization server supplying the user ID to find the user's role, but for our purposes, just returning a hard-coded object will suffice. Listing 4.19 shows a generic function to return the `User` and then map the received user's role to an action using the string enum.

### **Listing 4.19 Mapping string enums**

```
interface User { ①
    name: string;
    role: UserRole;
}

enum UserRole { ②
    Administrator = 'admin',
    Manager = 'manager'
}

function loadUser<T>(): T { ③
    return JSON.parse('{ "name": "john", "role": "admin" }');
}

const user = loadUser<User>(); ④

switch (user.role) { ⑤
    case UserRole.Administrator: console.log('Show control panel'); break;
    case UserRole.Manager: console.log('Hide control panel'); break;
}
```

- ① Declaring a custom type `User`
- ② Declaring a string enum
- ③ Declaring a generic function
- ④ Invoking the generic function with a concrete type `User`

### ⑤ Switching on the user's role with a string enum

The script in listing 4.19 uses the type `User` during the invocation of `loadUser()`, and the generic type `T` declares as the return type of this function becomes a concrete type `User`. Note that the hard-coded object that this function returns, has the same structure as the interface `User`.

Here, `user.role` will always be `admin`, which maps to the enum member `UserRoleAdministrator`, and the script will print "Show control panel". you can see this script in action at [codepen.io/yfain/pen/LMpVKq?editors=0011](https://codepen.io/yfain/pen/LMpVKq?editors=0011).

#### TIP

In listing 10.15 in chapter 10, you'll see the code of the class `MessageServer` that uses a generic type `<T>` in several methods.

## ENFORCING THE RETURN TYPE OF HIGHER-ORDER FUNCTIONS

If a function can receive a function as argument or return another function, we call it *a higher order function*. In this section, we'll like to show you an example that enforces the return type of a higher-order function while allowing arguments of different types. Let's say we need to write a higher-order function that returns the function with the following signature:

```
(c: number) => number
```

This fat arrow function takes a number as an argument and returns a number as well. The higher-order function (we'll use the fat arrow notation) may look like this:

```
(someValue: number) => (multiplier: number) => someValue * multiplier;
```

We didn't write the `return` statement after the first fat arrow because in single-line fat arrow functions, return is implicit. The example of using such a function is shown below:

### Listing 4.20 Using a sample higher-order function

```
const outerFunc = (someValue: number) =>
  (multiplier: number) => someValue * multiplier; ①

const innerFunc = outerFunc(10); ②

let result = innerFunc(5); ③

console.log(result); ④
```

- ① Declaring the higher-order function
- ② `innerFunc` is a closure that know that `someValue = 10`
- ③ Invoking the returned function
- ④ This will print 50

Now let's make our task a bit more complex. We want to allow calling our higher-order function with arguments of different types ensuring that it always returns a function with the same signature:

```
(c: number) => number
```

Let's start with declaring the generic function that can take a generic type `T` but returns the function `(c: number) → number`:

```
type numFunc<T> = (arg: T) => (c: number) => number;
```

Now we can declare variables of type `numFunc`, and TypeScript will ensure that these variables are functions of type `(c: number) → number`. In the following code sample

### **Listing 4.21 Using the generic function numFunc<T>**

```
const noArgFunc: numFunc<void> = () => ①
    (c: number) => c + 5;
const numArgFunc: numFunc<number> = (someValue: number) => ②
    (multiplier: number) => someValue * multiplier;
const stringArgFunc: numFunc<string> = (someText: string) => ③
    (padding: number) => someText.length + padding;

const createSumString: numFunc<number> = () => (x: number) => 'Hello'; ④
```

- ① Invoking the function with no arguments
- ② Invoking the function with a numeric argument
- ③ Invoking the function with a string argument
- ④ Compiler error: `numFunc` expect another signature

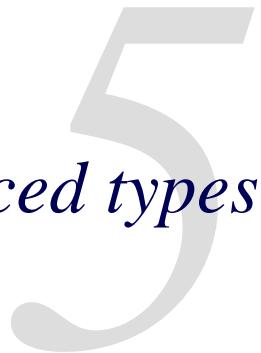
The last line won't compile because the signature of the returned function is `(c: number) → string`, which can't be assigned to the variable of type `numFunc`.

You can see this example in the Playground at [bit.ly/2NpIGmn](https://bit.ly/2NpIGmn)

## **4.3 Summary**

In this chapter you learned:

- What TypeScript enums are about
- How to give a name to a limited number of related values, which can be either numbers or strings
- Why using generic types is better than using the type `any`
- How to understand the code that uses generics
- How to write your own classes, interfaces, and functions that can work with parameterized types



# Decorators and advanced types

## This chapter covers:

- What the TypeScript decorators are for
- How to create a new type based on the existing one using mapped types
- How conditional types work
- Combining mapped and conditional types

In the previous chapters, we've covered the main types that should be sufficient for the most of your coding activities. But TypeScript goes further and offers you additional derivative types that can become quite handy in certain scenarios.

We used the word "advanced" in this chapter's title for a couple of reasons. First, you don't have to know them to be a productive member of your team. Second, their syntax may not be immediately obvious for a software developer who's familiar with other programming languages.

The materials described in this chapter are not required for understanding the rest of the book and may be skipped if you are short on time. You may want to read this chapter if:

- It's time to prepare for the next technical interview, where rarely used knowledge is expected.
- You look at a specific code and have a gut feeling that there should be a more elegant solution.
- You're just curious to learn what else is available as if dealing with interfaces, generics, and enums is not enough.

**NOTE**

In this chapter, we'll use the syntax of *generics* a lot, and we assume that you've read and understand the materials from the section 4.2 in chapter 4 related to the syntax of generics, which is a must for understanding mapped and conditional types described in this chapter.

## 5.1 Decorators

TypeScript doc defines a *decorator* as "a special kind of declaration that can be attached to a class declaration, method, accessor, property, or parameter. Decorators use the form `@expression`, where expression must evaluate to a function that will be called at runtime with information about the decorated declaration."

Say you have `class A {...}` and there is a magic decorator called `@Injectable()` that knows how to instantiate classes and inject their instances into other objects. Then, we can decorate one of our classes like this:

```
@Injectable() class A {}
```

As you could guess, the `@Injectable()` decorator would somehow change the behavior of `class A`. What would be the other way of changing the behavior of `class A` without modifying its code? We could create a subclass of `class A` and add or override the behavior there, but simply adding a decorator to the class definition looks more elegant.

We could also say that a decorator adds metadata to a specific target, which is a `class A` in our example. In general, a metadata is additional data about some other data. Take an mp3 file - we can say the song is the data. But the mp3 file may have additional properties like the name of the artist, album, an image, etc; this is the metadata of this mp3 file.

On a similar note, you can annotate a TypeScript class with some metadata describing additional features you'd like this class to have. Names of TypeScript decorators start with the `@`-sign, e.g. `@Component`. While you can write your own decorators, it's more likely that you'll be just using the decorators available in some library or a frameworks. Consider the following simple class:

```
class OrderComponent {
  quantity: number;
}
```

Imagine that you want to turn this class into a UI component. Moreover, you want to declare that the value for the property `quantity` will be provided by the parent component. If you'd be using the Angular framework with TypeScript, you'd use a built-in decorator `@Component()` for the class and `@Input()` for the property as seen in listing 5.1.

## Listing 5.1 An example of an Angular component

```

@Component({      ①
  selector: 'order-processor',    ②
  template: `Buying {{quantity}} items`  ③
})
export class OrderComponent {
  @Input() quantity: number;    ④
}

```

- ① Applying the `@Component` decorator
- ② This component can be used in HTML as `<order-processor>`
- ③ The browser should render this text
- ④ The value for this input property is provided by the parent component

In Angular, the `@Component()` decorator can be applied only to a class, and it supports various properties, e.g. `selector` and `template`. The `@Input()` decorator can be applied only to the properties of a class. We can also say that these two decorators provided the metadata about the class `OrderComponent` and the property `quantity` respectively.

To make the decorators useful, there should be some code that knows how to parse them and do what these decorators prescribe. In the example from listing 5.1, the Angular framework would parse these decorators and generate additional code to turn the class `OrderComponent` into a renderable UI component.

**NOTE**

In chapters 11 and 12, we'll be working with the Angular framework, and you'll see many examples of using decorators there. Decorators are heavily used by the server-side framework Nest.js (see [docs.nestjs.com/custom-decorators](https://docs.nestjs.com/custom-decorators)), state management library MobX (see [mobx.js.org/refguide/modifiers.html](https://mobx.js.org/refguide/modifiers.html)), and UI library Stencil.js (see [stenciljs.com/docs/decorators](https://stenciljs.com/docs/decorators)).

TypeScript doesn't come with any built-in decorators, but you can create your own or use the ones provided by the framework or a library of your choice.

Decorators in listing 5.1. allowed you to specify additional behavior for the class and its property in a concise and declarative manner. Of course, a decorator is not the only way for adding behavior to the object. For example, creators of Angular could have created an abstract class `UIComponent` with a particular constructor and force the developers to extend it every time they want to turn a class to a UI component. But using a decorator component is a nicer, more readable and declarative solution.

Decorators (unlike inheritance) separate the concerns and facilitate easier code maintenance,

since the framework is free to interpret them as it wants. In contrast, if a component was a subclass, it could override or just expect or rely on the certain behavior of the methods in the superclass.

**NOTE**

There is a proposal to add decorators to JavaScript, which is currently a Stage 2 draft (see [tc39.github.io/proposal-decorators](https://tc39.github.io/proposal-decorators)).

Although decorators were introduced back in 2015, they are still considered an experimental feature, and you have to compile your app with the tsc option `--experimentalDecorators`. If you use `tsconfig.json`, add the following compiler's option there:

```
"experimentalDecorators": true
```

A decorator can be used to observe or modify the definition of the target (e.g. a class, method, et al). The signatures of these special functions differ depending on what the target is. In this chapter, we'll show you how to create class and method decorators.

### 5.1.1 Creating class decorators

A class decorator is applied to the class, and the decorator function is executed when the constructor executes. The class decorator requires one parameter - a constructor function for the class. In other words, a class decorator will receive the constructor function of the decorated class. Listing 5.2 declares a class decorator that just logs the information about the class on the console.

#### **Listing 5.2 Declaring a custom decorator whoAmI**

```
function whoAmI (target: Function): void { ①
  console.log(`You are: \n ${target}`) ②
}
```

- ① Declaring the decorator, which takes a constructor function as an argument
- ② Logging the target class information

**NOTE**

If the return type of the class decorator is `void`, this function doesn't return any value (see section 2.1.1 in chapter 2). Such a decorator won't be replacing the class declaration - it'll just observe the class as in listing 5.3. But the decorator can modify the class declaration, in which case it would need to return the modified version of the class (i.e. the constructor function), and we'll show you how to do it at the end of this section.

To use this decorator, just prepend the class name with `@whoAmI` as shown in listing 5.3. The constructor function of the class will be automatically provided for the decorator. In our example, it has two arguments: a string and number.

### Listing 5.3 Applying the decorator `@whoAmI` to a class

```
@whoAmI
class Friend {

    constructor(private name: string, private age: number){}
}
```

When the TypeScript code is transpiled into JavaScript, tsc checks if any decorators were applied, in which case it'll generate additional JavaScript code that will be used during the runtime.

You can run this code sample in the Playground at [bit.ly/2SNnSq4](https://bit.ly/2SNnSq4) by pressing the button Run. It'll run the JavaScript version of the code producing the following output on the browser's console:

```
You are:
function Friend(name, age) {
    this.name = name;
    this.age = age;
}
```

**TIP** To see the effect of the decorator `@whoAmI` on the generated code, in the Playground, remove this decorator from the class `Friend` declaration and note that the TypeScript compiler generates different JavaScript.

You may say that the `@whoAmI` decorator is not overly useful, so let's create another one. Say, you're developing a UI framework and want to allow turning classes into UI components in a declarative way. You want to create a function that takes an arbitrary argument, e.g. an HTML string for rendering as seen in listing 5.4.

### Listing 5.4 Declaring a custom decorator `UIcomponent`

```
function UIcomponent (html: string): void { ①
    console.log(`The decorator received ${html}\n`); ②

    return function(target: Function) { ③
        console.log(`Someone wants to create a UI component from \n ${target}`);
    }
}
```

- ① This decorator factory has an argument
- ② Print the string received by the decorator
- ③ This is a decorator function

Here you see a decorator function inside another function. We can call the outer function *a decorator factory*. It can take any arguments and apply some app logic to decide which decorator to return. Listing 5.4 has just one `return` statement, and this code always returns the same

decorator function, but nothing stops you from conditionally returning the decorator you need based on the parameters provided to the factory function.

**NOTE**

A bit later in listing 5.9, you'll see the requirements to the decorator signature that depends on what it decorates. Then how come we were able to use an arbitrary argument in the function `UIComponent()`? The reason is that `UIComponent()` is not a decorator, but a decorator factory that returns the actual decorator with the proper signature `function (target: Function)`.

Listing 5.5 shows the class `Shopper` decorated as `UIcomponent`.

### **Listing 5.5 Applying a custom decorator `UIcomponent`**

```
@UIcomponent('<h1>Hello Shopper!</h1>') ①
class Shopper {

    constructor(private name: string) {} ②

}
```

- ① Passing an HTML to the decorator
- ② A class constructor that takes a shopper's name

Running this code will produce the following output:

```
The decorator received <h1>Hello Shopper!</h1>

Someone wants to create a UI component from
function Shopper(name) {
    this.name = name;
}
```

You can see this code in action at [codepen.io/yfain/pen/JxpwKQ?editors=0011](https://codepen.io/yfain/pen/JxpwKQ?editors=0011).

So far all our decorators examples were observing the classes - they didn't modify class declarations. In the next example, we'll show you a decorator that does it. But first, we'd like to show you a constructor mixin.

In JavaScript, a *mixin* is a class that implements a certain behavior. Mixins are not meant to be used alone, but their behavior can be added to other classes. While JavaScript doesn't support multiple inheritance, you can compose behaviors from multiple classes using mixins.

If a mixin has no constructor, mixing its code with other classes comes down to copying its properties/methods to the target class. But if a mixin has its own constructor, it needs to be capable of taking any number of parameters of any types, otherwise it won't be "mixable" with arbitrary constructors of target classes.

TypeScript supports a *constructor mixin* that has the following signature:

```
{ new(...args: any[]): {} }
```

It uses a single rest argument (three dots) of type `any[]` and can be mixed with other classes that have constructors. Let's declare a type alias for this mixin:

```
type constructorMixin = { new(...args: any[]): {} };
```

Accordingly, the following signature represents a generic type `T` that extends `constructorMixin`; in TypeScript, it also means that the type `T` is assignable to the type `constructorMixin`:

```
<T extends constructorMixin>
```

You'll use this signature for creating a class decorator that modifies the original constructor of a class. The class decorator signature will look like this:

```
function <T extends constructorMixin> (target: T) {
    // the decorator is implemented here
}
```

Now we are prepared for writing the decorator that modifies the declaration (and constructor) of a target class. Let's say we have the following class `Greeter`:

### **Listing 5.6 Undecorated class Greeter**

```
class Greeter {
    constructor(public name: string) { }
    sayHello() { console.log(`Hello ${this.name}`) };
}
```

We can instantiate and use it like this:

```
const grt = new Greeter('John');
grt.sayHello(); // prints "Hello John"
```

We want to create a decorator, that can accept a salutation parameter, and add to the class a new property `message` concatenating the given salutation and name. Also, we want to replace the code of the method `sayHello()` to print the message.

Listing 5.7 shows a *higher-order function* (returns a function) that implements our decorator. We can also call it a *factory function* because it constructs and returns a function.

## Listing 5.7 Declaring the decorator `useSalutation`

```
function useSalutation(salutation: string) { ①
  return function <T extends constructorMixin> (target: T) { ②
    return class extends target { ③
      name: string;
      private message = 'Hello ' + salutation + this.name; ④
      sayHello(){console.log(`>${this.message}`);} ⑤
    }
  }
}
```

- ① The factory function that takes one parameter - salutation
- ② The decorator's body
- ③ Redeclaring the decorated class
- ④ Adding a private property to the new class
- ⑤ Redeclaring the method

Starting from the line `return class extends target` we provide another declaration of the decorated class. In particular, we've added the new property `message` to the original class and replaced the body of the method `sayHello()` to use the salutation provided in the decorator.

In the next listing, we use the decorator `@useSalutation` with the class `Greeter`, and invoking `grt.sayHello()` will print "Hello Mr. Smith".

## Listing 5.8 Using the decorated class `Greeter`

```
@useSalutation("Mr. ") ①
class Greeter {

  constructor(public name: string) { }
  sayHello() { console.log(`Hello ${this.name} `) }
}

const grt = new Greeter('Smith');
grt.sayHello();
```

- ① Applying the decorator with an argument to the class

You can see and run this code at [bit.ly/2wKyz4c](https://bit.ly/2wKyz4c). Click on the button Run and open the browser console to see the output.

It's great that we have such a powerful mechanism that can replace the class declaration, but use it with caution. Try not to change the public API of the class because the static type analyzer won't be offering auto-complete for the decorator-added public properties or methods.

Say, you wanted to add a public method `sayGoodbye()` to the target class in the

`useSalutation()` function. After typing `grt.` as in listing 5.9, your IDE would still prompt you that this object only has the method `sayHello()` and the property `name`. It won't suggest `sayGoodbye()` even though you can write `grt.sayGoodbye()` and it'll work just fine.

## SIDE BAR Formal declarations of the decorators' signatures

A decorator is a function and its signature depends on the target. A signature for the class and method decorators won't be the same. After you install TypeScript, it'll include several files with type declarations. One of them is called `lib.es5.d.ts`, and it includes the types for decorators for various targets as seen in listing 5.9.

### Listing 5.9 Decorator signatures

```
declare type ClassDecorator = ①
    <TFunction extends Function>(target: TFunction) => TFunction | void;
declare type PropertyDecorator = ②
    (target: Object, propertyKey: string | symbol) => void;
declare type MethodDecorator = ③
    <T>(target: Object, propertyKey: string | symbol,
        descriptor: TypedPropertyDescriptor<T>) =>
        TypedPropertyDescriptor<T> | void;
declare type ParameterDecorator = ④
    (target: Object, propertyKey: string | symbol,
        parameterIndex: number) => void;
```

- ① The signature for a class decorator
- ② The signature for a property decorator
- ③ The signature for a method decorator
- ④ The signature for a parameter decorator

In section 4.2.3 in chapter 4, we explained generic functions and their syntax for the named functions as well as fat arrow ones. The materials from that section should help you in understanding the signatures shown in listing 5.7. Consider the following line:

```
<T>(someParam: T) => T | void
```

That's right, it declares that a fat arrow function can take a parameter of a generic type `T` and return either the value of type `T` or `void`. Now let's try to read the declaration of the signature of the `ClassDecorator`.

```
<TFunction extends Function>(target: TFunction) => TFunction | void
```

It declares that a fat arrow function can take a parameter of a generic type `TFunction`, which has an additional constraint: the concrete type must be a subtype of `Function`. Any TypeScript class is a subtype of a `Function`, which represents a constructor function. In other word, the target for this decorator must be a class, and the decorator can either return a value of this class's type or return no value.

Take another look at the class decorator `@whoAmI` shown in listing 5.2. We didn't use the fat arrow expression there but that function had the following signature, which is allowed for class decorators:

```
function whoAmI (target: Function): void
```

Since the signature of the `whoAmI()` function doesn't return a value, we can say that this decorator just observes the target. If you wanted to modify the original target in the decorator, you'd need to return the modified class, but its type would be the same as originally provided in lieu of `TFunction`.

### 5.1.2 Creating method decorators

Now let's create a decorator that can be applied to a class's method. For example, you may want to create a decorator `@deprecated` to mark the methods that will be removed soon. As you can see in listing 5.9, the function `MethodDecorator` requires three parameters:

- `target` - an object that refers to the method class
- `propertyKey` - the name of the method being decorated
- `descriptor` - a descriptor of the method being decorated

The parameter `descriptor` contains the object describing the method that your code is decorating. In particular, `TypedPropertyDescriptor` has the property `value` that will store the original code of the decorated method. By changing the value of this property inside the method decorator, you can modify the original code of the decorated method. Let's consider a class `Trader` that has a method `placeOrder()`:

```
class Trade {
  placeOrder(stockName: string, quantity: number, operation: string, traderID: number) {
    // the method implementation goes here
  }
  // other methods go here
}
```

Say, there is a trader with the ID 123, and she can place an order to buy 100 shares of IBM as follows:

```
const trade = new Trade();
trade.placeOrder('IBM', 100, 'Buy', 123);
```

This code worked fine for years, but the new regulation came in: "For audit purposes, all trades must be logged". One of the ways to do this is going through the code of all methods that are related to buying or selling financial products, which may have different parameters and add the code that logs their invocations. But creating a method decorator `@logTrade` that works with any method and logs the parameters would be a more elegant solution. Listing 5.10. shows the code of the method decorator `@logTrade`.

### **Listing 5.10 The method decorator `@logTrade`**

```
function logTrade(target, key, descriptor) {    ①
  const originalCode = descriptor.value;      ②
  descriptor.value = function () {    ③
    console.log(`Invoked ${key} providing:`, arguments);
    return originalCode.apply(this, arguments);  ④
  };
  return descriptor;    ⑤
}
```

- ① The method decorator must have three arguments
- ② Storing the original method's code
- ③ Modifying the code of the method being decorated
- ④ Invoking the target method
- ⑤ Returning the modified method

We stored the original method code, then modified the received descriptor by adding a `console.log()` statement. Then we used the JavaScript function `apply()` to invoke the decorated method. Finally, we returned the modified method descriptor.

### **Listing 5.11 Using the `@logTrade` decorator**

```
class Trade {
  @logTrade ①
  placeOrder(stockName: string, quantity: number,
             operation: string, tradedID: number) {
    // the method implementation goes here
  }
}

const trade = new Trade();
trade.placeOrder('IBM', 100, 'Buy', 123); ②
```

- ① Decorating the method `placeOrder()`
- ② Invoking `placeOrder()`

After invoking the decorated method `placeOrder()`, the console output will look similar to this:

```
Invoked placeOrder providing:
Arguments(4)
0: "IBM"
1: 100
2: "Buy"
3: 123
```

By creating a method decorator, we eliminated the need to update the code of potentially multiple trade-related methods that require auditing. Besides, the decorator `@logTrade` can work with any method that wasn't even written yet.

You can run this code in the TypeScript Playground at [bit.ly/2tdvGXQ](https://bit.ly/2tdvGXQ).

We've shown you how to write class and method decorators, which should give you a good foundation for mastering property and parameter decorators on your own.

## 5.2 Mapped types

Mapped types allow you to create new types from the existing ones. This is done by applying a transformation function to an existing type. Let's see how they work.

### 5.2.1 The mapped type `Readonly`

Imagine that you need to pass the objects of type `Person` (shown next) to the function `doStuff()` for processing.

```
interface Person {
  name: string;
  age: number;
}
```

The type `Person` is used in multiple places, but you don't want to allow the function `doStuff()` to accidentally modify some of the `Person`'s properties like `age` in listing 5.12.

#### Listing 5.12 The unlawful change of the age

```
const worker: Person = {name: "John", age: 22};

function doStuff(person: Person) {
  person.age = 25; ①
}
```

- ① We don't want to allow this

None of the properties of the type `Person` was declared with the `readonly` modifier. Should we declare another type just to be used with `doStuff()` as follows?

```
interface ReadonlyPerson {
  readonly name: string;
  readonly age: number;
}
```

Does it mean that you need to declare (and maintain) a new type each time when you need to have a read-only version of the existing one? There is a better solution. We can use a built-in mapped type  `Readonly` to turn all the properties of a previously declared type to be `readonly`. We'll just need to change the signature of the function `doStuff()` to take the argument of type  `Readonly<Person>` instead of `Person`, just like this:

### **Listing 5.13 Using the mapped type `Readonly`**

```
const worker: Person = {name: "John", age: 22};

function doStuff(person: Readonly<Person>) { ①
    person.age = 25; ②
}
```

- ① Modifying the existing type with the mapped type  `Readonly`
- ② This line generates a compiler error

To understand why an attempt to change the value of the property `age` generates a compiler error, you need to see how the type  `Readonly` is declared, which in turn requires an understanding of the lookup type  `keyof`.

## **KEYOF AND A LOOKUP TYPE**

Reading the declarations of the built-in mapped types in the file `typescript/lib/lib.es5.d.ts` helps in understanding their inner-workings and requires familiarity with the TypeScript's *index type query*  `keyof` and *a lookup type*.

You can find the following declaration of the  `Readonly` mapping function in `lib.es5.d.ts`:

### **Listing 5.14 The declaration of the mapped type `Readonly`**

```
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
};
```

We assume that you've read about generics in chapter 4, and you know what `<T>` in angle brackets means. Usually, the letter `T` in generics represents type, `K` - key, `V` - value, `P` - property et al.

`keyof` is called *index type query*, and it represents a union of allowed property names (the keys) of the given type. If the type `Person` would be our `T`, then  `keyof T` would represent the union of `name` and `age`. Figure 5.1 shows a screenshot taken while hovering the mouse over the custom type `propNames`. As you see, the type of  `propName` is a union of `name` and `age`.

```

interface Person {
  name: string;
  age: number;
}

type propNames = "name" | "age"
type propNames = keyof Person;

```

**Figure 5.1 Applying keyof to the type Person**

In listing 5.14, the fragment `[P in keyof T]` means "give me the union of all the properties of the given type `T`." This seems as if we're accessing the elements of some object, but actually, this is done for declaring types. The `keyof` type query can be used only in type declarations.

Now we know how to get access to the property names of a given type, but to create a mapped type from the existing one, we also need to know the property types. In case of the type `Person`, we need to be able to find out programmatically that property types are `string` and `number`.

This is what *lookup types* are for. The piece `T[P]` is a lookup type, and it means "Give me the type of a property `P`". Figure 5.2 shows a screenshot taken while hovering the mouse over the type `propTypes`. The types of the properties are `string` and `number`.

```

type propNames = keyof Person;
|
type propTypes = string | number
type propTypes = Person[propNames];

```

**Figure 5.2 Getting the types of the Person's properties**

Now let's read the code in listing 5.14 one more time. The declaration of the type  `Readonly<T>` means "Find the names and types of the properties of the provided concrete type and apply the `readonly` qualifier to each property".

In our example,  `Readonly<Person>` will create a mapped type that will look like this:

### Listing 5.15 Applying the mapped type `Readonly` to the type `Person`

```

interface Person {
  readonly name: string;
  readonly age: number;
}

```

Now you see why an attempt to modify the person's age results in the compiler's error "Cannot assign to age because it's a read-only property". Basically, we took an existing type `Person` and mapped it to a similar type but with the read-only properties. You can see that this code won't compile in the Playground at [bit.ly/2GC2TUN](https://bit.ly/2GC2TUN).

You may say, "OK, I understand how to apply the mapped type  `Readonly`, but what's the practical use of it?" In chapter 10 in listing 10.15 you can see two methods that use the

Readonly type with their message argument, for example:

```
replyTo(client: WebSocket, message: Readonly<T>): void
```

This method can send messages to blockchain nodes over the WebSocket protocol. The messaging server doesn't know what types of messages are going to be sent, and the message type is generic. To prevent accidental modification of the message inside `replyTo()`, we use the mapped type `Readonly` there.

Let's consider one more code sample that illustrates the benefits of using `keyof` and `T[P]`. Imagine, we need to write a function to filter a generic array of objects to keep only those that have a specified value in a specified property. In the first version, we won't use type checking and will write the function as follows:

### **Listing 5.16 Not a good version of the filter function**

```
function filterBy<T>(
  property: keyof T,
  value: any,
  array: T[])
{
  return array.filter(item => item[property] === value); ①
}
```

- ① Keep only those object that have provided value in the specified property

This solution can open the door to hard-to-find bugs if this function is called with the non-existing property name or the wrong value type. Listing 5.17 declares a type `Person` and the `filterBy()` functions. The last two lines invoke the function providing either the non-existing property `lastName` or the wrong type for `age`.

## Listing 5.17 A buggy version

```
interface Person {
    name: string;
    age: number;
}

const persons: Person[] = [
    { name: 'John', age: 32 },
    { name: 'Mary', age: 33 },
];

function filterBy<T>( ❶
    property: any,
    value: any,
    array: T[])
{
    return array.filter(item => item[property] === value); ❷
}

console.log(filterBy('name', 'John', persons)); ❸

console.log(filterBy('lastName', 'John', persons)); ❹

console.log(filterBy('age', 'twenty', persons)); ❺
```

- ❶ This function doesn't do any type checking
- ❷ Filtering data based on the property/value
- ❸ The correct invocation of the function
- ❹ The wrong invocation of the function
- ❺ The wrong invocation of the function

Each of the last two lines of the code will return zero object without any complains even though the type `Person` has no property `lastName` and the type of the `age` property is not a string. In other words, the code in listing 5.17 is buggy.

Let's change the signature of the function `filterBy()` so it would catch these bugs during the compile time. The new version of `filterBy()` is shown in listing 5.18:

## Listing 5.18 A better version of filterBy()

```
function filterBy<T, P extends keyof T>( ❶
    property: P, ❷
    value: T[P], ❸
    array: T[])
{
    return array.filter(item => item[property] === value);
}
```

- ❶ Check that provided property belongs to the union [keyof T]
- ❷ The property to filter by
- ❸ A value for filtering with checking the type of the provided property

First of all, the fragment `<T, P extends keyof T>` tells us that our function has accept two generic values: `T` and `P`. We also add a restriction `P extends keyof T`, i.e. that `P` must be one of the properties of the provided type `T`. If the concrete type for `T` is `Person` then `P` can be either `name` or `age`.

The function signature in listing 5.18 has yet another restriction: `value: T[P]`, which means the provided value must be of the same type as declared for `P` in type `T`. That's why the following lines will give compile errors:

### **Listing 5.19 These lines won't compile**

```
filterBy('lastName', 'John', persons) ①
filterBy('age', 'twenty', persons) ②
```

- ① Non-existing property `lastName`
- ② The value of `age` must be a number

As you see, introducing `indexof` and a lookup type in the function signature allows to catch possible errors during compile time. You can see this code sample in action at [bit.ly/2V6Q4pO](https://bit.ly/2V6Q4pO).

## **5.2.2 Declaring your own mapped types**

Listing 5.14 shows the transformation function for the built-in mapped type `Readonly`. You can define your own transformation functions using similar syntax. Let's try to define the type `Modifiable` - an opposite to `Readonly`.

We took a type `Person` made all of its properties read-only by applying `Readonly` mapped type: `Readonly<Person>`. Let's consider another scenario. Say, the properties of the type `Person` were originally declared with the `readonly` modifier as follows:

```
interface Person {
  readonly name: string;
  readonly age: number;
}
```

How can you remove the `readonly` qualifiers from the `Person` declaration if need be? There is no built-in mapped type for it, so let's declare one as in listing 5.20:

### **Listing 5.20 Declaring a custom mapped type Modifiable**

```
type Modifiable<T> = {
  -readonly[P in keyof T]: T[P];
};
```

The minus sign in front of the `readonly` qualifier removes it from all properties of the given type. Now you can remove the `readonly` restriction from all properties by applying the mapped type `Modifiable` as seen in listing 5.21:

## Listing 5.21 Applying the Modifiable mapped type

```
interface Person {
  readonly name: string;
  readonly age: number;
}

const worker1: Person = {name: "John", age: 25};

worker1.age = 27;      ①

const worker2: Modifiable<Person> = {name: "John", age: 25};

worker2.age = 27;      ②
```

- ① Results in the compiler error
- ② No errors here

You can see this code in the Playground at [bit.ly/2GMAf3c](https://bit.ly/2GMAf3c).

### 5.2.3 Other built-in mapped types

You know that if a property name in the type declaration ends with the modifier ?, this property is optional. Say we have the following declaration of the type Person:

```
interface Person {
  name: string;
  age: number;
}
```

Since none of the property names ends with a question mark, providing values for name and age is mandatory. What if you have a need in type that has the same properties as in Person, but all of its properties should be optional? This is what the mapped type Partial<T> is for. Its mapping function is declared in lib.es5.d.ts as in listing 5.22:

## Listing 5.22 The declaration of the mapped type Partial

```
type Partial<T> = {
  [P in keyof T]?: T[P];
};
```

Have you spotted the question mark there? Basically, we create a new type by appending the question mark to each property name of the given type. The mapped type Partial makes all properties in the given type optional. Figure 5.3 shows a screenshot taken while hovering the mouse over the declaration of the worker1 variable.

```

interface Person {
  name: string;
  age: number;
}

[ts]
Type '{ name: string; }' is not assignable to
type 'Person'.
  Property 'age' is missing in type '{ name:
string; }'. [2322]
const worker1: Person
const worker1: Person = { name: "John" };

const worker2: Partial<Person> = { name: "John"};

```

**Figure 5.3 Applying the Partial type**

It shows an error message because the variable `worker1` has the type `Person`, where each property is required, but the value for `age` was not provided. There are no errors in initializing `worker2` with the same object because the type of this variable is `Partial<Person>`, so all its properties are optional.

There is a way to make all properties of a type optional, but can you do the opposite? Can you take a type that was declared with some optional properties and make all of them required? You bet! This can be done with the mapped type `Required` that's declared as follows:

```

type Required<T> = {
  [P in keyof T]-?: T[P];
};

```

The `-?` means remove the modifier `?`.

Figure 5.4 shows a screenshot taken while hovering the mouse over the declaration of the `worker2` variable. The properties `age` and `name` were optional in the base type `Person` but are required in the mapped type `Required<Person>` hence the error about missing `age`.

```

interface Person {
  name?: string;
  age?: number;
}

const worker1: Person = { name: "John"};

[ts] 'worker2' is declared but its value is never read. [6133]
[ts] Property 'age' is missing in type '{ name: string; }' but required in type 'Required<Person>'. [2741]
  • main.ts(92, 3): 'age' is declared here.

const worker2: Required<Person>
const worker2: Required<Person> = { name: "John"};

```

**Figure 5.4 Applying the Required type**

**TIP**

The type `Required` was introduced in TypeScript 2.8. If your IDE doesn't recognize this type, make sure it uses the proper version of the TypeScript language service. In Visual Studio Code you can see its version in the bottom right corner. Click on it to change to a newer version if you have it installed.

You can apply more than one mapped type to a given type. In listing 5.23, we apply `Readonly` and `Partial` to the type `Person`. The former will make each property read-only and the latter will make each property optional.

### Listing 5.23 Applying more than one mapped type

```

interface Person {
  name: string;
  age: number;
}

const worker1: Readonly<Partial<Person>>      ①
  = {name: "John"}; ②

worker1.name = "Mary"; // compiler's error ③

```

- ① The `worker1` is still a `Person`, but its properties are a read-only and optional
- ② Initializing the property `name` but not the optional `age`
- ③ `name` is read-only and can be initialized only once

TypeScript offers yet another useful mapped type called `Pick`. It allows you to declare a new type by picking a subset of properties of the given type. Its transformation function looks like this:

```

type Pick<T, K extends keyof T> = {
  [P in K]: T[P];
}

```

```
};
```

The first argument expects an arbitrary type `T`, and the second - a subset `K` of the properties of this `T`. You can read it as "From `T`, pick a set of properties whose keys are in the union `K`". The next listing shows the type `Person` that has three properties. With the help of `Pick`, we declare a mapped type `PersonNameAddress` that has two string properties: `name` and `address` as seen in listing 5.24.

### Listing 5.24 Using the Pick mapped type

```
interface Person {  
    name: string;  
    age: number;  
    address: string;  
}  
  
type PersonNameAddress<T, K> = Pick<Person, 'name' | 'address' >;
```

- ① Declaring the type `Person` with 3 properties
- ② Declaring the mapped type `PersonNameAddress` with 2 properties

You may be thinking, "The discussion of the built-in mapped types is good and they do seem to be really useful, but do I need to know how to implement my own?" The answer is "Yes", and you'll see examples of using using the mapped type `Pick` for defining a custom mapped type in figure 5.5 in this chapter and in listing 10.22 in chapter 10.

Mapped types allow you to modify existing types, but TypeScript offer yet another way of changing the type based on some condition, and this will be the subject of our next discussion.

## 5.3 Conditional types

With any mapped type, the transformation function is always the same, but with conditional types, it depends on a specific condition. Many programming languages including JavaScript and TypeScript support conditional (ternary) expressions, for example:

```
a < b ? doSomething() : doSomethingElse()
```

If the value of `a` is less than `b`, invoke the function `doSomething()`, otherwise invoke `doSomethingElse()`. This expression *checks the values* and conditionally executes different code. A conditional type would also use a conditional expression, but it would *check the expressions type*. A conditional type will always be declared in the following form:

```
T extends U ? X : Y
```

Here, the keyword `extends` means "inherits from `U`" or "is `U`". As with generics, these letters can represent any types.

In object-oriented programming, the declaration `class Cat extends Animal` means that `Cat` is `Animal` and `Cat` has the same (or more) features as `Animal`. Another way to put it is that a `Cat` is a more specific version of an animal. This also means that the `Cat` object can be assigned to the variable of type `Animal`.

But can the `Animal` object be assigned to the variable of type `Cat`? No, it can't. Every `Cat` is an animal but not every animal is a `Cat`.

Similarly, the expression `T extends U` ? checks if `T` is assignable to `U`. If this is true, we'll use the type `x` otherwise `y`. The expression `T extends U` means that the value of type `T` can be assigned to the variable of type `U`.

### TIP

We already mentioned *assignable* types while discussing listing 2.20 in chapter 2 and while explaining listing 3.6 in chapter 3.

Let's consider an example of a function that can have different return types based on some condition. We want to write the function `getProducts()`, which should return the type `Product` if a numeric product ID was provided as an argument. Otherwise, this function has to return an array `Product[]`. Using conditional types, the signature of this function can look as follows:

```
function getProducts<T>(id?: T):
    T extends number ? Product : Product[]
```

If the type of the argument is `number` then the return type of this function is `Product`, otherwise - `Product[]`. Listing 5.25 includes a sample implementation of such a function. If the provided optional `id` is a number, we return one product; if not - an array of two products.

### **Listing 5.25 A function with a conditional return type**

```
class Product {
    id: number;
}

const getProducts = function<T>(id?: T):
    T extends number ? Product : Product[] { ①

    if (typeof id === 'number') { ②
        return { id: 123 } as any;
    } else {
        return [{ id: 123 }, { id: 567 }] as any;
    }
}

const result1 = getProducts(123); ③

const result2 = getProducts(); ④
```

- ① Declaring a conditional return type
- ② checking the type of the provided argument

- ③ Invoking the function with a numeric argument
- ④ Invoking the function with no arguments

The type of the variable `result1` is `Product`, and the type of the `result2` is `Product[]`. You can see it for yourself by hovering the mouse pointer over these variables in the Playground at [bit.ly/2BTD3Z3](https://bit.ly/2BTD3Z3).

In listing 5.21, we used the `as` type assertion that tells TypeScript that it shouldn't be inferring the type because you know about this type better than TypeScript. `as any` means "TypeScript, don't complain about this type". The problem is that the narrowed type of `id` is not picked up by the conditional type, so the function cannot evaluate the condition and narrow the return type to `Product`.

The section 3.1.6 in chapter 3 has a similar example implemented using method overloading. Conditional types can be used in many different scenarios. Let's discuss another use case.

TypeScript has a built-in conditional type `Exclude`, which allows you to discard the specified types. `Exclude` is declared in the file `lib.es5.d.ts` as follows:

```
type Exclude<T, U> = T extends U ? never : T;
```

This type excludes those types that are assignable to `U`. Note the use of the type `never`, which means "this should never exist; filter it out". If the type `T` is not assignable to `U` then keep it.

Let's say we have a class `Person`, and we use this class in multiple places in the app for the popular TV show "The Voice":

```
class Person {
  id: number;
  name: string;
  age: number;
}
```

As you know, all the vocalists must go through blind auditions, where the judges don't see them and know nothing about them. For blind auditions, we want to create another type, which is the same as `Person` except it won't have the properties `name` and `age`. In other words, we want to exclude the properties `name` and `age` from the type `Person`.

From the previous section, you remember that the lookup type `keyof` can give you the list of all properties from a type. Hence, the following type will contain all the properties of `T` except those that belong to the given type `K`.

```
type RemoveProps<T, K> = Exclude<keyof T, K>;
```

Let's create a new type that will be like `Person` minus `name` and `age`:

```
type RemainingProps = RemoveProps<Person, 'name' | 'age'>;
```

In this example, the type `K` is represented by the union `'name' | 'age'`, and the type `RemainingProps` represents the union of the remaining properties, which is only `id` in our example.

Now we can construct a new type that will contain just the `RemainingProperties` with the help of the mapped type `Pick` illustrated in the previous section in listing 5.24.

```
type RemainingProps = RemoveProps<Person, 'name' | 'age'>;
type PersonBlindAuditions = Pick<Person, RemainingProps>;
```

Figure 5.5 shows a screenshot taken while hovering the mouse pointer over the type `PersonBlindAuditions`. You can also see it in Playground at [bit.ly/2Wpr4d7](https://bit.ly/2Wpr4d7). The type `PersonBlindAuditions` is based on `Person` minus two properties.

```
class Person {
  id: number;
  name: string;
  age: number;
}

type RemoveProps<T, K> = Exclude<keyof T, K>;

type RemainingProps = RemoveProps<Person, 'name' | 'age'>;

type PersonBlindAuditions = {
  id: number;
}
type PersonBlindAuditions = Pick<Person, RemainingProps>;
```

**Figure 5.5 Combining Pick and Exclude**

You may say, wouldn't it be easier to create a separate type `PersonBlindAuditions` that has just the property `id`? This may be true in this simple case when the type `Person` has just three properties. But a person may be described by 30 properties and we may want to use it as a base class and create more descriptive conditional types based on it.

Actually, even with our 3-property class using the conditional types it can be beneficial. What if some time down the road a developer decides to replace the property `name` with `firstName` and `lastName` in the class `Person`? If you used the conditional type `PersonBlindAuditions`, its declaration will start giving you a compile error, and you'll fix it. But if you didn't declare `PersonBlindAuditions` as a conditional type but simply creating an independent class `PersonBlindAuditionsIndie` would require a developer who renames properties in `Person` to remember replicating the same changes in the class `PersonBlindAuditionsIndie`.

Besides, the type `RemoveProperties` is generic and you can use it to remove any properties

from any types.

### 5.3.1 The keyword infer

Our next challenge is to try to find the return type of a function and replace it with another one. Let's say we have an interface that declares some properties and methods, and we need to wrap each of the methods into a `Promise` so they run asynchronously. For simplicity, we'll consider an interface `SyncService`, which declares one property `baseUrl` and one method `getA()`.

```
interface SyncService {
    baseUrl: string;
    getA(): string;
}
```

There's nothing to do with the property `baseUrl` as is, but we want to *promisify* the method `getA()`. Here are the challenges:

1. How to differentiate properties from methods?
2. How to reach out to the original return type of the method before wrapping it into a `Promise`?
3. How to preserve the existing argument types of the methods?

Since we need to differentiate properties from methods, we'll use conditional types; mapped types will help us with modifying method signatures. Our goal is to create a type `Promisify` and apply it to `SyncService`. This way the implementation of `getA()` will have to return a `Promise`. The goal is to be able to write code as in listing 5.22.

#### **Listing 5.26 Promisifying synchronous methods**

```
class AsyncService implements Promisify<SyncService> { ①
    baseUrl: string; ②

    getA(): Promise<string> { ③
        return Promise.resolve('');
    }
}
```

- ① Mapping `SyncService` to `Promisify`
- ② No need to modify properties from `SyncService`
- ③ The original return type must be wrapped in `Promise`

We want to declare the new mapped type `Promisify` that will loop through all properties of a given type `T` and convert the signatures of its methods so they become asynchronous. The conversion will be done by a conditional type. The condition is that the type `U` (the supertype of `T`) has to be a function that can take any number of arguments of any types and may return any value:

```
T extends (...args: any[]) => any ?
```

After the question mark you provide the type if `T` is a function, and after the colon - if it's not a function.

The type `T` must be assignable to a type that looks like a function signature. If the above condition is true, we want to wrap the return of the function into a `Promise`. The problem is that if we'll use the type `any`, we'll lose the type information for function arguments as well as its return type.

Let's assume that a generic type `R` represents the return type of the function. Then we can use the keyword `infer` with this variable `R`.

```
T extends (...args: any[]) => infer R ?
```

By writing `infer R`, we instruct TypeScript to check the provided concrete return type (e.g. `string` for the method `getA()`) and replace `infer R` with this concrete type. Similarly, we can replace the type `any[]` in the function arguments with `infer A`:

```
T extends (...args: infer A) => infer R ?
```

Now we can declare our conditional type as follows:

```
type ReturnPromise<T> =
  T extends (...args: infer A) => infer R ? (...args: A) => Promise<R> : T;
```

This instructs TypeScript: "If a concrete type for `T` is a function, wrap its return type: `Promise<R>`. Otherwise, just preserve its type `T`." The conditional type `ReturnPromise<T>` can be applied to any type, and if we want to enumerate all properties of a class, interface et al, we can use the lookup type `keyof` to get a hold of all properties.

If you read the section on mapped types, the syntax of the next snippet should be familiar to you.

```
type Promisify<T> = {
  [P in keyof T]: ReturnPromise<T[P]>;
};
```

The mapped type `Promisify<T>` would iterate through properties of `T` and apply to them the conditional type `ReturnPromise`. In our example, we'll do `Promisify<SyncService>`, which won't do anything to the property `baseUrl`, but will change the return type of `getA()` to `Promise<string>`.

Figure 5.6 shows the entire script, which you can also see in the Playground at [bit.ly/2T51M7x](https://bit.ly/2T51M7x).

```

type ReturnPromise<T> =
  T extends (...args: infer A) => infer R ? (...args: A) => Promise<R> : T;

type Promisify<T> = {
  [P in keyof T]: ReturnPromise<T[P]>;
};

interface SyncService {
  baseUrl: string;
  getA(): string;
}

class AsyncService implements Promisify<SyncService> {
  baseUrl: string;

  getA(): Promise<string> {
    return Promise.resolve('');
  }
}

let service = new AsyncService();

let result: Promise<string> ←
let result = service.getA(); // hover over result: it's of type Promise<string>

```

**Figure 5.6 Combining conditional and mapped types**

## 5.4 Summary

In this chapter you learned:

- How to add metadata to a class or function
- How to create a type based on another one
- How to conditionally change a type

All these language features are not trivial for understanding, but they show the power of the language.

Decorators allow you to modify the type declaration or a behavior of the class, methods or parameters. Even if you won't write your own decorators, you need to understand their meaning if one of the frameworks (e.g. Angular) uses them.

Mapped types allow you to create apps that have a limited number of basic types and many derived types based on the basic ones.

Conditional types allow you to postpone making a decision of what type to use based on some condition. When you'll be reviewing the code of the blockchain app in chapter 10, you'll see the practical use of mapped and conditional types.

In the next chapter, we'll review the tools that help in debugging TypeScript in the browser and turn your TypeScript code into a runnable app.



# 6 Tooling

## ***This chapter covers***

- Debugging the TypeScript code with the help of source maps
- The role of linters
- Compiling and bundling TypeScript apps with Webpack
- Compiling TypeScript apps with Babel
- How to compile TypeScript with Babel and bundle with Webpack

TypeScript is one of the most loved languages. Yes, people love its syntax. But probably the main reason why it's being loved is tooling. TypeScript developers appreciate auto-completion, these squiggly lines indicating errors as you type, and refactoring offered by IDEs. And the best part is that most of these features are implemented by the TypeScript team and not by the IDE developers.

You'll see the same auto-complete and error messages in the online TypeScript Playground, in Visual Studio Code, or in WebStorm. When you install TypeScript, its bin directory includes two files: tsc and tsserver. The latter is the TypeScript Language Service that IDEs use to support these productivity features. When you type the TypeScript code, the IDEs communicate with tsserver that compiles the code in memory.

With the help of sourcemap files, you can debug the TypeScript code right in the browser. Linters allow you enforce the coding styles in your organization.

Type declaration files (.d.ts) allow the tsserver to offer context-sensitive help showing signatures of available functions or object properties. Thousands of publicly available type declaration files for popular JavaScript libraries allow you to be more productive even with the code that's not written in TypeScript.

All these conveniences add up and explain why people like TypeScript. But just having a great compiler is not enough for real-world projects, which consist of a diverse set of assets like JavaScript code, CSS, images, etc. We'll take a look at such essential tools for the modern web development like Webpack bundler and Babel. You'll also get a brief review of the emerging tools ncc and deno.

## 6.1 Source maps

The code written in TypeScript gets transpiled into the JavaScript, which is executed in a browser's or standalone JavaScript engine. To debug any program, you need to provide its source code to the debugger. But we have two versions of the source code: the executable code is in JavaScript, and the original one is in TypeScript. We'd like to debug the TypeScript code and source map files will allow us to do this.

Source map files have extensions .map, and they contain json-formatted data that map the corresponding code fragments in the generated JavaScript to the original language, which in our case is TypeScript. If you decide to debug a running JavaScript program that was written in TypeScript, have the browser download the source map files generated during the compilation, and you'll be able to place breakpoints in the TypeScript code even though the engine runs JavaScript.

Let's take a simple TypeScript program, and transpile it with the sourcemap generation option turned on. After that, we'll peek inside the generated source map. The source code of the file greeter.ts is shown in listing 6.1.

### Listing 6.1 greeter.ts

```
class Greeter {
    static sayHello(name: string) {
        console.log(`Hello ${name}`);
    }
}

Greeter.sayHello('John');
```

- ① Print the name on the console
- ② Invoke the method sayHello()

Let's transpile this file with generating the source map file:

```
tsc greeter.ts --sourceMap true
```

After the transpiling is complete, you'll see the files greeter.js and greeter.js.map. The latter is a source map file, and its fragment is shown in listing 6.2.

## Listing 6.2 The generated sourcemap file greeter.js.map

```
{
  "file": "greeter.js",
  "sources": ["greeter.ts"],
  "mappings": "AAAA;IAAA;IAMA,CAAC;IAJU,gBAAQ,...."
}
```

This file is not expected to be read by a human, but at least you see that it has the property `file` with the name of the generated JavaScript file and the property `sources` with the name of the source TypeScript file. The value in the `mapping` property contains mappings of the corresponding code fragments in the JavaScript and TypeScript files.

How the JavaScript engine would guess that the name of the file that contains the mapping is `greeter.js.map`? No guesses are needed. The TypeScript compiler would add the following line at the end of the generated `greeter.js`:

```
//# sourceMappingURL=greeter.js.map
```

The next step is to run our little greeter app in the browser and see if we can debug its TypeScript code. First, let's create an html file that loads `greeter.js`:

## Listing 6.3 index.html

```
<!DOCTYPE html>
<html>
  <body>
    <script src="greeter.js" /> ①
  </body>
</html>
```

- ① We load the JavaScript file here, not the TypeScript

Next, we need a WebServer that will serve the above html document to the browser. You can download and install a convenient npm package `live-server` as follows:

```
npm install -g live-server
```

Finally, start this server in the Terminal window from the directory where our greeter files are located as follows:

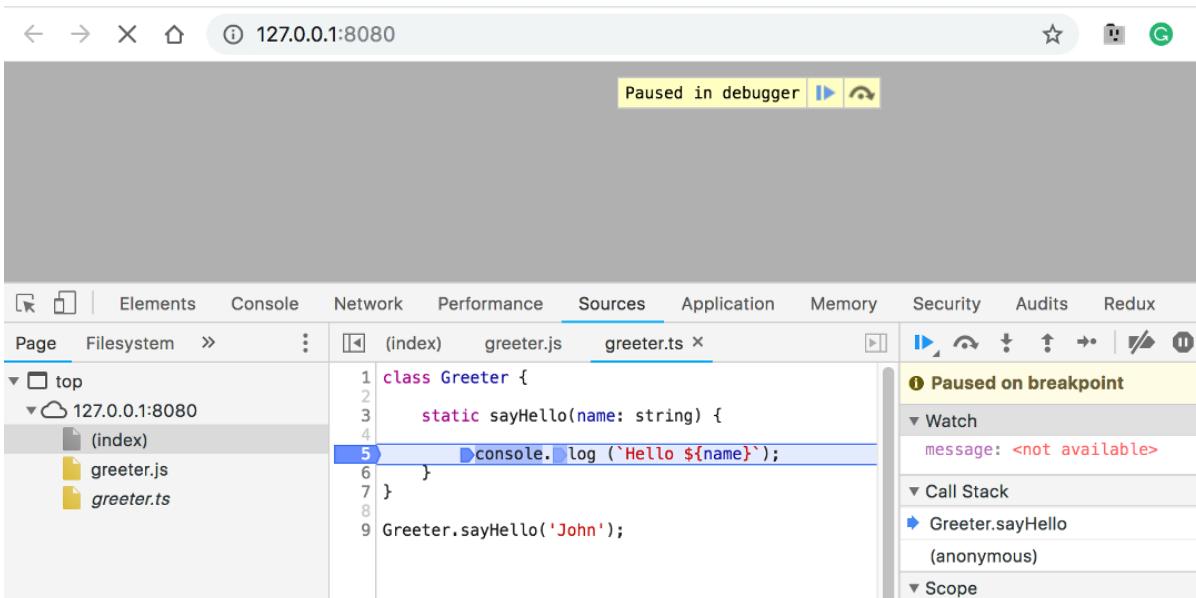
```
live-server
```

It'll open the Chrome browser at `localhost:8080` and will load the file `index.html` shown in listing 6.3. Open the Chrome Dev Tools in the tab Sources and select the file `greeter.ts`. In the source code, click to the left of the line 5 to place a breakpoint there. The Sources panel should look similar to the one in figure 6.1.



**Figure 6.1 Setting the breakpoint in the TypeScript code**

Refresh the page, and the execution of this code will stop at the breakpoint as seen in figure 6.2, and you'll be able to use familiar debugger's controls like step forward, step into, et al.



**Figure 6.2 The execution paused in the debugger**

#### NOTE

While each IDE comes with its own debugger, we prefer debugging the source code in Chrome Dev Tools. Moreover, you can even debug the code that runs as a standalone Node.js app in Chrome as well, and we explained how to do it in the sidebar "Debugging the Node.js code in the browser" at the end of the section 10.6.1 in chapter 10. This material also belongs to the TypeScript Tooling subject and could have been a part of this chapter, but we decided to cover debugging Node.js apps where we you could see its use in the blockchain app that has a Node.js server.

Now we'd like to show you the TypeScript compiler's option `--inlineSources`, which affects the process of generating source maps. With this option, the `.js.map` file will also include the TypeScript source code of your app. Try to compile the file `greeter.ts` as follows:

```
tsc greeter.ts --sourceMap true --inlineSources true
```

It's still produce the greeter.js as well as greeter.js.map but the latter will also include the code from the file greeter.ts. This mode eliminates the need to deploy separate .ts files under your web server, but you can still debug the TypeScript code.

**NOTE**

Deploying the js.map files in prod servers doesn't increase the size of the code downloaded by the browser. The browser downloads the sourcemap files only if the user opens the Dev Tools. You should not deploy the source maps in prod server only if you want to prevent the user from reading the TypeScript sources of your app.

## 6.2 The linter TSLint

Linters are the tools that check and enforce the coding style. For example, you may want to ensure that all string values are specified in single quotes or you may want to disallow unnecessary parentheses. Such restrictions are called rules and you configure them in text files.

JavaScript developers use several linters: JSLint, JSHint, and ESLint. TypeScript developers use the linter TSLint, which is an open source project maintained by Palantir (see [palantir.github.io/tslint](https://palantir.github.io/tslint)). There is a plan to merge the TSLint and ESLint to ensure the unified linting experience. You can read more about this effort in the blog titled "TSLint in 2019" at [medium.com/palantir/tslint-in-2019-1a144c2317a9](https://medium.com/palantir/tslint-in-2019-1a144c2317a9). After this effort is complete, JavaScript and TypeScript developers will use ESLint (see [eslint.org](https://eslint.org)).

Since many TypeScript teams continue using TSLint, we'll provide the basic introduction to this tool in this section. First of all, you need to install tslint in your project. Let's start from scratch. Create a new directory, open the Terminal window there and initialize a new npm project with the following command:

```
npm init -y
```

The `-y` option will silently accept all the default options while creating the package.json file there. Then install TypeScript and ts-lint there as follows:

```
npm install typescript tslint
```

This will create the directory `node_modules` and install TypeScript and tslint there. The tslint executable will be located in the directory `node_modules/.bin`. Now we want to create the configuration file `tslint.json` using the following command:

```
./node_modules/.bin/tslint --init
```

**TIP**

Starting from version 5.2, npm comes with npx command line that can run executables from node\_modules/.bin, e.g. `npx tslint --init`. You can read more about this useful command at [www.npmjs.com/package/npx](http://www.npmjs.com/package/npx).

This command will create the file `tslint.json` with the content shown in listing 6.4:

#### **Listing 6.4 The generated file `tslint.json`**

```
{
  "defaultSeverity": "error",
  "extends": [
    "tslint:recommended" ①
  ],
  "jsRules": {},
  "rules": {}, ②
  "rulesDirectory": [] ③
}
```

- ① Use the recommended rules
- ② Optional custom rules go here
- ③ An optional directory with custom rules

This configuration file states that `tslint` should extend the preset recommended rules, which you can find in the file `node_modules/tslint/lib/configs/recommended.js`. Figure 6.3 shows a snapshot taken from the file `recommended.js`.

```

117  "ordered-imports": {
118    options: {
119      "import-sources-order": "case-insensitive",
120      "module-source-path": "full",
121      "named-imports-order": "case-insensitive",
122    },
123  },
124  "prefer-const": true,
125  "prefer-for-of": true,
126  quotemark: {
127    options: ["double", "avoid-escape"],
128  },
129  radix: true,
130  semicolon: { options: ["always"] },
131  "space-before-function-paren": {
132    options: {
133      anonymous: "never",
134      asyncArrow: "always",
135      constructor: "never",
136      method: "never",
137      named: "never",
138    },
139  },
140  "trailing-comma": {
141    options: {
142      esSpecCompliant: true,
143      multiline: "always",
144      singleline: "never",
145    },
146  }

```

**Figure 6.3 A fragment from recommended.js**

The rule in lines 126-128 looks like this:

```

quotemark: {
  options: ["double", "avoid-escape"],
},

```

This enforces the use of double quotes around strings. The "avoid-escape" rule allows you to use the "other" quotemark in cases where escaping would normally be required. For double quotes, "other" would mean single quotes.

Figure 6.4 shows a screenshot taken from the WebStorm IDE; it shows a linting error on the first line. While using single quotes around strings is fine with the TypeScript compiler, the quotemark rule states that in this project you should use single quotes.

```

1 const customerName = 'Mary';
2
3 ~~~~~
4
5 const greeting = 'Hello "World"';
6

```

TSLint: ' should be " (quotemark) :

Figure 6.4 TSLint reports errors

**TIP**

Hover over the TSLint error and the IDE may offer you an auto-fix.

To avoid using escape characters for a string inside another string, in line 5 we surrounded World with double quotes, and the linter didn't complain because of the "avoid-escape" option.

**TIP**

To enable TSLint in VS Code, install its TSLint extension. Click on the Extension icon on the sidebar, and search for TSLint in the Marketplace. Make sure that VS Code uses the current version of TypeScript (it's shown at the bottom right corner in the status bar). Changing the TypeScript version is described in the VS Code documentation at [using-newer-typescript-versions](#).

In figure 6.4, you may notice a short squiggly line in the empty line 3. If you hover the mouse over this squiggly line, you'll see another linter's error: "TSLint: Consecutive blank lines are forbidden (no-consecutive-blank-lines)". The name of the rule is shown in parentheses here, and you can find the following rule in the file recommended.js:

```
"no-consecutive-blank-lines": true,
```

**TIP**

You can see the description of all available TSLint rules at [palantir.github.io/tslint/rules/](https://palantir.github.io/tslint/rules/)

Let's override it in the file tslint.json shown in listing 6.4. We'll add the rule there allowing consecutive blank lines as seen in listing 6.5.

### Listing 6.5 Overriding the rule in tslint.json

```
{
  "defaultSeverity": "error",
  "extends": [
    "tslint:recommended"
  ],
  "jsRules": {},
  "rules": {"no-consecutive-blank-lines": false}, ①
  "rulesDirectory": []
}
```

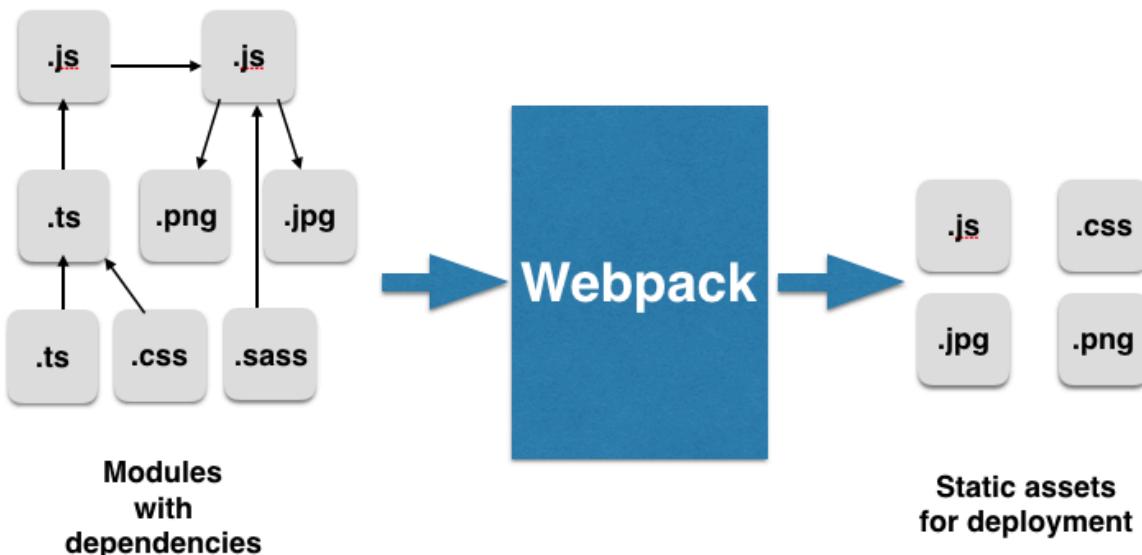
- ① We added this line

By assigning the value `false` to the `no-consecutive-blank-lines` rule we've overriden its value from the recommended.js file. Now that little squiggly line shown in figure 6.4 will disappear. You can override the recommended rules or add new ones that comply with your coding style or the style accepted in your project team.

## 6.3 Bundling code with Webpack

When a browser makes requests to the server, it gets HTML documents, which may include additional files like CSS, images, videos, and so on. In Part 2, we'll be working with a sample blockchain app that has multiple files. Pretty often, you'll be using one of the popular JavaScript frameworks, which could also add hundred files to your app.

If all these files of your app would be deployed separately, the browser would need to make hundreds of request to load it. The size of your app could be as large as several megabytes. Real-world applications consist of hundreds and even thousands of files, and we want to minimize, optimize, and bundle them together during deployment. Figure 6.5 shows how various files are given to Webpack, which produces smaller number of files for deployment.



**Figure 6.5 Bundling sources with Webpack**

Bundling multiple files into one can offer better performance and faster downloads. Technically, you can specify one output file in `tsconfig.json` and `tsc` can place the generated code of multiple `.ts` file into a single output file, but this only works if the `module` compiler's option is `System` or `AMD`; the output file won't include 3rd-party dependencies like JavaScript libraries.

Also, you want to be able to configure development and production builds differently. For prod, you will add the optimization and minimization, while in dev, you'll be just bundling files

together.

Several years ago, JavaScript developers were mostly using general task runners like Grunt and Gulp for orchestrating the build process. Today it's such bundlers as Webpack, Rollup, and Browserify. With the help of plugins, bundlers can also wear the hats of compilers if need be.

In this section, we'll introduce the Webpack bundler (see [github.com/webpack](https://github.com/webpack/webpack)) that was created for web applications running in a browser. Webpack supports many typical tasks required for preparing web application builds with minimal configuration.

You can install Webpack either globally or locally (in the `node_modules` directory of your project). Webpack also has a command line tool (CLI) `webpack-cli` (see [github.com/webpack/webpack-cli](https://github.com/webpack/webpack-cli)).

Prior to Webpack 4, properly configuring a Webpack project was not easy, but with `webpack-cli` it is. This scaffolding tool simplifies Webpack configuration for the projects that need it.

To install Webpack and its CLI globally on your computer, run the following command (`-g` is for global):

```
npm install webpack webpack-cli -g
```

**NOTE** Installing Webpack (an any other tool) globally allows you to use it with multiple projects. This is great, but in your organization the production build is done on a dedicated computer, and there could be restrictions as to what software can be installed globally. That's why, we'll use locally installed Webpack and Webpack CLI.

Webpack is probably the most popular bundler in the JavaScript ecosystem, and in the next section we'll bundle a simple JavaScript app.

### 6.3.1 Bundling JavaScript with Webpack

The goal of this section is to show you how to configure and run Webpack for a very basic JavaScript app, so you can see how this bundler works and what to expect as its output. The source code that comes with this chapter has several projects, and we'll start with the project called `webpack-javascript`.

This is an npm project, and you need to install its dependencies by running the following command:

```
npm install
```

In this project, we have a tiny JavaScript file that uses a 3rd-party library called chalk (see

[www.npmjs.com/package/chalk](https://www.npmjs.com/package/chalk)), which is listed in the dependencies section in package.json of this project shown in listing 6.6.

### Listing 6.6 package.json

```
{
  "name": "webpack-javascript",
  "description": "Code sample for chapter 6",
  "homepage": "https://www.manning.com/books/typescript-quickly",
  "license": "MIT",
  "scripts": {
    "bundleup": "webpack-cli"      ①
  },
  "dependencies": {
    "chalk": "^2.4.1"            ②
  },
  "devDependencies": {
    "webpack": "^4.28.3",        ③
    "webpack-cli": "^3.1.2"       ④
  }
}
```

- ① Defining a command to run Webpack
- ② The chalk library
- ③ The Webpack bundler
- ④ The Webpack command line interface

Webpack packages are located in the `devDependencies` section as we need them only on the developer's computer. We'll be running this app by entering the command `npm run bundleup`, which will run the executable `webpack-cli` located in `node_modules/.bin` directory.

The library `chalk` just paints the console output in different colors, but what this library does is irrelevant for our project. Our goal is to bundle our JavaScript code with some library. The code of `index.js` is shown in listing 6.7.

### Listing 6.7 webpack-javascript/src/index.js

```
const chalk = require('chalk');      ①
const message = 'Bundled by the Webpack';
console.log(chalk.black.bgGreenBright(message)); ②
```

- ① Load the library
- ② Use the library

Typically, developers use the configuration file `webpack.config.js` for creating custom configurations even though it's optional as of Webpack v4. This is a place where you configure the build for the project, and the file `webpack.config.js` of this project is shown in listing 6.8.

## Listing 6.8 webpack.config.js

```
const { resolve } = require('path');

module.exports = {
  entry: './src/index.js',      ①
  output: {
    filename: 'index.bundle.js', ②
    path: resolve(__dirname, 'dist') ③
  },
  target: 'node',               ④
  mode: 'production'           ⑤
};
```

- ① The source file name to bundle
- ② The name of the output bundle
- ③ The location of the output bundle
- ④ We'll run this app under Node.js; don't inline built-in Node.js modules
- ⑤ Optimize the file size of the output bundle

To create a bundle, Webpack needs to know the main module (the entry point) of your application, which may have dependencies on other modules or third-party libraries. Webpack loads the entry point module and builds a memory tree of all dependent modules if any.

In this config file, we use the Node's module `path` that can resolve the absolute path of the file with the help of the `{dbl\_}dirname` environment variable. Webpack runs under Node, and the `{dbl\_}dirname` variable will store the directory, where the executable JavaScript module (i.e. `webpack.config.js`) is located. So the fragment `resolve(__dirname, 'dist')` instructs Webpack to create a subdirectory named `dist` in the root of our project, and the bundled app will be located in the `dist` directory.

**TIP**

Storing the output files in a separate directory will allow you to configure your version control system to exclude the generated files. If you use Git, just add the `dist` directory to the `.gitignore` file.

We specified `production` as the value of the property `mode` to have Webpack minimize the size of the bundle.

Starting from Webpack 4, configuring project is much easier because of the introduction of the default modes `production` and `development`. In `production` mode, the size of the generated bundles is small, the code is optimized for runtime, and the development-only code is removed from the sources. In `development` mode, the compilation is incremental and you can debug the code in the browser.

`webpack-cli` allows you to bundle your providing the `entry`, `output` and other parameters right

on the command line. Besides, webpack-cli can generate a configuration file for your project.

Now we're ready to run Webpack to see how it'll bundle up our app. We'll be running the local version of Webpack installed in the `node_modules` directory of your project. In listing 6.6 we defined the npm command `bundleup` so we can run the locally installed webpack-cli:

```
npm run bundleup
```

The bundling will finish in several seconds and your console may look similar to figure 6.6.

```
$ npm run bundleup

> webpack-javascript@ bundleup /Users/yfain11/Documents/get_ts/code/getts/chapter6/webpack-javascript
> webpack-cli ← Running this executable

Hash: 80e7eecc06035e09db60
Version: webpack 4.28.3
Time: 213ms
Built at: 03/11/2019 7:32:02 AM
Asset           Size    Chunks      Chunk Names
index.bundle.js 22.4 KiB       0  [emitted]  main
Entrypoint main = index.bundle.js
[1] ./src/index.js 125 bytes {0} [built] ← The source code
[5] (webpack)/buildin/module.js 497 bytes {0} [built]
[10] external "os" 42 bytes {0} [built]
+ 10 hidden modules
```

**Figure 6.6 The console output of "npm run webpack"**

From this output we can see that Webpack built the bundle named `index.bundle.js`, and its size is about 22Kb. This is the main chunk (a.k.a. bundle). In our simple example we have the only one bundle, but larger apps are usually split into modules and Webpack can be configured to build multiple bundles.

Note that the size of the original `src/index.js` was only 125 bytes. The size of the bundle is much larger because it includes not only the three lines of our `index.js` but also the chalk library and all its transitive dependencies. Webpack also adds its own code to keep track of the bundle content.

Change the value of the property `mode` in `webpack.config.js` to be `development` and re-run the bundling. Under the hood, Webpack will apply different pre-defined configuration settings, and the size of the generated file will be more than 56Kb vs 22Kb in the production mode.

Open the file `dist/index.bundle.js` in any plain text editor. In the production version, you'll just see one very long optimized line, whereas the development bundle will have the readable content with comments.

The generated file `index.bundle.js` is a regular JavaScript file and you can use it in the `HTML` tag

<script> or any other place where the JavaScript file names are allowed. This particular app is not intended to run in the browser - it's for Node.js, and you can run it with the following command:

```
node dist/index.bundle.js
```

Figure 6.7 shows the console output, where the chalk library displayed the message "Bundled by the Webpack" on a bright-green background.

```
$ node dist/index.bundle.js
Bundled by the Webpack
```

**Figure 6.7 Running your first bundle**

We ran this example using the Node.js runtime, but for web apps, Webpack offers a dev server that can serve your web pages . It has to be installed separately:

```
npm install webpack-dev-server -D
```

After that, add the start npm command in the `scripts` section in `package.json`, and it'll look like this:

```
"scripts": {
  "bundleup": "webpack-cli --watch", ①
  "start": "webpack-dev-server"
}
```

- ① Run webpack in the watch mode so it rebuilds the bundle on code changes

Now let's create a super simple JavaScript file `index.js`:

```
document.write('Hello World!');
```

We'll ask Webpack to generate a bundle of this file and save it in `dist/bundle.js`. The `webpack.config.js` will be similar to the one shown in listing 6.8 with two changes: we'll add the property `devServer` and change the mode to development as seen in listing 6.9.

## Listing 6.9 Adding the devServer property

```
const { resolve } = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'index.bundle.js',
    path: resolve(__dirname, 'dist')
  },
  target: 'node',
  mode: 'development',      ①
  devServer: {               ②
    contentBase: '..'
  }
};
```

- ① Configuring Webpack for the dev mode
- ② Adding a section for webpack-dev-server

In `devServer`, you can configure any options that webpack-dev-server allows on the command line (see the Webpack product documentation at [mng.bz/gn4r](#)). We just use `contentBase` to specify that the files should be served from the current directory.

Accordingly, the HTML document will be referring `index.bundle.js` as in listing 6.10:

## Listing 6.10 index.html

```
<!DOCTYPE html>
<html>
<body>
  <script src="dist/index.bundle.js"></script>
</body>
</html>
```

We're ready to build the bundle and start the web server. Building the bundle goes first:

```
npm run bundleup
```

To start the server using webpack-dev-server, just run the following command:

```
npm start
```

You can find this app in the directory `webpack-devserver`. Open your browser at `localhost:8080` and it'll greet the word as seen in figure 6.8.



## Hello World!

**Figure 6.8** The browser renders index.html

When you serve your application with webpack-dev-server, it'll run on the default port 8080. Since we started Webpack in the watch mode, it'll recompile the bundle(s) each time you modify the code.

Now that you've seen how to bundle a pure JavaScript project, let's see how to use Webpack with the TypeScript code.

### 6.3.2 Bundling TypeScript with Webpack

The source code that comes with this chapter has several projects, and in this section, we'll work with the one located in the webpack-typescript directory. This project is almost the same as reviewed in the previous section. It includes a three-line TypeScript file index.ts (in the previous section it was index.js) that uses the same JavaScript library chalk.

Let's highlight the differences starting from the file index.ts shown in listing 6.11.

#### Listing 6.11 webpack-typescript/src/index.ts

```
import chalk from 'chalk';      ①
const message: string = 'Bundled by the Webpack';  ②
console.log(chalk.black.bgGreenBright(message));  ③
```

- ① Import the default object to access this library
- ② The type of the message variable is declared explicitly
- ③ Use the library

The library chalk explicitly exposes a default export. That's why instead of writing `import * as chalk from 'chalk'` we wrote `import chalk from chalk`.

We didn't have to explicitly declare the type of `message` but we wanted to make it obvious that this is a TypeScript code.

In this project, the file package.json has two additional lines (compared to listing 6.7). We've added `ts-loader` and `typescript` there as seen in listing 6.12.

## Listing 6.12 The devDependencies section in package.json

```
"devDependencies": {
    "ts-loader": "^5.3.2",      ①
    "typescript": "^3.2.2",     ②
    "webpack": "^4.28.3",
    "webpack-cli": "^3.1.2"
}
```

- ① Added the TypeScript loader
- ② Added the TypeScript compiler

Usually build-automation tools provide developers with a way to specify additional tasks that need to be performed during the build process, and Webpack offers *loaders* and *plugins* that allow customizing builds. Any Webpack loader pre-processes one file at a time while plugins can operate on a group of files.

**TIP**

You can find the “list of loaders” in the Webpack docs on GitHub at [mng.bz/U0Yv](https://mng.bz/U0Yv).

Webpack loaders are transformers that take a source file as input and produce another file as output (in memory or on disk). For example, the `json-loader` takes an input file and parses it as JSON. For transpiling TypeScript to JavaScript, the `ts-loader` uses TypeScript compiler, which explains why we added it to the `devDependencies` section in `package.json`. TypeScript compiler uses `tsconfig.json`, which has the following content:

## Listing 6.13 tsconfig.json

```
{
  "compilerOptions": {
    "target": "es2018",          ①
    "moduleResolution": "node"   ②
  }
}
```

- ① Transpile into the JavaScript syntax described in the ECMAScript 2018 spec
- ② Use the Node.js module resolution

The `moduleResolution` option tells tsc how to resolve modules if the code includes import statements. If your app includes a statement `import { a } from "moduleA"`, tsc needs to know where to look for the `moduleA`.

There are two strategies for module resolution: `Classic` and `Node`. For example, in case of `Classic`, tsc can look for the definition of `moduleA` in files `moduleA.ts` and in the type definition file `moduleA.d.ts`.

In case of the `Node` strategy, module resolution will also try to find the module in the files located in the directory `node_modules`, which is exactly what we need because the third-party library `chalk` is installed in `node_modules`.

**TIP**

To read more about module resolutions, visit the online documentation at [www.typescriptlang.org/docs/handbook/module-resolution.html](https://www.typescriptlang.org/docs/handbook/module-resolution.html).

Listing 6.14 shows how we added the `ts-loader` in the `rules` section of `webpack.config.js`.

### **Listing 6.14 webpack.config.js**

```
const { resolve } = require('path');

module.exports = {
  entry: './src/index.ts',
  output: {
    filename: 'index.bundle.js',
    path: resolve(__dirname, 'dist')
  },
  module: {
    rules: [               ①
      {
        test: /\.ts$/,
        exclude: /node_modules/,  ②
        use: 'ts-loader'          ③
      }
    ],
    resolve: {             ④
      extensions: [ '.ts', '.js' ]  ⑤
    },
    target: 'node',
    mode: 'production'
  };
}
```

- ① Rules for modules (configure loaders, parser options, etc.)
- ② Apply to files with the `.ts` extension
- ③ Ignore the `.ts` files from `node_modules` directory
- ④ Transpile TypeScript using options from the existing `tsconfig.json`
- ⑤ Add the `.ts` extension to the `resolve` property to be able to import your TypeScript files

In short, we say to Webpack, "If you see a file with the name extension `.ts`, use `ts-loader` to pre-process them. Ignore the `.ts` files located under the `node_modules` directory - there's no need to compile them."

In our simple example, the array `rules` has just one loader configured, i.e. `ts-loader`. In real-world project, it usually includes multiple loaders. For example, `css-loader` is used for

processing CSS files. `file-loader` resolves `import/require()` on a file into the URL and emits it into the output bundle; it's used for handling images or other files having specified name extensions.

**TIP**

There's an alternative Webpack loader for TypeScript called `awesome-typescript-loader` (see [github.com/s-panferov/awesome-typescript-loader](https://github.com/s-panferov/awesome-typescript-loader)), which may show better performance on large projects.

Now you can build the bundle `index.bundle.js` the same way as we did it in the previous section by running the following command:

```
npm run bundleup
```

To make sure that the bundled code works, just run it:

```
node dist/index.bundle.js
```

## SIDE BAR What Webpack plugins are for

If Webpack loaders transform files one at a time, plugins have access to all files, and they can process them before or after the loaders kick in. You can find the list of available Webpack plugins at [webpack.js.org/plugins](https://webpack.js.org/plugins). For example, the `SplitChunksPlugin` plugin allows you to break the bundle into separate chunks.

Say your app code is split into two modules `main` and `admin` and you want to build two corresponding bundles. Each of these modules uses some framework, e.g. Angular. If you just specify two entry points (`main` and `admin`), each bundle would include the application code as well as its own copy of Angular's code.

To prevent this from happening, you can process the code with the `SplitChunkPlugin`. With this plugin, Webpack won't include any of the Angular code in the `main` and `admin` bundles; it will create a separate shareable bundle with the Angular code only. This will lower the total size of your application because it includes only one copy of Angular shared between two application modules. In this case, your HTML file should include the vendor bundle first, followed by the application bundle.

The `UglifyJSPlugin` plugin performs code minification of all transpiled files. It's a wrapper for the popular UglifyJS minifier, which takes the JavaScript code and performs various optimizations. For example, it compresses the code by joining consecutive `var` statements, removes unused variables and unreachable code, and optimizes if-statements. Its mangle tool renames local variables to single letters.

The `TerserWebpackPlugin` also performs code minification using `terser` - a special JavaScript parser, mangle, optimizer and beautifier toolkit for ES6.

Using the option `mode: "production"` in the `webpack.config.js` file you implicitly engage a number of Webpack plugins that will optimize and minimize your code bundles. If you're interested which specific plugins are being used in the production mode, see [webpack.js.org/concepts/mode/#mode-production](https://webpack.js.org/concepts/mode/#mode-production).

The configuration file presented in listing 6.14 is rather small and simple, but in real-world projects, the file `webpack.config.js` can be complex and include multiple loaders and plugins. While we just used the TypeScript loader in our tiny app, most likely, you'll be also using loaders for HTML, CSS, images and others.

Your project may have multiple `entry` files and you might want to use special plugins to create bundles in a special way. For example, if your app will be deployed as 10 bundles, Webpack can extract the common code (from the framework you use) into a separate bundle so nine others won't duplicate it.

As the complexity of the bundling process increases, the JavaScript file `webpack.config.js` grows too, and it becomes more difficult to write and maintain. Providing a value of a wrong type can result in errors during the bundling process, and the error description may not be easy to understand. The good news is that you can write the Webpack configuration file in TypeScript (e.g. `webpack.config.ts`), getting all the help of the static type analyzer as with any other TypeScript code. You can read about using TypeScript for configuring Webpack in the product documentation at [webpack.js.org/configuration/configuration-languages](https://webpack.js.org/configuration/configuration-languages).

In this section, we presented a project where the Typescript code uses a JavaScript library chalk. In chapter 7, we'll provide a more detail discussion of having mixed TypeScript-JavaScript projects. Meanwhile let's see how to use TypeScript with another popular tool called Babel.

## 6.4 Using the Babel transpiler

Babel is a popular JavaScript transpiler, which works as a remedy for a well known issue: not every browser support every language feature declared in ECMAScript. If you are developing a new Web app, you want to test it against all the browsers that your users may have.

We're not even talking about the full implementation of a specific ECMAScript version. At any given time, one browser may implement a specific subset of ECMAScript 2018, while another still understands only ECMAScript 5. Visit the site [caniuse.com](https://caniuse.com) and search for "arrow functions". You'll see that Internet Explorer 11, Opera Mini, and some others do not support them.

Babel allows writing in modern JavaScript and transpile it down to the older syntax. While `tsc` allows you to specify a particular ECMAScript spec as a target for transpiling (e.g. ES2018), Babel is more fine-grained; it allows you to selectively pick the language features that should be transformed to JavaScript supported by older browsers. Figure 6.9 shows a fragment of the browser compatibility table (see [kangax.github.io/compat-table/es2016plus](https://kangax.github.io/compat-table/es2016plus)).

Feature name	Compilers/polyfills														
	95%	5%	49%	52%	61%	47%	48%	56%	9%	1%	44%	49%	49%	66%	81%
• <a href="#">Asynchronous Iterators</a>	Current browser	Traceur + core-js 2	Babel 6+ + core-js 2	Babel 7+ + core-js 3	Babel 7+ + core-js 3	Closure 2019.03	TypeScript + core-js 2	TypeScript + core-js 3	es7-shim	IE 11	Edge 17	Edge 18	Edge 19 Preview	FF 60 ESR	FF 65
• <a href="#">template literal revision</a>	2/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2
2018 misc															
• <a href="#">Object.fromEntries</a>	Yes	No	No	No	Yes	No	No	No	No	No	No	No	No	Yes	Yes
2019 features															
• <a href="#">Object.fromEntries</a>	No	No	No	Yes <sup>[5]</sup>	No	No <sup>[26]</sup>	Yes <sup>[6]</sup>	No	No	No	No	No	No	No	Yes
• <a href="#">string.trimming</a>	4/4	0/4	4/4	4/4	0/4	4/4	4/4	2/4	0/4	2/4	2/4	2/4	2/4	2/4	4/4
• <a href="#">Array.prototype.flatMap<sup>[28]</sup></a>	2/3	0/3	1/3	1/3	3/3	2/3	1/3	3/3	0/3	0/3	0/3	0/3	0/3	0/3	2/3
2019 misc															
• <a href="#">optionalCatchBinding</a>	3/3	0/3	0/3	3/3	3/3	3/3	3/3	0/3	0/3	0/3	0/3	0/3	0/3	3/3	3/3
• <a href="#">Symbol.prototype.description</a>	3/3	0/3	0/3	3/3	2/3	0/3	3/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3	3/3
• <a href="#">Function.prototype.toString.revision</a>	7/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	1/7	4/7	4/7	4/7	7/7	7/7
• <a href="#">JSON.superset</a>	2/2	0/2	0/2	0/2	2/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2
• <a href="#">Well-formed JSON.stringify</a>	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	Yes

Figure 6.9 A fragment of the browser compatibility table

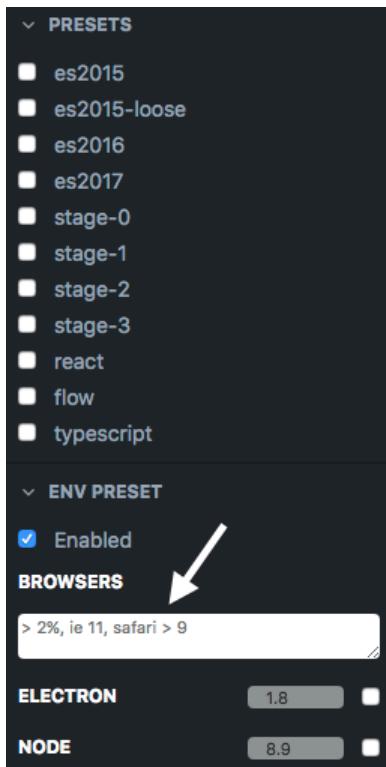
On top, you see the names of the browsers and compilers. On the left, a list of features. A browser, compiler, or a server's runtime may fully or partially support some of the features, and Babel plugins allow you to specify that only certain features should be transformed into the older code. The complete list of plugins is available at [babeljs.io/docs/en/plugins](https://babeljs.io/docs/en/plugins).

For the sake of discussion, we picked a feature "string trimming" from the ES2019 spec (see the black arrow on the left). Let's say our app needs to work in the browser Edge. Follow the vertical arrow and you'll see that Edge 18 only partially (2/4) implements string trimming at this time.

We can use the string trimming feature in our code, but we'll need to ask Babel to transpile this feature into the older syntax. Sometime later, when Edge will fully support this feature, and no transpiling would be needed, and Babel is flexible enough to help you with this.

Babel consists of many plugins, and each one is used for transpiling a particular feature of the language, but trying to find and map features to plugins would be a time-consuming task. That's why Babel plugins are combined into *presets*, which are lists of plugins that you want to apply for transpiling. In particular, `preset-env` allows you to specify the ECMAScript features and the browsers that your app should support.

In section A12 in Appendix A, we included a screenshot from [babeljs.io](https://babeljs.io) illustrating its REPL tool. Let's take another look at Babel's "Try it out" menu shown in figure 6.10 concentrating on the left navigation bar that allows you to configure presets.



**Figure 6.10 Configuring the ENV Preset**

Each preset is just a group of plugins, and if you want to transpile the code into the ES2015 syntax, just select the es2015 checkbox. Instead of using ECMAScript spec names, you can configure specific versions of the browsers or other runtimes using the ENV PRESET option. The white arrow shows the editable box with the suggested values for the ENV preset: 2%, ie 11, safari 9. This means that you want Babel to transpile the code so it'll run in all browsers with the market share of 2% or more, and also in Internet Explorer 11 and Safari 9.

Neither IE 11 nor Safari 9 support arrow functions, and if you enter `(a,b) => a + b;`, Babel will transform it to JavaScript that these browsers understand as seen in figure 6.11 on the right.

```

BABEL
Docs Setup Try it out Videos Blog Search Do
es2015-100%
es2016
es2017
stage-0
stage-1
stage-2
stage-3
react
flow
typescript
ENV PRESET
Enabled
BROWSERS
> 2%, ie 11, safari > 9
<
1 (a,b) => a + b;
1 "use strict";
2
3 (function (a, b) {
4   return a + b;
5 });

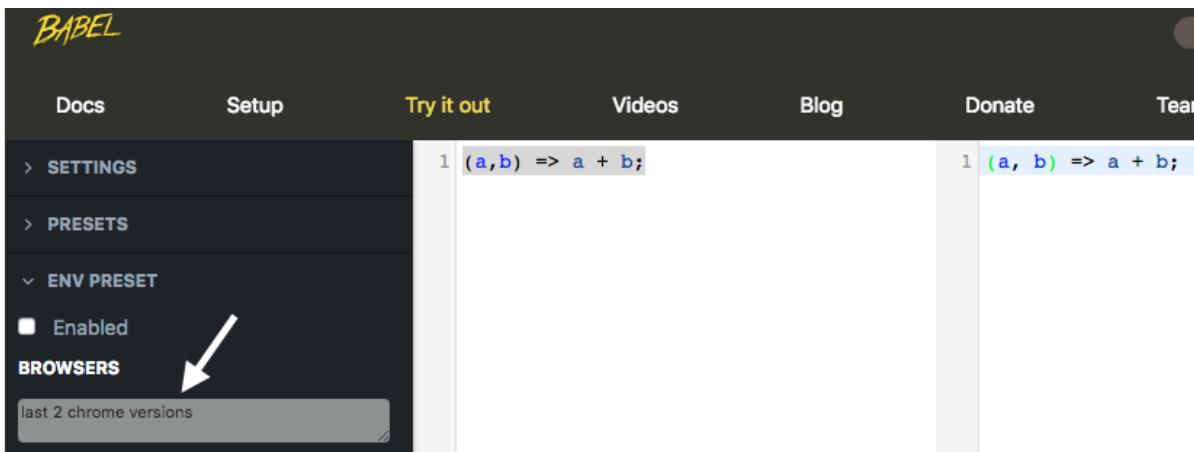
```

Figure 6.11 Applying the ie and safari presets

**TIP**

If you see errors after entering the names of the browsers, uncheck the Enabled checkbox after entering the browsers and versions. This seems like a bug but it may be fixed by the time you read this.

Now let's change the preset to be "last 2 chrome versions". Babel is smart enough to understand that the last two versions of Chrome support arrow functions and there's no need to do any transformation as seen in figure 6.12.



**Figure 6.12 Applying the Chrome preset**

The ENV preset comes with the browsers list, and you need to use proper names and phrases (e.g. "last 2 major versions", "Firefox >= 20", or "> 5% in US") to specify the constraints. These phrases are listed in the `browserslist` project available at [github.com/browserslist/browserslist](https://github.com/browserslist/browserslist).

**NOTE**

We used ENV preset in the Babel REPL to play with target environments, but these options can be configured and used in Babel that runs from a command line. In the next section in listing 6.15, we'll add `@babel/preset-env` to the configuration file `.babelrc`. In listing 6.17, you'll see the file `.browserslistrc`, where you can configure specific browsers and versions like we did in the Babel REPL. You can read more about `preset-env` in the product documentation at [babeljs.io/docs/en/next/babel-preset-env.html](https://babeljs.io/docs/en/next/babel-preset-env.html).

Babel can be used to transpile such programming languages as the modern JavaScript, TypeScript, CoffeeScript, Flow, and more. For example, the React framework uses the JSX syntax, which is not even a standard JavaScript, and Babel understands JSX too. In chapter 12, we'll use Babel with a React app.

When Babel transpiles TypeScript, unlike `tsc`, it doesn't perform type checking, and only transpiles the TypeScript syntax to JavaScript. Babel creators didn't implement the full-featured TypeScript compiler. Babel just parses the TypeScript code and generates the corresponding JavaScript syntax.

You may be wondering, "We're pretty happy with the TypeScript transpiler. Why include the section on the JavaScript-to-JavaScript transpiler in the TypeScript book?"

The reason is that you may join the project where some of the modules were written in JavaScript and some in TypeScript. In such projects, the chances are that Babel is already a part of the development-deployment workflow. For example, Babel is popular among developers who use React framework, which only recently started supporting TypeScript.

As any npm package, you can install Babel either locally or globally (with the `-g` option). Installing it locally within the project directory makes your project self-sufficient because, after running `npm install`, you can use Babel without expecting that the computer has it installed elsewhere (someone may work with your project using different computer).

```
npm install @babel/core @babel/cli @babel/preset-env
```

Here, `@babel/core` is Babel transpiler, `@babel/cli` the command line interface, and `@babel/preset-env` is the ENV preset discussed earlier.

#### **NOTE**

At the [npmjs.org](https://npmjs.org) registry, JavaScript packages could be organized as *organizations*. For example, `@babel` is the organization for Babel-related packages. `@angular` is the organization for packages that belong to Angular framework. `@types` is the place for TypeScript type definition files for various popular JavaScript libraries.

In the following sections, we'll introduce you to three small projects. The first one will use Babel with JavaScript, the second - Babel with TypeScript, and the third - Babel, TypeScript and Webpack.

### **6.4.1 Using Babel with JavaScript**

In this section we'll review a simple project that uses Babel with JavaScript, and it's located in the directory `babel-javascript`. We'll continue working with this three-line script introduced in listing 6.7 that uses the JavaScript library `chalk`. The only change in listing 6.15 is that the message reads "Transpiled with Babel" now.

#### **Listing 6.15 babel-javascript/src/index.js**

```
const chalk = require('chalk');
const message = 'Transpiled with Babel';
console.log(chalk.black.bgGreenBright(message));
```

Listing 6.16 shows the npm script that we'll use to run Babel and dependencies that should be installed on the developer's machine.

## Listing 6.16 A fragment from babel-javascript/package.json

```

"scripts": {
  "babel": "babel src -d dist"      ①
},
"dependencies": {
  "chalk": "^2.4.1"
},
"devDependencies": {                ②
  "@babel/cli": "^7.2.3",
  "@babel/core": "^7.2.2",
  "@babel/preset-env": "^7.2.3"
}

```

- ① The npm script to transpile the code from src to dist
- ② Locally installed dev dependencies

Babel is configured in the file .babelrc, and our configuration file will be very simple. We just want to use preset-env for transpiling.

## Listing 6.17 The file .babelrc

```
{
  "presets": [
    "@babel/preset-env"
  ]
}
```

We didn't configure any specific browsers' versions here, and without any configuration options, @babel/preset-env behaves exactly the same as @babel/preset-es2015, @babel/preset-es2016 and @babel/preset-es2017. In other words, all language features introduced in ECMAScript 2015, 2016, and 2017 will be transpiled to ES5.

**TIP**

We configure Babel in the file named .babelrc, which is fine for static configurations like ours. If your project would need to create Babel configurations programmatically, you'd need to use the file babel.config.js (see [babeljs.io/docs/en/config-files#project-wide-configuration](https://babeljs.io/docs/en/config-files#project-wide-configuration) for details). If you'd like to see how Babel transpiles our file src/index.js, install the dependencies of this project by running `npm install`, and then run the npm script from package.json:

```
npm run babel
```

Listing 6.18 shows the transpiled version of index.js that is created in the dist directory, and it'll have the following content (compare with listing 6.13):

**Listing 6.18 dist/index.js**

```
"use strict";
var chalk = require('chalk');①
var message = 'Transpiled with Babel';②
console.log(chalk.black.bgGreenBright(message));
```

- ① Babel added this line
- ② Babel replaced const with var

**NOTE**

The transpiled file still invokes `require('chalk')` and this library is located in a separate file. Keep in mind that Babel is not a bundler, and we'll use Webpack with Babel in section 6.4.3.

You can run the transpiled version as follows:

```
node dist/index.js
```

The console output will look similar to figure 6.13.

```
$ node dist/index.js
Transpiled with Babel
```

**Figure 6.13** Running the program transpiled by Babel

If we wanted to ensure that Babel generates the code that works in specific browsers' versions, we'd need to add an additional config file `.browserslistrc`. For example, let's imagine that we want to ensure that our code work only in the two latest versions of Chrome and Firefox. Then we could create the file in the root of our project:

**Listing 6.19 A sample file .browserslistrc**

```
last 2 chrome versions
last 2 firefox versions
```

Now, running Babel won't convert `const` to `var` as in listing 6.16 because both Firefox and Chrome support the `const` keyword for a while. Try it out and see for yourself.

### 6.4.2 Using Babel with TypeScript

In this section we'll review a simple project that uses Babel with TypeScript, and it's located in the directory `babel-typescript`. We'll continue working with this three-line script introduced in listing 6.9 that uses the JavaScript library `chalk`. The only change is that the message reads "Transpiled with Babel" now.

## Listing 6.20 babel-javascript/src/index.ts

```
import chalk from 'chalk';
const message: string = 'Transpiled with Babel';
console.log(chalk.black.bgGreenBright(message));
```

Compared to the package.json from pure JavaScript project (see listing 6.14), our TypeScript project adds the dev dependency `preset-typescript` that strips the TypeScript types from the code, so Babel can treat it as plain JavaScript. We'll also add an option to read .ts files to the npm script that runs Babel as in listing 6.21.

## Listing 6.21 A fragment from package.json

```
"scripts": {
  "babel": "babel src -d dist --extensions '.ts'"      ①
},
"dependencies": {
  "chalk": "^2.4.1"
},
"devDependencies": {
  "@babel/cli": "^7.2.3",
  "@babel/core": "^7.2.2",
  "@babel/preset-env": "^7.2.3",
  "@babel/preset-typescript": "^7.1.0"                  ②
}
```

- ① Instruct Babel to process files with the .ts extensions
- ② Add the dependency `preset-typescript`

Typically, presets include a number of plugins, but `preset-typescript` includes just one: `@babel/plugin-transform-typescript`, which in turn uses `@babel/plugin-syntax-typescript` to parse TypeScript and `@babel/helper-plugin-utils` with general utilities for plugins.

While `@babel/plugin-transform-typescript` turns the TypeScript code into the ES.Next syntax, it's not a TypeScript compiler. As strange as it sounds, Babel simply erases TypeScript. For example, it'll turn `const x: number = 0` into `const x = 0`. `@babel/plugin-transform-typescript` is a lot faster than TypeScript compiler because it does not type-check input files.

### NOTE

`@babel/plugin-transform-typescript` has several minor limitations listed at [babeljs.io/docs/en/babel-plugin-transform-typescript](https://babeljs.io/docs/en/babel-plugin-transform-typescript) (e.g. it doesn't support `const enum`). For better TypeScript support consider using plugins `@babel/plugin-proposal-class-properties` and `@babel/plugin-proposal-object-rest-spread`.

After reading the first five chapters of this book, you started to like type checking and

compile-time errors that real TypeScript compiler offers. And now the authors suggest using Babel to simply erase the TypeScript-related syntax?

Not really. During development, you can continue using tsc (with tsconfig.json) and an IDE with full TypeScript support. At the deployment stage, you may still introduce Babel and its ENV preset (you already started to like the flexibility in configuring target browsers offered by the ENV preset, aren't you?).

In your build process, you can even add an npm script (in package.json) that runs tsc:

```
"check_types": "tsc --noEmit src/index.ts"
```

Now you can run sequentially `check_types` and `babel` assuming you have tsc locally installed:

```
npm run check_types && npm run babel
```

The option `--noEmit` is to ensure that tsc won't output any files (i.e. `index.js`) because this will be done by the `babel` command that runs right after `check_types`. If there are compile errors in `index.ts`, the build process will fail and the `babel` command won't even run.

**TIP**

If you use `&&` (double ampersand) between two npm scripts, they run sequentially. Use `&` (single ampersand) for parallel execution.

In this project, the config file `.babelrc` includes `@babel/preset-typescript`:

### Listing 6.22 The file `.babelrc`

```
{
  "presets": [
    "@babel/preset-env",
    "@babel/preset-typescript"
  ]
}
```

Comparing with the babel-javascript project, we made the following TypeScript-related changes:

1. Added the option `--extensions '.ts'` to the command that runs Babel
2. Added TypeScript-related dev dependencies to `package.json`
3. Added `"@babel/preset-typescript"` to the config file `.babelrc`

TIP: To transpile our simple script `index.ts`, run the following npm script from `package.json`:

```
npm run babel
```

You'll find the transpiled version of `index.js` in the `dist` directory. You can run the transpiled code the same way as we did in the previous section:

```
node dist/index.js
```

Now let's add Webpack to our workflow to bundle together our script index.js and the JavaScript library chalk.

### 6.4.3 Using Babel with TypeScript and Webpack

Babel is a transpiler, but it's not a bundler, which is required for any real-world app. There are different bundlers to choose from (e.g. Webpack, Rollup, Browserify), but we'll stick to Webpack. In this section we'll review a simple project that uses Babel with TypeScript and Webpack, and it's located in the directory webpack-babel-typescript.

In section 6.3.2, we reviewed the TypeScript-Webpack setup, and we'll continue using our three-line source code from that project as seen in listing 6.23.

#### **Listing 6.23 webpack-babel-typescript/src/index.ts**

```
import chalk from 'chalk';
const message: string = 'Built with Babel bundled with Webpack';
console.log(chalk.black.bgGreenBright(message));
```

The devDependency section from package.json is shown in listing 6.24.

#### **Listing 6.24 The devDependencies section in package.json**

```
"devDependencies": {
  "@babel/core": "^7.2.2",
  "@babel/preset-env": "^7.2.3",
  "@babel/preset-typescript": "^7.1.0",
  "babel-loader": "8.0.5", ①
  "webpack": "^4.28.3",
  "webpack-cli": "^3.1.2"
}
```

- ① Adding the Webpack Babel loader

Compare Babel dependencies in listings 6.24 and 6.21. There are two changes in 6.24:

1. We added `babel-loader`, which is a Webpack loader for Babel
2. We removed `babel-cli` because we won't be running Babel from the command line because Webpack will use the `babel-loader` as a part of the bundling process.

As you remember from section 6.3, Webpack uses the configuration file `webpack.config.js`. While configuring TypeScript with Webpack, we used the `ts-loader` as seen in listing 6.14. This time, we want `babel-loader` to handle the files with extension, `.ts`. Listing 6.25 shows the Babel-related section from `webpack.config.js`.

### Listing 6.25 A fragment from webpack-babel-typescript/webpack.config.js

```
module: {
  rules: [
    {
      test: /\.ts$/,          ①
      exclude: /node_modules/, ②
      use: 'babel-loader'
    }
  ],
},
```

- ① Apply this rule for files ending with .ts
- ② Process .ts files with babel-loader

The file `.babelrc` will look exactly the same as in previous section as shown in listing 6.20.

After installing dependencies with `npm install`, we're ready to build the bundle running the `bundleup` command from `package.json`:

```
npm run bundleup
```

This command will build `index.bundle.js` in the `dist` directory. This file will contain the transpiled (by Babel) version of our `index.ts` plus the code from the JavaScript library `chalk`. You can run this bundle as usual:

```
node dist/index.bundle.js
```

The output will look familiar as well:

```
$ node dist/index.bundle.js
Built with Babel bundled with Webpack
```

**Figure 6.14** Running the program transpiled by Babel

The main message from this section is that you don't have to select either Babel or `tsc` for generating JavaScript. They can live happily together in the same project.

#### NOTE

People who don't like TypeScript often use this argument: "If I just write in plain JavaScript, I wouldn't need to use a compiler. I can run my JavaScript program as soon as it's written." This statement is plain wrong. Unless you're ready to ignore the new JavaScript syntax introduced since 2015, you'd need to come up with a process that would compile down the code written in modern JavaScript to the code that each browser understands. Most likely, you'll introduce a transpiler in your project anyway being that Babel, TypeScript, or something else.

## 6.5 Tools to watch

In this section, we'd like to mention a couple of tools that were not officially released at the time of this writing, but could become useful additions to the toolbox of a TypeScript developer.

### 6.5.1 Deno

Every JavaScript developer knows about the Node.js runtime. We also use it in the book for running apps outside the browsers. Everybody likes Node.js... except its original creator Ryan Dahl. In 2018, he delivered a presentation titled "10 Things I Regret About Node.js" (see [youtu.be/M3BM9TB-8yA](https://youtu.be/M3BM9TB-8yA)) and started working on Deno, a secure run-time environment that's built on top of the V8 engine (just like Node) and has a built-in TypeScript compiler.

**NOTE**

At the time of this writing, Deno is still an experimental piece of software, and you can check its current status at [deno.land](https://deno.land).

Some of the regrets of Ryan Dahl were that Node apps need package.json, node\_modules, npm for module resolution, and a central repository for distributing packages. Deno doesn't need any of these. If your app needs a package, it should be able to get it directly from the source code repository of that package. We'll show you how this works in a small project named deno, which comes with this chapter's code.

This project has just one script index.ts as seen in listing 6.26.

#### Listing 6.26 /deno/index.ts

```
import { bgGreen, black } from 'https://deno.land/std/colors/mod.ts'; ①
const message: string = 'Ran with deno!'; ②
console.log(black(bgGreen(message))); ③
```

- ① Including the library colors
- ② Using the TypeScript type string
- ③ Using the API from the library colors

Note that we're importing the library named *colors* right from the source. There is no package.json that would list this library as a dependency, and no npm install is required. You may say, "It's dangerous using a direct link to a third party library. What if their code changes breaking your app?"

It's not going to happen because Deno locally caches each third-party library when it's loaded for the first time. Every subsequent run of your app will reuse the same version of each library unless you specify a special option --reload.

**NOTE** We couldn't use the library chalk for this example, because it's not packaged for being consumed by Deno.

Nothing else is needed for this script to run as long as you have the deno executable, which can be downloaded from [github.com/denoland/deno/releases](https://github.com/denoland/deno/releases). Just pick the latest release and get the zip file for the platform you use. For example, for MAC OS, download and unzip the file deno\_osx\_x64.gz.

For simplicity, just download it in the deno directory. You can use the following command to launch the app once you have deno downloaded (tested with v0.3.4):

```
./deno_osx_x64 index.ts
```

**TIP** In MAC OS, you may need to add a permission to execute this file: `chmod +x ./deno_osx_x64.`

**TIP** If you run this on Windows, make sure you have PowerShell of at least version 6 and Windows Management Framework. Otherwise, you may see the error "TS5009: Cannot find the common subdirectory path for the input files".

As you see, Deno runs a TypeScript program out of the box, and it won't require npm or package.json. The first time you run this app it'll produce the following output:

```
Compiling file: ...chapter6/deno/index.ts
Downloading https://deno.land/std/colors/mod.ts...
Compiling https://deno.land/std/colors/mod.ts
```

Deno compiled index.ts, and then downloaded and compiled the library colors. After that, it ran our app producing the following output:



Ran with deno!

**Figure 6.15** Running the app under Deno

Deno cached the compiled library colors, so it won't need to download/compile colors next time you run the app. As you see, we didn't need to configure project dependencies and there was nothing to install or configure prior to running the app.

Deno doesn't understand the format of the npm packages, but if it will gain traction, maintainers of the popular JavaScript libraries will be packaging their product in the format acceptable by Deno. Let's keep an eye on this tool.

### 6.5.2 ncc

The second tool to watch is ncc (see [github.com/zeit/ncc](https://github.com/zeit/ncc)). It's a command line interface for compiling a Node.js module into a single file, together with all its dependencies. This tool can be used by TypeScript developers who write the apps that run on the server side.

Just to give you some background on ncc, it's a product of the company called Zeit, which is a serverless cloud providers. You may have heard of their product Now (see [zeit.co/now](https://zeit.co/now)) that offers a super-easy serverless deployment of web apps.

Zeit also develops software that allows you to split any app into as many small pieces as possible. For example, if you're writing an app that uses Express frameworks, they want to represent each endpoint by a separate bundle, which contains only the code needed for the functionality of this endpoint.

They need it to avoid running any live servers. If the client hits the endpoint, it returns the serverless bundle in a miniature container, and response time is only 100 milliseconds, which is pretty impressive. And ncc is the tool that can package a server-side app into a small bundle.

ncc can take any JavaScript or TypeScript as an input and produce a bundle as an output. It requires either minimal configuration or no configuration at all. The only requirement is that your code should use ES6 modules or `require()`.

We'll show you a small app that has minimal configuration, which is required because we use TypeScript. If we wrote this app in JavaScript, no configuration would be needed. This app is located in the directory ncc-typescript, which has the file package.json shown in listing 6.27, tsconfig.json, and index.ts that uses our old buddy: the chalk library.

#### **Listing 6.27 A fragment of ncc/package.json**

```
{
  "name": "ncc-typescript",
  "description": "A code sample for the TypeScript Quickly book",
  "homepage": "https://www.manning.com/books/typescript-quickly",
  "license": "MIT",
  "scripts": {
    "start": "ncc run src/index.ts",          ①
    "build": "ncc build src/index.ts -o dist -m" ②
  },
  "dependencies": {
    "chalk": "^2.4.1"
  },
  "devDependencies": {
    "@zeit/ncc": "^0.16.1"                  ③
  }
}
```

- ① Using ncc in the run mode
- ② Transpiling TypeScript with ncc (-m is for prod optimization)

### ③ The ncc tool

You won't see tsc as a dependency in this package.json file because the TypeScript compiler is an internal dependency of ncc. Still, you can list the compiler's options in the tsconfig.json if needed. From the development perspective, ncc allows you compile and run your TypeScript code in one process.

Note the `scripts` section, where we defined two commands: `start` and `build`. Accordingly, TypeScript developers can use ncc two modes:

1. The run mode where ncc runs the TypeScript code without explicit compilation (it'll compile it internally)
2. The build mode where the TypeScript is transpiled into JavaScript

And the good part is that you don't need to use a bundler like Webpack because ncc will build the bundle for you. Try it for yourself by doing `npm install` and running the sample app located in `index.ts`:

```
npm run start
```

As per package.json shown in listing 6.25, the `start` command will run ncc that will compile and run `index.ts`, and the console output will look as in figure 6.16:

```
> ncc run src/index.ts

ncc: Using typescript@3.2.2 (ncc built-in)
  46kB  index.js
  58kB  index.js.map
121kB  sourcemap-register.js
167kB  [1880ms] - ncc 0.16.1
Built with ncc
```

**Figure 6.16 Running the app with ncc**

Our `start` command transpiled `index.ts` with the generation of source maps (default). In the run mode, the transpiled file was not generated. If you run the `build` command, ncc will generate the bundle `index.js` in the `dist` directory, but the app won't run:

```
npm run build
```

The console output of the `build` command will look similar to this:

```
ncc: Using typescript@3.2.2 (ncc built-in)
24kB  dist/index.js
24kB  [1313ms] - ncc 0.16.1
```

The size of the optimized bundle is 24Kb (ncc uses Webpack internally) as seen in figure 6.6. The

ncc generated bubble contains the code that we wrote as well as the code of the chalk library, and you can run our bundled app as usual:

```
node dist/index.js
```

The output is as expected:

```
$ node dist/index.js
Built with ncc
```

**Figure 6.17** Running the app with ncc

To summarize, the ncc benefits are:

1. Zero configuration for building and running apps
2. You can use either the run or build modes. In case of run, it spares you from the explicit compilation of the TypeScript code.
3. It supports the hybrid projects where some code is written in JavaScript and some in TypeScript.

In this section, we mentioned two interesting tools: Deno and ncc, but the TypeScript ecosystem is evolving fast, and we should be watching for the new tools that will make us more productive, our apps more responsive and build and deployment processes straightforward.

## 6.6 Summary

In this chapter you learned:

- Which tools TypeScript developers use on daily basis
- What role bundlers play in JavaScript or TypeScript projects
- Why it may make sense to use both TypeScript and Babel
- Which upcoming tools could be considered for adding to your toolbox

This chapter should be useful for any TypeScript developer, especially to those who have limited experience with JavaScript/Web tooling. Knowing the syntax of any programming language is important, but understanding the process of how your program could be turned into a working runnable app is equally important.

In this chapter, we didn't mention yet another useful package called ts-node, which is used for running both tsc and Node.js runtime as a single process. We'll use it the section 10.4.2 in chapter 10 while starting a server written in TypeScript.

In the next chapter, you'll see how to bring JavaScript libraries in your TypeScript app.



# Using TypeScript and JavaScript in the same project

## This chapter covers:

- How to enjoy the TypeScript benefits when working with a JavaScript library
- The role of type definition files
- How to upgrade an existing JavaScript app to TypeScript

In this chapter, we'll show how you can benefit from such TypeScript features as getting compilation errors and auto-complete even while using third-party libraries written in JavaScript. We'll start by explaining the role of type definition files, and then will discuss a concrete use case where an app written in TypeScript uses a JavaScript library. Finally, we'll go over the things to consider for implementing the gradual upgrade of your app from JavaScript to TypeScript.

## 7.1 Type definition files

The JavaScript language was created in 1995, and gazillions lines of code were written in this language since then. Developers from around the globe have released thousands of libraries written in JavaScript, and the chances are that your TypeScript app could also benefit from using one of these libraries.

It would be naive to expect that creators of JavaScript libraries would invest time in re-writing their libraries or frameworks in TypeScript, but we want to be able to use the JavaScript's heritage in our TypeScript apps. Moreover, we're spoiled by TypeScript conveniences like static type analyzer, auto-complete, and immediate reports on compilation errors. Can we continue enjoying these features while working with the API of JavaScript libraries? Yes, we can with the help of *type definition files*.

**NOTE**

In the previous chapter in section 6.3.2, we already had a project where the TypeScript code used a JavaScript library called chalk. But the goal of that example was just to show how to bundle the TypeScript and JavaScript together.

### 7.1.1 Getting familiar with type definition files

The purpose of type definition files is to let the TypeScript compiler know the types expected by the APIs of a specific JavaScript library. Type definition files just include the names of the variables (with types) and function signatures (with types) used by a particular JavaScript library. The big idea is to let TypeScript know the names and types of the members of a particular JavaScript library.

In 2012, Boris Yankov created a Github repository for type definition files (see [github.com/DefinitelyTyped/DefinitelyTyped](https://github.com/DefinitelyTyped/DefinitelyTyped)). Other people started contributing, and currently more than 8000 contributors work with this project. Then, the site DefinitelyTyped.org was created, and a couple of years ago, the new *organization* @types was created at npmjs.org, which became another repository for type definition files, and all declaration files from DefinitelyTyped.org are published automatically to the @types organization.

The suffix of any definition filename is *.ts*, and you can find these files for more than 6000 JavaScript libraries at [www.npmjs.com/~types](https://www.npmjs.com/~types). Just go there and search for the JavaScript library you're interested in. For example, you can find the information on type definitions of jQuery at [@types/jquery](https://www.npmjs.com/package/@types/jquery), and figure 7.1 shows a screenshot of this web page.

The screenshot shows the npmjs.com package page for `@types/jquery`. At the top, there's a navigation bar with links for Nautical Pea Maker, npm Enterprise, Products, Solutions, Resources, Docs, Support, Search, Join, and Log In. Below the header, a banner promotes npm Enterprise. The main content area shows the package name `@types/jquery`, version 3.3.29 (published 3 months ago), and tabs for Readme, 1 Dependencies, 735 Dependents, and 90 Versions. The Installation section contains the command `npm install --save @types/jquery`. The Summary section notes that the package contains type definitions for jQuery. The Details section provides additional information, including the URL for the sources (`https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/types/jquery`) and a list of global values (`$`, `Symbol`, `jQuery`). Annotations with arrows point from labels to specific parts of the page: a box labeled "Command to install" points to the installation command; a box labeled "Sources" points to the GitHub URL; and a box labeled "Global values" points to the list of global values.

**Figure 7.1** jQuery type definitions at [npmjs.org](https://www.npmjs.com/package/@types/jquery)

On top right in figure 7.1, you see the command that installs the type definition file(s) for jQuery, but we like adding the `-D` option so npm would add `@types/jquery` to the `devDependencies` section of the project's `package.json` file.

```
npm install @types/jquery -D
```

#### NOTE

The above command doesn't install the jQuery library; it just installs the type definitions of jQuery members.

In general, you install type definitions specifying the organization name `@types` followed by the name of the package. After installing `@types/jquery`, you can find several files with extension `.d.ts`, e.g. `jQuery.d.ts` and `jQueryStatic.d.ts` located in the `node_modules/@types/jquery` directory of your project, and TypeScript compiler (and static analyzer) will use them to help you with auto-complete and type errors.

In the middle in figure 7.1, you see the URL of the sources of jQuery type definitions, and at the bottom - the name of global values offered by jQuery. For example, you can use `$` for accessing jQuery API when it's installed. Let's see if after installing type definition files IDEs will start

helping us with jQuery API.

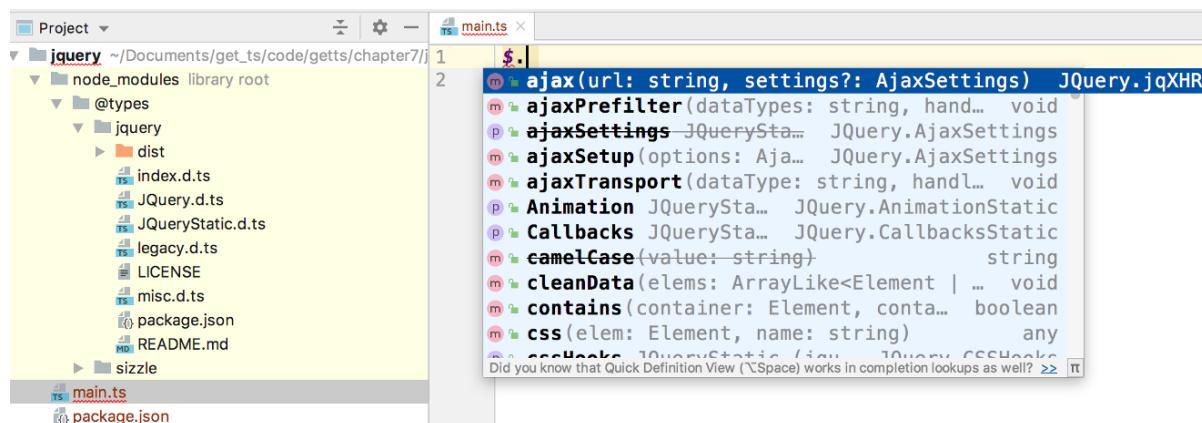
You can create a new directory and turn it into an npm project by running the command `npm init -y` there. This command creates the file `package.json`, and then you can install type definition for jQuery:

```
npm install @types/jquery -D
```

Let's see what happens next.

### 7.1.2 Type definition files and IDEs

Now let's see how IDEs use type definition files. Open the npm project created in the previous section in your IDE, and create and open `main.ts` and press **CTRL-Space** after entering `$..`. If you use WebStorm IDE, you'll see available jQuery API as seen in figure 7.2.



**Figure 7.2** WebStorm's auto-complete for jQuery

On top, you can see `JQuery`'s method `ajax()` with strongly typed argument just like in any TypeScript program. Keep in mind that we didn't even install jQuery, which is written in JavaScript anyway; we just have the type definitions. This is great, but let's open the same project in VS Code. You may not see any auto-complete as seen in figure 7.3.



**Figure 7.3** VS Code can't find type definitions for jQuery

The reason is that WebStorm IDE automatically shows all definitions it can find in the project while VS Code prefer us to explicitly configure which d.ts files to use. OK, let's play by the VS Code rules and create the file `tsconfig.json` with the compiler's option `types`. In this array, you can specify which type definitions to use for auto-complete (e.g. the names of the directories under `node_modules/@types`). Listing 7.1 shows `tsconfig.json` that we added to the project.

### **Listing 7.1 tsconfig.json**

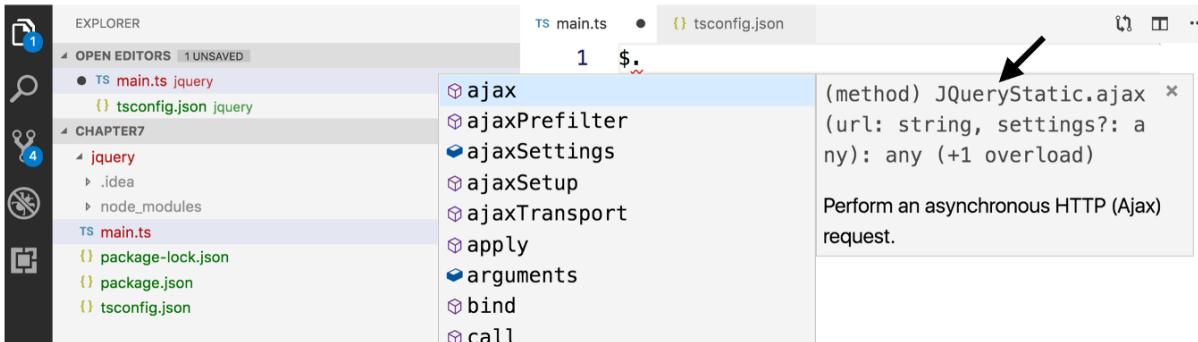
```
{
  "compilerOptions": {
    "types" : [ "jquery" ]
  }
}
```

While `types: [ jquery ]` works for this example, if you had to add type definition files for several JavaScript libraries, you'd need to list all of them in the `types` compiler option, e.g. `types: [ jquery, lodash ]`. Adding the compiler's option `types` is not the only way to help the compiler in finding type definitions, and we'll show you the `reference` directive in section 7.1.4.

#### **TIP**

Module resolution is the process that the compiler goes through to figure out what an import refers to. For details, refer to TypeScript documentation at [www.typescriptlang.org/docs/handbook/module-resolution.html](http://www.typescriptlang.org/docs/handbook/module-resolution.html).

Now press Ctrl-Space after `$ .`, and the auto-complete starts working properly. After clicking on the `ajax()` function, VS Code shows its program documentation as seen in figure 7.4.



**Figure 7.4 VS Code shows auto-complete and doc for JQuery's method ajax()**

#### **TIP**

In WebStorm, to see program documentation for an item selected in the auto-complete list, select it and click **Ctrl-J**.

No matter what IDE you use, having a d.ts file for a JavaScript code gives you an intelligent help from TypeScript compiler and static analyzer. Let's peek inside the type definition file. The arrow in figure 7.4 points at the interface name where TypeScript found the type definitions for

the `ajax()` function. Coincidentally, it's defined in the file called `JQueryStatic.d.ts` located in the directory `node_modules/@types/jquery`. Hover the mouse pointer over the name of the function `ajax()` and press CTRL-Click. Both VS Code and WebStorm will open the fragment of `ajax()` type definitions shown in listing 7.2.

### **Listing 7.2 A fragment from `JQueryStatic.d.ts`**

```
/**  
 * Perform an asynchronous HTTP (Ajax) request. ①  
 * @param url A string containing the URL to which the request is sent. ②  
 * @param settings A set of key/value pairs that configure the Ajax request. All settings are optional.  
 ②  
 * A default can be set for any option with $.ajaxSetup(). See jQuery.ajax( settings ) below  
 * for a complete list of all settings.  
 * @see \`{@link https://api.jquery.com/jQuery.ajax/ }\`  
 * @since 1.5  
 */  
ajax(url: string, settings?: JQuery.AjaxSettings): JQuery.jqXHR; ③
```

- ① Description of the function `ajax()`
- ② Description of the `ajax()` parameters
- ③ Signature of `ajax()` with types

Type definition files can contain only type declarations. In case of jQuery, its type declarations are wrapped in an interface with multiple properties and method declarations, e.g. `ajax()` in listing 7.2. In some d.ts files, you'll see the use of the word `declare`, for example:

```
declare const Sizzle: SizzleStatic;  
  
export declare function findNodes(node: ts.Node): ts.Node[];
```

We're not declaring `const Sizzle` or the function `findNode()` here, but just stating that we're going to use a JavaScript library that contains the declaration of `const Sizzle` and `findNode()`. In other words, this line tries to calm down tsc: "Don't scream if you'll see `Sizzle` or `findNode()` in my TypeScript code. During the runtime, my app will include the JavaScript library that has these types".

#### **NOTE**

The term `ambient declaration` means that the variable in question will exist in runtime. If you wouldn't have type definitions for jQuery, you can simply write `declare var $: any` in your TypeScript code and use the variable `$` to access jQuery API. Just don't forget to load jQuery along with your app.

As you see, type definition files allow us to kill two birds with one stone: use the existing JavaScript libraries while enjoying the benefits of a strongly-typed language.

**TIP**

Some JavaScript libraries include d.ts files and there is no need to install them separately. A good example is a library moment.js used for validating, manipulating, and formatting dates. Visit its repository at [github.com/moment/moment](https://github.com/moment/moment), and you'll see the file moment.d.ts there.

## SIDE BAR Using JavaScript libraries without type definition files

While type definition files are a preferable way of using JavaScript libraries in a TypeScript app, you can use them even without having type definition files. If you know the global variable of the selected JavaScript framework (e.g. `$` in jQuery), you can use it as is. Modern JavaScript libraries may use module systems, and instead of offering a global variable, they may require that a particular module has to be imported in your code. Refer to the product documentation of the library of your choice.

Let's take jQueryUI, which is a set of UI widgets and themes built on top of jQuery. Let's assume that the type definition file for jQueryUI doesn't exist (even though it does).

The Getting Started guide of JQueryUI (see [learn.jquery.com/jquery-ui/getting-started](http://learn.jquery.com/jquery-ui/getting-started)) states that to use this library in a Web page, you need to install locally add the code shown in listing 7.3 to the HTML document:

### Listing 7.3 Adding jQueryUI to a Web page

```
<link rel="stylesheet" href="jquery-ui.min.css">      ①
<script src="external/jquery/jquery.js"></script>    ②
<script src="jquery-ui.min.js"></script>            ③
```

- ① Adding CSS
- ② Adding jQuery
- ③ Adding jQueryUI

After this is done, you can add jQueryUI widgets to the TypeScript code. To get access to jQueryUI you'd still use `$` (the global variable from jQuery). For example, if you have an HTML dropdown `<select id="customers">`, you can turn it into the jQueryUI dropdown `selectMenu()` like this:

```
$( "#customers" ).selectMenu();
```

The above code will work, but without the type definition file you won't get any help from TypeScript, and your IDE will be highlighting jQueryUI API as erroneous. Of course, you can "fix" all tsc errors with following ambient type declaration:

```
declare const $: any;
```

It's always better to use the type definition file, if available.

### 7.1.3 Shims and type definitions

A *shim* is a library that intercepts API calls and transforms the code so the old environment (e.g. IE 11) can support newer API (e.g. ES6). For example, ES6 has introduced the method `find()` for arrays, which finds the first element that meets the provided criteria. In the code snippet in listing 7.3, the value of `index` will be 4 because it's the first value that's greater than 3.

#### Listing 7.4 Using the method `find()` on the array

```
const data = [1, 2, 3, 4, 5];

const index = data.find(item => item > 3); // index = 4
```

If your code has to run in Internet Explorer 11, which doesn't support ES6 API, you'd add the compiler's option "target": "ES5" in `tsconfig.json`. In this case, your IDE will underline the `find()` method with a squiggly line as an error because ES5 didn't support it, and the IDE won't even offer the method `find()` in the auto-complete list as shown in figure 7.5.

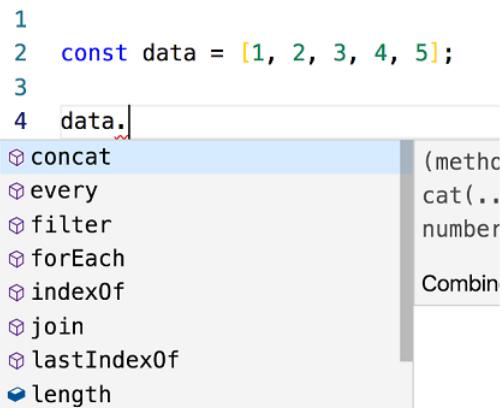


Figure 7.5 No method `find()` in ES5 arrays

Can you still use the newer API and see it in the auto-complete list? You can, if you install the type definition file `es6-shim.d.ts` and add it to the `types` compiler option in `tsconfig.json`:

```
npm install @types/es6-shim -D
```

Add this shim to our `tsconfig.json` ("`types` : ["`jquery`", "`es6-shim`"]), and your IDE won't complain and will show you the method `find()` in the auto-complete list as seen in figure 7.6.

```

2 const data = [1, 2, 3, 4, 5];
3
4 const array2 = data.find( item => item > 3 );
5 concat          (method) Array<number>.fin ×
6 copyWithin      d(predicate: (value: numbe
7 entries        r, index: number, obj: num
8 every          ber[][]) => boolean, thisAr
9 fill            g?: any): number
10 filter         Returns the value of the first element in
11 find           the array where predicate is true, and
12 findIndex       undefined otherwise.
13 forEach

```

**Figure 7.6 es6-shim helps with ES6 API**

**TIP**

There is a newer shim called core-js (see [www.npmjs.com/package/core-js](http://www.npmjs.com/package/core-js)), which can be used not only for the ES6 syntax, but for the newer versions of the ECMAScript specs as well.

### 7.1.4 Creating your own type definition files

Let's say sometime ago you've created a JavaScript function `greeting()`, which is located in the file `hello.js`.

#### Listing 7.5 hello.js

```

function greeting(name) {
  console.log("hello " + name);
}

```

You want to continue using this excellent function (with auto-complete and type checking) in your TypeScript project. In the `src` directory, create a file `typings.d.ts` with the following content:

#### Listing 7.6 The file ./src/typings.d.ts

```
declare function greeting(name: string): void;
```

Finally, you need to let TypeScript know where this type definition file is located. Since our `greeting()` function is not overly useful for the JavaScript community, we have not published it at `npmjs.org`, and no one have created a `.d.ts` file in the `@types` organization either. If this is the case, you can use a special TypeScript reference directive (a.k.a. triple-slash directive), which have to be placed on the top of your `.ts` file that uses `greeting()`. Figure 7.7 shows a screenshot taken while we were typing `greeti` in the file `main.ts` in VS Code.

```

1  /// <reference path="src/typings.d.ts" />
2
3  greeti
4      ⚡ greeting
5          ••WebGLRenderingContext
6

```

function greeting(name: string): void

**Figure 7.7 Getting auto-complete in main.ts**

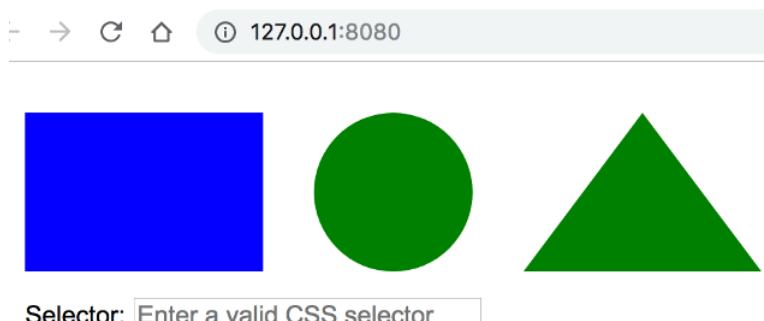
As you see, the auto-complete prompts us with the argument and return types of the JavaScript function `greeting()`.

**TIP**

If you want to write a type definition file for a JavaScript library, read the TypeScript documentation at [www.typescriptlang.org/docs/handbook/declaration-files/introduction.html](https://www.typescriptlang.org/docs/handbook/declaration-files/introduction.html). You can read more about the triple-slash directives at [www.typescriptlang.org/docs/handbook/triple-slash-directives.html](https://www.typescriptlang.org/docs/handbook/triple-slash-directives.html).

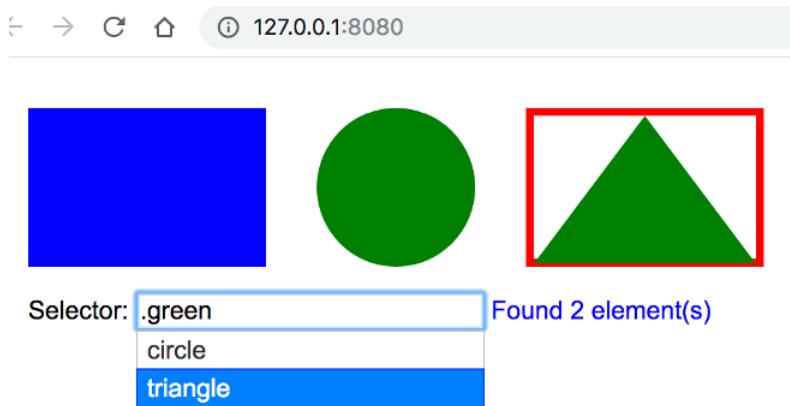
## 7.2 A sample TypeScript app that uses JavaScript libraries

In this section, we'll review the code of a sample app that's written in TypeScript and uses the JavaScript library jQuery UI. This simple app will display three shapes: a rectangle, circle, and triangle as seen in figure 7.8.



**Figure 7.8 Three shapes are rendered by jQuery UI**

If you read the printed version of this book, you need to know that the rectangle is blue, and both the circle and triangle are green. The user can enter a valid CSS selector, and the input field will render a dropdown list with the shape name(s) that have the provided selector. Figure 7.9. shows a screenshot taken after the user entered `.green` in the input field and selected the triangle in the dropdown; the triangle became surrounded with a red border.



**Figure 7.9 Finding element that has the CSS class .green**

Finding the HTML elements that have one of specific selectors is implemented using jQuery, and rendering shapes is done by jQuery UI. This sample project is located in the directory chapter7/jquery-ui-example, and it includes four files: package.json, tsconfig.json, index.ts, and index.html. The content of the file package.json is shown in listing 7.7.

### Listing 7.7 package.json

```
{
  "name": "jquery-ui-example",
  "description": "Code sample for the TypeScript Quickly book",
  "homepage": "https://www.manning.com/books/typescript-quickly",
  "license": "MIT",
  "devDependencies": {
    "@types/jquery": "^3.3.29",          ①
    "@types/jqueryui": "^1.12.7",        ②
    "typescript": "^3.4.1"               ③
  }
}
```

- ① Type definitions for jQuery
- ② Type definitions for jQueryUI
- ③ The TypeScript compiler

As you see, we didn't add the jQuery and jQuery UI libraries to package.json, because we've added the three lines shown in listing 7.8 to the <head> section of index.html.

### Listing 7.8 Adding jQuery and jQueryUI to index.html

```
<link rel="stylesheet" href="//code.jquery.com/ui/1.12.1/themes/base/jquery-ui.css"> ①
<script src="//code.jquery.com/jquery-3.3.1.min.js"></script> ②
<script src="//code.jquery.com/ui/1.12.1/jquery-ui.min.js"></script> ③
```

- ① Adding the jQuery UI styles
- ② Adding the jQuery library
- ③ Adding the jQuery UI library

You may ask, why didn't you add the `dependencies` section to `package.json` like we do with all other npm packages? The locally installed jQuery UI didn't include its bundled version, and we didn't want to complicate this app by adding Webpack or other bundler. That's why we decided to find the URL of the content delivery network (CDN) for these libraries.

The home page of jQuery ([jquery.com](https://jquery.com)) includes the button Download, which brings you to the page [jquery.com/download](https://jquery.com/download), which in turn includes the link [jquery.com/download](https://jquery.com/download) with the required URLs. If you need to include any JavaScript library to your project, you'd need to go through a similar discovery process.

The `<head>` section of our file `index.html` also includes the styles shown in listing 7.9. In our TypeScript code we'll use jQuery to get references to the HTML elements with IDs `#shapes`, `#error`, and `#info`.

**NOTE** In this demo app, we use jQuery selectors to find elements on the page, but these selectors that are already supported by the standard `document.querySelector()` or `document.querySelectorAll()` methods. We use jQuery just for the sake of demoing how TypeScript code can work with JavaScript libraries.

The user will be able to enter in the input field any valid CSS style that exists within the DOM element with ID `#shapes` and see the results in the auto-complete list as shown earlier in figure 7.9.

### Listing 7.9 A fragment from the <styles> tag in index.html

```

<style>
    #shapes {
        display: flex;
        margin-bottom: 16px;
    }

    #shapes > *:not(:last-child) {
        margin-right: 32px;
    }

    @media (max-width: 640px) { ❶
        #shapes {
            flex-direction: column;
            align-items: center;
        }

        #shapes > *:not(:last-child) {
            margin-bottom: 16px;
            margin-right: 0;
        }
    }

    #rectangle {
        background-color: blue; ❷
        height: 100px;
        width: 150px;
    }

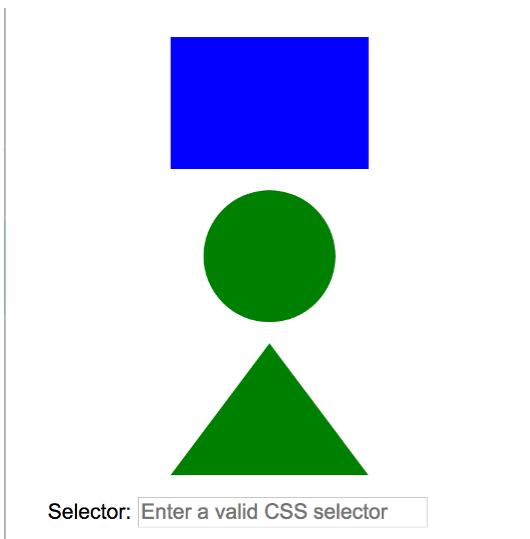
    #circle {
        background-color: green; ❸
        border-radius: 50%;
        height: 100px;
        width: 100px;
    }

    #triangle {
        color: green; ❸
        height: 100px;
        width: 150px;
    }
</style>

```

- ❶ Changing page layout for devices having width less than 640 pixels
- ❷ The rectangle is blue
- ❸ The circle and triangle are green

The media query `@media (max-width: 640px)` instructs the browser to change the layout on small devices (less than 640 pixels in width). The style `flex-direction: column` will render our shapes vertically, and `align-items: center;` will center the shapes in the page as shown in figure 7.10.



**Figure 7.10 Finding the DOM element that has the CSS class .green**

The `<body>` section of `index.html` has two containers implemented as `<div>` tags as shown in listing 7.10. The top one `<div id="shapes">` has the child `<div>` tags that represent our shapes. The bottom container includes the input field for entering the search criteria and two areas for displaying an error or info message (e.g. "Found 2 element(s)" as in figure 7.9).

### **Listing 7.10 The body section of index.html**

```

<body>
  <div id="shapes">
    <div id="rectangle"
        class="blue"
        hasAngles>①
    </div>

    <div id="circle"
        class="green"></div>②

    <div id="triangle"
        class="green"
        hasAngles>
      <svg viewBox="0 0 150 100">
        <polygon points="75, 0, 150, 100, 100, 0, 75" fill="currentColor"/>
      </svg>
    </div>
  </div>③

  <div class="ui-widget">
    <label for="selector">Selector:</label>
    <input id="selector" placeholder="Enter a valid CSS selector">④
    <span id="error"></span>
    <span id="info"></span>
  </div>

  <script src="dist/index.js"></script>⑤
</body>

```

- ① The container with shapes
- ② Any of these CSS attributes can be used for finding a shape with our UI

- ③ An error message will be rendered here
- ④ An info message will be rendered here
- ⑤ This script is a compiled version of index.ts

Any of the valid selectors can be entered in the input field, e.g. div, .green, hasAngles, and the only reason we've added the attribute hasAngle to the rectangle and triangle was to allow searching for these shapes by entering the selector [hasAngles] in the input field.

The main goal of this app is to illustrate the use of the Autocomplete widget offered by jQuery UI. It enables users to quickly find and select from a pre-populated list of values as they type, leveraging searching and filtering. You see this widget in action at the bottom of the figure 7.9. If the user enters .green, we'll find the DOM elements that have this CSS selector and will add them to the source list of values in the Autocomplete widget.

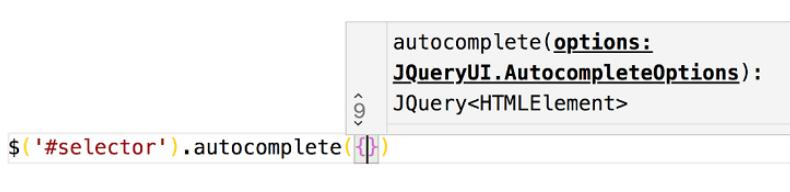
The Autocomplete widget is described in jQuery UI documentation at (see [api.jqueryui.com/autocomplete](https://api.jqueryui.com/autocomplete)), and it requires the object options with a mandatory property source, which defines the data to use.

### **Listing 7.11 A sample of using the Autocomplete widget**

```
$('#selector')
  .autocomplete({
    source: (request,
      response) => { ... });
  ^1
  ^2
  ^3
  ^4
```

- ① Our <input> field has id="#selector"
- ② Append the jQuery UI Autocomplete widget
- ③ A function that gets the data and returns a callback
- ④ A callback to invoke when a value from the list is selected

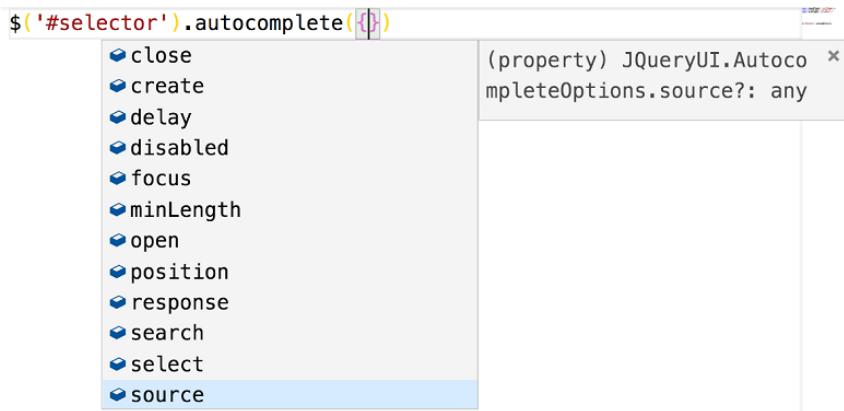
The code in listing 7.11 has no types, but since we have the jQuery UI type definition file installed, VS Code starts leading us through the API of autocomplete as seen in figure 7.11. Note the digit 9 with the up and down arrows. jQuery UI offers many different ways of invoking autocomplete(), and by clicking on the arrows, you can pick the API you like.



**Figure 7.11 The first prompt by VS Code**

Figure 7.11 prompts us that the type of the option object is `JQueryUI.AutocompleteOptions`.

You can always press the keys Ctrl-Space and the IDE will continue helping you. According to the widget's documentation, we need to provide the source of the auto-complete values, and VS Code list the `source` option among others as shown in figure 7.12.



**Figure 7.12 VS Code keeps prompting**

Perform Cmd-Click (or Ctrl-Click) on `autocomplete`, and it'll open the file `index.d.ts` with all possible options. Do Cmd-Click on `JQueryUI.AutocompleteOptions`, and you'll see its type definitions as shown in figure 7.13.

```
interface AutocompleteOptions extends AutocompleteEvents {
    appendTo?: any; //Selector;
    autoFocus?: boolean;
    delay?: number;
    disabled?: boolean;
    minLength?: number;
    position?: any; // object
    source?: any; // [], string or () ←
    classes?: AutocompleteClasses;
}
```

**Figure 7.13 JQueryUI.AutocompleteOptions**

The IDE's typeahead help may not be perfect; it's only as good as the provided type definition file. Take another look at the property `source` in figure 7.13. It's declared as `any`, and then the comment stated that it could be an array, string, or function. This declaration could be improved by declaring a union type that would allow only these types, for example:

```
type arrayOrFunction = Array<any> | string | Function;

let source: arrayOrFunction = (request, response) => 123;
```

Introducing the type `arrayOrFunction` would eliminate the need of writing that comment `// []`, `string`, or `()`. Of course, you'd need to replace `123` with some code that handles `request` and `response`.

**TIP**

As you can imagine, the library code and the API listed in its d.ts file may go out of sync. This all depends on the good will of people who maintain the code to keep type definitions up to date.

Now let's review the TypeScript code in our file index.ts shown in listing 7.12. The readers who were developing web apps ten years ago would recognize the jQuery style of coding: We start with getting references to the browser's DOM elements on the page. For example, `$('#shapes')` means that we want to find a reference to the DOM element with `id="shapes"`.

### **Listing 7.12 index.ts**

```
const shapesElement = $('#shapes');      ①
const errorElement = $('#error');        ①
const infoElement = $('#info');          ①

$('#selector')                         ①
    .autocomplete({                  ②
        source: (request: { term: string }, ③
                  response: () => void) => {
            try {                      ④
                const elements = $(request.term, shapesElement);
                const ids = elements.map((_index, dom) => ({ label: $(dom).attr('id'), ⑤
                                                               value: request.term })).toArray();

                response(ids);           ⑥
            }

            infoElement.text(`Found ${elements.length} element(s)`); ⑦
            errorElement.text('');
        } catch (e) {                  ⑧
            response([]);
            infoElement.text('');
            errorElement.text('Invalid selector');
            $('*', shapesElement).css({ border: 'none' });
        }
    },
    focus: (_event, ui) => {            ⑨
        $('*', shapesElement).css({ border: 'none' });
        `#${ui.item.label}`, shapesElement).css({ border: '5px solid red' });
    }
);

$('#selector').on('input', (event: JQuery.TriggeredEvent<HTMLInputElement>) => { ⑩
    if (!event.target.value) {
        $('*', shapesElement).css({ border: 'none' });
        errorElement.text('');
        infoElement.text('');
    }
});
```

- ① Using jQuery to find references to DOM elements
- ② The first parameter of the function is the search criterion
- ③ The second parameter is a callback to modify the DOM
- ④ Find the elements with the shapes element that meet the search criterion
- ⑤ Find the IDs of the shapes that meet the criterion

- ⑥ Invoke the callback passing the autocomplete values
- ⑦ Handle event fired when the focus moves to one of the IDs
- ⑧ Remove the borders from the shapes, if any
- ⑨ Add the red border to the DOM element with selected ID
- ⑩ Reset all previous selections and messages

We pass to the Autocomplete widget an object with two properties: `source` and `focus`. Our `source` property is a function that takes two parameters:

1. `request` - an object with the search criterion, e.g. `{term: '.green'}`
2. `response` - a callback that implements finding IDs of the DOM element that meet the search criterion. In our case, we pass the callback function, which contains one `try/catch` block.

The `focus` property is an event handler that will be invoked when you move the mouse over one of the item in the rendered list. There, we clear previously bordered shapes and add a border to the currently selected one.

Before running this app, we need to compile TypeScript to JavaScript, and our TypeScript compiler will use the compiler options shown in listing 7.13.

### **Listing 7.13 tsconfig.json**

```
{
  "compilerOptions": {
    "outDir": "dist",      ①
    "target": "es2018"     ②
  }
}
```

- ① Where to place the compiled JavaScript
- ② Compile into the JavaScript compatible with the ES2018 spec

In previous chapters, we would create an npm script command in `package.json` to run the locally installed version of the executable we want to run. For example, adding the command `"tsc": "tsc"` to the `scripts` section in `package.json` would allow us to run locally installed compiler as follows:

```
npm run tsc
```

This time we were lazy and didn't configure this command. We'll use the `npx` command (it comes with `npm`) to compile our `index.ts` using the locally installed `tsc`:

```
npx tsc
```

After running this command, you'll see the file index.js in the dist directory. This file is used in index.html as follows:

```
<script src="dist/index.js"></script>
```

We're almost there. The only missing player is a web server that would serve our app to the browser. One of the simple web servers you can install is live-server (see [www.npmjs.com/package/live-server](https://www.npmjs.com/package/live-server)). Let's install it:

```
npm i live-server -g
```

**TIP**

Instead of installing live-server manually, you can run it as `npx live-server`. If live-server is not found in the project's node\_modules, npx will download it from the npmjs.com, cache globally and run the live-server binary.

To run the server, enter the following command in the Terminal window in root directory of our project:

```
live-server
```

Point your browser to localhost:8080 and you'll see our app running in the browser. The best way to understand how the code works is by running it through a debugger, and figure 7.14 shows our app running in the Chrome. It paused in the running script dist/index.js at the breakpoint placed on line 22, which is invoked when the `focus` event is fired. In the Watch panel on the right, we've added `ui.item.label`, which has the value `circle` matching the selection in the UI.

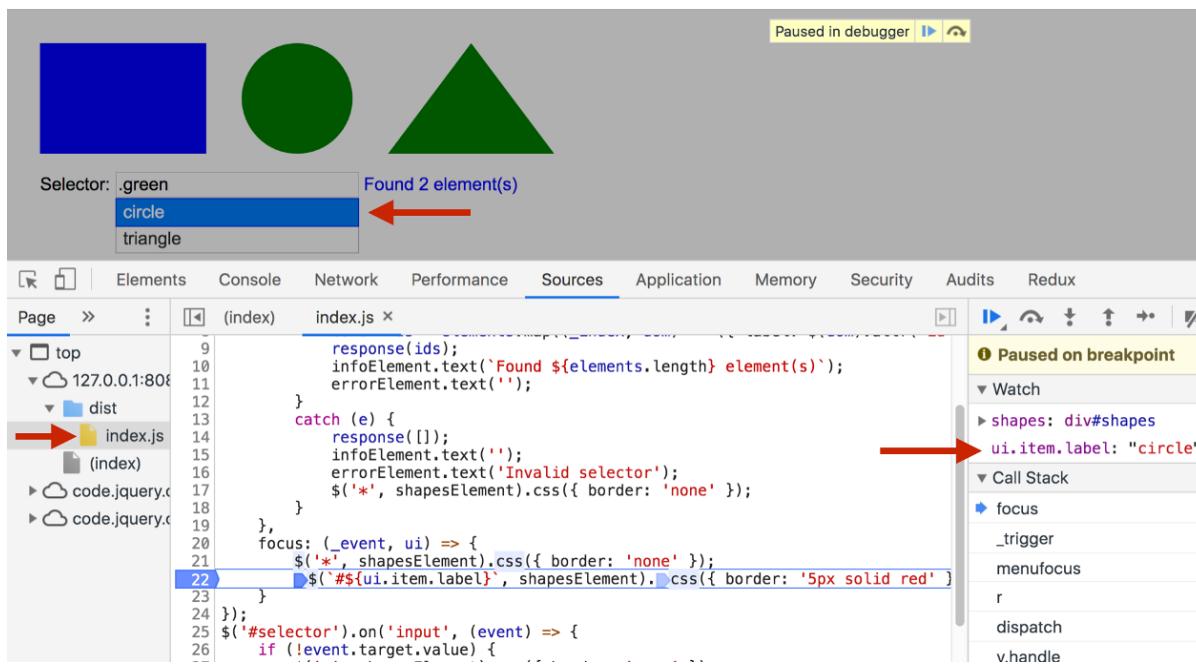


Figure 7.14 Hitting the breakpoint in dist/index.js

### TIP

In the section "Source maps" in chapter 6, we explained that having the source map files allow to debug the TypeScript code. Just add the line "sourceMap": true to the file tsconfig.json, and you'll be able to debug index.ts while running index.js.

After covering the subject of using third-party JavaScript libraries with your TypeScript code, let's discuss another scenario: you already have an app written in JavaScript and are considering switching to TypeScript.

## 7.3 Introducing TypeScript in your JavaScript project

In ideal world, you always work with the latest languages and technologies, but in reality, you have a mix of old and new ones. You are an enterprise developer, and your team is working on a JavaScript app for the last several years, but after reading this book, you have a strong desire for writing code in TypeScript. Of course, you can always come up with some pet project and start developing it in TypeScript after hours, but is it possible to bring TypeScript to your main JavaScript project at work?

TypeScript supports optional typing, which means you don't have to modify your JavaScript code declaring types of each variable or a function parameter. Why won't you start with bringing the TypeScript compiler to your JavaScript app? But the code base of your JavaScript app may have tens of thousands lines of code, and trying to use tsc to compile all of them may reveal hidden bugs and slow down the deployment process, which may not be the best way to start.

Select a part of your app that implements some isolated functionality (e.g. adding a new

customer or a shipping module) and run it through tsc as is. Most likely, your app already has some build process that uses such tools as Grunt, Gulp, Babel, Webpack et al. Find the right place and incorporate tsc into this process.

You don't even need to rename JavaScript files to give them the .ts extensions. Just use the TypeScript compiler option "allowJs": true, which tells tsc: "Please compile not only the .ts files but .js files as well and don't perform type checking - just transpile it according to the setting in the compiler's option target."

**TIP**

If you don't change the file extensions from .js to .ts, your IDE will still highlight the types in the JavaScript files as erroneous, but tsc will compile them if you use the option "allowJs": true.

You may say, "Why even ask tsc to skip type checking if types are optional anyway?" One of the reasons is that tsc still might not be able to fully infer all the types info from your JavaScript code will report errors. Another reason is that your existing code can be buggy (ok, they are not showstoppers), and allowing tsc to work at its fullest may reveal lots of compile errors, and you may not have time or resources to fix them. If it ain't broke, don't fix it, right?

Well, you may have a different approach: "If it ain't broke, improve it", and if you're confident that your JavaScript code is written well, opt in to type checking by adding the tsc compiler's option "checkJs": true to tsconfig.json. But if some of your JavaScript files would still generate errors, you can skip checking them by adding //@ts-nocheck comment to these files; conversely, you can choose to check only a few .js files by adding a comment //@ts-check to them without setting checkJs": true. You can even turn off type checking for a specific line of code by adding //@ts-ignore on the preceding line.

To illustrate the effect of type checking of the existing JavaScript code, we opened in VS Code the code of a random file OpenAjax.js from the GitHub repository of the Dojo framework (see [github.com/dojotoolkit/dojox/blob/master/OpenAjax.js](https://github.com/dojotoolkit/dojox/blob/master/OpenAjax.js)). Let's pretend that we want to start turning this code to TypeScript and added the //@ts-check comment to the top of this file, and you'll see some of the lines with squiggly underlines as shows in figure 7.15.

```

1 // @ts-check
2 import { isMoment } from './constructor';
3 import { normalizeUnits } from '../units/aliases';
4 import { createLocal } from '../create/local';
5 import isUndefined from '../utils/is-undefined';
6
7 if(!window["OpenAjax"]){
8     OpenAjax = new function(){
9         // summary:
10        //   the OpenAjax hub
11        // description:
12        //   see http://www.openajax.org/member/wiki/OpenAjax\_Hub\_S
13
14     var libs = {};
15     var ooh = "org.openajax.hub.";
16
17     var h = {};
18     this.hub = h;
19     h.implementer = "http://openajax.org";
20     h.implVersion = "0.6";
21     h.specVersion = "0.6";
22     h.implExtraData = {};
23     h.libraries = libs;

```

The diagram shows a snippet of JavaScript code with several lines underlined by red squiggly lines, indicating type check errors. Red arrows point from these error-prone lines to a red-bordered box containing the text 'Type check errors'. The lines affected include the import statements at the top, the definition of the OpenAjax object, and its properties like implementer, implVersion, specVersion, and libraries.

**Figure 7.15 Adding the // @ts-check comment to the top of a JavaScript file**

Let's ignore the errors on the import statements on top; we wouldn't see them if all these files were present. It looks like the error in line 8 is not an error either. It seems that the object `OpenAjax` will be present at the runtime. Adding `// @ts-ignore` right above line 8 would remove the squiggly.

But to fix the errors in lines 19-23 we'd need to declare a `type` or an `interface` with all these properties. By the way, while changing this code to TypeScript, you may want to rename the variable `h` and give it a more meaningful name.

Let's consider another piece of a JavaScript code shown in figure 7.16. We want to get the price of some product, and if it less than \$20, we'll buy it. The IDE doesn't complain and the code seems to be legitimate.

```

1 const getPrice = () => Math.random()*100;
2
3 if (getPrice < 20) {
4     console.log("Buying!");
5 }

```

**Figure 7.16 A buggy JavaScript code**

Let's add the `//@ts-check` comment on top so the static type analyzer can check this code for validity as seen in figure 7.17.

```

1  //@ts-check
2  const getPrice = () => Math.random()*100;
3
4  if (getPrice < 20) {
5      const getPrice: () => number
6  }
7  Operator '<' cannot be applied to types '() => number' and
8  'number'. ts(2365)
9  Quick Fix... Peek Problem

```

**Figure 7.17 @ts-check found a bug**

Oops! Our JavaScript code had a bug - we forgot to add parentheses after the `getPrice` in the if-statement (i.e. we never invoked this function). If you were wondering why this code would never give you OK to buy the product, now you know the reason: the expression `getPrice < 20` was never evaluated to `true!` Simply adding the `//@ts-check` on top helped us to find a runtime bug in a JavaScript program.

There's another tsc option `noImplicitAny` that could help you with JavaScript-to-TypeScript migration. If you're not planning to specify types of function parameters and return types, tsc may have hard times inferring the right types, and you may temporarily keep the compiler's option `"noImplicitAny": false` (the default). In this mode, if tsc cannot infer the variable type based on how it's used, the compiler silently defaults the type to `any`. That's what is meant by implicit any. Just don't forget to turn it back on when the migration to TypeScript is complete.

TypeScript compiler will go easy on your `.js` files. It'll allow adding properties to a class or a function after their declaration. The same applies to object literals in `.js` files: you can add properties to the object literal even if they were not defined originally. TypeScript supports the CommonJS module format, and will recognize the `require()` function calls as module imports. All function parameters are optional by default, and calls with fewer arguments than the declared number of parameters are allowed.

You can also help tsc with type inference by adding the JSDoc annotations (e.g. `@param`, `@return`) to your JavaScript code, and TypeScript compiler will understand them. Read more on the subject in the document "JSDoc support in JavaScript" at [github.com/Microsoft/TypeScript/wiki/JSDoc-support-in-JavaScript](https://github.com/Microsoft/TypeScript/wiki/JSDoc-support-in-JavaScript).

The process of upgrading your JavaScript project to TypeScript is not overly complicated, and in this section, we gave you a high-level overview of one approach of how to do it gradually. For more details, read the document "Upgrading from JavaScript" in the TypeScript documentation at [www.typescriptlang.org/docs/handbook/migrating-from-javascript.html](https://www.typescriptlang.org/docs/handbook/migrating-from-javascript.html). There you can find specifics on integration with the Gulp, Webpack, converting a React app to TypeScript and

more. In chapter 12, we'll show you an app developed in TypeScript with the React.js framework, and in chapter 13, we'll use the Vue.js library with TypeScript as well.

#### SIDEBAR Once again: Why TypeScript?

TypeScript is not the first attempt to create an alternative to JavaScript that could run either in a browser or in a standalone JavaScript engine. TypeScript is only seven years old, but it's already included in the top ten programming languages in various ratings. Why don't top 10 languages include older languages like CoffeeScript or Dart that were supposed to become an alternative way of writing JavaScript?

In our opinion, there are three major forces that make TypeScript standout:

1. It strictly follows the ECMAScript standards. If a proposed feature made it to the stage 3 of the TS39 process, it'll be included in TypeScript today.
2. TypeScript IDEs works with the same static type analyzer offering you a consistent help as you're writing code.
3. TypeScript easily inter-operates with the JavaScript code, which means that you can use thousands of existing JavaScript libraries in your TypeScript apps, and you learned how to do it in this chapter.

This concludes Part 1 of the book where we introduced the TypeScript language. We didn't cover each and every feature of the language, and we didn't plan to. The book title is "TypeScript Quickly" isn't it? If you understand all the materials from Part 1, you'll be able to easily pass a TypeScript technical interview, but to become a really productive TypeScript developer, we encourage you to study and run all sample apps described in Part 2.

## 7.4 Summary

In this chapter you learned:

- How to use existing JavaScript code in your TypeScript projects
- How having type definition files make you more productive in writing code
- How to use a JavaScript library that doesn't have a type definition file
- How to use TypeScript and JavaScript together using the library jQuery UI as an example
- How to gradually upgrade your existing JavaScript code to TypeScript

## *Part 2: Applying TypeScript in a blockchain app*

P1



# *Developing your own blockchain app*

## **This chapter covers:**

- The principles of blockchain apps
- What the hashing functions are for
- What's block mining
- The process of developing a primitive blockchain app

Part 2 of this book is a collection of various versions of an app, where we apply different TypeScript elements and techniques introduced in Part 1. We're not going to sell you TypeScript in Part 2 assuming that you like it by now. We'll simply roll up our sleeves and will start using it. Our goal is to show you how to develop a more than just a small example and see how the language really works.

Prior to starting writing this book, we had to decide which sample app to create to show you TypeScript in action. We didn't want it to be yet another ToDo app. We were looking for some hot technology, and this is how the idea of creating a blockchain app came about.

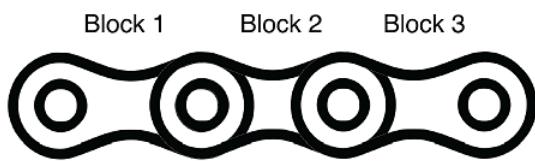
In Part 2 of this book, we'll develop several blockchain apps, e.g. a standalone app, a browser app, an Angular app, a React.js app, a Vue.js app - feel free to read only those chapters that interest you. But if you're planning to read Part 2, chapters 8 and 10 are a must. Since the whole idea of blockchains is still new, we'll start by explaining you the basics of how blockchains operate and what they are for.

## 8.1 Blockchain 101

While blockchains can be used for various types of applications, financial apps made the blockchain a buzzword. Most likely, you've heard about cryptocurrencies in general and Bitcoins in particular. Often, you see the words Bitcoin and blockchain in the same sentence, but if blockchain is a special decentralized way of storing immutable data, Bitcoin is a concrete cryptocurrency that uses a concrete implementation of a blockchain. In other words, Bitcoin is to blockchain as your app data is to a DBMS.

A cryptocurrency has no physical bills or coins, but it can be used to buy or sell things or services. Also, transactions made using a cryptocurrency don't use brick and mortar institutions for the record keeping. OK, if there are no bills and banks are not involved, how can one party be sure that the other paid for the provided services or goods?

In a blockchain, financial transactions are combined into blocks, which then are validated and linked into a chain. Hence the term *blockchain*. Try visualizing a bicycle chain: Figure 8.1 depicts a blockchain that consists of three blocks (chain links).

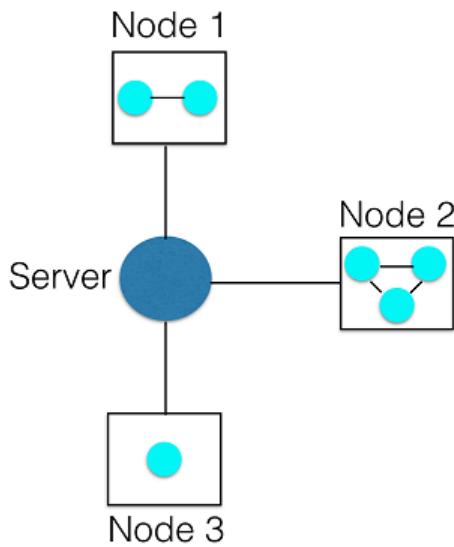


**Figure 8.1 A blockchain of three blocks**

If a record of new transactions has to be added to a blockchain, an app creates a new block, which is given to the blockchain nodes (computers) for validation using a specific to the blockchain algorithm. If the block is valid, it's added to the blockchain, otherwise it's rejected.

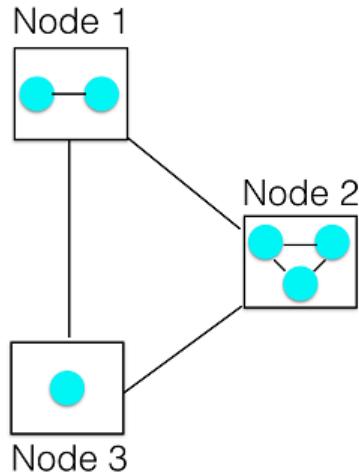
Where is the data about transactions stored and what's the meaning of the word *decentralized* in this context? A typical blockchain is decentralized because no single person or a company controls or owns the data. Being decentralized also means that there is no single point of failure.

Imagine a server that has information about available seats of some airlines. Multiple travel agencies connect to the same server to browse and book the air tickets. Some agencies (nodes) are small and have only one computer connected to the server. Some have two, three computers or even more computers in the same node, but still they all depend on the data from that single server. This is centralized data processing as illustrated in figure 8.2. If the server is down, no one can book air tickets.



**Figure 8.2 Centralized data processing**

In case of decentralized data processing as in most of the blockchains, there is no central data server. Full copies of the blockchain are stored on nodes of a peer-to-peer network, which could include your computer(s) as well if you decide to join a blockchain. A single node doesn't mean one computer, and your may be the owner of the computer cluster that represents one node. Figure 8.3 illustrates decentralized network. If any of the nodes is down, the system remains operational as long as there is at least one node running.



**Figure 8.3 Decentralized data processing**

Is it even secure to store copies of my transactions on multiple computers that belong to some people or organizations? What if one of these computer owners (the bad guy) will modify my transaction changing the paid amount to zero? The good news is that it's not possible. After the block is added to the chain, its data can't be changed - the data in a blockchain are immutable.

Think of a blockchain as a storage where the data is "written in stone". Literally, after a new piece of data is added to the store, you can neither remove nor update it. The insertion of a new

block is allowed only after some node in the network solves a math problem. All this may sound like a Voodoo magic, and the best way to understand how a blockchain works is by building one, which we'll start in next section.

Basically, a blockchain is a decentralized immutable ledger represented by a collection of blocks. Each block can store any types of data, e.g. the information about financial transactions, voting results, medical records et al. Each block is linked to the previous one by storing the hash value of the previous block in the chain. In the next section, we'll give you a mini primer on hashing.

## 8.2 Hashing functions

As per Wikipedia (see [en.wikipedia.org/wiki/Hash\\_function](https://en.wikipedia.org/wiki/Hash_function)), "a hash function is any function that can be used to map data of arbitrary size to data of a fixed size. The values returned by a hash function are called hash values, hash codes, digests, or simply hashes." It also states that hash functions "are also useful in cryptography. A cryptographic hash function allows one to easily verify that some input data maps to a given hash value, but if the input data is unknown, it is deliberately difficult to reconstruct it (or any equivalent alternatives) by knowing the stored hash value."

An encryption is a two-directional function that can take some value and apply a secret key to return the encrypted value. Accordingly, using the same key, the previously encrypted value can be decrypted back.

But a hashing utility use a unidirectional function that applies an algorithm that doesn't allow to reverse the process and reveal the original value. Let's consider a very basic and not overly secure hash function to understand the unidirectional nature of hashes.

A hash function always produces the same hash value if provided the same input value, and complex hashing algorithms are used to minimize the probability that more than one input will provide the same hash value.

Let's say an app has a numbers-only passwords and we don't want to store them in clear in the database. We want to write a hash function that applies the modulo operator to the provided password and adds 10 to it:

### **Listing 8.1 A simplest hash function**

```
function hashMe(password: number): number {
    const hash = 10 + (password % 2);      ①
    console.log(`Original password: ${password}, hashed value: ${hash}`);
    return hash;
}
```

- ① Create a modulo-based hash

For any even number, the expression `input % 2` will produce zero. Now let's invoke the function `hashMe()` several times providing different even numbers as an input parameters:

```
hashMe (2);
hashMe (4);
hashMe (6);
hashMe (800);
```

Each of these invocation will produce the hash value of 10 and the output will look as follows:

```
Original password: 2, hashed value: 10
Original password: 4, hashed value: 10
Original password: 6, hashed value: 10
Original password: 800, hashed value: 10
```

You can see this function in action on CodePen at [codepen.io/yfain/pen/qKwwmo?editors=0111](https://codepen.io/yfain/pen/qKwwmo?editors=0111).

When a hashing function generates the same output for more than one input, it's called *collision*. In cryptography, various Secure Hash Algorithms (SHA) offer more or less secure ways for creating hashes, and they are created to be *collision resistant*, i.e. to make it extremely hard to find two inputs that would produce the same hash value. A block in a blockchain is represented by a hash value, and it's very important that cyber criminals won't be able to replace one block with another by preparing the fraudulent content that produces the same hash as in the legitimate block. Our hash function shown in listing 8.1 has no resistance to collision attacks. Blockchains use collision-resistant hashing algorithms, and one of them is called SHA-256, which takes a string of any length and produces the hash value of the *fixed length* of 256 bits or 64 hexadecimal characters.

Even if you decide to generate a SHA-256 hash for the entire text of this book, its length will be 64 numbers in hexadecimal representation. There are  $2^{256}$  possible combinations of bits in SHA-256 hashes, which is more than a number of grains of sand in the world.

### TIP

### Tip

To learn more about the SHA-256 algorithm, visit Wikipedia at [en.wikipedia.org/wiki/SHA-2](https://en.wikipedia.org/wiki/SHA-2)

For example, to calculate a SHA-256 hash on Linux-based OS, you can use the utility `shasum`. On Windows, you can use the program `certUtil`. There are multiple online SHA-256 generators as well. For creating hashes programmatically, you can use the `crypto` module in Node.js apps or the `crypto` object in modern browsers.

Here's how you can calculate the SHA-256 hash for the text *hello world* on Mac OS:

```
echo -n 'hello world' | shasum -a 256
```

This command produces the following hash:

```
b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
```

No matter how many times you repeat the above command for the string *hello world* you'll always get the same hash value, but changing any character in the input string produces a completely different SHA-256 hash. Applying the functional programming terminology we can say that a hash function is a *pure function* because it always returns the same value for a given input.

**TIP**

**Tip**

If you're interested in hashing methodologies and algorithms, visit the site [www.partow.net/programming/hashfunctions](http://www.partow.net/programming/hashfunctions) or read the Wikipedia article at [en.wikipedia.org/wiki/Hash\\_function](https://en.wikipedia.org/wiki/Hash_function).

We've stated already that blocks in a blockchain are linked using hashes, and in the next section, we'll get familiar with the block internals.

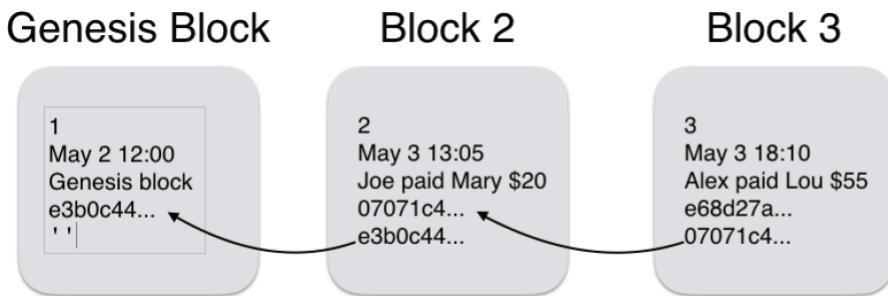
### 8.3 What a block is made out of

You can think of a block as a record in a ledger. While each block in a blockchain contains the app-specific data, it also has a timestamp, its own hash value, and the hash value of the previous block. In a very simple (and easy to hack) blockchain, an app can perform the following actions to add a new block to the chain:

1. Find the hash of the most recently inserted block and store it as a reference to the previous block.
2. Generate a hash value for the newly created block
3. Submit the new block to the blockchain for validation

Let's discuss these steps. At some point, the blockchain was created, and the very first block was inserted into this chain. Obviously, there are no blocks before the first one, and the previous block's hash doesn't exist. The very first block in a chain is called a *genesis block*.

Figure 8.4 shows a sample blockchain, where each block has a unique index, the timestamp, the data, and two hashes - its own and the previous block's. Note that the genesis block has an empty string instead of the previous block's hash.



**Figure 8.4 A sample blockchain**

In figure 8.4 in the genesis block, we used empty quotes to show the lack of the previous block's hash. Our data is represented by the text describing a transaction, e.g. "Joe paid Mary \$20".

**TIP**

**Tip**

You can think of a blockchain as a special type of a linked list where each node has a reference only to the previous one. It's not a classical singly linked list where each node has a reference to the next one.

Now let's imagine that there is a bad guy named Rampage, and he found out that the hash of some block in our blockchain is "e68d27a...". Can he modify its data stating that he paid to Mary \$1000 and regenerate the other blocks so the hash values play well? To prevent this from happening, a blockchain enforces solving algorithms that take time and resources. That's why blockchain members are required to spend time and resources to mine a block rather than quickly generating a hash value.

## 8.4 What's block mining

Everyone understands what's gold mining. The goal is to perform some work to get gold, which can be exchanged for money or goods. The more people mine gold the more gold exists in the world. With real money, the more products are produced the more money government prints.

The cryptocurrency (e.g. Bitcoin) is produced as the incentive for people (a.k.a. miners) who solve math problems required by a particular blockchain. Since every new block has to be approved by other miners, more miners mean more secure blockchain.

In our distributed blockchain, we want to make sure that only the blocks that have certain hashes can be added. For example, our blockchain may require each hash to start with 0000. The hash is calculated based on the content of the block, and it won't start with four zeros unless someone will find an additional value to be added to the block to produce such a hash. Finding such a value is called block mining.

To be added to the blockchain, a new block will be given for processing to all nodes on the network distributed across the world, and these nodes will start working hard trying to calculate

the special value that produces a valid hash. The first one to guess this value wins. Wins what? A blockchain may offer rewards, e.g. in the Bitcoin blockchain, a successful data miner may earn bitcoins.

Let's assume that our blockchain has the requirement that the hash of each block must start with 0000 otherwise the block will be rejected. Say, an app wants to add to our blockchain the new block (number 4) that has the following data:

```
4
June 4 15:30
Simon refunded Al $200
```

Prior to adding any block, its hash must be calculated, so we'll concatenate the above values into one string and generate the SHA-256 hash by running the following command:

```
echo -n '4June 4 15:30 Simon refunded Al $200' | shasum -a 256
```

The generated hash will look as follows:

```
d6f9255c5fc579594bef56403778d475ab441abbd56bfff788d597ae1e8d4ad22
```

This hash doesn't start with 0000, hence it would be rejected by our blockchain. We need to do some data mining trying to find a value that if added to our block, would result in generation of the hash that starts with 0000. Brute force to the rescue! We can write a program that will be adding sequential numbers (1,2,3 et al.) to the end of our input string until the generated hash starts with 0000.

After trying for a while, we found that secret value - it's 236499. Let's append this value to our input string and recalculate the hash:

```
echo -n '4June 4 15:30 Simon refunded Al $200236499' | shasum -a 256
```

Now the generated hash starts with four zeros and looks as follows:

```
0000696c2bde5add287a7b6ccf9a7e57c9d69dad8a6a93922b0451a5150e6696
```

Perfect! We know the number to include in the block's content, so the generated hash will conform to the blockchain requirements. The only problem is that this number can be used just once with this particular input string. Changing any character in the input string will generate a hash that won't start with 0000.

In cryptography, a number that can be used just once is called *nonce* and the value 236499 is such a nonce for our specific block. How about this: We'll add the property called nonce to the block object, and let the data miners calculate its value for each new block?

A miner has to spend some computational resources to calculate the nonce - this is how he or she

will get a *proof of work*, which is a must to have for any block to be considered for adding to the blockchain. We'll write a program to calculate nonces while developing our own little blockchain in the next section. Meanwhile, let's agree on the following structure of a block:

### **Listing 8.2 A sample block type**

```
interface Block {
  index: number;          ①
  timestamp: number;      ②
  data: string;            ③
  nonce: number;           ④
  hash: string;            ⑤
  previousBlockHash: string; ⑥
}
```

- ① A sequential block number
- ② Date and time when the block was added to the blockchain
- ③ Data about one or more app-specific transactions
- ④ A number to be figured out by miners
- ⑤ This block's hash
- ⑥ The hash value of the previous block in the blockchain

Please note that we used the TypeScript `interface` to declare the custom type `Block`. In the next section, we'll see if it should remain an `interface` or become a `class`.

**SIDE BAR**    **Bitcoin mining**

Now that you've seen how the new block can be verified and added to a blockchain, you can understand how the Bitcoin data mining works. Say there is a new transaction in bitcoins between Joe and Mary, and this transaction needs to be added to the bitcoin blockchain (the ledger). This transaction has to be placed into a block and the block has to be verified first.

Anyone who participates in the Bitcoin's blockchain can become a data miner - a person (or a firm) who wants to use his/her hardware to be the first who solved the computationally difficult puzzle. This puzzle will require a lot more computational resources than our puzzle with four zeros.

Overtime, the number of miners grows, the computational resources increase, and the Bitcoin blockchain may increase the difficulty of the puzzle used for block mining. This is done to have the time required to mine a block to remain the same, e.g. ten minutes, which may require finding hashes with 15-20 leading zeros.

The first person (e.g. the bitcoin miner Peter) who will solve the puzzle for a specific transaction (e.g. Joe pays Mary five bitcoins) will earn a newly released bitcoin(s). Peter may also earn money from the transaction fees associated with adding transactions to the bitcoin blockchain. What if Peter likes Mary and decided to commit a fraud increasing the transaction amount from five to fifty? It's not possible, because Joe's transaction will be digitally signed using public-private key cryptography.

Earlier in this section we were concatenating the block number, the time, and a single transaction's text to calculate hash. The bitcoin's blockchain stores blocks with multiple transactions (about 2500) per block.

Bitcoins can be used to pay for services and can be bought and sold using conventional money or other crypto currencies. But the bitcoin mining is the only process that results in releasing new bitcoins into circulation

## SIDE BAR The ledger and cooking the books

Every business has to keep track of its transactions. In the past, these transactions would be manually recorded in a big fat book and then categorized, e.g. sales, purchases etc. These days, such records are stored in files but the concept of a *ledger* remains unchanged. Visit Wikipedia at [en.wikipedia.org/wiki/Ledger](https://en.wikipedia.org/wiki/Ledger), and you'll find the following definitions there:

*The ledger is a permanent summary of all amounts entered in supporting journals which list individual transactions by date. Every transaction flows from a journal to one or more ledgers. A company's financial statements are generated from summary totals in the ledgers.* For our purposes, you can think of a blockchain as a representation of a ledger. Now let's remind you about cooking books.

Investopedia defines an idiom "cook the books" (see [www.investopedia.com/terms/c/cookthebooks.asp](https://www.investopedia.com/terms/c/cookthebooks.asp)) as follows:

*Cook the books is an idiom describing fraudulent activities performed by corporations in order to falsify their financial statements. Typically, cooking the books involves augmenting financial data to yield previously nonexistent earnings.*

Guess what, if the ledger is implemented as a blockchain, cooking the books becomes literally impossible! Actually, there is a slight chance that more than fifty percent of the blockchain node will conspire to approve the illegal modification of the block. This is more of a theoretical possibility than a practical one.

## 8.5 A mini project with hash and nonce

Modern browsers come with the `crypto` object to support cryptography. In particular, you can use the API `crypto.subtle.digest()` to generate hash (see [developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/digest](https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/digest)). We'd like to give you a little assignment to work on. After the assignment, we'll provide you a solution but without any explanations. Try to understand the code on your own.

1. Write the function `generateHash(input: string)` that takes a string input and finds its SHA-256 hash using the browser's `crypto` API.
2. Write the function `calculateHashWithNonce(nonce: number)` that will concatenate the provided nonce with an input string and will invoke `generateHash()`.
3. Write the function `mine()` that will invoke `calculateHashWithNonce()` in a loop until the generated hash starts with `0000`.

It should print the generated hash and the calculated nonce, for example:

"Hash: 0000bfe6af4232f78b0c8eba37a6ba6c17b9b8671473b0b82305880be077edd9, nonce:

107105"

Listing 8.3 presents our solution to this assignment. In this solution, we used the JavaScript keywords `async` and `await`, which are explained in appendix A. Just read the code - you should be able to understand how it works.

### Listing 8.3 The solution to the hash and nonce project

```
import * as crypto from 'crypto';

let nonce = 0;

async function generateHash(input: string): Promise<string> {    ①
  const msgBuffer = new TextEncoder().encode(input);    ②
  const hashBuffer = await crypto.subtle.digest('SHA-256', msgBuffer);    ③
  const hashArray = Array.from(new Uint8Array(hashBuffer));    ④
  const hashHex = hashArray.map(b => ('00' + b.toString(16)).slice(-2)).join('');    ⑤
  return hashHex;
}

async function calculateHashWithNonce(nonce: number): Promise<string> {    ⑥
  const data = 'Hello World' + nonce;
  return generateHash(data);
}

async function mine(): Promise<void> {    ⑦
  let hash: string;
  do {
    hash = await this.calculateHashWithNonce(++nonce);    ⑧
  } while (hash.startsWith('0000') === false);

  console.log(`Hash: ${hash}, nonce: ${nonce}`);
}

mine();
```

- ① This function generates the SHA-256 hash from the provided input
- ② Encode as UTF-8
- ③ Hash the message
- ④ Convert ArrayBuffer to Array
- ⑤ Convert bytes to a hex string
- ⑥ This function adds the nonce to the string and then calculates hash
- ⑦ This function will come up with the nonce that would result in generating a hash that starts with four zeros
- ⑧ We use await because this function is asynchronous

You can see it in action in Codepen at [codepen.io/yfain/pen/gjNqWW?editors=0011](https://codepen.io/yfain/pen/gjNqWW?editors=0011). Initially, the console panel will be empty but after several seconds of work it'll print the following:

Hash: 0000bfe6af4232f78b0c8eba37a6ba6c17b9b8671473b0b82305880be077edd9, nonce: 107105

If you change the code of the method `mine()` replacing four zeros with five, the calculation may take minutes. Try it with ten zeros, and the calculation may take hours.

Calculating the nonce is time consuming but verification is fast. To check if the program from listing 8.3 calculated the nonce 107105 correctly, we used the MAC OS utility `shasum` as follows:

```
echo -n 'Hello World107105' | shasum -a 256
```

This utility printed the same hash as our program:

```
0000bfe6af4232f78b0c8eba37a6ba6c17b9b8671473b0b82305880be077edd9
```

In the method `mine()` in listing 8.3, we hard-coded the required number of zeros to 0000. To make this method more useful, we could add an argument to it, for example:

```
mine(difficulty: number): Promise<void>
```

The value of `difficulty` could be used to represent the number of zeros to be added to the beginning of the hash value. Increasing the `difficulty` would substantially increase the time required to find the nonce.

In the next section, we'll start applying our TypeScript skills and build a simple blockchain app.

## 8.6 Developing your first blockchain

Reading and understanding the materials from the previous section is a prerequisite for understanding the content of this section, where we'll create two blockchain apps: with and without requiring a proof of work.

The first app will create a blockchain and provide an API for adding blocks to it. Prior to adding a block to the chain, we'll calculate a SHA-256 hash for the new block (no algorithm solving) and will store the reference to the hash of the previous block. We'll also add an index, timestamp, and some data to the block.

The second program won't accept blocks with arbitrary hashes, but will require mining to calculating nonces to produce hashes that start with 0000.

Both programs will run from the command line using the `node.js` runtime and will use the module `crypto` (see [nodejs.org/api/crypto.html](https://nodejs.org/api/crypto.html)) for generating SHA-256 hashes.

### 8.6.1 The project structure

Each chapter from Part 2 is a separate project with its own file package.json (dependencies) and the file tsconfig.json (TypeScript compiler's options). You can find the source code of this project at [github.com/yfain/getts](https://github.com/yfain/getts).

Figure 8.5 shows the screenshot taken from the Visual Studio Code IDE after we opened the project that comes with chapter 8, ran `npm install`, and compiled the code with the `tsc` compiler. This project has two apps, and their source code is located in the files `src/bc101.ts` and `src/bc101_proof_of_work.ts`.

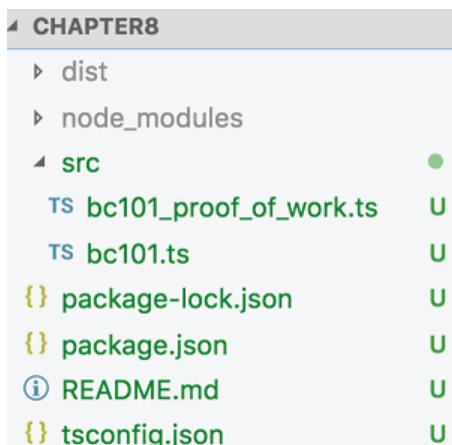


Figure 8.5 The blockchain project structure

Running the `tsc` compiler will create the directory `dist` with the JavaScript code. Running the command `npm install` from the project root directory installs all project dependencies in the directory `node_modules`. This project's dependencies are listed in the file `package.json` shown in listing 8.4:

#### Listing 8.4 chapter8/package.json

```
{
  "name": "chapter8_blockchain",
  "version": "1.0.0",
  "license": "MIT",
  "scripts": {
    "tsc": "tsc"          ①
  },
  "devDependencies": {
    "@types/node": "^10.5.1", ②
    "typescript": "~3.0.0"     ③
  }
}
```

- ① The npm script command to run the locally installed `tsc` compiler
- ② The type definition file for the Node.js
- ③ The TypeScript compiler

The `tsc` compiler is one of the dependencies of this project, and we define a custom npm command to run it in the `scripts` section. Npm scripts allow you to redefine some npm commands or define your own. You can define a command of any name and ask npm to run it by entering `npm run command-name`. As per the content of the `scripts` section of our project, you can run the `tsc` compiler as follows:

```
npm run tsc
```

You may ask, why not just run the `tsc` command from the command line? You can just run the `tsc` command if the `tsc` compiler is globally installed on the computer where you run it. This may be true if you run this command on your computer where you have full control on the globally installed tools. But this may not be the case if your firm has a dedicated team responsible for building and deploying projects on their computers.

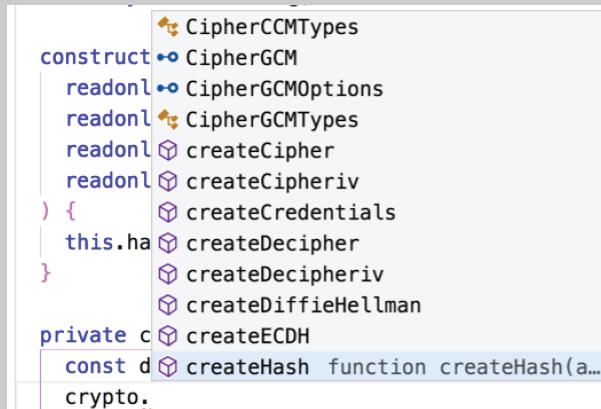
They may require you to provide the application code as well as the build utilities in one package. When you run any program using the `npm run` command, npm will look for the specified program in the `node_modules/bin` directory. In our project, after running `npm install`, the `tsc` compiler will be installed locally in `node_modules/bin`, so our package includes the tooling required to build our app.

**SIDE BAR****A type definition file in action**

We've introduced type definition files in chapter 6. These files include type declarations for public API of the JavaScript code, and in this project, we use the file describing the Node.js API. These files have extensions \*.d.ts.

Type definition files allow the TypeScript type checker to warn you if you're trying to use the API in a wrong way. For example, a function expect a numeric argument and you're trying to invoke this function providing a string one. This wouldn't be possible if you'd be using just the JavaScript library, which has no type annotations. If type definition files for a JavaScript library or a module are present, IDEs can offer context-sensitive help.

Figure 8.6 shows the screenshot taken in the VS Code editor after we entered the word `crypto.`, and the IDE offered context-sensitive help of the API from the module `crypto` that comes with Node.js.



**Figure 8.6 An auto-complete for the crypto module**

If we wouldn't install the type-definition file `@types/node`, we wouldn't get this help. The repository [npmjs.org](https://npmjs.org) has a special `@type` section (a.k.a. organization) that stores thousands of type definition files for popular JavaScript libraries.

Our project also includes the following configuration file `tsconfig.json` file with the TypeScript compiler's options.

## Listing 8.5 chapter8/tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs", ①
    "outDir": "./dist", ②
    "target": "es2017", ③
    "lib": [
      "es2017" ④
    ]
  }
}
```

- ① How to generate code for JavaScript modules
- ② A directory where compiled JavaScript is stored
- ③ Compile into the ES2017 syntax
- ④ This project will use API described in the library es2017

While the compiler's options `outDir` and `target` options are self-explanatory, `module` and `lib` require additional explanations.

Prior to ES6, JavaScript developers were using different syntax for splitting code into modules. For example, the AMD format was popular for the browser-based app, and CommonJS was used by Node.js developers. ES6 introduced `import` and `export` keywords, so a script from one module can import whatever was exported in another.

In TypeScript, we always use ES6 modules, and if a script needs to load some code from a module, we use the `import` keyword. For example, to use the code from the Node.js module `crypto`, we could add the following line to our script:

```
import * as crypto from 'crypto';
```

But the Node.js runtime implements the CommonJS spec for modules, which requires you to write the JavaScript as follows:

```
const crypto = require("crypto");
```

By specifying the compiler's option `"module": "commonjs"`, we instruct `tsc` to turn the `import` statement into `require()`, and all modules members with the `export` qualifier will be added to the `module.exports={...}` construct as prescribed by the CommonJS spec.

TypeScript comes with a set of libraries that describe APIs provided by browsers and JavaScript specs of different versions. You can selectively make these libraries available for your program via compiler's option `lib`. For example if you want to use a `Promise` (introduced in ES2015) in your program, and run it in the target browsers that support promises, you may use the following compiler's option:

```
{
  "compilerOptions": {
    "lib": [ "es2015" ]
  }
}
```

However, the `lib` option includes only *type definitions*, it doesn't provide the actual implementation of the API. Basically, you tell the compiler, "Don't worry when you see `Promise` in my code - my runtime JavaScript engine natively implements this API". But you have to either run your code in the environment that natively supports `Promise` or include a polyfill library that provides `Promise` implementation for older browsers.

**TIP****Tip**

You can find the list of available libraries at [github.com/Microsoft/TypeScript/tree/master/lib](https://github.com/Microsoft/TypeScript/tree/master/lib).

Now that we went through the configuration files of our project, let's review the code in two TypeScript files located in the `src` directory as seen in figure 8.5.

### **8.6.2 Creating a primitive blockchain**

The script `chapter8/src/bc101.ts` is the very first version of our blockchain. It contains classes `Block` and `Blockchain` as well as short script that uses the API of `Blockchain` to create a blockchain of three blocks. In this chapter, we're not going to use web browsers, and our scripts will run under the `Node.js` runtime. Let's review the code of `bc101.ts` starting from the class `Block`.

The class `Block` declares the properties required for each block (e.g. index, hash values of the current and previous blocks) as well as the method to calculate its hash with the help of the `Node.js` module `crypto` that among other things can generate SHA-256 hashes. During the instantiation of the `Block` object, we calculate its hash based on the concatenated values of all its properties.

## Listing 8.6 The class Block from bc101.ts

```

import * as crypto from 'crypto';

class Block {
    readonly hash: string;           ①

    constructor (
        readonly index: number,      ②
        readonly previousHash: string, ③
        readonly timestamp: number,   ④
        readonly data: string        ⑤
    ) {
        this.hash = this.calculateHash(); ⑥
    }

    private calculateHash(): string {
        const data = this.index + this.previousHash + this.timestamp + this.data;
        return crypto
            .createHash('sha256')          ⑦
            .update(data)                ⑧
            .digest('hex');             ⑨
    }
}

```

- ① The hash of this block
- ② The sequential number of the block
- ③ The hash of the previous block
- ④ The time of the block creation
- ⑤ The app-specific data
- ⑥ Calculate the hash of this block on its creation
- ⑦ Create an instance of the Hash object for generating SHA-256 hashes
- ⑧ Compute and update the hash value inside the Hash object
- ⑨ Convert the hash value into a hexadecimal string

The constructor of the class `Block` invokes the method `calculateHash()`, which starts with concatenating the values of the block's properties `index`, `previousHash`, `timestamp`, and `data`. This value is given to the module `crypto` that takes this concatenated sting and calculates its hash in the form of a string of hexadecimal characters. This hash is assigned to the property `hash` of the newly created `Block` object, which will be given to the object `blockchain` for addition to the chain.

The blocks transaction data are stored in the `data` property, which in our class `Block` has the `string` type. Realistically, the `data` property should have some custom type that describes the structure of the data. But in our primitive blockchain, using the `string` type is fine.

Now let's create the class `Blockchain` that uses an array to store blocks and has a method `addBlock()` that does three things:

1. Creates an instance of the `Block` object
2. Gets the hash value of the most recently added block and stores it in the new block's property `previousHash`
3. Adds the new block to the array

When the object `Blockchain` is instantiated, its constructor will create the genesis block, which won't have a reference to the previous block.

### **Listing 8.7 The class Blockchain**

```
class Blockchain {
    private readonly chain: Block[] = [];❶

    private get latestBlock(): Block {❷
        return this.chain[this.chain.length - 1];
    }

    constructor() {
        this.chain.push(❸
            new Block(0, '0', Date.now(),
            'Genesis block'));
    }

    addBlock(data: string): void {
        const block = new Block(❹
            this.latestBlock.index + 1,
            this.latestBlock.hash,
            Date.now(),
            data
        );
❺
        this.chain.push(block);
    }
}
```

- ❶ Our blockchain is stored here
- ❷ The getter to get the reference to the most recently added block
- ❸ Create the genesis block and add it to the chain
- ❹ Create the new instance of the `Block` and populate its properties
- ❺ Add the block to the array

Now we can invoke the method `Blockchain.addBlock()` to mine blocks. The following code creates an instance of `Blockchain` and invokes the method `addBlock()` twice adding two blocks with the data "First block" and "Second block" respectively. The genesis block is created in the constructor of `Blockchain`.

## Listing 8.8 Creating a 3-block blockchain

```
console.log('Creating the blockchain with the genesis block...');  
const blockchain = new Blockchain(); ①  
  
console.log('Mining block #1...');  
blockchain.addBlock('First block'); ②  
  
console.log('Mining block #2...');  
blockchain.addBlock('Second block'); ③  
  
console.log(JSON.stringify(blockchain, null, 2)); ④
```

- ① Create a new blockchain
- ② Add the first block
- ③ Add the second block
- ④ Print the content of the blockchain

The file bc101.ts includes the scripts shown in listings 8.6 - 8.8. To run this script, compile it and run its JavaScript version bc101.js under the Node.js runtime:

```
npm run tsc  
node dist/bc101.js
```

The script bc101.js will print on the console the content of our blockchain as shown in listing 8.9. The chain array stores three blocks of our primitive blockchain.

## Listing 8.9 The console output produced by bc101.js

```
Creating the blockchain with the genesis block...  
Mining block #1...  
Mining block #2...  
{  
  "chain": [  
    {  
      "index": 0,  
      "previousHash": "0",  
      "timestamp": 1532207287077,  
      "data": "Genesis block",  
      "hash": "cc521dd5bbf1786977b14d16ce5d7f8da0e9f3353b3ebe0762ad9258c8ab1a04"  
    },  
    {  
      "index": 1,  
      "previousHash": "cc521dd5bbf1786977b14d16ce5d7f8da0e9f3353b3ebe0762ad9258c8ab1a04",  
      "timestamp": 1532207287077,  
      "data": "First block",  
      "hash": "52d40c33a8993632d51754c952fdb90d61b2c8bf13739433624bbf6b04933e52"  
    },  
    {  
      "index": 2,  
      "previousHash": "52d40c33a8993632d51754c952fdb90d61b2c8bf13739433624bbf6b04933e52",  
      "timestamp": 1532207287077,  
      "data": "Second block",  
      "hash": "0d6d43368772e2bee5da8a1cc92c0c7f28a098bfef3880b3cc8caa5f40c59776"  
    }  
  ]  
}
```

Note that the `previousHash` of each block (except the genesis block) has the same value as the `hash` property of the previous block in the chain. This program runs very fast, but the real-world blockchain would require the data miners to spend some CPU cycles and solve an algorithm so the generated hashes conform to the certain requirements as described in the section "What's data mining is for" earlier. Let's make the block mining process a bit more realistic by introducing a problem solving.

**NOTE****Note**

When you run the script `bc101.js`, you won't see the same hash values as in listing 8.9 because we use timestamp for hash generation, and for each reader it will be different.

### **8.6.3 Creating a blockchain with the proof of work**

The next version of our blockchain is located in the file `bc101_proof_of_work.ts`. This script has a lot of similarities with `bc101.ts`, but it has some extra code to force the data miners to provide the proof of work, so their blocks can be considered for adding to the blockchain.

The script `bc101_proof_of_work.ts` also has the classes `Block` and `Blockchain` but if the latter is exactly the same as seen in listing 8.7, the class `Block` has additional code.

In particular, the class `Block` has the property `nonce` that is calculated in the new method `mine()`. The nonce will be concatenated to other properties of the block to produce the hash that starts with five zeros.

The process of calculating nonce that meets our requirements will take some time, and the method `mine()` will keep invoking `calculateHash()` multiple times with different nonce values until the generated hash starts with 00000. The new version of the class `Block` is shown in listing 8.10.

### Listing 8.10 The class Block from bc101\_proof\_of\_work.ts

```

class Block {
    readonly nonce: number; ①
    readonly hash: string;

    constructor (
        readonly index: number,
        readonly previousHash: string,
        readonly timestamp: number,
        readonly data: string
    ) {
        const { nonce, hash } = this.mine(); ②
        this.nonce = nonce;
        this.hash = hash;
    }

    private calculateHash(nonce: number): string {
        const data = this.index + this.previousHash + this.timestamp + this.data + nonce; ③
        return crypto.createHash('sha256').update(data).digest('hex');
    }

    private mine(): { nonce: number, hash: string } {
        let hash: string;
        let nonce = 0;

        do {
            hash = this.calculateHash(++nonce); ④
        } while (hash.startsWith('00000') === false); ⑤

        return { nonce, hash };
    }
}

```

- ① The new property nonce
- ② Calculate nonce and hash
- ③ Nonce is a part of the input for calculating hash
- ④ Using brute force for data mining
- ⑤ Run this loop until the hash starts with 00000

Note that the method `calculateHash()` is almost identical to the one from listing 13. The only difference is that we append the value of the given nonce to the input string used for calculating hash. The method `mine()` keeps calling `calculateHash()` in a loop providing the sequential numbers 0, 1, 2... as the nonce argument. Sooner or later, the calculated hash will start with 00000 and the method `mine()` will return the hash as well as the calculated nonce. We'd like to bring your attention to the line that invokes the method `mine()`:

```
const { nonce, hash } = this.mine();
```

The curly braces on the left of the equal sign represent JavaScript destructuring (see Appendix A). The method `mine()` returns the object with two properties and we extract their values into two variables `nonce` and `hash`.

We've reviewed only the class `Block` from the script `bc101_proof_of_work.ts` because the rest of the script is the same as in `bc101.ts`. Let's run this program as follows:

```
node dist/bc101_proof_of_work.js
```

This script won't end as fast as `bc101.ts` and may take several seconds to complete as we spend time doing block mining. The output of this script is shown next.

### **Listing 8.11 The console output produced by `bc101_proof_of_work.ts`**

```
Creating the blockchain with the genesis block...
Mining block #1...
Mining block #2...
{
  "chain": [
    {
      "index": 0,
      "previousHash": "0",
      "timestamp": 1532454493124,
      "data": "Genesis block",
      "nonce": 2832,
      "hash": "000005921a5611d92cdc81f89d554743d7e33af2b35b4cb1a0a52cd4664445ca"
    },
    {
      "index": 1,
      "previousHash": "000005921a5611d92cdc81f89d554743d7e33af2b35b4cb1a0a52cd4664445ca",
      "timestamp": 1532454493140,
      "data": "First block",
      "nonce": 462881,
      "hash": "000009da95386579eee5e944b15eab2539bc4ac223398ccef8d40ed83502d431"
    },
    {
      "index": 2,
      "previousHash": "000009da95386579eee5e944b15eab2539bc4ac223398ccef8d40ed83502d431",
      "timestamp": 1532454494233,
      "data": "Second block",
      "nonce": 669687,
      "hash": "0000017332a9321b546154f255c8295e4e805417e50b78609ff59a10bf9c237c"
    }
  ]
}
```

Once again, the `chain` array stores the blockchain of three blocks but this time, each block has a different value in its `nonce` property and the hash value of each block starts with `00000`, which serves as a proof of work: we did the block mining and solved the algorithm for the block!

Let's take another look at the code in listing 8.10 and identify familiar TypeScript syntax elements. Each of the six class properties has a type and is marked as `readonly`. The properties `nonce` and `hash` were explicitly declared on this class, and four more properties were created by the TypeScript compiler because we used the `readonly` qualifier with each argument of the constructor.

Both class methods explicitly declare the types of their arguments and return values. Both methods we declared with the `private` access level, which means that they can be invoked only within the class.

The return type of the method `mine()` is declared as `{ nonce: number, hash: string }`. Since this type is used only once, we didn't create a custom data type for it.

## 8.7 Summary

In this chapter, we've introduced you to the inner workings of blockchains. You've learned that each block is identified by a hash value and is linked to the previous block by storing its hash value. You also know that before inserting a new block to the blockchain, a mathematical problem is offered to each member (a.k.a. node) of the blockchain who's interested in calculating the acceptable hash value for a reward. When we use the term node, we mean a computer, network, or a farm that represents one member of a blockchain.

In this chapter's blockchain, the acceptable hash value had to start with five zeros, and we were calculating a special number called nonce that would ensure that the hash of the block indeed starts with five zeros. Calculating such a nonce takes time, which delays the insertion of the new block to the blockchain, but it can be used as a proof of work needed to award the blockchain node that did it faster than others.

The two apps developed in this chapter were standalone apps that we launched under the Node.js runtime. In the next chapter, we'll create a web app with a UI that's more user-friendly.



# *Developing a browser-based blockchain node*

## ***This chapter covers:***

- Creating a web client to a blockchain
- Creating a small library for hash generation
- Running the blockchain web app and debugging TypeScript in the browser

Just to remind you, in chapter 8, we developed an app that would create an instance of the blockchain and provided the script for adding blocks to it. We launched that app from a command line, and it ran under the Node.js runtime.

In this chapter, we'll review code of the blockchain app that runs in the browser. It has a pretty basic UI as we didn't use any web framework here; we used the standard browser API like `document.getElementById()` or `addEventListener()` for implementing UI.

In this app, each block can store data about several transactions, which won't be simple strings as in chapter 8 but will be implemented as TypeScript custom types. We accumulate several transactions and then create the block to be inserted in the blockchain. Also, we created a small library that contains the code for mining blocks and generating the proper hash, and this library can be re-used in a browser as well as in the Node.js environment.

We'll start by presenting the project structure and running our blockchain web app. After that, we'll do a detailed code review.

## 9.1 Running the blockchain web app

In this section, we'll start by showing you how this blockchain project is configured. Then you'll see the commands to compile and deploy it, and finally, you'll see how the user can work with this app in the browser.

### 9.1.1 The project structure

You can find the source code of this project at [github.com/yfain/getts](https://github.com/yfain/getts), and figure 9.1 shows how this project is structured.

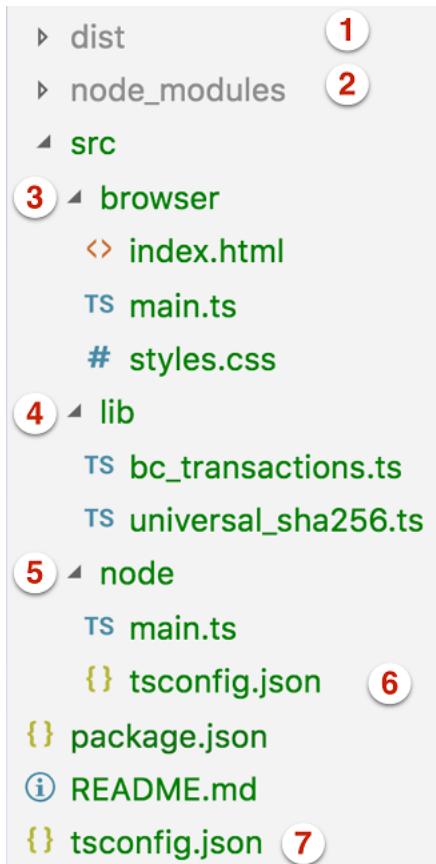


Figure 9.1 Project structure of the blockchain app

1. Compiled code is here
2. Project dependencies are installed here
3. Sources of the web client
4. Sources of reusable hashing library
5. The source code of a standalone client
6. The child TypeScript compiler's config file
7. The main TypeScript compiler's config file

The sources (TypeScript, HTML, and CSS) are located in the subdirectories called browser, lib, and node. The main machinery for creating the blockchain is implemented in the directory lib, but we provided two demo apps - one for the web browser and another for the Node.js runtime.

Here's what subdirectories of src contain:

- The lib directory implements the blockchain creation and block mining. It also has a universal function for generating hashes for both the browser and Node.js environments.
- The browser directory contains the code that implements the UI of our blockchain app that. This code uses the code from the directory lib.
- The node directory is a small demo of how you can run the app that also uses the code from the directory lib.

This project will have some new for you dependencies in the file package.json shown in listing 9.1. Since this is a web app, we need a web server, and in this chapter, we'll use the package called `serve` available on npmjs.org. From deployment perspective, this app will have not only the JavaScript files but also HTML and CSS files that will be copied into the deployment directory dist. The package `copyfiles` will do this job.

Finally, to avoid manual invocations of the `copyfiles` script, we'll add a couple of commands to the npm scripts section of the file package.json. We started using npm scripts in section 8.2.1 in chapter 8, but the `scripts` section in this chapter's package.json file will have more commands.

### **Listing 9.1 The package.json file**

```
{
  "name": "TypeScript_Quickly_chapter9",
  "version": "1.0.0",
  "license": "MIT",
  "scripts": {
    "start": "serve",      ①
    "compileDeploy": "tsc && npm run deploy",  ②
    "deploy": "copyfiles -f src/browser/*.html src/browser/*.css dist"  ③
  },
  "devDependencies": {
    "@types/node": "^10.5.1",
    "serve": "^10.0.1",    ④
    "copyfiles": "^2.1.0",  ⑤
    "typescript": "~3.0.0"
  }
}
```

- ① The npm start command will start the web server
- ② Combining two commands: tsc and deploy
- ③ The deploy command will copy HTML and CSS files
- ④ The package serve is a new dev dependency
- ⑤ The package copyfiles is a new dev dependency

After you run the command `npm install`, all project dependencies will be installed in the `node_modules` directory, and the executables `serve` and `copyfiles` will be installed in `node_modules/.bin`. The sources of this project are located in the directory `src`, while the deployed code will be saved in `dist`.

Note that this project has two tsconfig.json files that are used by the TypeScript compiler. The base tsconfig.json is located in the project root directory, and it usually defines the common-to-project compiler's options, e.g the JavaScript target and which libraries to use. The content of the base tsconfig.json is shown in listing 9.2.

### **Listing 9.2 Base tsconfig.json**

```
{
  "compilerOptions": {
    "sourceMap": true,      ①
    "outDir": "./dist",    ②
    "target": "es5",
    "module": "es6",       ③
    "lib": [
      "dom",               ④
      "es2018"             ⑤
    ]
  }
}
```

- ① Generate the source maps files
- ② Compiled JavaScript files go to the dist dir
- ③ Use the ES6 modules syntax
- ④ Use type definitions for the browser's DOM API
- ⑤ Use type definitions supported by ES2018

For the code that runs in the browser, we want the TypeScript compiler to generate JavaScript that uses modules explained in section A.11 in appendix A. Also, we want to generate the source map files that map the lines in the TypeScript code to the corresponding lines in the generated JavaScript.

With source maps, you can debug your TypeScript code while running the web app in the browser, even though the browser executes JavaScript. We'll show you how to do this in section 9.5. If the browser has the Developer Tools panel open, it loads the source map file along with the JavaScript file, and you can debug your TypeScript code there as if the browser runs TypeScript.

There's also another tsconfig.json in the directory src/node shown in listing 9.3.

### **Listing 9.3 src/node/tsconfig.json**

```
{
  "extends": "../../tsconfig.json",   ①
  "compilerOptions": {
    "module": "commonjs"
  }
}
```

- ① Inherit properties from this file

This file inherits all the properties from tsconfig.json located in the root of our project as specified in the option `extends`. In the base tsconfig.json, the property `module` has a value `es6`, which is fine for generating JavaScript that runs in the browser.

The child configuration file in the node directory, overrides the `module` property to have the value `commonjs`, so the TypeScript compiler will generate module-related code as per the CommonJS rules. In general, tsc will load the base tsconfig.json first, and then the inherited one(s) overriding/adding properties based on its content.

**TIP**

You can find the description of all options allowed in tsconfig.json in the product documentation at [www.typescriptlang.org/docs/handbook/tsconfig-json.html](https://www.typescriptlang.org/docs/handbook/tsconfig-json.html).

### **9.1.2 Deploying the app using npm scripts**

To deploy our web app we need to compile the TypeScript files into the dist directory and copy index.html and styles.css there as well.

The scripts section of package.json (see listing 9.1) has three commands: `start`, `compileDeploy`, and `deploy`.

```
"scripts": {
  "start": "serve", ①
  "compileDeploy": "tsc && npm run deploy", ②
  "deploy": "copyfiles -f src/browser/*.html src/browser/*.css dist" ③
},
```

- ① Start the web server
- ② Runs commands `tsc` and `deploy`.
- ③ copies the HTML and CSS files from the `src/browser` directory to `dist`

The `deploy` command just copies the HTML and CSS files from the `src/browser` directory to `dist`. The `compileDeploy` runs two commands: `tsc` and `deploy`. In npm scripts, double ampersand (`&&`) is used for specifying command sequences. So to compile the TypeScript files and copy index.html and styles.css to the dist directory, we need to run the following command:

```
npm run compileDeploy
```

After running this command the `dist` directory will contain the files shown in figure 9.2.



**Figure 9.2** The files for our web app deployment

**NOTE** The source code of a real-world app contains hundreds files, and prior to deploying them in a web server, we use tools to optimize and bundle the code as a smaller number of files. One of the most popular bundlers is webpack (see [webpack.js.org/](https://webpack.js.org/)).

Now you can start our web server by running the following command:

```
npm start
```

**TIP** npm support a limited number scripts (see [docs.npmjs.com/misc/scripts](https://docs.npmjs.com/misc/scripts)), and `start` is one of them. With these scripts, you don't have to specify use the option `run`. That's why we didn't use the word `run` as in `npm run start`. But custom scripts like `compileDeploy` require the word `run`.

This command will start the web server on the localhost on the port 5000. The console will show the output as in figure 9.3.

```

Serving!

- Local:          http://localhost:5000
- On Your Network: http://10.0.0.6:5000

Copied local address to clipboard!

```

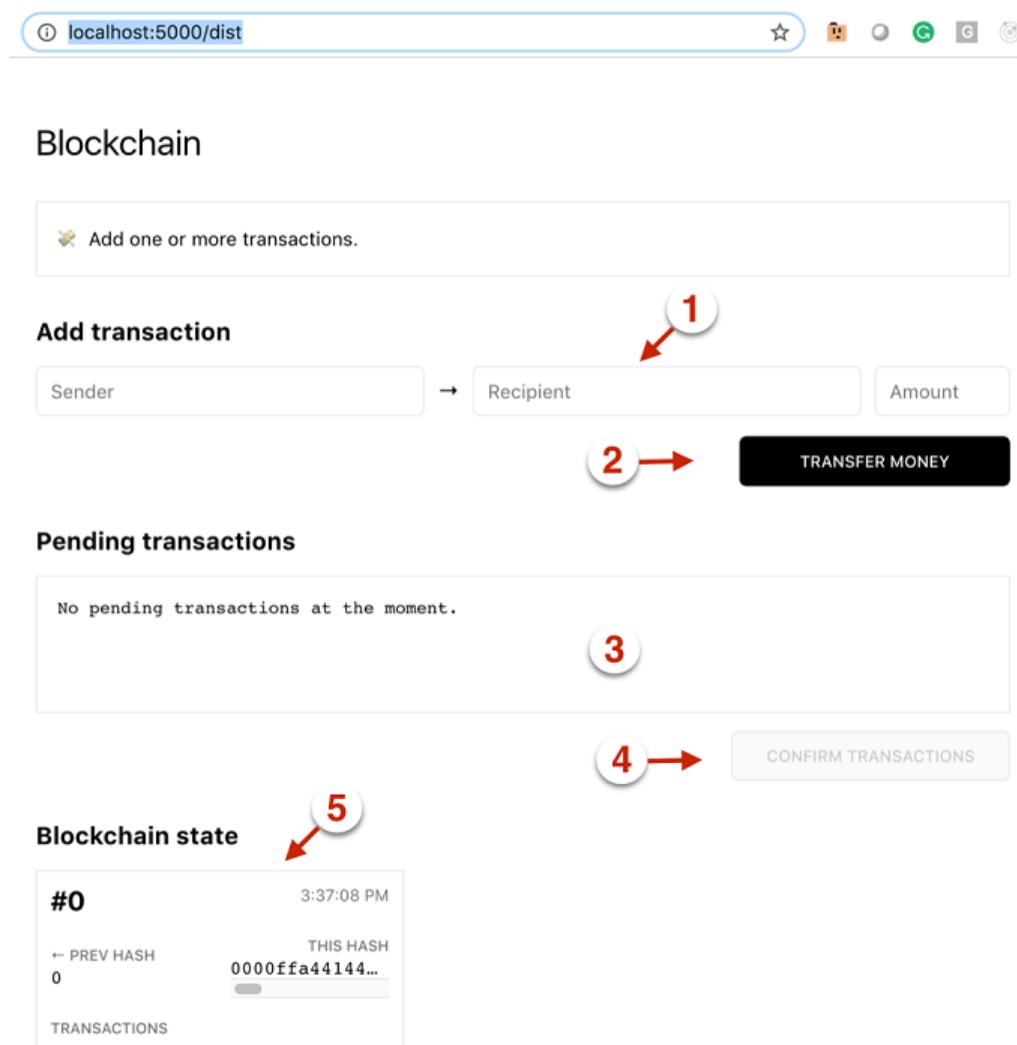
**Figure 9.3** Running the web server

**NOTE**

In chapter 10, we'll develop the app that will have separate code for client and server. There we'll develop and start the server using an npm package called nodemon.

### 9.1.3 Working with the blockchain web app

Open your browser at [localhost:5000/dist](http://localhost:5000/dist), and the web server will send dist/index.html to the browser. The file index.html will load CSS the compiled JavaScript files, and after a couple of seconds spent on genesis block mining, the UI of our blockchain web app will look as seen in figure 9.4.



**Figure 9.4** Running the web server

The landing page of this app shows the initial state of our blockchain with a single genesis block (5). Then, the user can add transactions (1) and click on the button TRANSFER MONEY (2) after each transaction, which will add it to the *Pending transactions* field (3). The button

CONFIRM TRANSACTIONS (4) becomes enabled, and the browser's window will look as in figure 9.5.

The screenshot shows a web-based blockchain application. At the top, the URL bar displays "localhost:5000/dist". Below the header, there is a message box containing a checked checkbox labeled "Ready to mine a new block." Underneath, a section titled "Add transaction" includes fields for "Sender" and "Recipient" with a value of "0", and a "TRANSFER MONEY" button. A "Pending transactions" section lists two entries: "John → Mary: \$200" and "Alex → Bill: \$500". At the bottom right of this section is a large black "CONFIRM TRANSACTIONS" button. Below this, a "Blockchain state" section shows a single block entry: "#0" with a timestamp of "8:51:09 PM". It displays the previous hash ("0") and the current hash ("00009c773601..."). The word "TRANSACTIONS" is visible at the bottom of this section.

**Figure 9.5 Creating pending transactions**

At this point, we added two pending transactions (1) and we neither created nor submitted a new block to our blockchain yet. This is what the button CONFIRM TRANSACTIONS is for. Figure 9.6 shows the browser's window after we clicked on this button, and the app spent some time performing data mining.

**NOTE**

For simplicity, we assume that if John pays Mary a certain amount of money, he has this amount. In real-world apps, the account balance is checked first and only after, the pending transaction is created.

The screenshot shows a web-based blockchain application. At the top, there's a header bar with the URL 'localhost:5000/dist' and several browser icons. Below the header, the title 'Blockchain' is displayed. A button labeled 'Add one or more transactions.' with a plus icon is visible. The main area is titled 'Add transaction' and contains three input fields: 'Sender' (empty), 'Recipient' (empty), and a balance field showing '0'. A large black button labeled 'TRANSFER MONEY' is centered below these fields. Below this section, a heading 'Pending transactions' is shown with a message 'No pending transactions at the moment.' To the right of this message is a button labeled 'CONFIRM TRANSACTIONS'. The final section, 'Blockchain state', displays two blocks. Block #0 has a previous hash of '0' and a current hash of '00009c773601...'. Block #1 has a previous hash of '00009c773601...' and a current hash of '0000366bde92...'. Both blocks have a 'TRANSACTIONS' section. Block #1 lists two transactions: 'John → Mary - \$200' and 'Alex → Bill - \$5'. The entire interface is styled with a clean, modern look with light gray backgrounds and white text.

**Figure 9.6 Adding a new block to the blockchain**

Similarly to what we did in the app from chapter 8, the new block #1 was created (1) with the hash value that starts with four zeros. But now the block includes two transactions: one between John and Mary and the other between Alex and Bill, and it was added to the blockchain. As you see, there are no pending transactions left (2).

You may be wondering, why seeming unrelated transactions are placed in the same block? The reason is that the process of adding a block to the blockchain is a slow operation, and creating a new block for each transaction would slow down the process even more.

To add some context to our blockchain, let's imagine that this blockchain is created for a large real-estate agency. John is buying an apartment from Mary and needs to provide a proof that he paid for it. Similarly, Alex is paying Bill for a house, and this is another transaction. While these transactions are in a pending state, they are not considered as a proof of payment, but when they are added to the block and the block is added to the blockchain, it's consider a done deal.

Now that we've seen the app running, let's review the code starting with UI.

## 9.2 The web client

The browser directory has three files: index.html, main.ts, and styles.css. We'll review the code of the first two files starting with index.html shown in listing 9.4.

## Listing 9.4 browser/index.html

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Blockchain</title>
  <link rel="stylesheet" href="dist/styles.css" style="color: red; font-size: 1em; font-weight: bold; margin-left: 10px;"> ①
  <script type="module" src="dist/browser/main.js"></script> ②
</head>
<body>
  <main>
    <h1>Blockchain</h1>
    <aside>
      <p id="status">Initializing the blockchain, creating the genesis block...</p>
    </aside>

    <section> ③
      <h2>Add transaction</h2>
      <form>
        <input id="sender" type="text" autocomplete="off" disabled placeholder="Sender">
        <span></span>
        <input id="recipient" type="text" autocomplete="off" disabled placeholder="Recipient">
        <input id="amount" type="number" autocomplete="off" disabled placeholder="Amount">
        <button id="transfer" type="button" class="ripple" disabled>
          TRANSFER MONEY ④
        </button>
      </form>
    </section>

    <section> ⑤
      <h2>Pending transactions</h2>
      <pre id="pending-transactions">No pending transactions at the moment.</pre>
      <button id="confirm" type="button" class="ripple" disabled>
        CONFIRM TRANSACTIONS ⑥
      </button>
      <div class="clear"></div>
    </section>

    <section> ⑦
      <h2>Blockchain state</h2>
      <div class="wrapper">
        <div id="blocks"></div>
        <div id="overlay"></div>
      </div>
    </section>
  </main>
</body>
</html>

```

- ① Including CSS
- ② Including the main module of our app
- ③ This section is for adding transactions
- ④ Add transaction to the list of pending ones
- ⑤ Pending transactions are displayed in this section
- ⑥ Start mining a new block with pending transactions
- ⑦ The content of the blockchain is rendered here

The `<head>` section of `index.html`, includes the tags to load `styles.css` and `main.js`. The latter is a

compiled version of main.ts, which is not the only TypeScript file of our app, but since we modularize our app, the script main.ts starts with importing members of another JavaScript module lib/bc\_transactions.ts as seen in listing 9.5.

Note that we use the attribute `type="module"` in the `<script>` tag that loads main.ts. See more on type `module` in section A.11 in appendix A.

### Listing 9.5 The first part of browser/main.ts

```
import { Blockchain, Block } from '../lib/bc_transactions.js'; ①

enum Status { ②
    Initialization = 'Initializing the blockchain, creating the genesis block...',
    AddTransaction = 'Add one or more transactions.',
    ReadyToMine = 'Ready to mine a new block.',
    MineInProgress = 'Mining a new block...'
}

// Get HTML elements ③
const amountEl = document.getElementById('amount') as HTMLInputElement;
const blocksEl = document.getElementById('blocks') as HTMLDivElement;
const confirmBtn = document.getElementById('confirm') as HTMLButtonElement;
const pendingTransactionsEl = document.getElementById('pending-transactions') as HTMLPreElement;
const recipientEl = document.getElementById('recipient') as HTMLInputElement;
const senderEl = document.getElementById('sender') as HTMLInputElement;
const statusEl = document.getElementById('status') as HTMLParagraphElement;
const transferBtn = document.getElementById('transfer') as HTMLButtonElement;
```

- ① Import the classes Block and Blockchain
- ② Declare possible statuses of the app
- ③ Get references to all important HTML elements

In chapter 4, we explained enums, which are named constants. In listing 9.5, we use enums to declare a finite number of statuses of our app: `Initialization`, `AddTransaction`, `ReadyToMine`, and `MineInProgress`. The constant `statusEl` represents the HTML element where we want to display the current status of the app.

#### NOTE

These little icons that you see as a part of the enum strings are emoji symbols. You can insert them into strings on MAC OS by pressing CMD-CTRL-Space and on Window 10 by pressing Win and a period or Win and semicolon.

The rest of the listing 9.5 shows the code that gets the references to various HTML elements that either store the values entered by the user or display the blocks of our blockchain. Each of these HTML elements has a unique id (see listing 9.4), and we use the browser's API `getElementById()` to get a hold of these DOM objects.

Listing 9.6 shows the immediately invoked function expression (IIFE) `main()`, and we use `async/await` keywords here (see section A.10.4 in appendix A). This function creates a new

blockchain with the initial genesis block and assigns event listeners to the buttons that initiate adding pending transactions and a new block mining.

### **Listing 9.6 The second part of browser/main.ts**

```
(async function main(): Promise<void> {
    transferBtn.addEventListener('click', addTransaction);      ①
    confirmBtn.addEventListener('click', mineBlock);      ①

    statusEl.textContent = Status.Initialization;      ②

    const blockchain = new Blockchain();      ③
    await blockchain.createGenesisBlock();      ④
    blocksEl.innerHTML = blockchain.chain.map((b, i) => generateBlockHtml(b, i)).join('');      ⑤

    statusEl.textContent = Status.AddTransaction;
    toggleState(true, false);

    function addTransaction() {      ⑥
        blockchain.createTransaction({
            sender: senderEl.value,
            recipient: recipientEl.value,
            amount: parseInt(amountEl.value),
        });
    }

    toggleState(false, false);
    pendingTransactionsEl.textContent = blockchain.pendingTransactions.map(t =>
        `${t.sender} ${t.recipient}: ${t.amount}`).join('\n');      ⑦
    statusEl.textContent = Status.ReadyToMine;

    senderEl.value = '';      ⑧
    recipientEl.value = '';      ⑧
    amountEl.value = '0';      ⑧
}

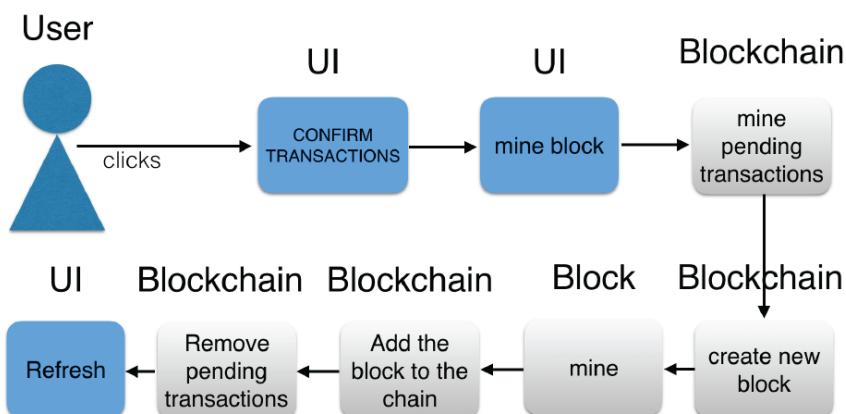
async function mineBlock() {      ⑨
    statusEl.textContent = Status.MineInProgress;
    toggleState(true, true);
    await blockchain.minePendingTransactions();      ⑩

    pendingTransactionsEl.textContent = 'No pending transactions at the moment.';
    statusEl.textContent = Status.AddTransaction;
    blocksEl.innerHTML = blockchain.chain.map((b, i) => generateBlockHtml(b, i)).join('');      ⑪
    toggleState(true, false);
}
})();
```

- ① Add event listeners to buttons
- ② Show the initial status using enum
- ③ Create an instance of Blockchain
- ④ Create the genesis block
- ⑤ Generate HTML for rendering block(s)
- ⑥ Add a new pending transaction
- ⑦ Render pending transactions as strings
- ⑧ Reset the form's values
- ⑨ Mine the block and render it on the web page

- ⑩ Create a new block, calculate hash, and add it to the blockchain
- ⑪ Render the newly inserted block on the web page

In listing 9.6, we use the class `Blockchain` (reviewed in section 9.3) that has an array called `chain`, which stores the content of our blockchain. The class `Blockchain` has a method `minePendingTransactions()` that adds them to the new block. This mining process starts when the user clicks on the button `CONFIRM TRANSACTIONS`. In a couple of places, we invoke `blockchain.chain.map()` to convert the blocks into the text or HTML elements for rendering on the web page. This workflow is shown in figure 9.7.



**Figure 9.7 The user-blockchain workflow**

Whenever we invoke an asynchronous function, we change the content of the HTML element `statusEl` to keep the user informed on the current status of the web app. Listing 9.7 shows the remaining part of `main.ts`, which contains two functions: `toggleState()` and `generateBlockHtml()`.

### Listing 9.7 The third part of browser/main.ts

```

function toggleState(confirmation: boolean, transferForm: boolean): void { ①
  transferBtn.disabled = amountEl.disabled = senderEl.disabled = recipientEl.disabled = transferForm;
  confirmBtn.disabled = confirmation;
}

function generateBlockHtml(block: Block, index: number) { ②
  return `
    <div class="block">
      <span class="block__index">#${index}</span>
      <span class="block__timestamp">${new Date(block.timestamp).toLocaleTimeString()}</span>
      <div class="prev-hash">
        <div class="hash-title"> PREV HASH</div>
        <div class="hash-value">${block.previousHash}</div>
      </div>
      <div class="this-hash">
        <div class="hash-title">THIS HASH</div>
        <div class="hash-value">${block.hash}</div>
      </div>
      <div class="block__transactions">
        <div class="hash-title">TRANSACTIONS</div>
        <pre class="transactions-value">${block.transactions.map(t => `${t.sender} ${t.recipient} - ${t.value}`)}</pre>
      </div>
    `;
}

```

- ① Disable/enable the form and the confirm button
- ② Generate block's HTML

The function `toggleState()` has two boolean parameters. Depending on the value of the first parameter, we either enable or disable the form, which consists of the input fields Sender, Recipient, Amount and the button TRANSFER MONEY. The second parameter enables or disables the button CONFIRM TRANSACTIONS.

The function `generateBlockHtml()` returns the `<div>` container with the information on each block as seen at the bottom of figure 9.6. We use several CSS `class` selectors, which are defined in the file `browser/styles.css` (not shown here).

To summarize, the code in the script `browser/main.ts` is responsible for the user interaction, and it performs the following:

1. Gets access to all HTML elements
2. Sets the listeners for the buttons to add pending transactions and create a new block
3. Creates the instance of the blockchain with the genesis block
4. Defines the method to create pending transactions
5. Defines the method to mine the new block with pending transactions
6. Defines a method to generate HTML for rendering blocks of the blockchain

Now that we understand how the browser's UI is implemented, let's review the code that creates the blockchain, adds the blocks and handles hashes.

## 9.3 Mining blocks

Our project contains the lib directory, which is a small library that supports creating blocks with multiple transactions. Also, this library checks if our blockchain app runs in the browser or in the standalone JavaScript engine so the appropriate API for generating SHA-256 hashes is used. This library consists of two files:

- bc\_transactions.ts - implements creation of blocks with transactions
- universal\_sha256.ts - checks the environment and exports the proper function `sha256()` that will use either the browser's or Node's API for generating SHA-256 hashes.

Revisit the code that uses Node's `crypto` API in listing 8.6 in chapter 8 - that code synchronously invokes three functions:

```
crypto.createHash('sha256').update(data).digest('hex');
```

Now we'd like to be able to generate hashes in Node.js as well as in browsers. But the browser's `crypto` API is `async` unlike the package we use with Node.js, hence the hash-generation code should be wrapped into asynchronous functions. For example, the function `mine()` that invokes `crypto` API returns a `Promise` and is marked as `async`:

```
async mine(): Promise<void> { ... }
```

Basically, instead of simply returning the value from a function, we'll wrap it into a `Promise`, which will make the function `mine()` asynchronous.

We'll show you the code of `bc_transactions.ts` in two parts. Listing 9.8 shows the code that imports the SHA-256 generating function, declares the `Transaction` interface and the class `Block`. A block may contain more than one transaction, and the interface `Transaction` declares the structure of one transaction.

## Listing 9.8 The first part of lib/bc\_transactions.ts

```

import {sha256} from './universal_sha256.js'; ①

export interface Transaction { ②
  readonly sender: string;
  readonly recipient: string;
  readonly amount: number;
}

export class Block { ③
  nonce: number = 0;
  hash: string;

  constructor (
    readonly previousHash: string,
    readonly timestamp: number,
    readonly transactions: Transaction[] ④
  ) {}

  async mine(): Promise<void> { ⑤
    do { ⑥
      this.hash = await this.calculateHash(++this.nonce);
    } while (this.hash.startsWith('0000') === false);
  }

  private async calculateHash(nonce: number): Promise<string> { ⑦
    const data = this.previousHash + this.timestamp + JSON.stringify(this.transactions) + nonce;
    return sha256(data); ⑧
  }
}

```

- ① Importing the function for hash generation
- ② A custom type representing a single transaction
- ③ A custom type representing a single block
- ④ While a Block instance, pass an array of transactions
- ⑤ The asynchronous function to mine the block
- ⑥ Use the brute force to find the proper nonce
- ⑦ The asynchronous wrapper function for hash generation
- ⑧ Invoking the function that uses crypto API and generates hash

The UI of our blockchain allows the user to create several transactions using the button TRANSFER MONEY (see figure 9.5), and only the click on the button CONFIRM TRANSACTIONS creates the `Block` instance passing the `Transaction` array to its constructor.

In chapter 2, we showed you how to declare TypeScript custom types using classes and interfaces. In listing 9.8, you see both: the type `Transaction` is declared as an interface, but the type `Block` as a class. We couldn't make `Block` an interface, because we wanted it to have a constructor, so we could use the `new` operator as in listing 9.9.

The `Transaction` type is a TypeScript interface, which will prevent us from making type errors

during development, but the lines that declare `Transaction` won't make it into the compiled JavaScript.

Our `Transaction` type is rather simple, but in the real-world blockchain, we could introduce more properties to the interface `Transaction`. For example, buying real estate is a multi-step process that requires multiple payments over time, e.g. initial deposit, payment for the search that there is no lien on the house, payment for the property insurance et al. If this would be a real-estate blockchain, we could introduce `propertyID` to identify the property (e.g. house, land, apartment) and the `type` to identify the transaction type.

But we'd like to stress that every transaction will include properties that describe the business domain of a particular blockchain - you'll always have a transaction type. A block would also need properties required by a blockchain implementation, e.g. creation date, hash value, previous block's hash value, et al. Also, a block type may include some utility methods. Our class `Block` from listing 9.8 has the methods `mine()` and `calculateHash()`.

**NOTE**

We already had the class `Block` in chapter 8 (see listing 8.6), which stored its data as a `string` in the property `data`. This time, the data is stored in a more structured way in a property of type `Transaction[]`. By the way, have you noticed that three of the properties of the class `Block` were declared implicitly via the constructor's arguments?

Our class `Block` has two methods: `mine()` and `calculateHash()`. The method `mine()` keeps increasing the nonce and invoking `calculateHash()` until it returns the hash value that starts with four zeros.

As per the signature of the function `mine()`, it returns a `Promise`, but where's a `return` statement of this function? We don't want this function to return anything - the loop will simply end when the proper hash is generated. The thing is that any function marked with the `async` keyword must return a `Promise`, which is a generic type (explained in chapter 4) and must be used with a type parameter. By using `Promise<void>` we specify that this function returns a `Promise` with an empty value hence the `return` statement is not required.

Since the method `calculateHash()` shouldn't be used by the external to `Block` scripts, we declared it as `private`. This function invokes `sha256()`, which takes a `string` and generates hash. Note that we use `JSON.stringify()` to turn an array of type `Transaction` into a `string`. The function `sha256()` is implemented in the script `universal_sha256.ts`, and we'll discuss it in section 9.4.

**NOTE**

For simplicity, in our function `calculateHash()`, we concatenate multiple transactions into a string and then calculate hash. In the real-world blockchains, a more efficient algorithm called Merkle Tree (see [en.wikipedia.org/wiki/Merkle\\_tree](https://en.wikipedia.org/wiki/Merkle_tree)) is used for calculating hashes of multiple transactions. Using this algorithm, a program can build a tree of hashes (one per two transactions), and if someone will try to tamper with one transaction, there is no need to traverse all transactions to recalculate/verify the final hash.

The first line in listing 9.8 imports from the JavaScript file `./universal_sha256.js` even though the directory `lib` has only the TypeScript version of this file. TypeScript doesn't allow using the `.ts` extensions in referencing imported file names. This ensures that references to external scripts won't change after the compilation where all files have extensions `.js`. This import statement looks as if we import a single function `sha256()`, but under the hood, it'll import different functions that uses different crypto API depending on the environment the app runs in. We'll show you how it's done in section 9.4 while reviewing the code of `universal_sha256.ts`.

In the second part of the file `bc_transactions.ts`, we declare the class `Blockchain` as seen in listing 9.9.

## Listing 9.9 The second part of lib/bc\_transactions.ts

```

export class Blockchain {
    private readonly _chain: Block[] = [];
    private _pendingTransactions: Transaction[] = [];

    private get latestBlock(): Block { ①
        return this._chain[this._chain.length - 1];
    }

    get chain(): Block[] { ②
        return [ ...this._chain ];
    }

    get pendingTransactions(): Transaction[] { ③
        return [ ...this._pendingTransactions ];
    }

    async createGenesisBlock(): Promise<void> { ④
        const genesisBlock = new Block('0', Date.now(), []);
        await genesisBlock.mine(); ⑤
        this._chain.push(genesisBlock); ⑥
    }

    createTransaction(transaction: Transaction): void { ⑦
        this._pendingTransactions.push(transaction);
    }

    async minePendingTransactions(): Promise<void> { ⑧
        const block = new Block(this.latestBlock.hash,
            Date.now(), this._pendingTransactions); ⑨
        await block.mine(); ⑩
        this._chain.push(block);
        this._pendingTransactions = [];
    }
}

```

- ① The getter for the latest block in the blockchain
- ② The getter for all blocks in the blockchain
- ③ The getter for all pending transactions
- ④ Creating the genesis block
- ⑤ Creating the hash for the genesis block
- ⑥ Adding the genesis block to the chain
- ⑦ Adding a pending transaction
- ⑧ Turning pending transaction into a block and add it to the blockchain
- ⑨ Creating the hash for the new block
- ⑩ Add the new block to the blockchain

Adding a new block to the blockchain takes time. For example, Bitcoin keeps this time around 10 minutes by controlling the complexity of the algorithm to solve by the blockchain nodes. It's done on purpose to prevent double-spending attacks (see the sidebar), so creating a new block for each transaction would make any blockchain extremely slow. That's why a block can contain

multiple transactions, and we accumulate pending transactions in the property `pendingTransactions`, and then create a new block that stores all of them. For example, one Bitcoin block contains about 2500 of transactions.

## SIDE BAR

### Double spending attacks

Let's say you have only two dollar bills in your pocket and want to buy a cup of coffee that cost \$2. You handed two dollar bills to the barista and he gave you the coffee, and you have no money in your pocket, which means that you can't buy anything anymore. You can spend your dollars only once unless you steal the money. Counterfeiting money also takes time and can't be done on the spot in the coffee shop.

The digital currency is just a file that's easier to duplicate. For example, a dishonest person may try to use the same amount of money to different recipients. For example, Joe has only one Bitcoin and pays it to Mary (creates a block with the transaction "Joe Mary 1"), and immediately pays one Bitcoin to Alex (creates another block with the transaction "Joe Alex 1"). This is an example of a *double-spending* attack.

If the Bitcoin blockchain would allow immediate insertion of new blocks, double-spending would be quite possible. Forcing the nodes to spend a long time calculating hash (proof of work) and get multiple block approvals from other nodes helps in solving the double-spending problem.

When the user keeps adding transactions using the UI shown in figure 9.5, for each transaction, we invoke the method `createTransaction()` that adds one transaction to the array `pendingTransactions`. When the new block is added to the blockchain, we remove all pending transactions from this array at the end of the method `minePendingTransactions()`.

Note that we declared the class variable `_pendingTransactions` as `private`, which can be modified only via the method `createTransaction()`. We also provided a public getter that returns an array of pending transactions that looks like this:

```
get pendingTransactions(): Transaction[] {
  return [ ...this._pendingTransactions ];
}
```

This method has just one line that creates a clone of the `_pendingTransactions` array using the JavaScript spread operator explained in section A.7 in appendix A. By creating a clone, we provide a copy of the transactions' data. Also, each property of the interface `Transaction` is `readonly`, so any attempt to modify the data in this array will result in TypeScript error. This getter is used in the file `browser/main.ts` that displays pending transactions on the UI as seen in listing 9.6.

The same cloning technique is used with the getter `chain()` that returns the clone of the blockchain.

Creation of the genesis block is done by invoking the method `createGenesisBlock()`, and the script `browser/main.ts` does it (see listing 9.6). The genesis block has an empty array `transactions`, and invoking `mine()` on this block calculates its hash. Mining the block may take some time and we don't want the UI to freeze hence this method is asynchronous. Also, we added `await` to the invocation of the method `mine()` to ensure that the block is added to the blockchain only after the mining is complete.

Our blockchain is created for the educational purpose, so every time the user launches our app, the blockchain will contain only the genesis block. Then the user starts creating pending transactions, and at some point, she clicks on the button CONFIRM TRANSACTIONS. This will result in the invocation of the method `minePendingTransactions()`, which creates a new `Block` instance, calculates its hash, adds this block to the blockchain, and resets the array `_pendingTransactions`.

**TIP**

After mining a new block the miner should be rewarded. We'll take care of this in the blockchain app in chapter 10.

You may want to take another look at the code of the method `mineBlock()` from listing 9.6 to have a better understanding of how the results of block mining are rendered on the UI.

Being that mining of the genesis block or a block with transactions, this process invokes the method `mine()` defined in the class `Block` (see listing 9.8). The method `mine()` runs a loop invoking `calculateHash()`, which in turn invokes `sha256()` discussed next.

## 9.4 Using crypto APIs for hash generation

Since we wanted to create a blockchain that could be used by web and standalone apps, we need to use two different crypto APIs for generating SHA-256 hashes:

- For web apps, we can invoke API of the `crypto` object supported by all browsers.
- The standalone apps will run under the `node.js` runtime that comes with the module `crypto` (see [nodejs.org/api/crypto.html](https://nodejs.org/api/crypto.html)), and we'll use it for generating hashes just like we did it in chapter 8 in listing 8.6.

But we want our little library to make the decision which API to use during runtime, so the client apps will just use one function without knowing which specific crypto API is used. The file `lib/universal_sha256.ts` shown in listing 9.10 declares three functions: `sha256_node()`, `sha256_browser()`, and `sha256()`. Note that only the last one is exported.

## Listing 9.10 lib/universal\_sha256.ts

```

function sha256_node(data: string): Promise<string> {    ①
  const crypto = require('crypto');
  return Promise.resolve(crypto.createHash('sha256').update(data).digest('hex'));  ②
}

async function sha256_browser(data: string): Promise<string> {  ③
  const msgUint8Array = new TextEncoder().encode(data);   ④

  const hashByteArray = await crypto.subtle.digest('SHA-256',
msgUint8Array);  ⑤

  const hashArray = Array.from(new Uint8Array(hashByteArray));  ⑥

  const hashHex = hashArray.map(b => ('00' + b.toString(16)).slice(-2)).join('');  ⑦
  return hashHex;
}

export const sha256 = typeof window === "undefined" ?  ⑧
  sha256_node : ⑨
  sha256_browser; ⑩

```

- ① The function to be used in Node.js runtime
- ② Generate SHA-256 hash
- ③ The function to be used in browsers
- ④ Encode the provided string as UTF-8
- ⑤ Hash the data
- ⑥ Convert the ArrayBuffer to Array
- ⑦ Convert bytes to the hexadecimal string
- ⑧ Check if the runtime has a global variable window
- ⑨ Export the hashing function for Node
- ⑩ Export the hashing function for browsers

You know from appendix A that when a JavaScript file contains import or export statements, it becomes an ES6 module. The module `universal_sha256.ts` declares the functions `sha256_node()` and `sha256_browser()` but doesn't export them. These functions become private and can be used only inside the module.

The function `sha256()` is the only one that is exported and can be imported by other scripts. This function has a very simple mission - to find out if this module runs in the browser or not. Depending on this, the we want to export either `sha256_browser()` or `sha256_node()` but under the name `sha256()`.

We came up with a simple solution: if the runtime environment has a global variable `window`, we assume that this code runs in the browser and export `sha256_browser()` under the name

`sha256()`. Otherwise we export `sha256_node()` under the same name `sha256()`. In other words, this is an example of a dynamic export.

We already used the Node.js crypto API in chapter 8, but this time we wrapped this code in a `Promise`:

```
Promise.resolve(crypto.createHash('sha256').update(data).digest('hex'));
```

We did it to align the signatures of the functions `sha256_node()` and `sha256_browser()`.

The function `sha256_browser()` runs in the browser and uses asynchronous crypto API, which makes this function asynchronous as well. As per the browser's crypto API requirements, we start by using the Web API `TextEncoder.encode()` (see [developer.mozilla.org/en-US/docs/Web/API/TextEncoder](https://developer.mozilla.org/en-US/docs/Web/API/TextEncoder)) that encodes the string as UTF-8 and returns the result in a special typed JavaScript array of unsigned 8-bit integers (see [developer.mozilla.org/en-US/docs/Web/JavaScript/Typed\\_arrays](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays)).

Then, the browser's crypto API generates hash in the form of an array-like object (an object with the `length` property and indexed elements). After that, we use the method `Array.from()` to create a real array. Figure 9.8 shows a snapshot taken in the Chrome's debugger. It shows fragments of data in the variables `hashByteArray` and `hashArray`. The breakpoint was placed right before calculating the value of the variable `hashHex`.

```

▼ hashByteArray: ArrayBuffer(32)
  ► [[Int8Array]]: Int8Array(32) [-115, 59, -7, 33, -81, 116, 44, 110, -75, -109,
    ► [[Int16Array]]: Int16Array(16) [15245, 8697, 29871, 28204, -27723, 23200, -13
    ► [[Int32Array]]: Int32Array(8) [569981837, 1848407215, 1520473013, 1086638873,
    ► [[Uint8Array]]: Uint8Array(32) [141, 59, 249, 33, 175, 116, 44, 110, 181, 147
      byteLength: (...)
```

1

```

  ► __proto__: ArrayBuffer
▼ hashArray: Array(32) ← 2
  0: 141
  1: 59
  2: 249
  3: 33
  4: 175
  5: 116
  6: 44
  7: 110
  8: 181
  9: 147
```

2

**Figure 9.8 Variables `hashByteArray` and `hashArray` in the debugger**

1. A special JavaScript typed array provides access to binary raw data
2. The `ArrayBuffer` is converted into `Array`

We'll explain how to debug your TypeScript code in the browser in section 9.6.

Finally, we want to turn the calculated hash into a string of hexadecimal values, and this is done in the following statement:

```
const hashHex = hashArray.map(
    b => ('00' + b.toString(16)).slice(-2))
    .join('');
```

We want to convert each element of the `hashArray` into a hexadecimal value and use the `Array.map()` method for this. Some hexadecimal values are represented by one character, while others need two. To ensure that a one-character values are prepended with a zero, we concatenate `00` and the hexadecimal value and then use `slice(-2)` to take just the two characters from the right. For example, the hexadecimal value `a` becomes `00a` and then `0a`.

The method `join('')` concatenates all converted elements of `hashArray` into a string `hashHex` specifying an empty string as a separator, i.e. no separator. Figure 9.9 shows a fragment of the result of applying `map()` and the final hash in the variable `hashHex`.

The diagram illustrates the mapping process. At the top, the variable `hashHex` is shown as a string: "8d3bf921af742c6eb593a05a19cbc440a595a98ce34be5229a520a57ede9a8ea". Below it, the expression `hashArray.map(b => ('00' + b.toString(16)).slice(-2))` is shown. A red arrow points from the first character of the string "8" down to the first element of the array, which is "8d". Another red arrow points from the number "1" down to the same array element, indicating the index. The array itself is shown with indices 0 through 12, containing the following values: "8d", "3b", "f9", "21", "af", "74", "2c", "6e", "b5", "93", "a0", "5a", and "19".

**Figure 9.9 Applying map() and join()**

1. The elements of the `hashArray` are turned into a string `hashHex`

You may ask, "How come the signature of the function `sha256_browser()` declares the return type `Promise` but it actually returns a string?" Since we declare this function as `async`, it automatically wraps its returned value into a `Promise`.

By now, you should understand the code in the `lib` directory and how the web client uses it. The last piece to review is a standalone client that can be used instead of the web client.

## 9.5 The standalone blockchain client

The directory node of this project has a little script that can create a blockchain without the web UI, but we wanted to show you that the code in the lib directory is reusable, and if the app runs under Node.js, the proper crypto API will be engaged.

### **Listing 9.11 node/main.ts**

```
import { Blockchain } from '../lib/bc_transactions';

(async function main(): Promise<void> {
    console.log('Initializing the blockchain, creating the genesis block...');

    const bc = new Blockchain();      ①
    await bc.createGenesisBlock();    ②

    bc.createTransaction({ sender: 'John', recipient: 'Kate', amount: 50 });  ③
    bc.createTransaction({ sender: 'Kate', recipient: 'Mike', amount: 10 });  ③

    await bc.minePendingTransactions();  ④

    bc.createTransaction({ sender: 'Alex', recipient: 'Rosa', amount: 15 });  ⑤
    bc.createTransaction({ sender: 'Gina', recipient: 'Rick', amount: 60 });

    await bc.minePendingTransactions();  ④

    console.log(JSON.stringify(bc, null, 2));  ⑤
})();
```

- ① Create a new blockchain
- ② Create the genesis block
- ③ Create a pending transaction
- ④ Create a new block and add it to the blockchain
- ⑤ Print the content of the blockchain

This program starts with printing the message that creating of the new block chain is in progress. This process of creating the genesis block can take some time, and the first `await` waits for it to complete.

What would happen if we didn't use `await` in the line that invokes `bc.createGenesisBlock()`? The code would proceed with mining blocks before the genesis block was created, and the script would fail with runtime errors.

After the genesis block is created, the script proceeds with creating two pending transactions and mining a new block. Then again `await` will wait for this process to complete, and then we create two more transactions and mine another block. Finally, this program prints the content of the blockchain.

#### **TIP**

Node.js supports `async` and `await` starting from version 8.

We'd like to remind you that the directory node has its own file `tsconfig.json` as seen in figure 9.1. Its content is shown in listing 9.3.

To launch this program, you need to make sure that the code is compiled by running `tsc`. It's important that you run `tsc` from the `src/node` directory to ensure that the compiler picks up the option from all the hierarchy of the `tsconfig.json` files. As per the base `tsconfig.json`, the compiled code will be saved in the `dist` directory.

**TIP**

The compiler's option `-p` allows you to specify the file path to a valid JSON configuration file. For example, you can compile the TypeScript code by running the following command: `tsc -p src/node/tsconfig.json`.

Now you can ask Node.js to launch the JavaScript version of the code from listing 9.11. If you're still in the directory `src/node`, you can run our app as follows:

```
node ../../dist/node/main.js
```

Below is the console output of this command.

## Listing 9.12 Creating the blockchain in a standalone app

```

❶ Initializing the blockchain, creating the genesis block...
{
  "_chain": [
    {
      ❷
      "previousHash": "0",
      "timestamp": 1540391674580,
      "transactions": [],
      "nonce": 239428,
      "hash": "0000d1452c893a79347810d1c567e767ea55e52a8a5ffc9743303f780b6c308f"
    },
    {
      ❸
      "previousHash": "0000d1452c893a79347810d1c567e767ea55e52a8a5ffc9743303f780b6c308f",
      "timestamp": 1540391675729,
      "transactions": [
        {
          "sender": "John",
          "recipient": "Kate",
          "amount": 50
        },
        {
          "sender": "Kate",
          "recipient": "Mike",
          "amount": 10
        }
      ],
      "nonce": 69189,
      "hash": "00006f79662bde59ff46cd57cff928977c465d931b2ba2d11e05868afcfee836"
    },
    {
      ❹
      "previousHash": "00006f79662bde59ff46cd57cff928977c465d931b2ba2d11e05868afcfee836",
      "timestamp": 1540391676138,
      "transactions": [
        {
          "sender": "Alex",
          "recipient": "Rosa",
          "amount": 15
        },
        {
          "sender": "Gina",
          "recipient": "Rick",
          "amount": 60
        }
      ],
      "nonce": 33462,
      "hash": "0000483b745526f48afde33435c21517dd72ea0a25407bc35be3f921029a3209"
    }
  ],
  "_pendingTransactions": [] ❺
}

```

- ❶ The genesis block
- ❷ The second block
- ❸ The third block
- ❹ The array of pending transactions is empty

While the web version of this app is more interactive, its standalone version runs the entire process in the batch mode. Still the main focus of this chapter was developing a web app, and now we'll show you how to debug the TypeScript code of web apps in the browser.

## 9.6 Debugging TypeScript in the browser

Writing code in TypeScript is fun but web browsers don't understand this language. They load and run only the JavaScript version of the app. Moreover, the executable JavaScript may be optimized and compressed as well, which makes it unreadable. But there is a way to load the original TypeScript code into the browser as well.

For that, you need to generate the source map files, which map the executable lines of code back to the corresponding source code, which is in TypeScript in our case. If the browser loads the source map file(s), we can debug the original sources!

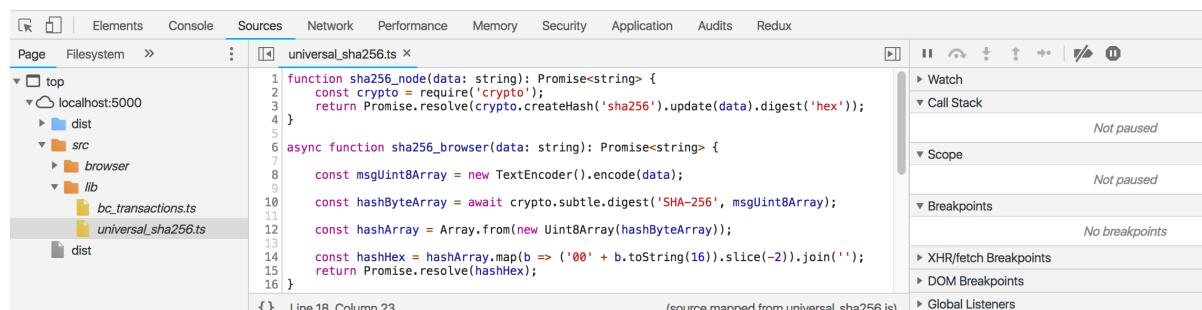
In the `tsconfig.json` shown in listing 9.2, we asked the compiler to generate source maps, which are files with the extension `.js.map` as seen in figure 9.2.

When a browser loads the JavaScript code of a web app, it loads only the `.js` files even if the deployed app includes the `.js.map` files. But if you open the browser's dev tools, then the browser will load the source maps as well.

### TIP

If your browser doesn't load the source map files of your app, check the settings of the dev tools and ensure that the option JavaScript source maps is enabled.

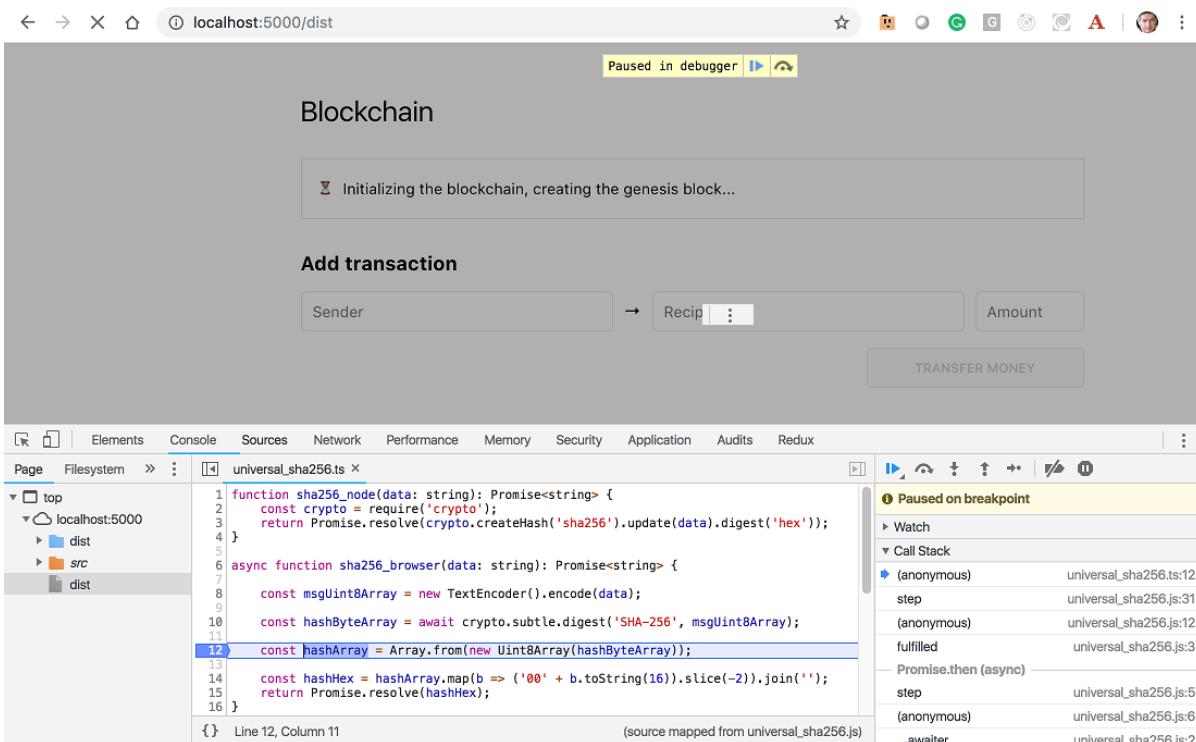
Now let's load our web client as explained in section 9.1 in the Chrome browser and open dev tools in the Sources tab. It'll split the screen into three parts as seen in figure 9.10.



**Figure 9.10 The Sources panel of Chrome Dev Tools**

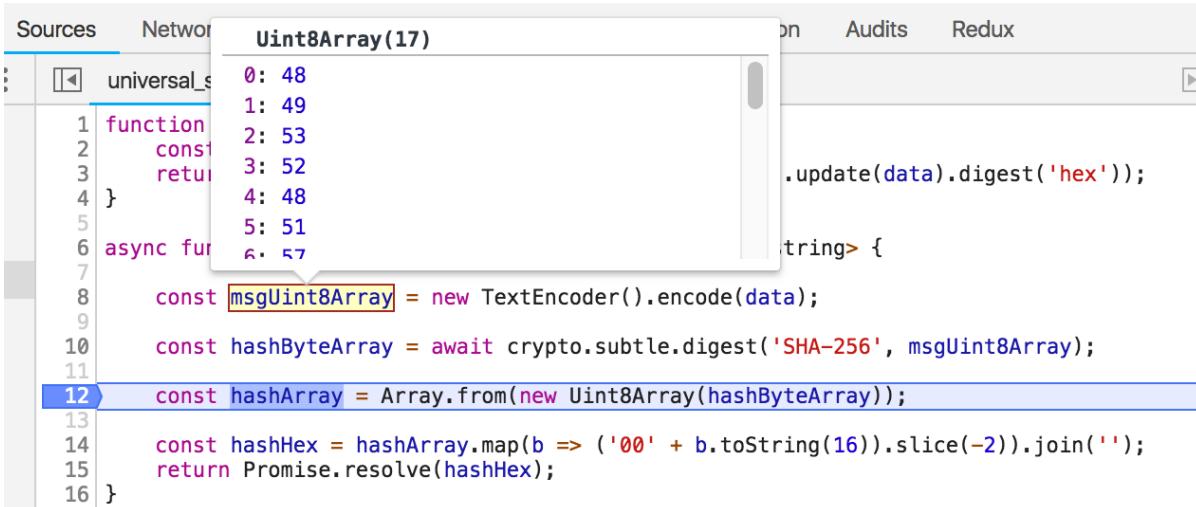
On the left you can browse and select a source file from your project. We've selected `universal_sha256.ts` and its TypeScript code is shown in the middle. On the right, you see the debugger's panel.

Let's set a breakpoint at the line 12 by clicking to the left of the line number and refresh the browser's window. The app will stop at the breakpoint and the browser window will look as in figure 9.11.



**Figure 9.11 The program paused at the breakpoint**

You can see the values of the variables by hovering the mouse over the variable name. Figure 9.12 shows the values of the variable `msgUint8Array` as we hover the mouse over its name in line 8.



**Figure 9.12 Watching the variable values by hovering the mouse**

You can also watch the values of any variable or expression by adding it to the Watch area in the debugger's panel on the right. Figure 9.13 shows the program paused at line 15. We've added a couple of variable names and one expression by clicking on the little plus sign in the Watch area on the right.

The screenshot shows the Chrome DevTools debugger interface. On the left is the source code for `universal_sha256.ts`. On the right is the debugger pane, which is paused on a breakpoint. The `Watch` tab is selected. It displays several variables and their values. One variable, `hashHex`, is highlighted. To the right of `hashHex` is a minus sign (`-`), indicating it can be removed from the watch list.

**Figure 9.13** Watching the variables in the Watch area

If you want to remove the watch, click on the little minus sign on the right of the variable/expression that you want to remove. In figure 9.13, you can see this minus sign to the right of the variable `hashHex` in the watch area.

Chrome browser comes with a full featured debugger, and to learn more about it, watch the video "Debugging JavaScript - Chrome DevTools 101" at [www.youtube.com/watch?v=H0XScE08hy8](https://www.youtube.com/watch?v=H0XScE08hy8). With source maps, you'll be able to debug TypeScript even if the JavaScript was optimized/minimized/uglified.

**NOTE**

IDEs also come with debuggers, and in chapter 10, we'll use the VS Code's debugger for debugging a standalone TypeScript server, but when it comes to web apps, we prefer debugging right in the browser because we may need to know more information about the execution context to figure out the problem, e.g. network requests, application storage, session storage, etc.

## 9.7 Summary

In this chapter, you learned a way to create, load and debug your very own web app in the browser. Using yet another implementation of the blockchain app, you learned how to create and configure a TypeScript project that is a web app. This time, we illustrated how to organize the project by splitting the code into several directories, and one of them (named `lib`) was reused by two different client apps - a web app (in the `browser` directory) and the standalone app (in the `node` directory).

In this app, you could see examples of a practical use of the following TypeScript (and JavaScript) syntax elements covered in Part 1 of this book:

- Using `private` and `readonly` keywords
- Using TypeScript interfaces and classes for declaring custom types
- Cloning objects using the JavaScript spread operator
- Using the `enum` keyword
- Using `async`, `await`, and a `Promise`

You also had a chance to use the web browser's API, HTML, and a simple Web server.

We showed you how to use npm scripts for creating custom commands for the app deployment and starting the web server.

Finally, we've explained how to debug TypeScript in the Chrome browser, which is our browser of choice when it comes to development of web apps.

In the next chapter, you'll see another facet of the blockchain app - how the nodes can broadcast newly created blocks to their peers for approval. You'll see more examples of a practical use of TypeScript in the web app that uses the WebSocket protocol.

# *Client-server communications using Node.js, TypeScript, and WebSockets*

## **This chapter covers:**

- Why a blockchain may need a server
- The meaning of the longest chain rule
- How to create a Node.js WebSocket server in TypeScript
- Practical illustrations of using TypeScript interfaces, abstract classes, access qualifiers, enums, and generics

In the previous chapter, we learned that each block miner can take a number of pending transactions, create a valid block that includes the proof-of-work, and add the new block to the blockchain. This workflow is easy to follow when there is only one miner creating the proof-of-work. Realistically, there could be thousands of miners around the world trying to find the valid hash for a block with the same transactions, which may cause conflicts.

In this chapter, we'll use TypeScript to create a server that utilizes WebSocket protocol for broadcasting messages to the blockchain's nodes. The web clients can also make requests to this server. While writing code in TypeScript remains our main activity, we'll use several JavaScript packages for the first time in this book:

- ws - a Node.js library that supports WebSocket protocol
- express - a small Node.js framework offering HTTP support
- nodemon - a tool that restarts node.js-based apps when the script file changes are detected
- lit-html - HTML templates in JavaScript for rendering to the browser's DOM

These packages are included in the file package.json as dependencies (see section 10.4.2).

Before explaining the TypeScript code of the blockchain app that comes with this chapter, we need to cover the following subjects:

- The blockchain concept known as *a longest chain rule*.
- How to build and run this blockchain app emulating more than one block miner

We'll also need to go over the infrastructure-related subjects:

- The project structure, its configuration files, and npm scripts.
- What's the WebSocket protocol about and why it's better than HTTP for implementing a notification server. As an illustration, we'll create a simple WebSocket server that can push messages to the web client.

Let's start with the longest chain rule.

## 10.1 Resolving conflicts using the longest chain rule

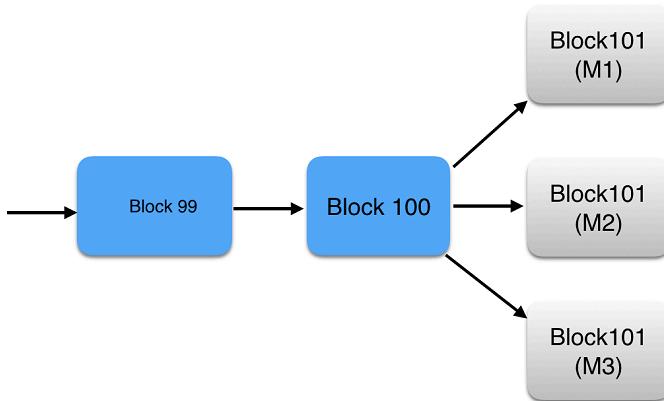
In chapter 9, we had a simplified way to start block mining - the user would just click on the button Confirm Transaction to create a new block. This time, let's consider a more realistic example of a blockchain that has already 100 blocks, and a pool of pending transactions. Multiple miners could grab pending transactions (e.g. 10 transactions each) and start mining blocks.

**NOTE**

While reading about block mining in this chapter, keep in mind that we are talking about a decentralized network. The blockchain nodes work in parallel, which may result in conflict situations if more than one node claims to have mined the next block. That's why a consensus mechanism is required to resolve conflicts.

Let's pick three arbitrary miners M1, M2, and M3 and assume they found the proper hash (i.e. the proof of work) and broadcasted their versions of the new block number 101 as a candidate for adding to the blockchain. Each of their candidate blocks may have different transactions, but wants to become the block number 101.

Each of the miners is located in different continents, and the forks are created in the nodes of these miners as seen in figure 10.1. Temporarily, three forks exists, which are identical in the first 100 blocks, but the block 101 is different in each fork. At this point, each of these three blocks 101 is validated (they have the right hashes) but not confirmed yet.

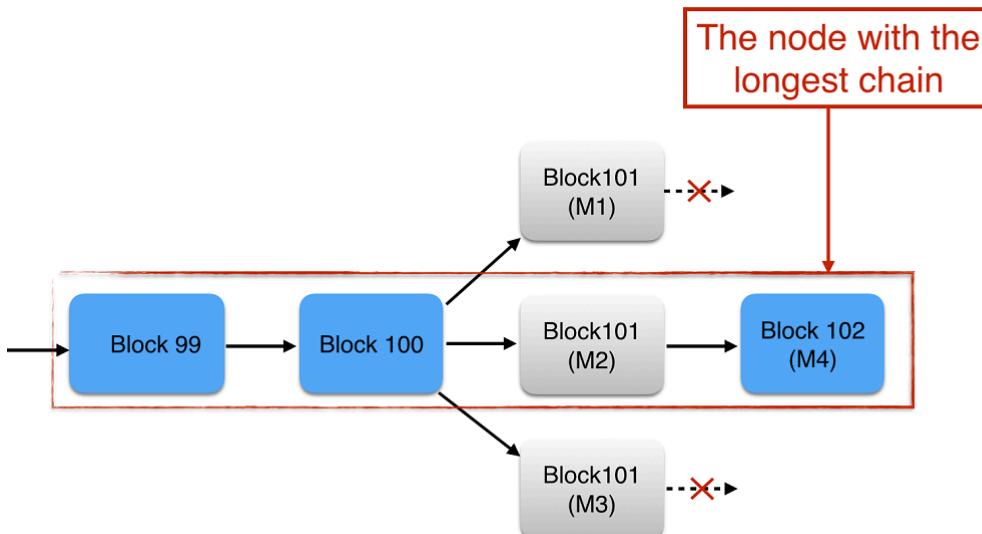


**Figure 10.1 A forked blockchain**

How to decide which one of these three should be added to the blockchain? We'll use the longest chain rule. A multi-node chain has multiple blocks that are candidate for adding to the network, but the blockchain is like a living a growing organism that keeps adding nodes all the times, and by the time all of the miners finished mining, one of their chains has been used already to add more blocks and became the longest chain.

For simplicity, we consider just three miners, that there could be hundreds of thousands of them, and at the same time, other miners are mining blocks 102, 103 et all. Let's assume that out of these three miners, M2 had more powerful CPUs and at some point its fork was the longest.

We may not know that some other miner M4 already requested the longest chain and picked the block 101 from the M2's fork and M4 already calculated the hash for the next candidate block 102. We assumed that M2 had a powerful CPU so the fork of the miner M2 was the longest (had an extra block 101) at some point, and that's why the miner M4 linked it to the block 101 as shown in figure 10.2.



**Figure 10.2 M2 has the longest chain**

Now, even though we have three forks starting from block 101, the longest chain is the one that ends with the block 102 and it will be adopted by all other nodes. The forks created by the miners M1 and M3 will be discarded and their transactions will be placed back into the pool of pending transactions so other miners could pick them up while creating other blocks.

The miners M1 and M3 just wasted the electricity for calculating the hash for the block 101, but M2 will get the reward for mining the block 101. As a matter of fact, when M4 selected the M2's block with an unconfirmed, but valid hash, he was risking useless work as well.

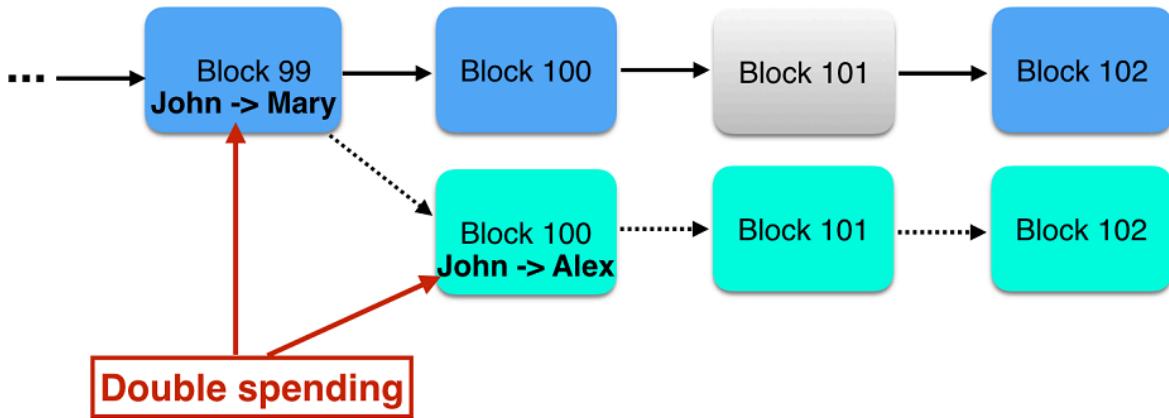
**NOTE** A blockchain would implement the mechanism to ensure that if a transaction is included in a valid block by any node, it won't be placed in the pool of pending transactions while some blocks are discarded.

We use a small number of the blocks for simplicity, but public blockchains may have thousands and thousands of nodes. In our scenario, there was a moment, when the fork of the miner M2 had a longest chain that was just one block longer than the other forks. In the real-world blockchains, there could be multiple forks of different length.

**TIP** Later in this chapter, we'll go over the process of requesting the longest chain and getting the responses. Figures 10.5 - 10.8 show the communication between two nodes while requesting the longest chain and announcing newly mined blocks.

Since the blockchain is a distributed network, blocks are being added on multiple nodes, so each node may have a chain of different length. The longest chain is considered to be the correct one.

Now let's see how the longest chain rule helps in preventing the *double spending* problem or other fraud. Let's say one miner has a friend John who has \$1000 in his account. One of the transactions in the block 99 is "John paid Mary \$1000". What if the miner decides to commit a fraud by forking the chain and adding another transaction "John paid Alex \$1000" in the block 100? Technically, this bad miner is trying to cheat the blockchain by making it appear that John spent the same \$1,000 twice — once in a transaction to Mary and again on a transaction to Alex. Figure 10.3 shows an attempt by the criminal miner to convince the others that the correct view of the blockchain is the fork that contains the John-to-Alex transaction.



**Figure 10.3 An attempt to double spend**

Remember, the block's hash value is easy to check but is time-consuming to calculate, hence our criminal miner has to recalculate the hashes for blocks 100, 101, and 102. Meanwhile, other miners continue mining new blocks (103, 104, 105, et al.) adding them to the chain shown on the top of figure 10.3. There's no way that the criminal miner can recalculate all hashes and create a longer chain faster than all other nodes. The chain with the most work (a.k.a. the longest chain) wins. In other words, the chances of modifying the content of the existing block(s) are close to zero, which makes a blockchain immutable.

#### SIDE BAR    What's a consensus

Any decentralized system needs to have rules allowing all the nodes to agree that a valid event happened. What if two nodes calculated the hash of a new block at the same time? Which block has to be added to the chain, and which node has to be rewarded? In other words, all nodes of the blockchain have to come to a general agreement - *consensus* - while deciding who's the winner.

Consensus is required from all members because there is no "system administrator" who could modify or delete a block. Blockchains use different rules (a.k.a. *consensus protocols*), and we'll use the proof of work combined with the longest chain to reach a consensus on the true state of the blockchain. The aim of the consensus protocol is to guarantee that a single chain is used.

So far, we never discussed how blockchain nodes communicate with each other if need be, and we'll do it in the next section.

## 10.2 Adding a server to the blockchain

Adding a server? Didn't we praise the blockchain technology for being completely decentralized without a server? We did, and our server won't be the central authority for creating, validating, and storing blocks. The blockchain can remain decentralized and still use a server for such utility services as caching hashes, broadcasting new transactions, requesting the longest chain, or announcing the newly created blocks. In this chapter, we'll review the code of such a server, and the process of communications between the clients via this server is described below.

The node M1 mined the block and requested the longest chain from the server, which broadcasted this request to all other nodes on the blockchain. Each of the nodes responds with their chains (just the block headers), and the server forwards these responses to M1.

The M1's node receives the longest chain and validates it by checking hashes of each block. Then the newly mined block (by M1) is added to this longest chain, and this chain is saved locally. For some time, M1's node will enjoy the status of "The source of truth" as it'll have the longest chain until someone else mines the new block(s).

Let's say the miner M1 wants to create a new transaction "Joe sent Mary \$1000". Creating a new block for each transaction would be too slow (lots of hashes would need to be calculated) and expensive (the electricity cost). Typically, one block includes multiple transactions, and M1 can simply broadcast its new transaction(s) to the rest of the blockchain members. The miners M2, and M3 and any future miners will do the same with their transactions. All broadcasted transactions comprise a pool of pending ones, and any node can pick, say 10 transactions from there and start mining.

**TIP**

For our version of blockchain, we'll use Node.js to create a web server that implements broadcasting via WebSockets. As an alternative, you could go completely serverless by implementing broadcasting using some peer-to-peer technology like WebRTC (see [en.wikipedia.org/wiki/WebRTC](https://en.wikipedia.org/wiki/WebRTC) ).

## 10.3 The project structure

The blockchain app that comes with this chapter consists of two parts: the server and the client, and both of them are implemented using TypeScript. We'll show you the project structure and explain how to run this app first, and then we'll discuss the selected code fragments that illustrate the practical use of specific TypeScript syntax constructs.

In your IDE, open the project in the directory chapter10 and run `npm install` in the terminal window. The structure of this project is shown in figure 10.4.

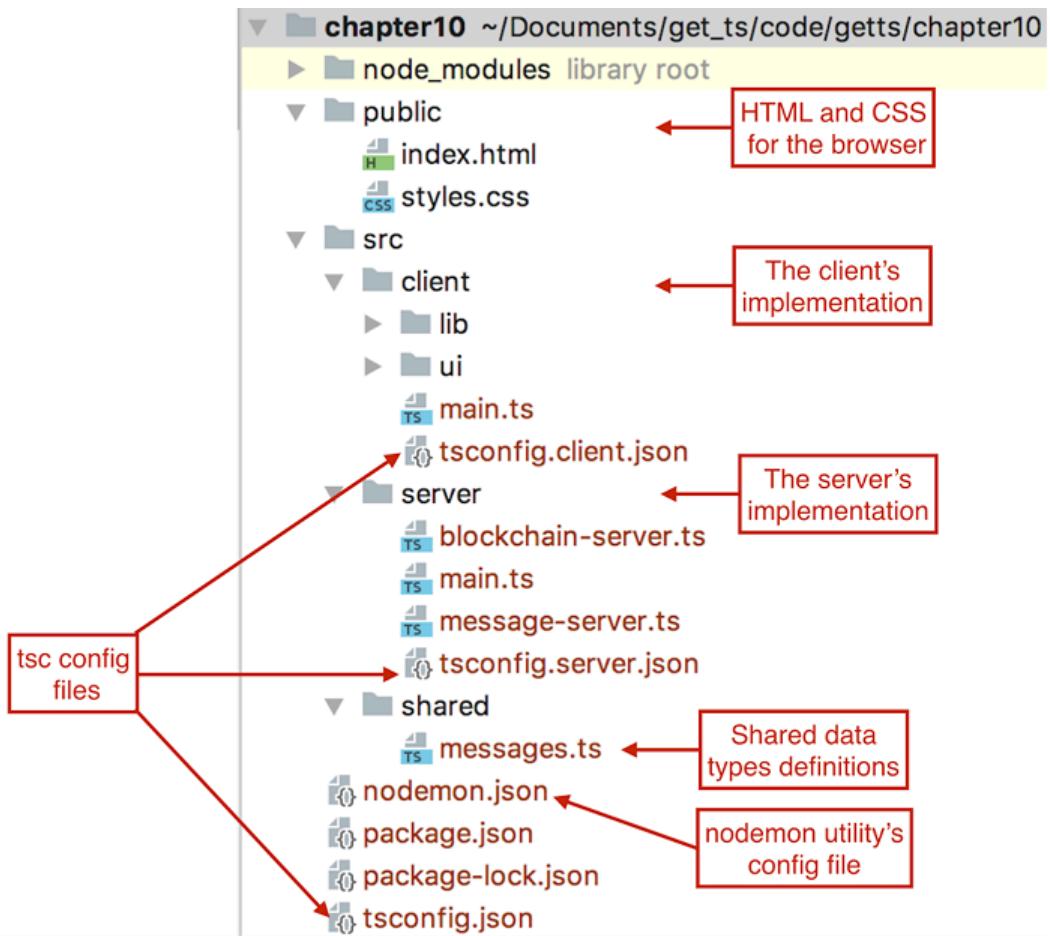


Figure 10.4 The project structure

The public directory is created during the build process, and the file `public/index.html` loads the compiled version of the file `client/main.ts` along with all its imported scripts. This is the web client that we'll use to illustrate the blockchain node.

The file `server/main.ts` contains the code that imports additional scripts and starts the WebSocket and the blockchain notification servers.

Note that there are three config files for the TypeScript compiler. The file `tsconfig.json` located in the root directory contains the compiler's options that are common to both the client and server. The files `tsconfig.client.json` and `tsconfig.server.json` extend `tsconfig.json` and add the compiler's options specific to the client and server respectively. We'll review their content in the next section.

Our server runs under the Node.js runtime, and its compiled code will be stored in the directory `build/server`. You could start the server with the following command:

```
node build/server/main.js
```

But the executable `node` doesn't support live reload, i.e. if you keep changing (and recompiling) the TypeScript code of the server, it won't restart. That's why we use the utility `nodemon` (see

[nodemon.io](#)) that monitors the Node.js app's codebase and restarts the Node.js server on file changes. If you have the `nodemon` utility installed, you can use it instead of `node`. For example, this is how you can start the Node.js runtime and run the code :

```
nodemon build/server/main.js
```

In our project, we be starting the server with the `nodemon` utility, which will be configured in the file `nodemon.json` reviewed in the next section. The file `package.json` includes the `npm scripts` section, and we'll review it in the next section as well.

## 10.4 Configuration files of the project

The project that comes with this chapter includes several configuration JSON files reviewed in this section.

### 10.4.1 Configuring the TypeScript compilation

A `tsconfig.json` file can inherit configurations from another file using the `extends config` property. In our project, we have three config files:

- `tsconfig.json` contains the common `tsc` compiler options for the entire project.
- `tsconfig.client.json` contains the options for compiling the client portion of the project
- `tsconfig.server.json` includes the options for compiling the server portion of the project.

The contents of each of these config files are shown next, and listing 10.1. shows the content of the base `tsconfig.json`.

#### Listing 10.1 `tsconfig.json` - common tsc options

```
{
  "compileOnSave": false,    ①
  "compilerOptions": {
    "target": "es2017",     ②
    "sourceMap": true       ③
  }
}
```

- ① Don't auto-compile on each modification of the TypeScript files
- ② Compile into the JavaScript using the syntax supported by ECMAScript 2017
- ③ Generate the sourcemap files

We specified `es2017` as the compilation target because we're sure that the users of this app will use modern browsers that support all features included in the ECMAScript 2017 specification.

Listing 10.2 shows the file `tsconfig.client.json`, which contains the `tsc` options that we want to use for compiling the client's portion of the code. This file is located in the directory `src/client`.

## Listing 10.2 tsconfig.client.json - tsc options for the client

```
{
  "extends": "../../tsconfig.json",    ①
  "compilerOptions": {
    "module": "es2015",      ②
    "outDir": "../../public", ③
    "inlineSources": true,   ④
    "plugins": [
      {
        "name": "typescript-lit-html-plugin" ⑤
      }
    ]
  }
}
```

- ① Inherit the config options from this file
- ② Use import/export statements in generated JavaScript
- ③ The compiled code has to go in the *public* directory
- ④ Emit the original TypeScript code and its sourcemaps within a single file
- ⑤ This plugin enables HTML auto-completion for the template strings tagged with html

Sourcemaps allow you to debug TypeScript while the browser executes JavaScript. They work by instructing the browser's Dev Tools which lines of the transpiled code (JavaScript) correspond to which lines of the source (TypeScript). However, the source code needs to be available for the browser, and we can either deploy TypeScript along with JavaScript to the web server, or use the `inlineSources` option which embeds the original TypeScript right inside the sourcemaps files.

The web part of this web uses lit-html (see [github.com/Polymer/lit-html](https://github.com/Polymer/lit-html)), a templating library for JavaScript, and `typescript-lit-html-plugin` will enable auto-complete (a.k.a IntelliSense) in your IDE.

**TIP**

If you don't want to reveal the sources of your app to the users, don't deploy sourcemaps in production.

Listing 10.3 shows the file `tsconfig.server.json`, which contains tsc options for compiling the server's code. This file is located in the directory `src/server`.

### Listing 10.3 tsconfig.server.json - tsc options for the server

```
{
  "extends": "../../tsconfig.json",      ①
  "compilerOptions": {
    "module": "commonjs",            ②
    "outDir": "../../build"          ③
  },
  "include": [
    "**/*.ts"                      ④
  ]
}
```

- ① Inherit config options from this file
- ② Convert import/export statements to commonjs compliant code
- ③ Compiled code has to go in the build directory
- ④ Compile all .ts files from all subdirectories

**TIP**

In the perfect world, you would never see the same compiler's option in base and inherited config files. But sometimes, IDEs don't handle the tsc config inheritance properly, and you may need to repeat the same option in more than one file.

Now that we have more than one tsc config file, the question is how to let the TypeScript compiler know which one to use? This is done using the `-p` option. The following command compiles the web client code using the options from `tsconfig.client.json`:

```
tsc -p src/client/tsconfig.client.json
```

The next command compiles the server code using `tsconfig.server.json`

```
tsc -p src/server/tsconfig.server.json
```

**NOTE**

If you introduce tsc configuration files named other than `tsconfig.json`, you need to use the `-p` option and specify the path to the file you want to use. For example, if you'd run the `tsc` command in the directory `src/server` wouldn't use the options from the file `tsconfig.server.json`, and you may experience unexpected compilation results.

Now let's take a look at the dependencies and the npm scripts of this project.

### 10.4.2 What's in package.json

Listing 10.4 shows the content of package.json. The `scripts` section contains custom npm commands, and the `dependencies` section lists only three packages that are required to run our app (both the client and server):

- `ws` - a Node.js library that supports WebSocket protocol
- `express` - a small Node.js framework offering HTTP support
- `lit-html` - HTML templates in JavaScript for rendering to the browser's DOM

The `devDependencies` section includes packages that are needed only during development.

#### Listing 10.4 package.json

```
{
  "name": "blockchain",
  "version": "1.0.0",
  "description": "Chapter 10 sample app",
  "license": "MIT",
  "scripts": {❶
    "build:client": "tsc -p src/client/tsconfig.client.json",
    "build:server": "tsc -p src/server/tsconfig.server.json",
    "build": "concurrently npm:build:*",
    "start:client": "tsc -p src/client/tsconfig.client.json --watch", ❷
    "start:server": "nodemon --inspect src/server/main.ts",
    "start": "concurrently npm:start:*",
    "now-start": "NODE_ENV=production node build/server/main.js"
  },
  "dependencies": {
    "express": "^4.16.3", ❸
    "lit-html": "^0.12.0", ❹
    "ws": "^6.0.0" ❺
  },
  "devDependencies": {
    "@types/express": "^4.16.0", ❻
    "@types/ws": "^6.0.1", ❼
    "concurrently": "^4.0.1", ❼
    "nodemon": "^1.18.4", ❼
    "ts-node": "^7.0.1", ❼
    "typescript": "^3.1.1",
    "typescript-lit-html-plugin": "^0.6.0" ❽
  }
}
```

- ❶ Our custom npm script commands
- ❷ Running tsc for the client in a watch mode
- ❸ The web framework for Node.js
- ❹ The templating library for the client
- ❺ The package to support WebSocket in Node.js apps
- ❻ Type definition files
- ❼ The package to run multiple commands concurrently
- ❽ The utility for the live reload of the Node.js runtime

- ⑨ ts-node runs both tsc and node as a single process
- ⑩ The plugin to enable the IntelliSense for the lit-html tags

ts-node launches a single process Node. After Node is launched, it registers a custom extension/loader pair using the Node's `require.extensions` mechanism. When Node's `require()` call resolves to a file with the extension `.ts`, Node invokes a custom loader, which transpiles TypeScript into JavaScript on the fly using tsc's programmatic API without launching a separate tsc process.

Note that the command `start:client` runs tsc in the watch mode (see the option `--watch`). This ensures that as soon as you modify and save any TypeScript code on the client, it'll get recompiled. What about recompiling the server's code?

### 10.4.3 Configuring nodemon

We could start the tsc compiler in the watch mode on the server as well and JavaScript would be regenerated on modification of the TypeScript code. But having the fresh JavaScript code is not enough on the server - we need to restart the Node.js runtime on each code change. That's why we've installed the nodemon utility, which will start the Node.js process and monitor the JavaScript files in the specified directory. The file `package.json` (see listing 10.4) includes the following command:

```
"start:server": "nodemon --inspect src/server/main.ts"
```

**TIP**

The `--inspect` option allows you to debug in Chrome dev tools the code that runs in Node.js (see [nodejs.org/en/docs/guides/debugging-getting-started](https://nodejs.org/en/docs/guides/debugging-getting-started) for detail). Nodemon just passes the `--inspect` option to Node.js so it'll be started in the debug mode.

It seems that the `start:server` command requests nodemon to start the TypeScript file `main.ts`, but since our project has the file `nodemon.json`, the nodemon will use the options from there. Our file `nodemon.json` contains the configuration options for nodemon as seen in listing 10.5.

#### **Listing 10.5 nodemon.json**

```
{
  "exec": "node -r ts-node/register/transpile-only", ①
  "watch": [ "src/server/**/*.*ts" ] ②
}
```

- ① How to start the node
- ② Watch all `.ts` files located in all server's subdirectories

The `exec` command allows us to specify the options for starting Node.js. In particular, the option `-r` is a shortcut for `--require module` used for preloading modules on startup. In our case, it asks the `ts-node` package to preloads the faster TypeScript's transpile-only module, which simply converts the code from TypeScript to JavaScript without performing the type check.

This module will automatically run the TypeScript compiler for each file with the extension `.ts` loaded by Node.js. By preloading the `transpile-only` module, we eliminate the need to start a separate `tsc` process for the server. Any TypeScript file will be loaded and auto-transpiled as the part of the single Node.js process.

**NOTE**

You can use the `ts-node` package in different ways. For example, you can use it to start Node.js with the TypeScript compilation: `ts-node myScript.ts`. Visit [www.npmjs.com/package/ts-node](http://www.npmjs.com/package/ts-node) for more details.

We did a high-level overview of the configuration of the blockchain app, and the next step is to see it in action.

#### **10.4.4 Running the blockchain app**

In this section, we'll show you how to run the server and two clients emulating the blockchain nodes. To start any processes we'll be using npm scripts, so let's take a closer look at the scripts section of the `package.json` file.

##### **Listing 10.6 The scripts section of package.json**

```
"scripts": {
  "build:client": "tsc -p src/client/tsconfig.client.json",      ①
  "build:server": "tsc -p src/server/tsconfig.server.json",      ②
  "build": "concurrently npm:build:*",                            ③
  "start:tsc:client": "tsc -p src/client/tsconfig.client.json --watch", ④
  "start:server": "nodemon --inspect src/server/main.ts",        ⑤
  "start": "concurrently npm:start:*",                           ⑥
}
```

- ① Perform the compilation of the client's code
- ② Perform the compilation of the server's code
- ③ Run concurrently all commands that start with `npm:build`
- ④ Starts `tsc` in the watch mode the client's code
- ⑤ Start the server with `nodemon`
- ⑥ Run concurrently all commands that start with `npm:start`

The first two build commands start the `tsc` compiler for the client and server respectively. These processes transpile TypeScript to JavaScript. The compilation of the client and server code can be done in parallel, so the third command uses the `npm` package called `concurrently` (see

[www.npmjs.com/package/concurrently](http://www.npmjs.com/package/concurrently)), which allows to run multiple commands concurrently.

The command `start:tsc:client` compiles the clients code in a watch mode, and `start:server` starts the server using nodemon as described in the previous section. Then again, the `start` command runs both `start:tsc:client` and `start:server` concurrently.

In general, you could start more than one npm command by simply adding the ampersand(s) between the commands. For example, you can define two custom commands "first" and "second" and then run them concurrently with `npm start`. In npm scripts, the ampersand means "run these commands concurrently".

### Listing 10.7 Concurrent execution with the ampersand

```
"scripts": {
  "first": "sleep 2; echo First",      ①
  "second": "sleep 1; echo Second",    ②
  "start": "npm run first & npm run second" ③
},
```

- ① Sleep for 2 second and print First
- ② Sleep for 1 second and print Second
- ③ Run the first and second commands concurrently

If you run `npm start`, it'll print Second and then First, which proves that the commands ran concurrently. Replacing `&` with `&&` will print First and then Second indicating the sequential execution.

But using the package `concurrently` gives you a clean separation of all the messages printed by each concurrent process. Listing 10.8 shows the terminal output of the command `npm start`, where we run concurrently two commands: `start:tsc:client` and `start:server`. Each line in the terminal output starts with the name of the process (in square brackets) that produced this message.

To start the blockchain app, run the `npm start` in the Terminal window, and it should produce the output as in listing 10.8:

## Listing 10.8 Starting the blockchain app

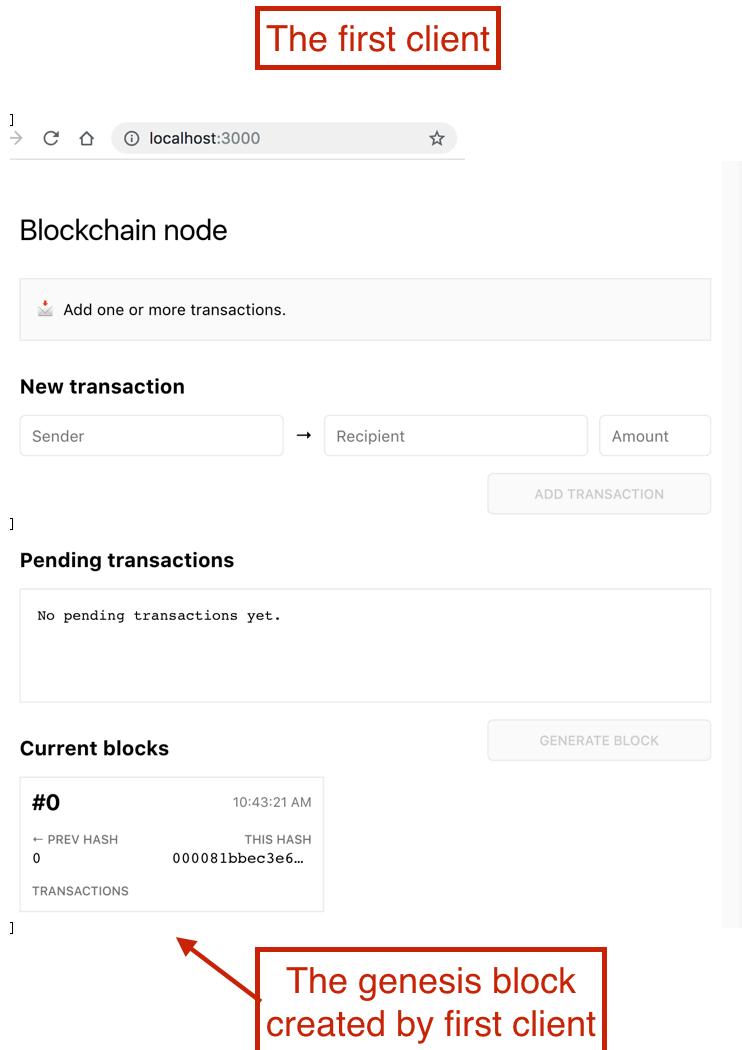
```
> blockchain@1.0.0 start /Users/yfain11/Documents/get_ts/code/getts/chapter10
10:47:18 PM - Starting compilation in watch mode...
[start:tsc:client] ①
[start:server] [nodemon] 1.18.9 ②
[start:server] [nodemon] to restart at any time, enter `rs` ③
[start:server] [nodemon] watching: src/server/**/*.ts ④
[start:server] [nodemon] starting `node -r ts-node/register/transpile-only --inspect src/server/main.ts` ⑤
[start:server] Debugger listening on ws://127.0.0.1:9229/2254fc00-3640-4390-8302-1e17285d0d23 ⑥
[start:server] For help, see: https://nodejs.org/en/docs/inspector ⑦
[start:server] Listening on http://localhost:3000 ⑧
[start:tsc:client] 10:47:21 PM - Found 0 errors. Watching for file changes. ⑨
```

- ① The output of the process start:tsc:client
- ② The output of the process start:server
- ③ The Node.js debugger runs locally on port 9229
- ④ The server's up and running on port 3000

**NOTE**

The URL of the Node.js debugger is `ws://127.0.0`. It starts with `ws` indicating that the dev tools connect to the debugger using the WebSocket protocol, which we'll introduce in the next section. When Node.js debugger is running, you'll see a green hexagon on the Chrome Dev Tools toolbar. Read more about Node.js debugger in the article by Paul Irish "Debugging Node.js with Chrome Dev Tools" at [medium.com/@paul\\_irish/debugging-node-js-nightlies-with-chrome-devtools-7c4a1b95](https://medium.com/@paul_irish/debugging-node-js-nightlies-with-chrome-devtools-7c4a1b95).

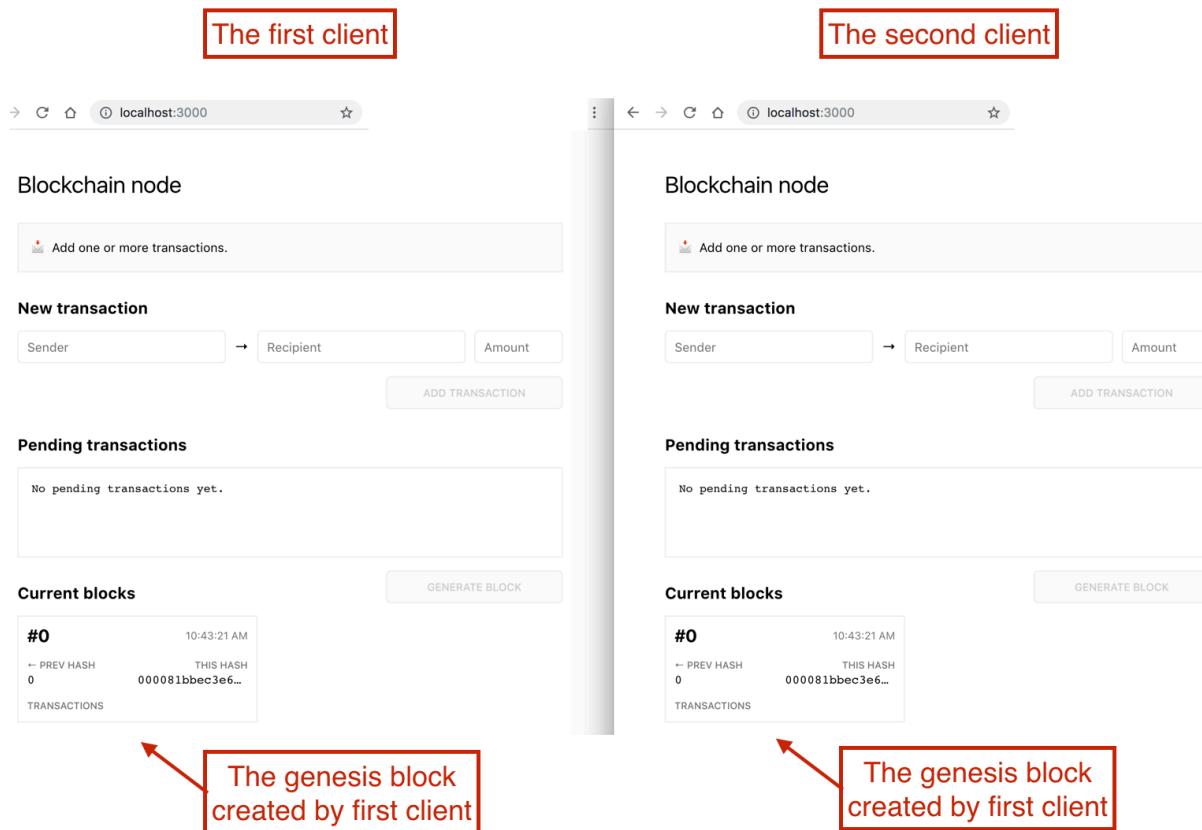
The server is up and running, and entering `localhost:3000` will start the first client of our blockchain. After a couple of seconds, the genesis block will be generated and you'll see the web page as in figure 10.5.



**Figure 10.5 The view of the very first blockchain client**

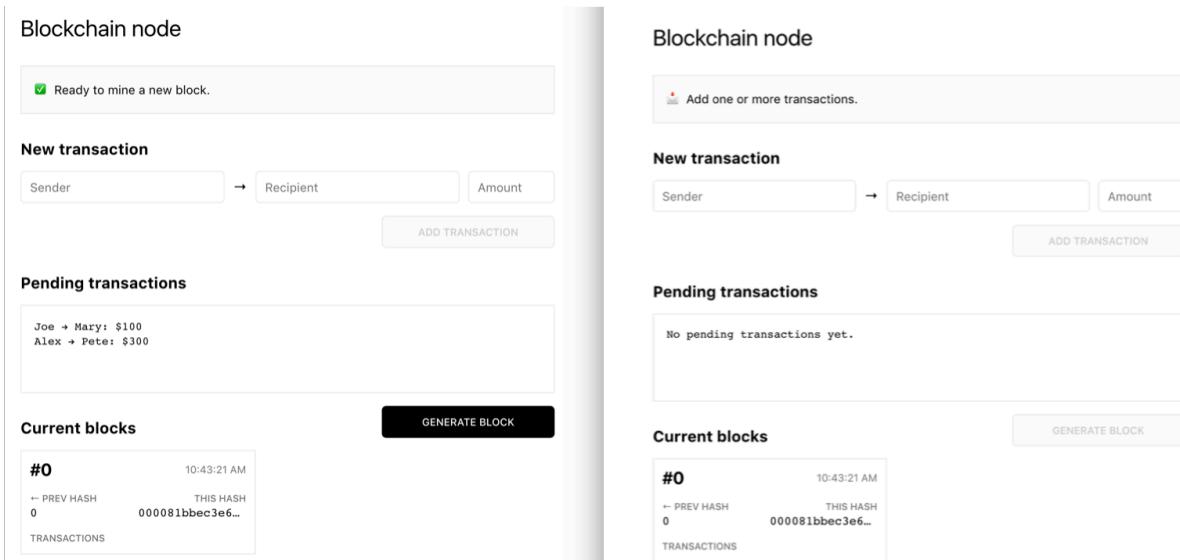
We'll show what's happening under the hood in section 10.6 after explaining how the clients communicate with each other via the WebSocket server. At this point, suffice it to say that before creating any block, the client makes a request to the server to find the longest chain.

Now let's start the second client by opening a separate browser window at localhost:3000 as seen in figure 10.6.



**Figure 10.6 The view of the first two blockchain clients**

Now let's discuss the use case when only the first client adds pending transactions, starts block mining and invites the other nodes to do the mining too. By this time, our blockchain already had a genesis block, and the server broadcasted it to all connected clients (in our case, the one shown on the right). Note that both clients see the same block that was mined by the first client. Then, the first client entered two transactions as seen in figure 10.7. No requests for the new blocks generation have been made yet.



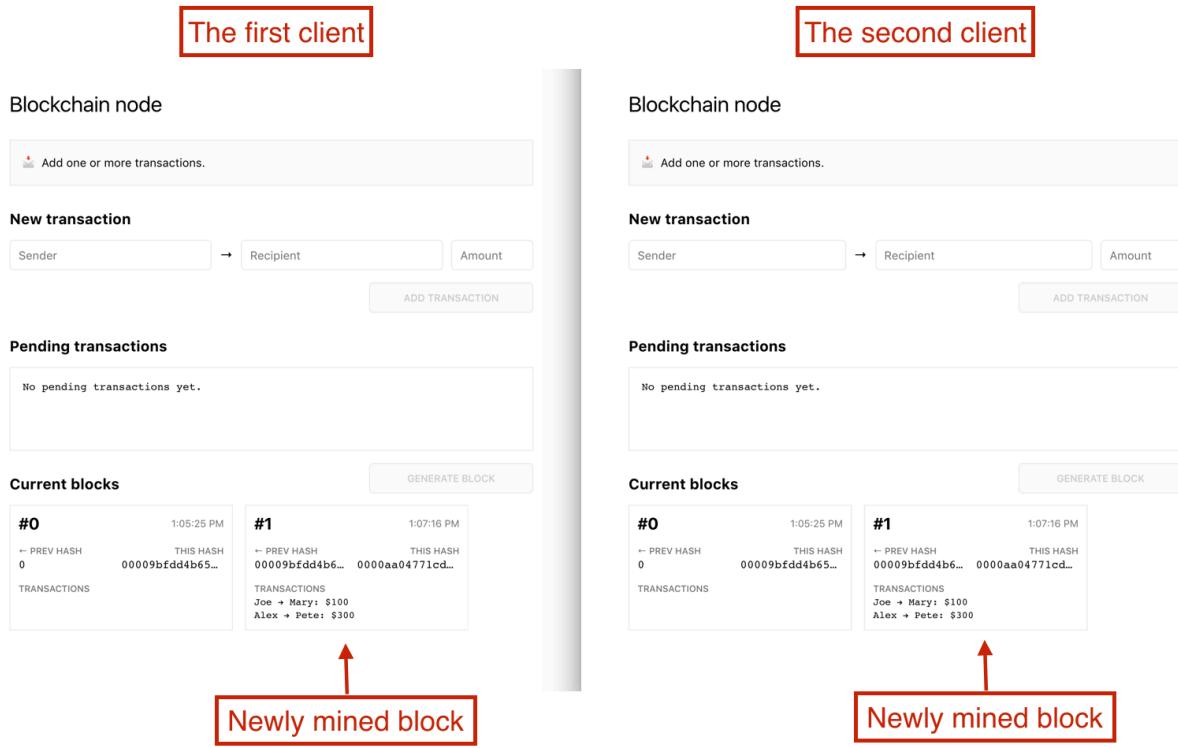
**Figure 10.7 Both clients created pending transactions**

**NOTE**

While a client is adding pending transactions, there are no communications with other clients and the messaging server is not being used.

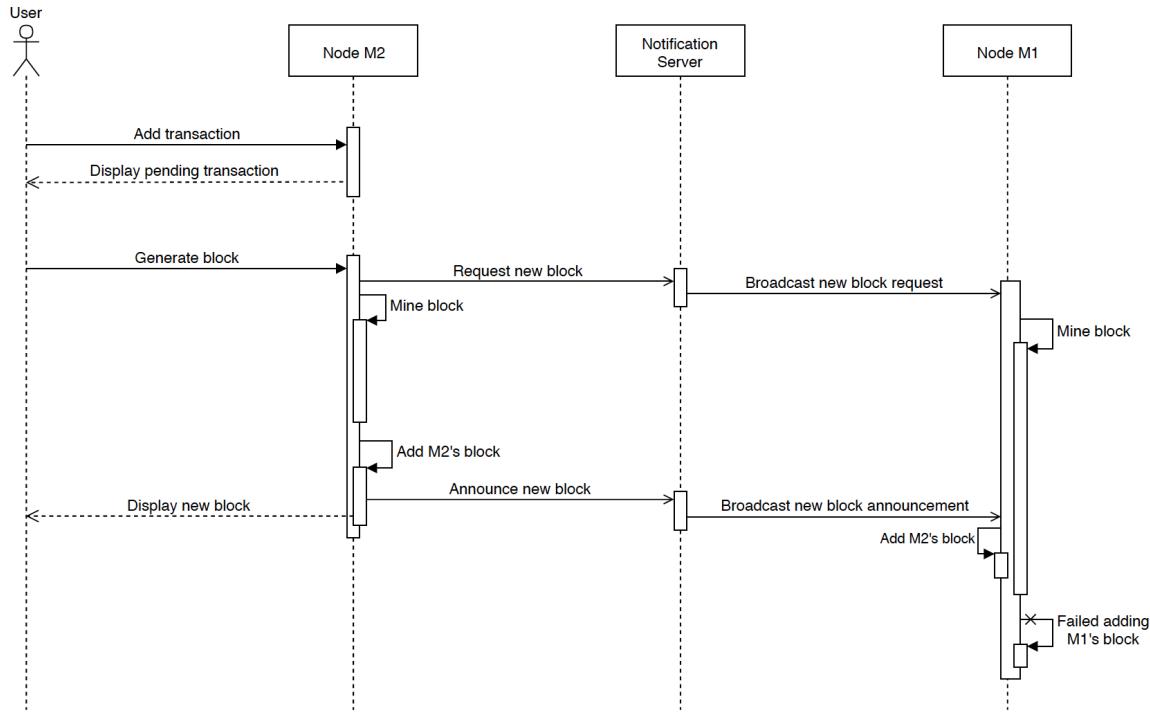
Now, the first client starts mining by clicking on the button GENERATE BLOCK. Under the hood, the first client sent the message with the block content to the server announcing that the first client started mining this block. The server broadcasted this message to all the connected clients and they started mining the same block as well.

One of our clients was faster and its new block was added to the blockchain and broadcasted to other connected clients so they could add this block to their versions of the blockchain. After the blocks are accepted to the blockchain, all clients will contain the same blocks as seen in figure 10.8.



**Figure 10.8 Each client added the same block**

In section 10.6, we'll discuss the messages that go over the sockets as the blocks are being mined so the process of mining in the multi-node network won't look like Voodoo Magic. Figure 10.9 shows a sequence diagram that should help you in following the messaging exchange during assuming that there are just two nodes: M1 and M2.



**Figure 10.9 The mining process in a two-node blockchain**

In our blockchain, the user works with the M2 node. When he or she creates pending transactions, no messages are being sent to other nodes. The messaging starts when the user initiates the generate block operation. The Node M2 sends a Request New Block message and starts mining at the M2 node. The server broadcasts this message to other nodes (i.e. M1), which also starts mining competing with M2.

In this diagram, the M2 node was faster and was the first to announce the new block, and the server broadcasted this message across the blockchain. The M1 node added the M2's block in its local blockchain. A bit later, M1 finished mining too but adding the M1's block failed cause it already accepted and added the M2's block and existing chain is longer. The code that corresponds to the failed attempt of adding the new node is shown later in this chapter in listing 10.27.

In other words, the M2's block was approved as the winner block by consensus between M1 and M2. Finally, the new block was rendered on the user's UI.

**NOTE**

In our simplified blockchain the UI and the block creation logic are implemented in the same web app. A real-world app that uses the blockchain technology would have separate apps for adding transactions and mining blocks.

Now that you've seen how the app works, let's get familiar with the client-server communications over the WebSockets. If you're familiar with WebSockets, skip the section

10.5.

## 10.5 A brief introduction to WebSockets

Since the blockchain app from this chapter will be pushing notification using the WebSocket protocol, we'd like to give you a brief overview of this low-overhead binary protocol supported by all modern web browsers (see [en.wikipedia.org/wiki/WebSocket](https://en.wikipedia.org/wiki/WebSocket)) as well as all Web servers written in Node.js, .Net, Java, Python et al. The WebSocket protocol allows bidirectional message-oriented streaming of text and binary data between browsers and web servers. In contrast to HTTP, WebSocket is not a request-response based protocol, and both the server and the client apps can initiate the data push to the other party as soon as the data becomes available, in real time. This makes the WebSocket protocol a good fit for various apps, for example:

- Live trading/auctions/sports notifications
- Controlling medical equipment over the web
- Chat applications
- Multiplayer online games
- Real-time updates in social streams
- Blockchain

All of these apps have one thing in common: there is a server (or a device) that may need to send an immediate notification to the user because some important event happened elsewhere. This is different from the use case when the user decides to send a request for fresh data to the server. For example, a stock trade happened on the stock exchange, and the notification has to be immediately sent to all users.

Another example is a server that broadcasts notifications from one blockchain node to others. It's important to understand that the server can push a notification(s) to the client(s) without the need to receive the client's request for data. For example the miner M1 starts or ends mining the block, and all other nodes must know about it right away. M1 sends the message to the WebSocket server announcing the new block, and the server can push this message to all other nodes immediately.

### 10.5.1 Comparing HTTP and WebSocket protocols

With the request-based HTTP protocol, a client sends a request over a connection and waits for a response to come back. Both request and response use the same browser-server connection. First, the request goes out and then the response comes back via the same "wire." Think of a narrow bridge over a river where cars from both sides have to take turns crossing the bridge. Cars on the server-side can only go over the bridge after a car from the client-side passes. In the web realm, this type of communications is called *half-duplex*.

The WebSocket protocol allows data to travel in both directions simultaneously (*full-duplex*)

over the same connection, and any party can initiate the data exchange. It's like a two-lane road. Another analogy is a phone conversation where two callers can speak and be heard at the same time. The WebSocket connection is kept alive, which has an additional benefit: low latency in the interaction between the server and the client.

A typical HTTP request/response adds several hundred bytes (HTTP headers) to the application data. Say you want to write a web app that reports the latest stock prices every second. With HTTP, such an app would need to send an HTTP request (about 300 bytes) and receive a stock price that would arrive with additional 300 bytes of an HTTP response object.

With WebSockets, the overhead is as low as a couple of bytes. Besides, there is no need to keep sending requests for the new price quote every second — this stock may not be traded for a while. Only when the stock price changes, the server will push the new value to the client.

Every browser supports a `WebSocket` object for creating and managing a socket connection to the server (see [mng.bz/1j4g](#)). Initially, the browser establishes a regular HTTP connection with the server, but then your app requests a connection upgrade specifying the server's URL that supports the WebSocket connection. After that, the communication goes without the need of HTTP. The URLs of the WebSocket endpoints start with `ws` instead of `http` — for instance, `ws://localhost:8085`. Similarly, for secure communications, you'd use `wss` instead of `https`.

The WebSocket protocol is based on events and callbacks. For example, when your browser app establishes a connection with the server, it receives the `connection` event, and your app invokes a callback to handle this event. To handle the data that the server may send over this connection, expect the `message` event providing the corresponding callback. If the connection is closed, the `close` event is dispatched so your app can react accordingly. In case of an error, the `WebSocket` object gets the `error` event.

On the server side, you'll have to process similar events. Their names may be different depending on the WebSocket software you use on the server. In the blockchain app that comes with this chapter, we use the Node.js runtime for implementing notifications using a WebSocket server.

### **10.5.2 Pushing data from a Node server to a plain client**

To get you familiar with the WebSockets let's consider a simple use case: the server pushes data to a tiny browser client as soon as the client connects to the socket. Our client won't need to send a request for data — the server will initiate the communications.

To enable WebSocket support, we'll use the npm package called `ws` ([www.npmjs.com/package/ws](http://www.npmjs.com/package/ws)) as seen in `package.json` shown in listing 10.4. The type definitions `@types/ws` are needed so the TypeScript compiler won't complain when we use the API from the `ws` package.

This section shows a pretty simple WebSocket server: it'll push the message *This message was pushed by the WebSocket server* to a plain HTML/JavaScript client as soon as the client connects to the socket. We purposely don't want the client to send any requests to the server so you can see that the server can push the data without any request ceremony.

Our app creates two servers. The HTTP server (we'll use the Express framework to implement it) runs on port 8000 and is responsible for sending the initial HTML page to the browser. When this page is loaded, it immediately connects to the WebSocket server that runs on port 8085. This server will push the message with the greeting as soon as the connection is established. The code of this app is located in the file server/simple-websocket-server.ts and is shown in listing 10.9.

### **Listing 10.9 simple-websocket-server.ts**

```
import * as express from "express";
import * as path from "path";
import { Server } from "ws";      ①

const app = express();          ②

// HTTP Server
app.get('/', (req, res) => res.sendFile(path.join(__dirname,
    '/simple-websocket-client.html')));  ③

const httpServer = app.listen(8000, "localhost", () => {      ④
    console.log(`HTTP server is listening on localhost:8000`);
});

// WebSocket Server
const wsServer = new Server({port: 8085});      ⑤
console.log('WebSocket server is listening on localhost:8085');

wsServer.on('connection', ⑥
    wsClient => {
        wsClient.send('This message was pushed by the WebSocket server');  ⑦

        wsClient.onerror = (error) =>  ⑧
            console.log(`The server received: ${error['code']}`);
    }
);

```

- ① We'll use Server from the ws module to instantiate a WebSocket server
- ② Instantiate the Express framework
- ③ When the HTTP client connects with the root path, the HTTP server sends back this HTML file
- ④ Start the HTTP server on port 8000
- ⑤ Start the WebSocket server on port 8085
- ⑥ Listen to the connection event from clients
- ⑦ Push the message to the newly connected client
- ⑧ Handle connection errors

**NOTE**

We could start two server instances on the same port, and we'll do it later in listing 10.13. For now, to simplify the explanations, we'll keep HTTP and WebSocket servers on different ports.

**SIDE BAR****Resolving paths in Node.js**

The line that starts with `app.use()` maps the URL of the HTTP request (could be GET, POST, et al) to a specific endpoint in the code or a file on disk. Let's consider this code fragment:

**Listing 10.10 Mapping the GET request to a file**

```
app.get('/', ❶
    (req, res) => res.sendFile( ❷
        path.join(__dirname, '../../../../../simple-websocket-client.html'))); ❸
```

- ❶ The server received an HTTP GET with the base URL
- ❷ Send the file back to the client via HTTP Response object
- ❸ Build the absolute path to the HTML file

The method `path.join()` uses the Node.js environment variable `_dirname` as a starting point and then builds the full absolute path. `_dirname` represents the directory name of the main module. Let's say, we start the server with the following command:

```
node build/server/simple-websocket-server.js
```

In this case, the value of `_dirname` will be the path to the directory `build/server`. Accordingly, the following code will go two levels up from the `build/server` directory, where the file `simple-websocket-client.html` is located.

```
path.join(__dirname, '../../../../../simple-websocket-client.html')
```

To make this line 100% cross-platform, it's safer to write without using the forward slash as a separator

```
path.join(__dirname, '.', '..', 'simple-websocket-client.html')
```

As soon as any client connects to our WebSocket server via port 8085, the connection event is dispatched on the server, and it'll also receive a reference to the object that represent this particular client. Using the method `send()`, the server sends the greeting to this client. If another client connects to the same socket on port 8085, it'll also receive the same greeting.

**NOTE**

As soon as the new client connects to the server, the reference to this connection is added to the `wsServer.clients` array so you can broadcast messages to all connected clients if needed: `wsServer.clients.forEach(client client.send('...'));`

The content of the file `server/simple-websocket-client.html` is shown in listing 10.11. This is a plain HTML/JavaScript client that uses the browser's `WebSocket` object.

### **Listing 10.11 simple-websocket-client.html**

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
  </head>
  <body>
    <span id="messageGoesHere"></span>

    <script type="text/javascript">
      const ws = new WebSocket("ws://localhost:8085");           ①

      const mySpan = document.getElementById("messageGoesHere");   ②

      ws.onmessage = function(event){                                ③
        mySpan.textContent = event.data;                            ④
      };

      ws.onerror = function(event) {                                ⑤
        console.log(`Error ${event}`);
      }
    </script>
  </body>
</html>
```

- ① Establish the socket connection
- ② Get a reference to the DOM element for showing messages
- ③ The callback for handling messages
- ④ Display the message the `<span>` element.
- ⑤ In case of an error, the browser logs the error message on the console.

When the browser loads `simple-websocket-client.html`, its script connects to your `WebSocket` server at `ws://localhost:8085`. At this point, the server upgrades the protocol from HTTP to `WebSocket`. Note that the protocol is `ws` and not `http`.

To see this sample in action, run `npm install`, compile the code by running the custom command defined in `package.json`

```
npm run build:server
```

The compiled version of all TypeScript files from the `server` directory will be stored in the

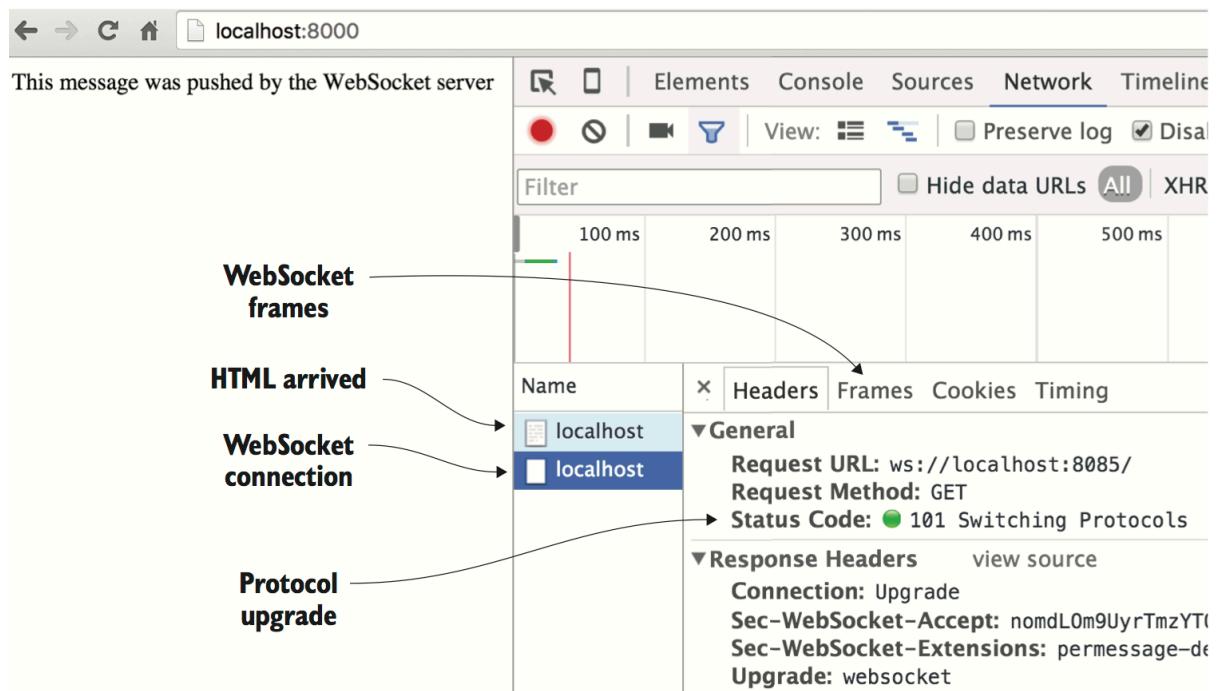
directory build/server. Run this simple WebSocket server as follows:

```
node build/server/simple-websocket-server.js
```

You'll see the following messages on the console:

```
WebSocket server is listening on localhost:8085
HTTP server is listening on localhost:8000
```

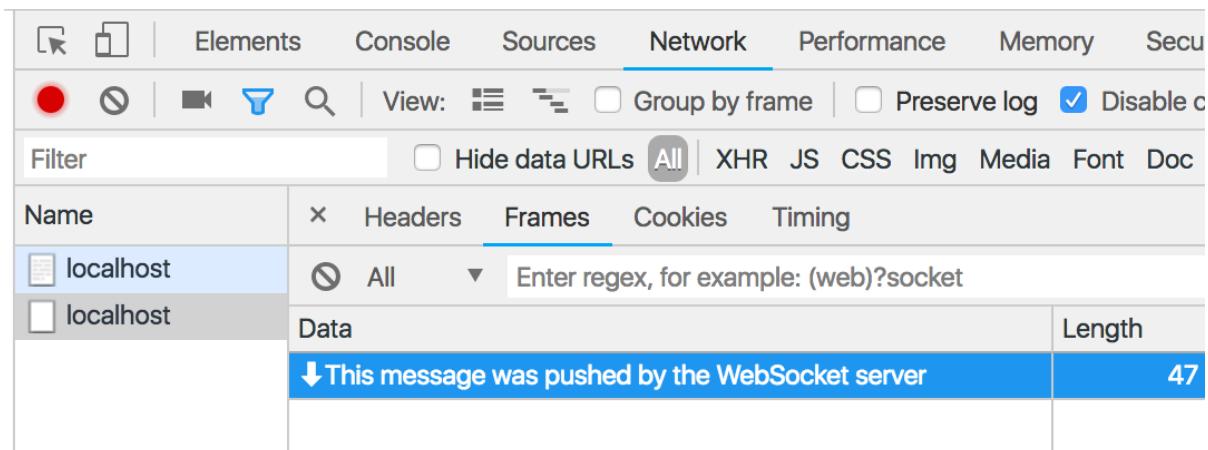
Open the Chrome browser and its Dev Tools at <localhost:8000>. You'll see the message, as shown in figure 10.10 at the top left. Under the Network tab on the right, you see two requests made to the servers running on localhost. The first one loads the file simple-websocket-client.html via HTTP, and the second request goes to the socket opened on the port 8085 on our server.



**Figure 10.10 Getting the message from the socket**

In this example, the HTTP protocol is used only to initially load the HTML file. Then the client requests the protocol upgrade to WebSocket (status code 101), and from then on this web page won't use HTTP.

Click on the tab Frames, and you'll see the content of the message that arrived over the socket connection from the server: "This message was pushed by the WebSocket server" as seen in figure 10.11.



**Figure 10.11 Monitoring the frame content**

Note the arrow pointing down by the message in the Frames tab. It denotes the incoming socket messages. Up-arrows mark the messages sent by the client.

**NOTE**

In this book, WebSocket clients are the apps that run in web browsers.

To send the message from the client to the server, invoke the method `send()` on the browser's `WebSocket` object:

```
ws.send("Hello from the client");
```

Actually, before sending messages, you should always check the status of the socket connection to ensure that it's still active. The `WebSocket` object has a property `readyState`, which can have one of the values shown in table 10.1.

**Table 10.1 Possible values of `WebSocket.readyState`**

Value	State	Description
0	CONNECTING	Socket has been created. The connection is not yet open.
1	OPEN	The connection is open and ready to communicate.
2	CLOSING	The connection is in the process of closing.
3	CLOSED	The connection is closed or couldn't be opened.

You'll see the usage of the `readyState` property in the code of the message server shown in listing 10.17.

**TIP**

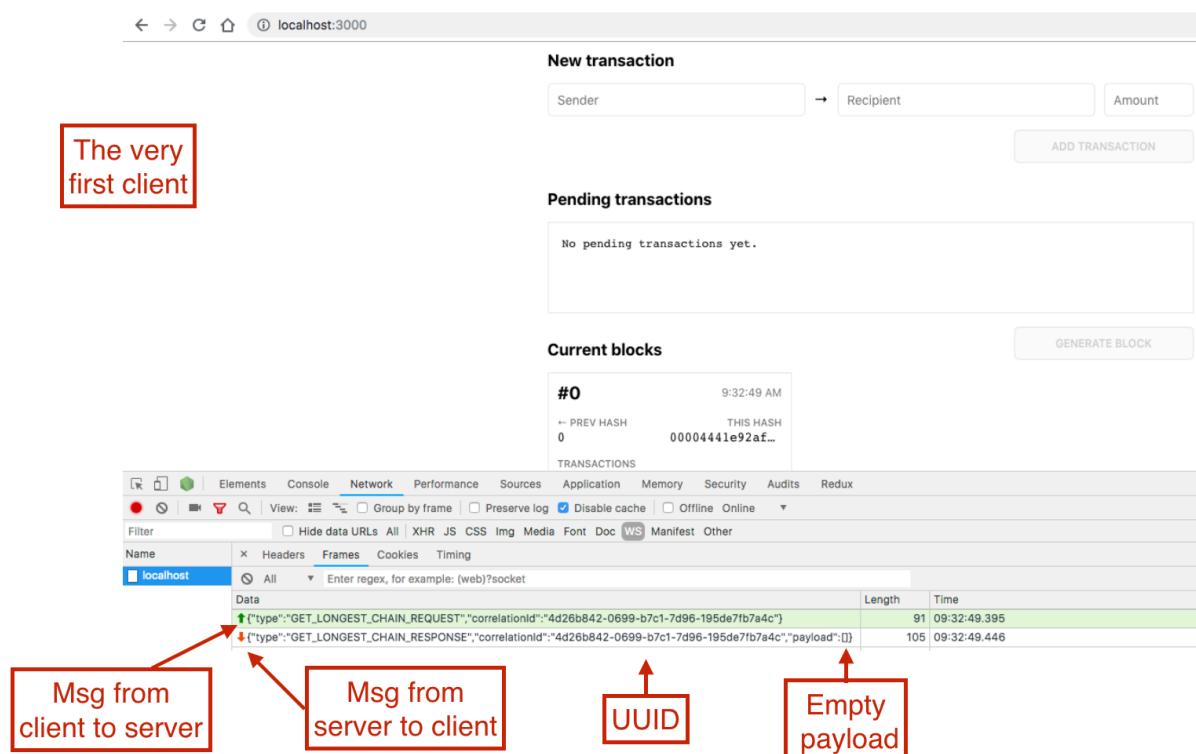
If you keep looking at this table with a limited set of constants, sooner or later TypeScript enums will come to mind, right?

In the next section, we'll go over the process of block mining, and you'll see how the WebSocket server is used there.

## 10.6 Reviewing notifications workflows

In this section, we'll review only those code fragments that are crucial for understanding how the server and communicates with the blockchain clients. We'll start with repeating the scenario of two clients communications illustrated earlier in this chapter, but this time, we'll keep the Chrome Dev Tools panel open to monitor the messages going over the WebSocket connections between the clients and server.

Once again, we'll launch the server on port 3000 using the `npm start` command. Then, we'll open the first client in the browser and connect to `localhost:3000` having the tabs Network | WS opened under the Chrome Dev Panel as seen in figure 10.12. Click on the word `localhost` on the bottom left, and you'll see the messages sent over the socket.



**Figure 10.12 The very first client connected**

The client connects to the server and makes a request to find the longest chain by sending the message of type `GET_LONGEST_CHAIN_REQUEST` to the server.

**TIP**

In the Frames panel, the arrow pointing up means the message went up to the server. The arrow pointing down means that the message arrived from the server.

This client happened to be the very first one on this blockchain, and it received the message `GET_LONGEST_TYPE_RESPONSE` from the server with an empty payload because the blockchain doesn't exist just yet and there are no other nodes just yet. If there were other nodes, the server would have broadcasted the request to other nodes, collected their responses and sent them to the original node-requestor.

The message format is defined the file `shared/messages.ts` shown in listing 10.12. Note that we define custom types using the TypeScript keywords `type`, `interface`, and `enum`.

### **Listing 10.12 shared/messages.ts**

```
export type UUID = string; ①

export interface Message {
  correlationId: UUID; ②
  type: string;
  payload?: any; ③
}

export enum MessageTypes { ④
  GetLongestChainRequest = 'GET_LONGEST_CHAIN_REQUEST',
  GetLongestChainResponse = 'GET_LONGEST_CHAIN_RESPONSE',
  NewBlockRequest = 'NEW_BLOCK_REQUEST',
  NewBlockAnnouncement = 'NEW_BLOCK_ANNOUNCEMENT'
}
```

- ① Declaring an alias type for `UUID`
- ② Declaring a custom `Message` type
- ③ The payload is optional
- ④ Declaring an enum with a set of constants

The messages are being sent asynchronously, and we added a property `correlationId` to be able to match the outgoing and arriving messages. If the client sends the message `GET_LONGEST_CHAIN_REQUEST` to the server, it'll contain a unique correlation ID. Sometime later, the server sends a message `GET_LONGEST_CHAIN_RESPONSE`, which will also contain the correlation ID. By comparing the correlation IDs of outgoing and incoming messages, we'll be able to find matching requests and responses. We use Universally Unique Identifiers (`UUID`) as the values for `correlationId`.

**SIDE BAR** **Generating UUIDs**

As per specification RFC 4122 (see [www.ietf.org/rfc/rfc4122.txt](http://www.ietf.org/rfc/rfc4122.txt)), "A UUID is an identifier that is unique across both space and time, with respect to the space of all UUIDs. Since a UUID is a fixed size and contains a time field, it is possible for values to rollover (around A.D. 3400, depending on the specific algorithm used)."

A UUID is a fixed-length string of ASCII characters in the following format: "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx" as seen in figure 10.12. We borrowed the code for generating UUIDs from StackOverflow (see [stack overflow .com/questions/105034/create-guid-uuid-in-javascript](http://stackoverflow.com/questions/105034/create-guid-uuid-in-javascript)).

In our app, all requests are initiated by the client, so UUID is generated in the client's code. You can find the function `uuid()` in the script `client/lib/cryptography.ts`.

Since UUID is a string of characters, we could declare the property `Message.correlationId` of type `string`. Instead, we created a type alias `UUID` using the `type` keyword (see listing 10.12), and declared `correlationId` of type `UUID`, which increased the code readability, don't you think?

When a client (a.k.a. a blockchain node) starts mining, it sends the message `NEW_BLOCK_REQUEST` inviting other nodes to do the same. A node announces that the mining is complete by sending the message `NEW_BLOCK_ANNOUNCEMENT` to other nodes, and this block becomes a candidate for adding to the blockchain.

### **10.6.1 Reviewing the server's code**

We start the server by loading the script `server/main.ts` shown in listing 10.13 in the Node.js runtime. In this script, we import the Express framework for configuring the HTTP endpoints and the directory where the web client's code is deployed. We also import the `http` object from Node.js, and the `ws` package for the WebSocket support. The script `server/main.ts` starts two server instances on the same port: `httpServer` that supports HTTP and `wsServer` that supports the WebSocket protocol.

## Listing 10.13 server/main.ts

```

import * as express from 'express';
import * as http from 'http';
import * as path from 'path';
import * as WebSocket from 'ws';      ①
import { BlockchainServer } from './blockchain-server';    ②

const PORT = 3000;
const app = express();    ③
app.use('/', express.static(path.join(__dirname, '..', '..', 'public')));   ④
app.use('/node_modules', express.static(path.join(__dirname, '..', '..', 'node_modules')));  ⑤

const httpServer: http.Server = app.listen(PORT, () => {
  if (process.env.NODE_ENV !== 'production') {
    console.log(`Listening on http://localhost:${PORT}`);
  }
});    ⑥

const wsServer = new WebSocket.Server({ server: httpServer });  ⑦
new BlockchainServer(wsServer);    ⑧

```

- ① Enabling our script with the WebSocket support
- ② Importing the class BlockchainServer
- ③ Instantiating Express
- ④ Specifying the location of the client's code
- ⑤ Specifying the location of node\_modules used by the client
- ⑥ Starting the HTTP server
- ⑦ Starting the WebServer
- ⑧ Starting our blockchain notification server

The lines that start with `app.use()` map the URLs that come from the client to the specific resources on the server. This was explained in the sidebar "Resolving paths in Node.js" in the previous section.

**TIP:** The fragment `'..', '..', 'public'` will resolve to `'../../public'` in Unix-based systems and to `'..\..\public'` on Windows.

While instantiating the `WebSocket.Server`, we pass the instance of the existing HTTP server to it. This allows us to run both HTTP and WebSocket servers on the same port. Finally, we instantiate the TypeScript class `BlockchainServer`, which uses WebSockets and its code is located in the script `blockchain-server.ts`. The class `BlockchainServer` is a subclass of `MessageServer` that encapsulates all the work related to WebSocket communications, and we'll review its code later in this section. The first part of the script `blockchain-server.ts` is shown in listing 10.14.

### Listing 10.14 The first part of server/blockchain-server.ts

```

import * as WebSocket from 'ws';
import { Message, MessageTypes, UUID } from '../shared/messages';
import { MessageServer } from './message-server';

type Replies = Map<WebSocket, Message>; ①

export class BlockchainServer extends MessageServer<Message> { ②
    private readonly receivedMessagesAwaitingResponse = new Map<UUID, WebSocket>(); ③

    private readonly sentMessagesAwaitingReply = new Map<UUID, Replies>(); // Used as accumulator
                           for replies from clients.

    protected handleMessage(sender: WebSocket, message: Message): void { ④
        switch (message.type) { ⑤
            case MessageTypes.GetLongestChainRequest :
                return this.handleGetLongestChainRequest(sender, message);
            case MessageTypes.GetLongestChainResponse :
                return this.handleGetLongestChainResponse(sender, message);
            case MessageTypes.NewBlockRequest :
                return this.handleAddTransactionsRequest(sender, message);
            case MessageTypes.NewBlockAnnouncement :
                return this.handleNewBlockAnnouncement(sender, message);
            default :
                console.log(`Received message of unknown type: "${message.type}"`);
        }
    }
}

private handleGetLongestChainRequest(requestor: WebSocket, message: Message): void {
    if (this.clientIsNotAlone) {
        this.receivedMessagesAwaitingResponse.set(message.correlationId, requestor); ⑥
        this.sentMessagesAwaitingReply.set(message.correlationId, new Map()); ⑦
        this.broadcastExcept(requestor, message); ⑧
    } else {
        this.replyTo(requestor, {
            type: MessageTypes.GetLongestChainResponse,
            correlationId: message.correlationId,
            payload: [] ⑨
        });
    }
}

```

- ① Replies from the blockchain nodes
- ② This class extends MessageServer
- ③ A collection of clients' messages waiting for responses
- ④ A handler for all message types
- ⑤ Invoke the appropriate handler based on the message type
- ⑥ Store the client's request using the correlation ID as a key
- ⑦ This map accumulates replies from clients
- ⑧ Broadcast the message to other nodes
- ⑨ There are no longest chains in a single-node blockchain

The method `handleMessage()` serves as a dispatcher for messages received from the clients. It

has a `switch` statement to invoke the appropriate handler based on the received message. This method was declared as `abstract` in the superclass `MessageServer`, and we'll review it later in this section.

For example, if one of the clients sends a message `GetLongestChainRequest`, the method `handleGetLongestChainRequest()` is invoked. First, it stores the request (the reference to the open WebSocket object) in a map using the correlation ID as a key. After that, this method broadcasts the message to other nodes requesting their longest chains. The method `handleGetLongestChainRequest()` can return an object with an empty payload only if the blockchain has one and only node.

**NOTE**

The method signature of `handleMessage()` includes the `void` keyword, which means that it doesn't return a value. Then why its body has a number of return statements? Typically, every `case` clause in the JavaScript `switch` statement has to end with `break`, so the code doesn't "fall through" executing the code in the next `case` clause. Using `return` statements in every `case` clause allows us to avoid these `break` statements and still guarantee that the code doesn't fall through.

Listing 10.15 shows the second part of `blockchain-server.ts`. It has the code that handles the responses from other nodes sending their longest chains.

### Listing 10.15 The second part of server/blockchain-server.ts

```

private handleGetLongestChainResponse(sender: WebSocket, message: Message): void {
    if (this.receivedMessagesAwaitingResponse.has(message.correlationId)) { ①
        const requestor = this.receivedMessagesAwaitingResponse.get(message.correlationId); ②

        if (this.everyoneReplied(sender, message)) {
            const allReplies = this.sentMessagesAwaitingReply.get(message.correlationId).values();
            const longestChain = Array.from(allReplies).reduce(this.selectTheLongestChain); ③
            this.replyTo(requestor, longestChain); ④
        }
    }
}

private handleAddTransactionsRequest(requestor: WebSocket, message: Message): void {
    this.broadcastExcept(requestor, message);
}

private handleNewBlockAnnouncement(requestor: WebSocket, message: Message): void {
    this.broadcastExcept(requestor, message);
}

private everyoneReplied(sender: WebSocket, message: Message): boolean { ⑤
    const repliedClients = this.sentMessagesAwaitingReply
        .get(message.correlationId)
        .set(sender, message);

    const awaitingForClients = Array.from(this.clients).filter(c => !repliedClients.has(c));

    return awaitingForClients.length === 1; ⑥
}

private selectTheLongestChain(currentlyLongest: Message, ⑦
    current: Message, index: number) {
    return index > 0 && current.payload.length > currentlyLongest.payload.length ?
        current : currentlyLongest;
}

private get clientIsNotAlone(): boolean {
    return this.clients.size > 1; ⑧
}
}

```

- ① Find the client that requested the longest chain
- ② Get the reference to the client's socket object
- ③ Find the longest chain
- ④ Relay the longest chain to the client that requested it
- ⑤ Check if every node replied to the request
- ⑥ Only the one who requested the chain left
- ⑦ This method is used while reducing the array of longest chains
- ⑧ Check if there is more than one node in the blockchain

When nodes send their `GetLongestChainResponse` messages, the server uses the correlation ID to find the client who requested the longest chain. When all node replied, the method `handleGetLongestChainResponse()` turns the set `allReplies` into an array and uses the

method `reduce()` to find the longest chain. Then it sends the response back to the client-requestor using the method `replyTo()`.

This is all good, but where's the code that supports the WebSocket protocol and defines such methods as `replyTo()` or `broadcastExcept()`? All this machinery is located in the abstract superclass `MessageServer` shown in listings 10.16 and 10.17.

### **Listing 10.16 The first part of server/message-server.ts**

```
import * as WebSocket from 'ws';

export abstract class MessageServer<T> {

    constructor(private readonly wsServer: WebSocket.Server) {
        this.wsServer.on('connection', this.subscribeToMessages); ①
        this.wsServer.on('error', this.cleanupDeadClients); ②
    }

    protected abstract handleMessage(sender: WebSocket, message: T): void; ③

    protected readonly subscribeToMessages = (ws: WebSocket): void => {
        ws.on('message', (data: WebSocket.Data) => { ④
            if (typeof data === 'string') {
                this.handleMessage(ws, JSON.parse(data)); ⑤
            } else {
                console.log('Received data of unsupported type.');
            }
        });
    };

    private readonly cleanupDeadClients = (): void => { ⑥
        this.wsServer.clients.forEach(client => {
            if (this.isDead(client)) {
                this.wsServer.clients.delete(client);
            }
        });
    };
}
```

- ① Subscribe to messages from a newly connected client
- ② Clean up the references to disconnected clients
- ③ This method is implemented in the class `BlockchainServer`
- ④ The message from the client has arrived
- ⑤ Pass handle the message
- ⑥ Purge the disconnected clients

While reading the code of `MessageServer` you'll recognize many of the TypeScript syntax elements covered in Part 1 of this book. First of all, this class is declared as `abstract` (see section 3.1.5 in chapter 3).

```
export abstract class MessageServer<T>
```

You can't instantiate an abstract class but have to declare a subclass that will provide a concrete implementation of all abstract members. In our case, the subclass `BlockchainServer`

implements the only abstract member `handleMessage()`. Also, the class declaration uses the generic type `T` (see section 4.2 in chapter 4), which is also used as an argument type in methods `handleMessage()`, `broadcastExcept()`, and `replyTo()`. In our app, the concrete type `Message` replaces the generic `<T>` as seen in listing 10.12, but in other apps, it could be a different type.

By declaring the method `handleMessage()` as `abstract`, we state that any subclass of `MessageServer` is free to implement this method any way it likes, as long as the signature of this method will look like this:

```
protected abstract handleMessage(sender: WebSocket, message: T): void;
```

Since we enforced this method signature in the abstract class, we know how the method `handleMessage()` should be invoked when implemented, and we do it in `subscribeToMessages()` as follows:

```
this.handleMessage(ws, JSON.parse(data));
```

Strictly speaking, we can't invoke an abstract method, but during runtime, the keyword `this` will refer to the instance of a concrete class `BlockchainServer` where the method `handleMessage` won't be abstract any longer. Listing 10.17 shows the second part of the class `MessageServer`. These methods implement broadcasting and replying to clients.

### **Listing 10.17 The second part of server/message-server.ts**

```
protected broadcastExcept(currentClient: WebSocket, message: Readonly<T>): void { ①
  this.wsServer.clients.forEach(client => {
    if (this.isAlive(client) && client !== currentClient) {
      client.send(JSON.stringify(message));
    }
  });
}

protected replyTo(client: WebSocket, message: Readonly<T>): void { ②
  client.send(JSON.stringify(message));
}

protected get clients(): Set<WebSocket> {
  return this.wsServer.clients;
}

private isAlive(client: WebSocket): boolean {
  return !this.isDead(client);
}

private isDead(client: WebSocket): boolean { ③
  return (
    client.readyState === WebSocket.CLOSING ||
    client.readyState === WebSocket.CLOSED
  );
}
```

① Broadcast to all other nodes

- ② Send a message to a single node
- ③ Check if a particular client is disconnected

Earlier, in the last line of the script server/main.ts (see listing 10.13) we passed the instance of the WebSocket server to the constructor of the `BlockchainServer`. This object has a property `clients`, which is a collection of all active WebSocket clients. Whenever we need to broadcast a message to all clients, we iterate through this collection as in the method `broadcastExcept()`. If we need to remove a reference to a disconnected client, we also use the property `clients` in the method `cleanupDeadClients()`.

The signatures of the methods `broadcastExcept()` and `replyTo` have the argument of a mapped type  `Readonly<T>`, which we covered in section 5.2 in chapter 5. It takes a type `T` and marks all of its properties as `readonly`. We use the  `Readonly` to avoid accidental modifications of the provided value within the method. You can see more examples in the sidebar "Examples of conditional and mapped types" later in this chapter.

Now let's continue discussing the workflow started in figure 10.12. The second client joins the blockchain and sends a message to the WebSocket server requesting the longest chain, which is just the genesis block at the moment. Figure 10.13 shows the messages that go under the hood over the WebSocket connections. We added the numbers to the messages so it's easier to understand their sequence.

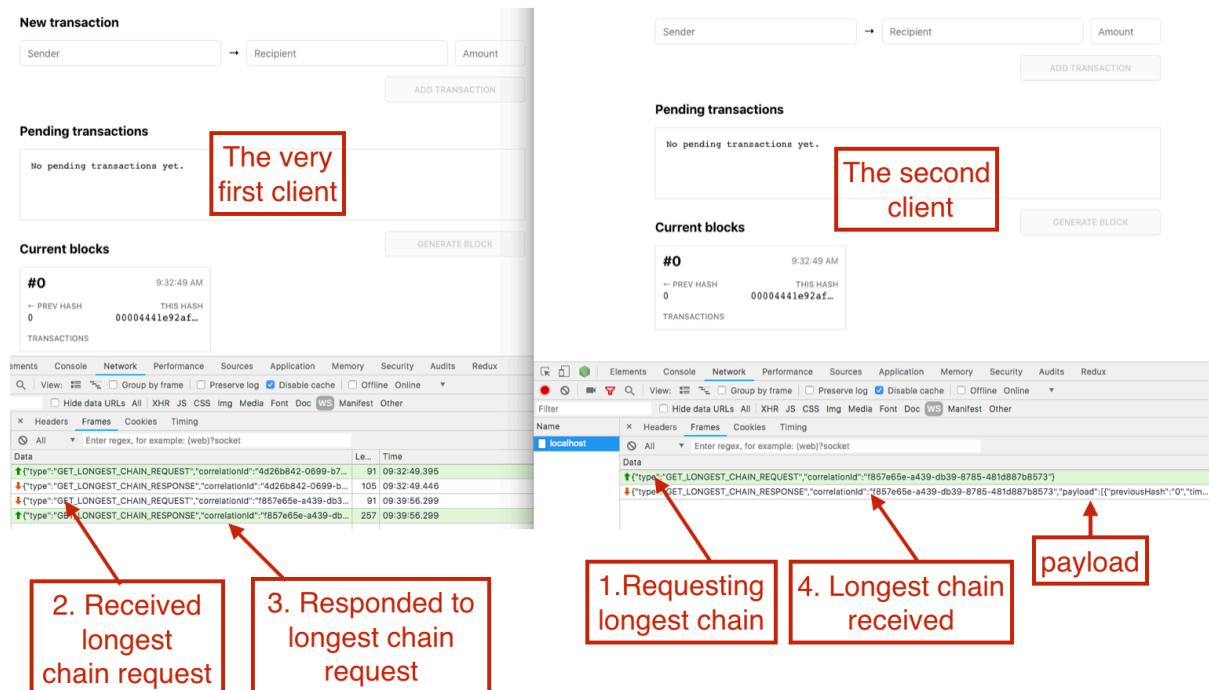


Figure 10.13 The second client connected

1. The second client (the right one) connects to our messaging server requesting the longest

chain. The server broadcasts it to other clients.

2. The first client (the left one) receives this request. This client has a single genesis block, which is its longest chain.
3. The left client sends the message back responding with its longest chain in the message payload.
4. The client on the right received the longest chain in the message payload.

In this example we have only two clients (the left one and the right one), but the WebSocket server broadcast the messages to all connected clients, so all of them would respond.

After that, the client on the left creates two pending transactions as shown in figure 10.14. This is a local event and no messages are being sent to the WebSocket server.

The screenshot shows a web-based blockchain application interface. At the top, there is a header bar with the URL 'localhost:3000' and several icons. Below the header, the main interface is divided into sections:

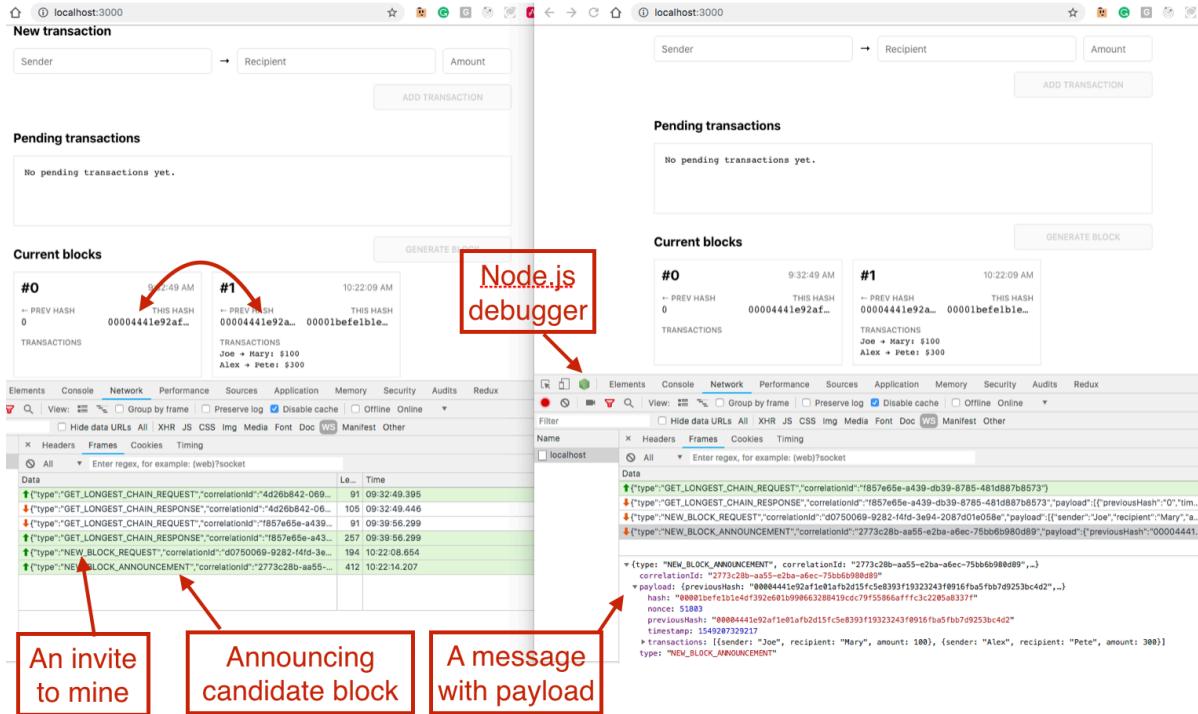
- New transaction:** A form with fields for 'Sender', 'Recipient', and 'Amount', and a 'Send' button labeled 'ADD TRANSACTION'.
- Pending transactions:** A list showing two entries: 'Joe → Mary: \$100' and 'Alex → Pete: \$300'.
- Current blocks:** A section showing the first block '#0' with details like 'PREV HASH' (0), 'THIS HASH' (00004441e92af...), and a 'GENERATE BLOCK' button.

At the bottom of the page, a browser developer tools Network tab is open, showing network traffic. The table lists several requests and responses, including several 'GET\_LONGEST\_CHAIN\_RESPONSE' messages, indicating that the client is receiving responses from the server without sending messages.

Data	Le...	Time
↑ {"type": "GET_LONGEST_CHAIN_REQUEST", "correlationId": "4d26b842-0699-b..."} 91	09:32:49.395	
↓ {"type": "GET_LONGEST_CHAIN_RESPONSE", "correlationId": "4d26b842-0699-..."} 105	09:32:49.446	
↓ {"type": "GET_LONGEST_CHAIN_REQUEST", "correlationId": "f857e65e-a439-db..."} 91	09:39:56.299	
↑ {"type": "GET_LONGEST_CHAIN_RESPONSE", "correlationId": "f857e65e-a439-d..."} 257	09:39:56.299	

**Figure 10.14 Adding transactions doesn't send messages**

Now the left client clicks on the button GENERATE BLOCK starting block mining and requesting other nodes to do the same. The message NEW\_BLOCK\_REQUEST is this invitation to mining. Some time later, the mining is complete (in our case the left miner finished first), and the left client announces the new candidate block by sending the message NEW\_BLOCK\_ANNOUNCEMENT. Figure 10.15 shows the messages related to the new block mining, and the content of the message NEW\_BLOCK\_ANNOUNCEMENT.



**Figure 10.15 New block messaging**

Now both clients show the same two blocks #0 and #1. Note that the hash value of the block #0 and the previous hash value of the block #1 are the same.

Listing 10.16 shows the content of the NEW\_BLOCK\_ANNOUNCEMENT message. We shortened the hash values to improve readability.

### Listing 10.18 A message containing the new block

```

correlationId: "2773c28b-aa55-e2ba-a6ec-75bb6b980d89"      ①
payload: {previousHash: "00004441e92af1",...}
  hash: "00001befb1ble4df392e601..."    ②
  nonce: 51803    ③
  previousHash: "00004441e92a..."    ④
  timestamp: 1549207329217
  transactions: [{sender: "Joe", recipient: "Mary", amount: 100},
    {sender: "Alex", recipient: "Pete", amount: 300}]    ⑤
type: "NEW_BLOCK_ANNOUNCEMENT"    ⑥

```

- ① The correlation ID (UUID)

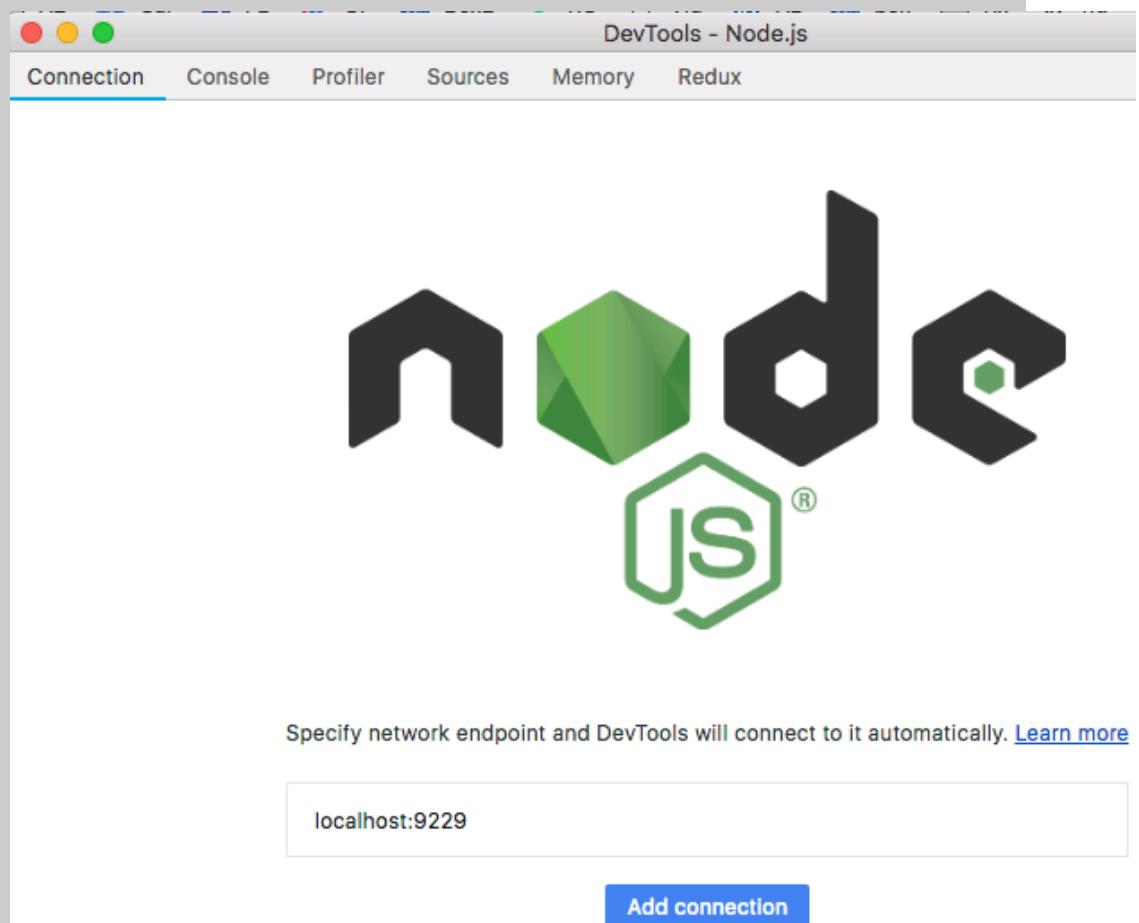
- ② The hash value of the block #1
- ③ The calculated nonce (the proof of work)
- ④ The hash value of the block #0
- ⑤ The block's transactions
- ⑥ The message type

Revisit listing 10.12 to find the definitions of custom data types used in this message.

#### SIDE BAR

#### Debugging the Node.js code in the browser

Since we started Node.js with the `--inspect` option, when you open Chrome Dev Tools, you'll see a green octagon to the left of the Elements tab as seen in figure 10.15. It represents the Node.js debugger. Click on it to open a detached window of the Node.js Dev Tools shown in figure 10.16.



**Figure 10.16 Opening Chrome Dev Tools for Node.js**

Revisit listing 10.8 that shows the output in the terminal window where we started the server. The Node.js connects with Dev Tools via the port 9229. By default, this window shows you the content from the Connection tab. Switch to the tab Sources, find the TypeScript code that you want to debug and put a breakpoint there. Figure 10.17 shows the breakpoint in line 9 in the script blockchain-server.ts.

```

import * as WebSocket from 'ws';
import { Message, MessageType, UUID } from '../shared/messages';
import { MessageServer } from './message-server';

export class BlockchainServer extends MessageServer<Message> {
  private readonly receivedMessagesAwaitingResponse = new Map<UUID, Message>();
  protected handleMessage(sender: WebSocket, message: Message): void {
    switch (message.type) {
      case MessageType.GetLongestChainRequest: return this.handleGetLongestChainRequest();
      case MessageType.GetLongestChainResponse: return this.handleGetLongestChainResponse();
      case MessageType.NewBlockRequest: return this.handleNewBlockRequest();
      case MessageType.NewBlockAnnouncement: return this.handleNewBlockAnnouncement();
      default: {
        console.log(`Received message of unknown type: ${message.type}`);
      }
    }
  }

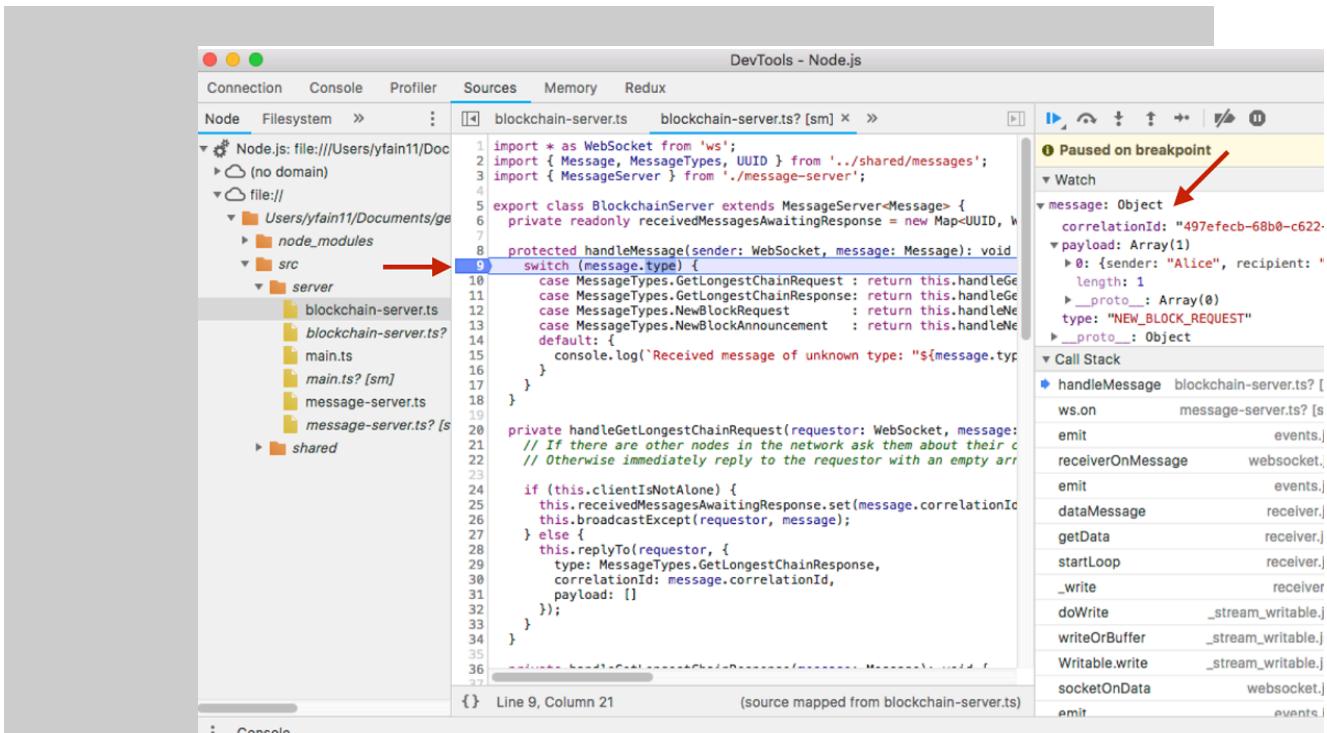
  private handleGetLongestChainRequest(requestor: WebSocket, message: Message) {
    // If there are other nodes in the network ask them about their chain
    // Otherwise immediately reply to the requestor with an empty array
    if (this.clientIsNotAlone) {
      this.receivedMessagesAwaitingResponse.set(message.correlationId, message);
      this.broadcastExcept(requestor, message);
    } else {
      this.replyTo(requestor, {
        type: MessageType.GetLongestChainResponse,
        correlationId: message.correlationId,
        payload: []
      });
    }
  }
}

```

Line 8, Column 62 (source mapped from blockchain-server.ts)

**Figure 10.17 Selecting the Sources tab**

We want to intercept the moment when the client will send a message to the server. The screenshot in figure 10.18 was taken after the client created a transaction "Alice Julie \$200" and clicked on the button GENERATE BLOCK.



**Figure 10.18 Debugging Node.js in Chrome browser**

The execution obediently stopped at line 9 as seen in figure 10.18. We added the variable `message` to the Watch panel on top right and can debug or monitor this variable. All the features of the Chrome Dev Tool's debugger are available for the Node.js code as well.

The next question is, "Who sent the message with the payload containing the newly mined block?" The web client did, and in the next section we'll review the relevant code fragments from the web client.

### 10.6.2 Reviewing the client's code

We use the word "client" because in this app we're discussing communications between a web app and a server. But "node" would be the right word here. Our node's code is implemented as a web app while in a real block-chain the nodes that mine blocks and the UI where users can add transactions would be separate app.

#### NOTE

The main subject of this chapter is introducing the messaging server. We're not going to review each and every line of the code but will show you the code fragments that are crucial for understanding the client's implementation.

The code that runs in the web browser is located in the directory `src/client`. For the HTML rendering, we use a small library called `lit-html` (see [www.npmjs.com/package/lit-html](http://www.npmjs.com/package/lit-html)), which

allows you to write HTML templates with the JavaScript template literals. This library uses tagged templates strings explained in section A.3.1 in appendix A.

The lit-html library uses tagged templates, which are plain JavaScript functions combined with HTML. In short, this library turns a string with HTML markup into the browser DOM node and re-renders it when it's data changes.

lit-html efficiently renders templates to DOM by updating only those nodes that need to display updated values. Below is the example from the lit-html documentation that defines tagged template with the expression \${name}, then passes Steve as a name to render `<div>Hello Steve</div>` and then, when the value of the variable name changes to Kevin, it updates only the name in that `<div>`.

### **Listing 10.19 How lit-html renders HTML**

```
import {html, render} from 'lit-html'; ①
const helloTemplate = (name) => html`<div>Hello ${name}!</div>`; ②
render(helloTemplate('Steve'), document.body); ③
render(helloTemplate('Kevin'), document.body); ④
```

- ① Import the functions `html` and `render`
- ② Declare the tagged template
- ③ Render the `<div>` greeting Steve
- ④ Replace the name Steve to Kevin in the `<div>`

#### **NOTE**

The main idea of lit-html is that it parses HTML and builds the DOM node once, and after that, it only renders the new values of the embedded variables when they change. It doesn't create the virtual DOM like React.js and it doesn't use a change detector like Angular. It just updates the variable values.

When you'll be reading code in the scripts located in the client/ui directory, you won't see `html` files there, but rather the invocations of the function `render()` similar to the example above. If you're familiar with Angular or React libraries, think of a class that has the `render()` function as a UI component. `html` is the main API of lit-html and you'll see that every `render()` function uses it.

Let's consider a small class `PendingTransactionsPanel` that knows how to render the panel with pending transactions. Listing 10.20 shows the file `ui/pending-transactions-panel.ts` that implements the function `render()`:

## Listing 10.20 pending-transaction-panel.ts

```
import { html, TemplateResult } from '../../../../../node_modules/lit-html/lit-html.js';
import { BlockchainNode } from '../lib/blockchain-node.js';
import { Callback, formatTransactions, Renderable, UI } from './common.js';

export class PendingTransactionsPanel implements Renderable<Readonly<BlockchainNode>> {

    constructor(readonly requestRendering: Callback) {} ①

    render(node: Readonly<BlockchainNode>): TemplateResult { ②
        const shouldDisableGenerate = node.noPendingTransactions || node.isMining;
        const formattedTransactions = node.hasPendingTransactions
            ? formatTransactions(node.pendingTransactions)
            : 'No pending transactions yet.';

        return html` ③
            <h2>Pending transactions</h2>
            <pre class="pending-transactions__list">${formattedTransactions}</pre>
            <div class="pending-transactions__form">${UI.button('GENERATE BLOCK',
                shouldDisableGenerate)}</form>
            <div class="clear"></div>
        `;
    }
}
```

- ① Provide the callback function to the constructor
- ② The argument type of the function render() is defined in blockchain-node.ts
- ③ lit-html needs the tagged template html

The class PendingTransactionsPanel implements the Renderable interface that forces it to have the property `requestRendering` and the method `render()` that are defined in the file `common.ts` as follows:

## Listing 10.21 A fragment from ui/common.ts

```
// other type definition go here
export type Callback = () => void; ①

export interface Renderable<T> { ②
    requestRendering: Callback; ③
    render(data: T): TemplateResult; ④
}
```

- ① Defining the custom type of a callback function
- ② Defining a generic interface
- ③ A callback function
- ④ The function to render HTML using lit-html

You'll find multiple `render()` functions in the client's code, but all of them work in a similar way: insert the values of JavaScript variable into HTML templates and refresh the corresponding portion of the UI when the value in these variables changes.

The client's top level class is called `Application`, and it also implements the interface `Renderable` so lit-html will know how to render it. The file `client/main.ts` creates an instance of the class `Application` and passes a callback to it that invokes its `render()` function.

### **Listing 10.22 ui/main.ts**

```
import { render } from '../../../../../node_modules/lit-html/lit-html.js';
import { Application } from './ui/application.js';

let renderingIsInProgress = false; ①
let application = new Application(async () => { ②

    if (!renderingIsInProgress) {
        renderingIsInProgress = true;
        await 0;
        renderingIsInProgress = false;
        render(application.render(), document.body); ③
    }
});
```

- ① A flag to prevent double rendering
- ② Passing a callback to the `Application` instance
- ③ Invoking the `render()` function

**TIP**

Even though the `render()` function takes the `document.body` as an argument, the library lit-html doesn't re-render the entire page but only the values in the templates' placeholders that have changed.

When the `Application` class is instantiated, it receives the callback function (the constructor's argument is called `requestRendering`), which invokes the `render()` function. We'll show you two fragments of the class `Application`, and when you see there `this.requestRendering()`, it invokes this callback. The first fragment of the script `application.ts` is shown in listing 10.23.

### Listing 10.23 The first fragment of ui/application.ts.

```

export class Application implements Renderable<void> {
    private readonly node: BlockchainNode;
    private readonly server: WebsocketController; ①

    private readonly transactionForm = new TransactionForm(this.requestRendering); ②
    private readonly pendingTransactionsPanel = new PendingTransactionsPanel(this.requestRendering);
    ②

    private readonly blocksPanel = new BlocksPanel(this.requestRendering); ②

    constructor(readonly requestRendering: Callback) { ③
        this.server = new WebsocketController(this.handleServerMessages); ④
        this.node = new BlockchainNode(); ⑤

        this.requestRendering();
        this.initializeBlockchain(); ⑥
    }

    private async initializeBlockchain() {
        const blocks = await this.server.requestLongestChain(); ⑦
        if (blocks.length > 0) {
            this.node.initializeWith(blocks);
        } else {
            await this.node.initializeWithGenesisBlock();
        }

        this.requestRendering();
    }

    render(): TemplateResult { ⑧
        return html`
            <h1>Blockchain node</h1>
            <aside>${this.statusLine}</aside>
            <section>${this.transactionForm.render(this.node)}</section> ⑨
            <section>
                <form @submit="${this.generateBlock}">
                    ${this.pendingTransactionsPanel.render(this.node)} ⑨
                </form>
            </section>
            <section>${this.blocksPanel.render(this.node.chain)}</section> ⑨
        </main>
    `;
    }
}

```

- ① This object is responsible for the WebSocket communications
- ② Passing the top-level callback to each UI component
- ③ The callback reference will be stored in the property requestRendering
- ④ Connect to the WebSocket server
- ⑤ All blockchain/node creation logic is here
- ⑥ Initialize the blockchain
- ⑦ Request the longest chain from all nodes
- ⑧ This method renders the UI components
- ⑨ Re-render the child component

The initial rendering is initiated from the constructor by invoking `this.requestRendering()`. After several seconds, the second rendering is done from the method `initializeBlockchain()`. Child UI components get the reference to the callback `requestRendering`, and they can decide when to refresh the UI.

The method `initializeBlockchain()` is invoked after the initial rendering of the UI, and this method requests the WebSocket server to give the longest chain. If this node is not the very first and only one, the longest chain is returned and is rendered in the block panel. Otherwise, the genesis block is generated and rendered. Both `requestLongestChain()` and `initializeWithGenesisBlock()` are asynchronous operations, and the code waits for their completion by using the JavaScript `await` keyword.

Listing 10.24 shows two methods from `application.ts`. The method `handleServerMessages()` is invoked when the WebSocket server sends a message to the client. Using the `switch` statement, it invokes the appropriate handler based on the message type. The method `handleGetLongestChainRequest()` is invoked when the client receives the request to send its longest chain.

#### **Listing 10.24 The second fragment of ui/application.ts.**

```
private readonly handleServerMessages = (message: Message) => { ①
  switch (message.type) { ②
    case MessageType.GetLongestChainRequest: return
      this.handleGetLongestChainRequest(message);
    case MessageType.NewBlockRequest : return this.handleNewBlockRequest(message);
    case MessageType.NewBlockAnnouncement : return this.handleNewBlockAnnouncement(message);
    default: {
      console.log(`Received message of unknown type: "${message.type}"`);
    }
  }
}

private handleGetLongestChainRequest(message: Message): void { ③
  this.server.send({
    type: MessageType.GetLongestChainResponse,
    correlationId: message.correlationId,
    payload: this.node.chain
  });
}
```

- ① Handle messages from the WebSocket server
- ② Dispatch the message to the right handler
- ③ The node sends its own chain to the server

#### **SIDE BAR Examples of conditional and mapped types**

In chapter 5, we introduced conditional and mapped types. In this sidebar, you'll see how we use them in the blockchain app. We're not going to review the entire script client/lib/blockchain-node.ts, but we'd like you to practice in reading and understanding generics and present a use case for the mapped type `Pick`. Let's discuss the custom types `Omit`, `WithoutHash`, and `NotMinedBlock` presented in listing 10.25.

### Listing 10.25 A fragment from blockchain-node.ts

```
export interface Block {      ①
  readonly hash: string;
  readonly nonce: number;
  readonly previousHash: string;
  readonly timestamp: number;
  readonly transactions: Transaction[];
}

export type Omit<T, K> = Pick<T, Exclude<keyof T, K>>;    ②
export type WithoutHash<T> = Omit<T, 'hash'>;            ③
export type NotMinedBlock = Omit<Block, 'hash' | 'nonce'>;   ④
```

- ① Declaring the `Block` type
- ② A helper type that uses `Pick`
- ③ Declare a type similar to `Block` but without hash
- ④ Declare a type similar to `Block` but without hash and nonce

The interface defines a custom type `Block` with five properties, and all of them are required and `readonly`, i.e. can be initialized only during the instantiation of the block. But creation of a block takes some time, and not all the values for these properties are available during instantiation yet.

On one hand, we'd like to benefit from strong typing, but on the other, we'd like to have some freedom in initializing some properties after the `Block` instance is created.

TypeScript comes with a mapped type `Pick` and conditional type `Exclude`, and we use them to define a new type excluding some of the properties of the existing ones. `Exclude` enumerates the remaining properties. The type `Pick` allows you to create a type from the provided list of properties.

Hence the following line means that we want to declare a generic type `Omit` that can take a type `T` and a key `K` and exclude from `T` the properties that match `K`:

```
type Omit<T, K> = Pick<T, Exclude<keyof T, K>>;
```

The generic type `Omit` can be used with any type `T`, and `keyof T` returns a list of properties of the concrete type. For example, if we provide `Block` as type `T`, the construct `keyof T` will represent the list of properties defined in the interface `Block`. Using `Exclude<keyof T, K>` allows us to remove some of the properties from the list, and `Pick` will create a new type from that new list of properties.

The `Omit` type in the following like means that we want to declare a type that will have all the properties of type `T` except `hash`:

```
type WithoutHash<T> = Omit<T, 'hash'>;
```

The process of generation of the block's hash value takes time, and until we have it, we can use the type `WithoutHash<Block>` that won't have the property `hash`. You can specify more than one property to be excluded as in the following line:

```
type NotMinedBlock = Omit<Block, 'hash' | 'nonce'>;
```

Using this type is illustrated below:

```
let myBlock: NotMinedBlock;

myBlock = {
  previousHash: '123',
  transactions: ["Mary paid Pete $100"]
};
```

Our type `NotMinedBlock` will always be `Block` minus `hash` and `nonce`. If at some point, someone will add another required `readonly` property to the interface `Block`, the assignment to the variable `myBlock` won't compile complaining that the newly added property must be initialized. In JavaScript, we'd get a runtime error in the same scenario.

Listings 10.23 and 10.24 show the most part of the application.ts, which communicates with the WebSocket server via the class `websocketController`, which gets the callback to handle messages via its constructor. Let's get familiar with the code of the `websocketController` by reviewing the workflow when the user clicks on the button GENERATE BLOCK. This should result in the following actions:

1. Broadcast the message `GET_LONGEST_CHAIN_REQUEST` to other nodes via the WebSocket server.
2. Broadcast the message `NEW_BLOCK_REQUEST` to other nodes via the WebSocket server.
3. Mine the block.
4. Process all `GET_LONGEST_CHAIN_RESPONSE` messages received from other nodes.
5. Broadcast the message `NEW_BLOCK_ANNOUNCEMENT` to other nodes and save the candidate block locally.

In listing 10.23, the `render()` method contains a form that looks like this:

```
<form @submit="${this.generateBlock}">
  ${this.pendingTransactionsPanel.render(this.node)}
</form>
```

The panel with pending transaction as well as the button GENERATE BLOCK are implemented in the class `PendingTransactionsPanel`. We use the directive `@submit` from `lit-html`, and when the user clicks on the button GENERATE BLOCK, the `async` method `generateBlock()` is invoked.

### **Listing 10.26 The method generateBlock() of the class Application**

```
private readonly generateBlock = async (event: Event): Promise<void> => {
  event.preventDefault();          ①

  this.server.requestNewBlock(this.node.pendingTransactions);  ②
  const miningProcessIsDone = this.node.mineBlockWith(this.node.pendingTransactions);  ③

  this.requestRendering();        ④

  const newBlock = await miningProcessIsDone;      ⑤
  this.addBlock(newBlock);            ⑥
};
```

- ① Prevent the page refresh
- ② Let all other nodes know that this one starts mining
- ③ Start the block mining
- ④ Refresh the status on the UI
- ⑤ Wait for the mining to complete
- ⑥ Add the block to the local blockchain

When we announce that we start block mining, we provide the list of pending transactions `this.node.pendingTransactions`, so other nodes can compete and try to mine the block for the same transactions faster. Then, this node starts mining as well.

## SIDE BAR When the new block is rejected

The method `Application.addBlock()` invokes the method `addBlock()` in the class `BlockchainNode`. We already discussed the process of block mining, in In chapters 8 and 9, but we'd like to highlight the code fragment that rejects an attempt to add a new block:

### Listing 10.27 A fragment from the `addNode()` method from `BlockchainNode`

```
const previousBlockIndex = this._chain.findIndex(b => b.hash === newBlock.previousHash);
if (previousBlockIndex < 0) { ①
    throw new Error(`#${errorMessagePrefix} - there is no block in the chain with the sp
}

const tail = this._chain.slice(previousBlockIndex + 1);
if (tail.length >= 1) { ②
    throw new Error(`#${errorMessagePrefix} - the longer tail of the current node takes pr
}
```

- ① Does the new block contains an existing previousHash?
- ② Does the chain already have at least one extra block

This code is related to the failed attempt of adding the block shown at the bottom right corner of figure 10.9. First we check if the new block's value of the `previousHash` even exists in the blockchain. It may exist, but not in the last block.

The second if-statement checks if there is at least one block after the one that contains this `previousHash`. This would mean that at least one new block was already added to the chain (e.g. received from other nodes). In this case the longest chain takes precedence and the newly generated block is rejected.

Now let's review the code related to WebSocket communications. For example, if you read the code of the method `addBlock()`, you'll see the following line there:

```
this.server.announceNewBlock(block);
```

This is how the node asks the WebSocket server to announce the fresh from the oven block candidate. The `Application` class also has a method `handleServerMessages()`, which handles the messages arriving from the server implemented in the file `client/lib/websocket-controller.ts`. This file declares the interface `PromiseExecutor` and class `WebsocketController`.

The class `Application` creates an instance of `WebsocketController`, which is our only contact for all communications with the server. When the client needs to send a message to the server

it'll use such methods as `send()` or `requestLongestChain()`, but sometimes, the server will be sending messages to the clients. That's why, we pass a callback method to the constructor of `WebsocketController` as seen in listing 10.28.

### Listing 10.28 The first fragment of the class `WebsocketController`

```
export class WebsocketController {
    private websocket: Promise<WebSocket>;
    private readonly messagesAwaitingReply = new Map<UUID, PromiseExecutor<Message>>(); ①

    constructor(private readonly messagesCallback: (messages: Message) => void) { ②
        this.websocket = this.connect(); ③
    }

    private connect(): Promise<WebSocket> {
        return new Promise((resolve, reject) => {
            const ws = new WebSocket(this.url);
            ws.addEventListener('open', () => resolve(ws)); ④
            ws.addEventListener('error', err => reject(err)); ④
            ws.addEventListener('message', this.onMessageReceived); ④
        });
    }

    private readonly onMessageReceived = (event: MessageEvent) => { ⑤
        const message = JSON.parse(event.data) as Message;

        if (this.messagesAwaitingReply.has(message.correlationId)) {
            this.messagesAwaitingReply.get(message.correlationId).resolve(message);
            this.messagesAwaitingReply.delete(message.correlationId);
        } else {
            this.messagesCallback(message);
        }
    }

    async send(message: Partial<Message>, awaitForReply: boolean = false): Promise<Message> {
        return new Promise<Message>(async (resolve, reject) => {
            if (awaitForReply) {
                this.messagesAwaitingReply.set(message.correlationId, { resolve, reject }); ⑥
            }
            this.websocket.then(
                ws => ws.send(JSON.stringify(message)),
                () => this.messagesAwaitingReply.delete(message.correlationId)
            );
        });
    }
}
```

- ① A map of WebSocket client waiting for responses
- ② Passing the callback to the constructor
- ③ Connecting to the WebServer
- ④ Assigning callbacks to WebSocket messages
- ⑤ Handling incoming messages
- ⑥ Storing the message that needs a response

**TIP**

The type `Partial` is covered in chapter 5.

The method `connect()` connects to the server and subscribes to the standard WebSocket

messages. All these actions are wrapped into a `Promise`, so if this method returns, we can be sure that the WebSocket connection is established and all handlers are assigned. An additional benefit is that now we could use the `async/await` keywords with these asynchronous functions.

The method `onMessageReceived()` handles the messages coming from the server. De facto, it's the message router. There, we deserialize the message and check its correlation ID. If the incoming message is a response to another message, the following code will return `true`:

```
messagesAwaitingReply.has(message.correlationId)
```

Every time the client sends the message out, we store the reference to its correlation ID mapped to the `PromiseExecutor` in `messagesAwaitingReply`. The `PromiseExecutor` knows which client waits for the response.

### **Listing 10.29 The interface PromiseExecutor**

```
interface PromiseExecutor<T> { ❶
  resolve: (value?: T | PromiseLike<T>) => void; ❷
  reject: (reason?: any) => void; ❸
}
```

- ❶ This type is used by tsc internally to initialize the promise
- ❷ Enforcing the signature of the `resolve()` method
- ❸ Enforcing the signature of the `reject()` method

For constructing a `Promise`, the interface `PromiseExecutor` (see the declaration in `lib.es2015.promise.d.ts`) uses the type `PromiseLike`. TypeScript has a number of files that end with "Like" (e.g. `ArrayLike`), which define subtypes that have less properties than the original type. Promises may have different constructor signatures, e.g. all them have `then()` but may or may not have `catch()`. The `PromiseLike` tells tsc, "I don't know what's the implementation of this thing, but at least it's *thenable*".

When the client sends a message that requires a response, it uses the method `send()` shown in listing 10.24, which stores the reference to the client's messages using the correlation ID and the object of type `PromiseExecutor`:

```
this.messagesAwaitingReply.set(message.correlationId, { resolve, reject });
```

The response is asynchronous and we don't know when it'll arrive. In such scenarios, a JavaScript caller can create a `Promise` providing the `resolve` and `reject` callbacks as explained in appendix A in section A.10.2. That's why, the method `send()` wraps the code sending a message into a `Promise`, and we store the references to the object containing these `resolve` and `reject` along with the correlation ID in `messagesAwaitingReply`.

For the messages expecting replies we need a way to push the reply when it arrives. We could

use a `Promise`, which could be manually resolved (or rejected) only from the callback that we pass to the `Promise((resolve, reject))` constructor. We pass the callback to the constructor, but to keep this pair of resolve/reject functions till the moment we get a reply, we create an object `PromiseExecutor` that has exactly these two properties - `resolve` and `reject` and puts them aside. In other words, `PromiseExecutor` is just a container for two callbacks.

The interface `PromiseExecutor` just describes the type of the object that we store in the map `messagesAwaitingReply`. When the response arrives, the method `onMessageReceived()` finds the `PromiseExecutor` object by the correlation ID, invoke `resolve()`, and deletes this message from the map:

```
this.messagesAwaitingReply.get(message.correlationId).resolve(message);
this.messagesAwaitingReply.delete(message.correlationId);
```

Now that you understand how the `send()` method works, you should be able to understand the code that invokes them. Listing 10.30 shows how the client can request the longest chain.

### Listing 10.30 Requesting the longest chain

```
async requestLongestChain(): Promise<Block[]> {
  const reply = await this.send(①
    {②
      type: MessageTypes.GetLongestChainRequest,
      correlationId: uuid()
    }, true); ③
  return reply.payload; ④
}
```

- ① Invoking the method `send()` and waiting for the response
- ② The first argument is a `Message` object
- ③ `true` means "waiting for a reply"
- ④ Return the payload from the response

The class `WebsocketController` has several other methods dealing with other types of messages, and they are implemented similar to `requestLongestChain()`.

To see this app in action, run `npm install` and then `ntp start`. Open a couple of browser windows and try to mine some blocks.

## 10.7 Summary

In this chapter you learned:

- How to configure a project that has the client and the server portions written in TypeScript
- How to have more than one file with the TypeScript compiler's options.
- What's the difference between HTTP and WebSocket protocols

- How to use TypeScript for developing a WebSocket server that runs under Node.js
- How to arrange communication between the blockchain node via a WebSocket server

Inter-node communication via a WebSocket server was the focus of this chapter. In the next one, we'll switch back to UI and create another version of the blockchain app with the front end developed using the Angular framework and TypeScript.



# *Developing Angular apps with TypeScript*

## **This chapter covers**

- A quick intro to the Angular framework
- How to generate, build and serve a Web app written in Angular and TypeScript
- How Angular implements dependency injection

In October of 2014, a team of Google developers was considering creating a new language AtScript, which would extend TypeScript and would be used for developing the all new Angular 2 framework. In particular, AtScript would support decorators, which were not in TypeScript back then.

Then one of the Googlers suggested to meet with the TypeScript team from Microsoft and ask if they would be willing to add decorators to TypeScript itself. The Microsoft folks agreed, and the Angular 2 framework was written in TypeScript, which also became a recommended language for developing Angular apps. Today, more than a million developers use Angular with TypeScript for developing web apps, and this gave a tremendous boost to popularity of this language.

At the time of this writing, Angular has 47K stars and 900 contributors on GitHub. Let's pay respect to this framework, and see how to use TypeScript for developing Angular apps. This chapter is a brief intro to the Angular ramework.

Today, the main players in the market for developing web apps (besides super-popular jQuery) are Angular and React.js, while Vue.js is getting more and more traction every month. Angular is a framework. React.js is a library that does one thing really well: rendering of the UI in the browser's DOM.

Chapter 13 will get you started with development of web apps using React.js and TypeScript. In chapter 14, we'll use React for developing another version of the blockchain client. Chapter 15 covers the basics of Vue.js, and in chapter 16, we'll write yet another version of the blockchain client in Vue.

**NOTE**

It's very difficult to provide a detailed coverage of Angular development with TypeScript in a 40-page chapter. If you're interested in learning Angular, read the second edition of our book "Angular Development with TypeScript" available at [www.manning.com/books/angular-development-with-typescript-second-edition](http://www.manning.com/books/angular-development-with-typescript-second-edition) and other book sellers.

The difference between a framework and library is that a framework forces you to write your apps in a certain way, while a library offers you certain features that you can use in your app written in any way you like. In this sense, Angular is definitely a framework or even more than a framework - it's an opinionated platform (a framework that can be extended) that has everything you need to develop a web app:

- Dependency injection support
- Angular Material - a library of modern-looking UI components
- The router for arranging user's navigation in the app
- A module to communicate with the HTTP servers
- Means for splitting the app into deployable modules that can be loaded either eagerly or lazily
- Powerful Forms support
- A library of reactive extensions (RxJS) to handle data streams
- A development web server that supports live code reloads
- The build tools for optimizing and bundling for deployment
- The command-line interface to quickly scaffold an app, a library, or smaller artifacts like components, modules, services, et al.

Let's quickly create and run a simple web app that would allow us to demo some of the Angular features.

## **11.1 Generating and running a new app with Angular CLI**

CLI stands for Command Line interface, and Angular CLI is a tool that can generate and configure a new Angular project in less than a minute. To install Angular CLI on your computer run the following command:

```
npm install @angular/cli -g
```

Now you can run CLI from the Terminal window using the command `ng` with parameters. The `ng` command can be used for generating a new workspace, application, library, component,

service and more. All parameters of the `ng` command are described at [angular.io/cli](https://angular.io/cli), and you can also see what's available by running `ng help` in the Terminal window. To get help on a specific parameter, run `ng help` followed by the name of the parameter you need help with, e.g. `ng help new`.

**NOTE**

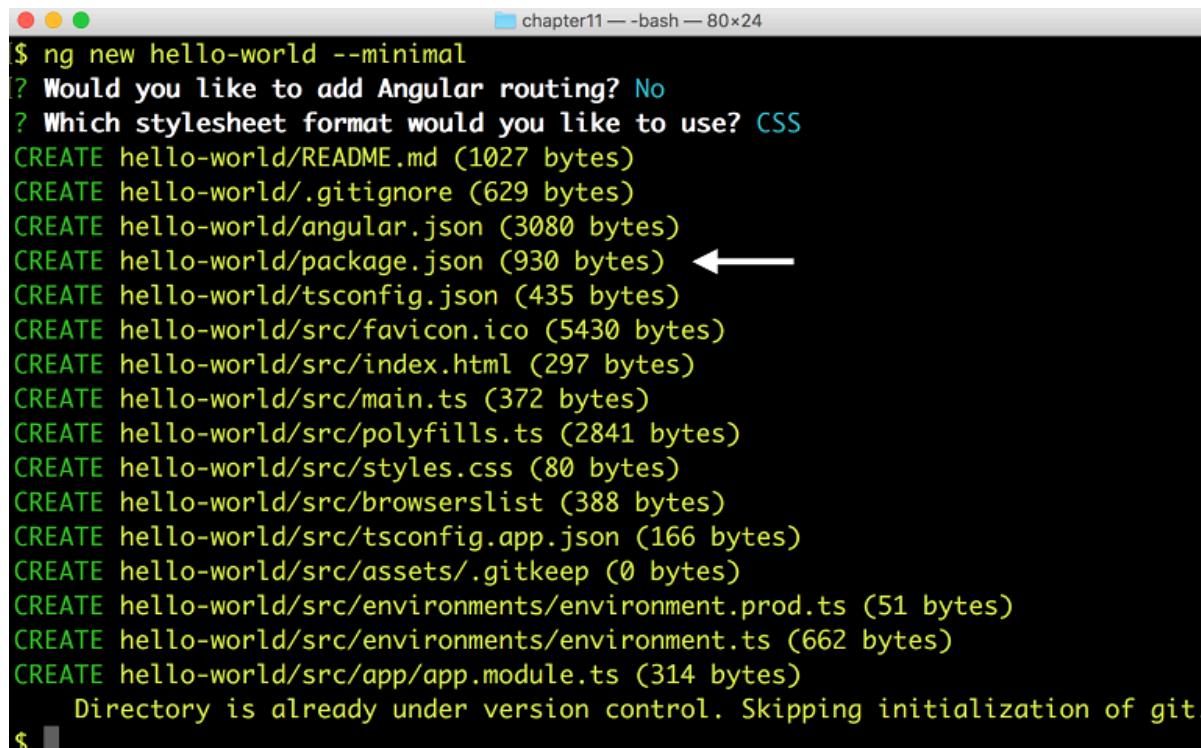
In this chapter we used Angular CLI 7.3. To see which version is installed on your computer, run `ng version`.

This chapter comes with four sample projects, which were generated by Angular CLI. Even though these projects are ready to be reviewed and run, we'll describe the process of generating and running the hello-world project so you can try it on your own.

To generate a new minimalistic project, you need to run the `ng new` command followed by the name of the project (a.k.a. workspace). To generate a simple hello-world project run the following command in the Terminal window:

```
ng new hello-world --minimal
```

It'll ask you "Would you like to add Angular routing? (y/N); answer N. Then you'll need to select the style sheets format; press Enter for CSS (default). In a second, you'll see the names of the generated files in the new directory `hello-world`, and one of them is `package.json`. In the next 30 seconds, it'll run `npm` installing all required dependencies. Figure 11.1 shows how your Terminal window may look like after the project is ready.



```
chapter11 — bash — 80x24
$ ng new hello-world --minimal
? Would you like to add Angular routing? No
? Which stylesheet format would you like to use? CSS
CREATE hello-world/README.md (1027 bytes)
CREATE hello-world/.gitignore (629 bytes)
CREATE hello-world/angular.json (3080 bytes)
CREATE hello-world/package.json (930 bytes) ←
CREATE hello-world/tsconfig.json (435 bytes)
CREATE hello-world/src/favicon.ico (5430 bytes)
CREATE hello-world/src/index.html (297 bytes)
CREATE hello-world/src/main.ts (372 bytes)
CREATE hello-world/src/polyfills.ts (2841 bytes)
CREATE hello-world/src/styles.css (80 bytes)
CREATE hello-world/src/browserslist (388 bytes)
CREATE hello-world/src/tsconfig.app.json (166 bytes)
CREATE hello-world/src/assets/.gitkeep (0 bytes)
CREATE hello-world/src/environments/environment.prod.ts (51 bytes)
CREATE hello-world/src/environments/environment.ts (662 bytes)
CREATE hello-world/src/app/app.module.ts (314 bytes)
  Directory is already under version control. Skipping initialization of git.
$
```

**Figure 11.1 Generating a minimal project**

In the Terminal window, switch to the newly generated directory hello-world and run the app in the browser with the `ng serve -o` command (-o means "Open the browser for me at the default host and port"):

```
cd hello-world
ng serve -o
```

This command builds the bundles for this app, starts the web server with your app, and opens the browser. At the time of this writing, the `ng serve` command uses Webpack to bundle your apps and Webpack Dev Server to serve it. By default, your app is served at `localhost:4200`. The command `ng serve -o` will produce the output as seen in figure 11.2.

```
$ cd hello-world
$ ng serve -o
** Angular Live Development Server is listening on localhost:4200, open your browser on h
http://localhost:4200/ **
Date: 201
9-04-10T10:35:12.530Z
Hash: 617767a50a6f77b8c833
Time: 7615ms
chunk {es2015-polyfills} es2015-polyfills.js, es2015-polyfills.js.map (es2015-polyfills)
284 kB [initial] [rendered]
chunk {main} main.js, main.js.map (main) 9.11 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 236 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.08 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 16.3 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 3.52 MB [initial] [rendered]
i 「wdm」: Compiled successfully.
```

**Figure 11.2 The app bundles are created**

If you read chapter 6, you can recognize the Webpack bundles and source map files. The app code is located in the chunk `main.js`, and the code from Angular framework is located in the chunk called `vendor.js`. Don't be scared by the size of the `vendor.js` (3.52MB) as the `ng serve` builds the bundles in memory without the optimization. Running the production build `ng serve --prod` would produce the bundles with the total size of a little over 100KB.

Congratulation! Your first Angular app is up and running as shown in figure 11.3.



## Welcome to hello-world!



**Here are some links to help you start:**

- [Tour of Heroes](#)
- [CLI Documentation](#)
- [Angular blog](#)

**Figure 11.3** Running hello-world

Although this is not the UI that your app needs, this project contains the basic TypeScript code, tsconfig.json, the HTML file that renders this UI, package.json with all required dependencies, a pre-configured bundler, and some other files. Having a bundled and running app within a couple of minutes without spending any time studying Angular is pretty impressive, isn't it? Unfortunately, you'd still need to learn this framework to develop your own app, and we'll help you with getting started with Angular.

First of all, terminate the running hello-world app by pressing Ctrl-C in the Terminal window. Let's open the hello-world directory in VS Code, which offers a lot more convenient dev environment than a Terminal window and a plain text editor.

## 11.2 Reviewing the generated app

Figure 11.4 is a VS Code screenshot showing the structure of the generated hello-world project.



**Figure 11.4 Looking at the hello-world workspace in VS Code**

All these files were generated by Angular CLI. We won't go through each file, but will describe those files that are crucial for understanding of how an Angular app works and what are the main players of any Angular app. We marked these files with arrows in figure 11.4.

The source code of our app is located in the src folder, and at the very minimum, it'll have one component (app.component.ts) and one module (app.module.ts). The content of the app.component.ts is shown in listing 11.1.

## Listing 11.1 app.component.ts

```

import { Component } from '@angular/core';      ①

@Component({
  selector: 'app-root',                      ②
  template: `                         ③
    <div style="text-align:center">
      <h1>
        Welcome to {{title}}!           ⑤
      </h1>
      `. Here, we bind the value of the variable `lastName` to the property `name` of the `CustomerComponent`. You'll see more examples of the property binding syntax in listings 11.26 and 11.32.

In listing 11.1, the property `styles` points at the empty array. If we wanted to add the CSS styles, we could either add them inline or specify file names of one or more CSS files in the property `styleUrls`, for example `styleUrls: [app.component.css]`.

But declaring a component class is not enough, because you app must have at least one Angular module (not to be confused with ECMAScript modules). An Angular module is a TypeScript class decorated with the `@NgModule()` decorator. It's like a registry of components, services, and possibly other modules that belong together. Typically, the code of the class is empty and the list of the module members is specified in the decorator's properties. Let's look at the code generated in the file `app.module.ts` shown in listing 11.2.

## Listing 11.2 The file src/app/app.module.ts

```

import { BrowserModule } from '@angular/platform-browser';      ①
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';                  ③

@NgModule({
  declarations: [   ④
    AppComponent
  ],
  imports: [  ⑥
    BrowserModule
  ],
  providers: [],   ⑦
  bootstrap: [AppComponent]                            ⑧
})
export class AppModule { }

```

- ① BrowserModule is needed for a web app
- ② We need the implementation of the decorator `NgModule()`
- ③ This is our app's only component
- ④ Applying the `NgModule()` decorator
- ⑤ Declaring all components that belong to this module
- ⑥ Importing other modules if needed
- ⑦ Declaring providers for Angular services, if any
- ⑧ Specifying the root component that has to be loaded

The `@NgModule()` decorator requires you to list all components and modules used in the app. In the property `declarations` we have only one component, but if we had several of them (e.g. `CustomerComponent`, `OrderComponent`), we'd have to list them as well:

```
declarations: [ AppComponent, CustomerComponent, OrderComponent ]
```

Of course, if you mention a class, interface, variable, or function name, they have to be imported on top of the module's file just like we did for the `AppComponent`. By the way, have you noticed the `export` keyword in listing 11.1? If we wouldn't have it there, we wouldn't be able to import `AppComponent` in the file `app.module.ts` or any other file for that matter.

The property `imports` is the place for listing other required modules. Angular itself is split into modules and, most likely, your real-world app will be modularized as well. Listing 11.1 has only `BrowserModule` in `imports`, which must be included in the root module of any app that runs in the browser. Listing 11.3 is an example of an `imports` property illustrating what can be included there.

### Listing 11.3 Importing other modules

```
imports: [ BrowserModule,      ①
          HttpClientModule, ②
          FormsModule,       ③
          ShippingModule,    ④
          BillingModule ]   ⑤
```

- ① Angular module for web apps
- ② Angular module for making HTTP calls
- ③ One of the Angular modules for supporting forms
- ④ The application's module that implements shipping functionality
- ⑤ The application's module that implements billing functionality

**NOTE** Besides listing the modules in the `imports` property of the `@NgModule()` decorator, you need to add an ES6 import statement for each of these modules to point at the files where they are implemented.

In the next section, we'll talk about services and dependency injection, and there we'll talk about service providers, that can be listed in the module's `providers` property.

The `bootstrap` property names the top level component that the module must load first. The root component may use child components, which Angular will recognize and load. The child components may have their own children, and Angular will find and load all of them. The Angular module shown in listing 11.2 has only one component in the `declaration` section, so it's listed in the `bootstrap` as well, but if this would have several components, you'd need to specify one to bootstrap when the module is loaded.

And where's the code to bootstrap the module with its `AppComponent` from listing 11.1? It's located in the CLI-generated file `main.ts` shown in listing 11.4.

### Listing 11.4 The file `src/main.ts`

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) { ①
  enableProdMode();
}

platformBrowserDynamic() ②
  .bootstrapModule(AppModule) ③
  .catch(err => console.error(err)); ④
```

- ① Check the environment variable `production`

- ② Create the entry point to the app
- ③ Bootstrap the root module
- ④ Catch errors if any

First, the code in `main.ts` reads one of the files in the `environments` directory to check the value of the boolean variable `environment.production`. Without going into details, we'll just state that this value affects how many times Angular's Change Detector will pass the app components tree to see what has to be updated on the UI. The Change Detector monitors each and every variable bound to the UI and gives a signal to the rendering engine on what to update.

Second, the `platformBrowserDynamic()` API creates a *platform*, which is an entry point to the web app. Then it bootstraps the module, which in turn loads the root and all child components required to render the module. It'll also render other modules listed in the `imports` property, and will create an injector that knows how to inject services listed in the `providers` property of the decorator `@NgModule()`.

OK, we went through the TypeScript code, we built the bundles, and you may guess that the file `index.html` shown in figure 11.4 will use the bundle, right? Wrong! This was the initial version of `index.html` and it only includes the selector of our root component as seen in listing 11.5.

### **Listing 11.5 src/index.html**

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>HelloWorld</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root> ①
</body>
</html>
```

- ① The root component of our app

In this file, you won't `<script>` tags pointing at the JavaScript bundles shown in figure 11.2. But when you run the app with `ng serve`, Angular will add the `<script>` tags to the HTML file, and you can see them during runtime in the Chrome Dev tools as shown in figure 11.5.

```

<!doctype html>
<html lang="en" class="gr__localhost">
  <head>...</head>
  ... <body data-gr-c-s-loaded="true"> == $0
    <app-root ng-version="7.2.12">...</app-root>
    <script type="text/javascript" src="runtime.js"></script>
    <script type="text/javascript" src="es2015-polyfills.js" nomodule></script>
    <script type="text/javascript" src="polyfills.js"></script>
    <script type="text/javascript" src="styles.js"></script>
    <script type="text/javascript" src="vendor.js"></script>
    <script type="text/javascript" src="main.js"></script>
  </body>
</html>

```

**Figure 11.5 The runtime version of the HTML doc has `<script>` tags**

**NOTE** The `ng serve` command builds and rebuilds the bundles in memory to speed up the development process, but if you'd like to see the actual files, run the `ng build` command, and it'll create the file `index.html` and all the bundles in the directory `dist`.

**TIP** The CLI-generated file `angular.json` contains all project configurations, and you can change the output directory as well as many other default options there.

This was a high-level overview of the project generated by the CLI command `ng new hello-world --minimal`. If we wouldn't use the `--minimal` option, CLI would also generate some boilerplate code for unit and end-to-end testing. You can use this simple project as a base for your app, which will require more components and services, and Angular CLI can help you in generating the boilerplate code for various artifacts of your app. Let's get familiar with the role of services in an Angular app.

## 11.3 Angular services and dependency injection

If a component is a class with UI, a service is just a class that implements the business logic of your app. You can create a service that calculates shipping cost. Another service may encapsulate all HTTP communications with the server. What all these services have in common is that they don't have UI, and Angular will instantiate and *inject* a service into our application's component(s) or another service.

**SIDE BAR****What Dependency Injection is about?**

If you've ever written a function that takes an object as an argument, you already wrote a program that instantiates this object and *injects* it into the function. Imagine a fulfillment center that ships products. An application that keeps track of shipped products can create a product object and invoke a function that creates and saves a shipment record:

```
var product = new Product();
createShipment(product);
```

The `createShipment()` function depends on the existence of an instance of the `Product` object. In other words, the `createShipment()` function has a dependency: `Product`. But the function itself doesn't know how to create `Product`. The calling script should somehow create and give (think *inject*) this object as an argument to the function.

Technically, you're decoupling the creation of the `Product` object from its use. But both of the preceding lines of code are located in the same script, so it's not real decoupling, and if you need to replace `Product` with `MockProduct`, it's a small code change in this simple example.

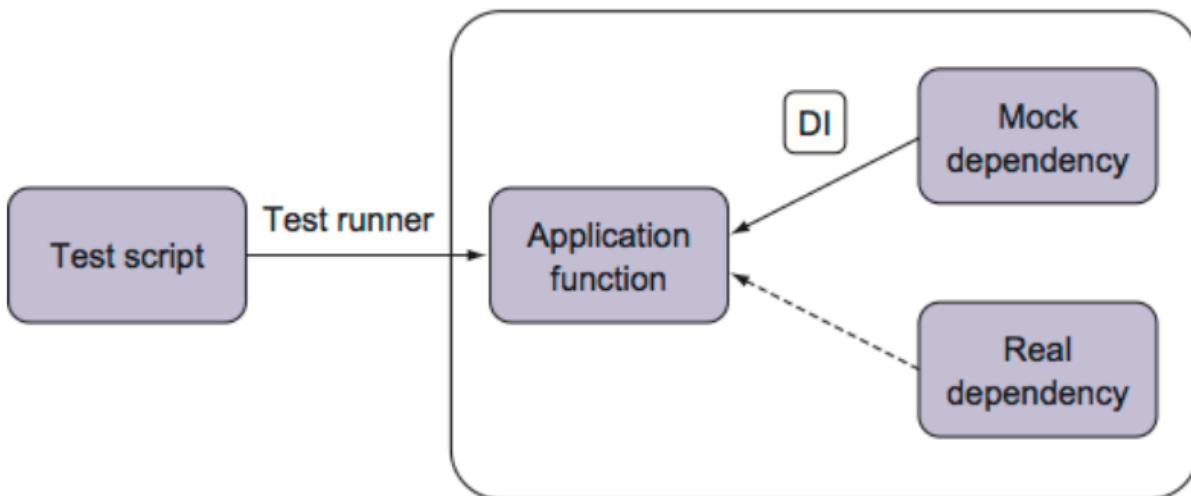
What if the `createShipment()` function had three dependencies (such as `product`, `shipping company`, and `fulfillment center`), and each of those dependencies had its own dependencies? In that case, creating a different set of objects for the function `createShipment()` would require many more manual code changes. Would it be possible to ask someone to create instances of dependencies (with their dependencies) for you?

This is what the Dependency Injection pattern is about: If object A depends on an object identified by a token (a unique ID) B, object A won't explicitly use the `new` operator to instantiate the object that B points at. Rather, it will have B *injected* from the operational environment.

Object A just needs to declare, "I need an object known as B; could someone please give it to me?" Object A doesn't request a specific object type (e.g. `Product`) but rather delegates to the framework the responsibility of what to inject to the token B. It seems that our object A doesn't want to be in control of creating instances and is ready to let the framework control this process, doesn't it? We're talking about the *Inversion of Control (IoC)* principle here. You, the framework, instantiate the object I need now and give it to me, will you?

Another good use of DI is writing unit tests, where the real services need to be replaced with mocks. In Angular, you can easily configure which objects (mocks or the real ones) have to be injected into your test scripts as shown in figure 11.6.

## IoC container (Angular)



**Figure 11.6 Injecting a mock service into unit tests**

**SIDE BAR**    Angular framework implements the Dependency Injection (DI) pattern and offers you a simple way of replacing one object with another if need be.

Let's ask Angular CLI to generate the class `ProductService` in our hello-world project. For this we'll use the CLI command `ng generate`, which can generate services, components, modules at al. To generate a service, you need to use the command `ng generate service` (or `ng g s`) followed by the name of the service. The following command will generate the new file `product.service.ts` in the directory `src/app`:

```
ng generate service product --skip-tests
```

**TIP**    To see all arguments and options of the command `ng generate`, run `ng --help generate` in the Terminal window.

If we wouldn't specify the option `--skip-tests`, the CLI would also generate a file with the boilerplate code for testing the product service. The content of the file `product.service.ts` is shown in listing 11.6.

### Listing 11.6 The generated file `product.service.ts`

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class ProductService {

  constructor() { }
}
```

- ① This decorator marks the class as injectable
- ② The instance of service must be available for all members of the root module

The decorator `@Injectable()` will instruct Angular to generate additional metadata required for instantiation and injecting the service. The property `providedIn` allows you to specify where this service has to be available. The value `root` means that we want this service to be *provided* on the app level and be a singleton, i.e. all other components and services can use a single instance of the `ProductService` object. If your app consists of several feature modules, you can restrict the service to be available only in a particular module, e.g. `providedIn: ShippingModule`.

An alternative to using the `providedIn` property, is specifying the provider in the module's `providers` property in `@NgModule`:

```
@NgModule({
  ...
  providers: [ProductService]
})
```

Say you have a `ProductComponent` that gets product details using the `ProductService` class. Without DI, your `ProductComponent` needs to know *how* to instantiate the `ProductService` class. This can be done multiple ways, such as using the `new` operator, calling `getInstance()` on a singleton object, or invoking some factory function `createProductService()`. In any case, `ProductComponent` becomes *tightly coupled* with `ProductService`, because replacing the `ProductService` with another implementation of this service requires code change in the `ProductComponent`.

If you need to reuse `ProductComponent` in another application that uses a different service to get product details, you must modify the code, e.g. `productService = new AnotherProductService()`. DI allows you to decouple application components and services by sparing them from knowing how to create their dependencies.

Angular documentation uses the concept of a *token*, which is an arbitrary key representing an object to be injected. You map tokens to values for DI by specifying providers. A *provider* is an instruction to Angular about *how* to create an instance of an object for future injection into a target component or another service.

We already mentioned that a provider can be specified in the module declaration (if you need a singleton service), but you can also specify a provider on a component level as shown in listing 11.7 where `ProductComponent` gets the `ProductService` injected.

Angular instantiates and injects any class used in the constructor's arguments. If you declared the provider in the `@Component()` decorator, such a component will get its own instance of the

service, which will be destroyed as soon as the component gets destroyed.

### **Listing 11.7 ProductService injected into ProductComponent**

```
@Component({
  providers: [ProductService] ①
})
class ProductComponent {
  product: Product;

  constructor(productService: ProductService) { ②
    this.product = productService.getProduct(); ③
  }
}
```

- ① Specify the token `ProductService` as a provider for injection
- ② Inject the object represented by the token `ProductService`
- ③ Use the API of the injected object assuming that `getProduct()` exists in the service

Often the token name matches the type of the object to be injected, so the preceding code snippet uses a shorthand `[ProductService]` for instructing Angular to provide a `ProductService` token by instantiating the class of the same name. The long version would look like this: `providers: [{provide: ProductService, useClass: ProductService}]`. You say to Angular, "If you see a class with a constructor that uses the token `ProductService`, inject the instance of the class `ProductService`."

`ProductComponent` doesn't need to know which concrete implementation of the `ProductService` type to use—it'll use whatever object is specified as a provider. The reference to the `ProductService` object will be injected via the constructor argument, and there's no need to explicitly instantiate `ProductService` in `ProductComponent`. Just use it as in the preceding code, which calls the service method `getProduct()` on the `ProductService` instance magically created by Angular.

If you need to reuse the same `ProductComponent` with a different implementation of the type `ProductService`, change the `providers` line, e.g. `providers: [{provide: ProductService, useClass: AnotherProductService}]`. Now Angular will instantiate `AnotherProductService`, but the code of the `ProductComponent` that uses `ProductService` doesn't require modification. In this example, using DI increases the reusability of `ProductComponent` and eliminates its tight coupling with `ProductService`.

## **11.4 An app with the ProductService injection**

This chapter comes with a sample project `di-products` that shows how to inject (and use) the `ProductService` into `ProductComponent`. Listing 11.8 shows the code of the `ProductService`

## Listing 11.8 The file product.service.ts from di-products

```
import { Injectable } from '@angular/core';
import { Product } from './product';

@Injectable({
  providedIn: 'root' ①
})
export class ProductService {

  getProduct(): Product { ②
    return { id: 0,
      title: "iPhone XI",
      price: 1049.99,
      description: "The latest iPhone" };
  }
}
```

- ① Create a singleton instance of the ProductService
- ② Return the hard-coded product data

In this service, we have the method `getProduct()` that returns hard-coded data of type `Product` shown in listing 11.9, but in real-world apps, we'd make an HTTP request to the server to get the data.

## Listing 11.9 The file product.ts

```
export interface Product {
  id: number,
  title: string,
  price: number,
  description: string
}
```

**NOTE** We declared the type `Product` as an interface, which enforces type checking in `getProduct()` but leaves no footprint in the transpiled JavaScript. We could declare the type `Product` as a class, but it would result in generation of the JavaScript code. For declaring custom types, it's better to use interfaces instead of classes if possible.

Now let's look at the product component, which we generated using the following CLI command:

```
ng generate component product --t --s --skip-tests
```

The option `--t` means that we don't want to generate a separate html file for the component's template. The option `--s` is for using inline styles instead of generating a separate CSS. file, and `--skip-tests` is for not generating the file with the boilerplate unit test code. After adding the template and injecting `ProductService` our `ProductComponent` looks as in listing 11.10.

## Listing 11.10 product.component.ts

```

import {Component} from '@angular/core';
import {ProductService} from '../product.service';
import {Product} from '../product';

@Component({
  selector: 'di-product-page',
  template: `<div>
    <h1>Product Details</h1>
    <h2>Title: {{product.title}}</h2>      ①
    <h2>Description: {{product.description}}</h2>  ②
    <h2>Price: \${{product.price}}</h2>      ③
  </div>`
})

export class ProductComponent {
  product: Product;          ④

  constructor( productService: ProductService) { ⑤
    this.product = productService.getProduct(); ⑥
  }
}

```

- ① Binding title to the UI
- ② Binding description to the UI
- ③ Binding price to the UI
- ④ This object's properties are bound to the UI
- ⑤ Injecting ProductService
- ⑥ Using the API of ProductService

When Angular will instantiate `ProductComponent` it'll also inject `ProductService` because it's a constructor's argument and will invoke `getProduct()` right away. The component's property `product` will be populated and the UI will be updated using binding.

**NOTE**

Please don't curse us for placing invoking `getProduct()` from the component's construction. In the real-word project we'd use a special `ngOnInit()` callback for this, but we wanted to show you the simplest possible code.

Finally, our `AppComponent` will have `ProductComponent` as a child by using its selector in the template as seen in listing 11.11.

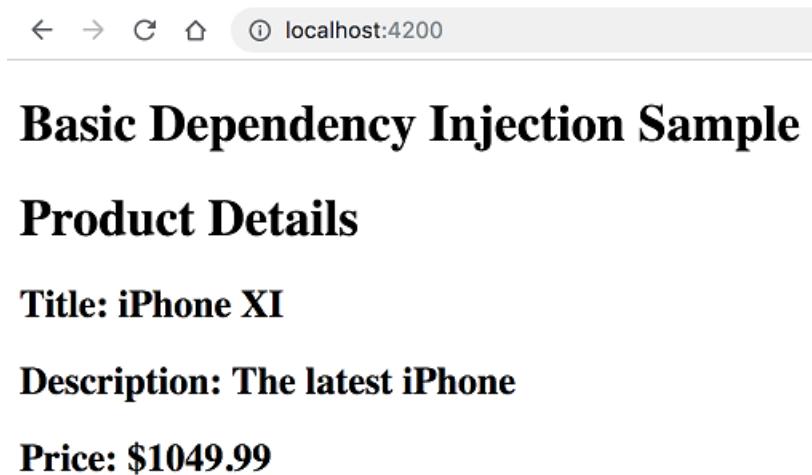
## Listing 11.11 app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `<h1> Basic Dependency Injection Sample</h1>
    <di-product-page></di-product-page>`①
})
export class AppComponent {}
```

- ① Adding the ProductComponent to the template

Build the bundle and start the dev server using the command `ng serve -o`, and the browser will render the UI as in figure 11.7.



**Figure 11.7** Rendering the product data

**TIP**

In the unlikely event that you don't like the UI of our component shown in figure 11.7, add the `styles` property to its `@Component()` decorator and use the styles that you like.

**SIDE BAR****State management in Angular**

State management is one of the most important parts of any app, and it deserves a chapter on its own, but we have room only for a sidebar. We placed this sidebar in the DI section only because in Angular, DI is widely used for implementing state management.

In a web app, one component can change the value of a specific variable(s), which is used in another component. This can be done either by the user's actions or as a result of a new server-generated data. For example, you are on Facebook and the top toolbar (one component) shows that you have 3 unread messages. When you click on the digit 3, it opens the messenger (another component), which shows you these three messages.

There's got to be something besides the toolbar and messenger components that stores and maintains the message counter during the user's session. This is an example of *app state management* that plays very important role in any app.

Angular injectable services (combined with RxJS) offer you a straightforward way to implement state management. If you create a service `AppState` and will declare its provider in the `@NgModule()` decorator, Angular will create a singleton `AppState`, and you can inject it in any components and services that need to have access to the current app state.

The `AppState` service may have a property `messageCounter`, and the `MessengerComponent` increments it every time a new message arrives. Accordingly, the `ToobarComponent` will get the current value of the `AppState.messageCounter` and render it in the UI. This way, the `AppState` becomes a *single source of truth* when it comes to storing and providing the app state values. Moreover, when one component updates the message counter, the `AppState` can broadcast its new value to other components that are interested in getting the new state.

While injectable services offer a clean solution for state management, some people prefer using third-party libraries (e.g. NGRX or NGXS) for implementing state management in Angular apps. These libraries may require you to write a lot of additional boilerplate code, and you should think twice before making a decision regarding the implementation of state management in your app. Poorly implemented state management can make your app buggy and costly to maintain.

Yakov Fain posted on YouTube a video titled "Angular: when ngrx is an overkill" (see [www.youtube.com/watch?v=xLTIDs0CDCM](https://www.youtube.com/watch?v=xLTIDs0CDCM)) where he compares a singleton service with NGRX library for implementing state management in a simple app.

## 11.5 Program to abstractions in TypeScript

In section 3.2.3 in chapter 3, we recommended to program to interfaces (a.k.a. abstractions). Since Angular DI allows you to replace the injectable objects, it would be nice if you could declare an interface `ProductService` and specify it as a provider. The injection point would look like `constructor( productService: ProductService)`, and you'd write several concrete classes that implement this interface and switch them in the providers declaration as needed.

You could do this in Java, C#, and other object-oriented languages. In TypeScript, the problem is that after transpiling the code into JavaScript, the interfaces are removed because JavaScript doesn't support them. In other words, if `ProductService` would be declared as an interface, the `constructor( productService: ProductService)` would turn into `constructor(productService)` and Angular wouldn't know anything about `ProductService`.

The good news is that TypeScript supports abstract classes, which can have some of the methods implemented, and some abstract, i.e. declared but not implemented (see section 3.1.5 in chapter 3). Then you'd need to implement some concrete classes that extend the abstract ones and implement all abstract methods. For example, you can have the classes as in listing 11.12:

### Listing 11.12 Declaring an abstract class and two descendants

```
export abstract class ProductService{ ①
    abstract getProduct(): Product; ②
}

export class MockProductService extends ProductService{ ③
    getProduct(): Product {
        return new Product('Samsung Galaxy S10');
    }
}

export class RealProductService extends ProductService{ ④
    getProduct(): Product {
        return new Product('iPhone XII');
    }
}
```

- ① Declaring an abstract class
- ② Declaring an abstract method
- ③ Creating the first concrete implementation of the abstract class
- ④ Creating the second concrete implementation of the abstract class

The good news is that you can use the name of the abstract class in the constructors, and during the JavaScript code generation, Angular will use a specific concrete class based on the provider

declaration. Having the classes `ProductService`, `MockProductService`, and `RealProductService` declared as earlier in listing 11.12 will allow you to write something like this:

### **Listing 11.13 Using an abstract class as provider**

```
// A fragment from app.module.ts
@NgModule({
  providers: [{provide: ProductService, useClass: RealProductService}], ①
  ...
})
export class AppModule { }

// A fragment from product.component.ts
@Component({...})
export class ProductComponent {
  constructor(productService: ProductService) {...}; ②
}
...
}
```

- ① Mapping a concrete type to an abstract token
- ② Using an abstract token at the injection point

Here we use an abstraction `ProductService` in declaring the provider and in the constructor's argument. This was not the case in listing 11.8 where `ProductService` was a concrete implementation of certain functionality. Replacing the providers works the same way as described earlier, and if you decide to switch from one concrete implementation of the service to another.

Without using abstract classes you'd need to be very careful in declaring classes `ProductService` and `MockProductService` so they'd have exactly the same API `getProducts()`. If you'd use the abstract class approach, the TypeScript compiler would give you an error if you'd try to implement a concrete class but would miss an implementation of one of the abstract methods. Program to abstractions!

## **11.6 Getting started with HTTP requests**

Angular applications can communicate with any web server supporting HTTP, and in this section, we'll show you how to start making HTTP requests, which will help you in understanding the code of the blockchain app presented later in this chapter.

Browser-based web apps run HTTP requests asynchronously, so the UI remains responsive, and the user can continue working with the application while HTTP requests are being processed by the server. In Angular, asynchronous HTTP are implemented using a special *observable* object offered by the RxJS library that comes with Angular.

If your app requires HTTP communications, you need to add `HttpClientModule` to the

imports section of the `@NgModule()` decorator. After that, you can use the injectable service `HttpClient` for invoking `get()`, `post()`, `put()`, `delete()`, and other requests. Each of these requests returns an `Observable` object.

In the context of client-server communications, you can think of `Observable` as a stream of data that can be pushed to your web app by the server. This concept is easier to grasp if used with WebSocket communications - the server keeps pushing the data into the stream over the open socket. With HTTP, you always get back just a single result set, but you can think of it as a stream of one piece of data.

**TIP****Tip**

Yakov Fain published a series of blogs about RxJS and observable streams, and this series is available at [yakovfain.com/2018/03/24/rxjs-essentials-part-8-pipeable-operators](http://yakovfain.com/2018/03/24/rxjs-essentials-part-8-pipeable-operators).

Let's see how a web client could make a request to the server's endpoint `/product/123` to retrieve the `Product` with the ID equal 123. Listing 11.14 illustrates one way of invoking the `get()` method of the `HttpClient` service, passing a URL as a string.

#### **Listing 11.14 Making an HTTP GET request**

```
interface Product { ①
  id: number,
  title: string
}
...
class ProductService {
  constructor(private httpClient: HttpClient) { } ②

  ngOnInit() { ③
    this.httpClient.get<Product>('/product/123') ④
      .subscribe(
        data => console.log(`id: ${data.id} title: ${data.title}`), ⑤
        (err: HttpErrorResponse) => console.log(`Got error: ${err}`) ⑥
      );
  }
}
```

- ① Defining the type `Product`
- ② Injecting the `HttpClient` service
- ③ This callback method is invoked by Angular
- ④ Declaring a `get()` request
- ⑤ Subscribing to the result of `get()`
- ⑥ Logging an error if any

The `HttpClient` service is injected in the constructor, and since we added a `private` qualifier, `httpClient` becomes a property of the object `ProductService` instantiated by Angular. We

placed the code that makes an HTTP request inside a so-called hook method `ngOnInit()`, which is invoked by Angular when a component is instantiated and all its properties are initialized.

In the `get()` method, we haven't specified the full URL (e.g. [localhost:8000/product/123](http://localhost:8000/product/123)) assuming that the Angular app makes a request to the same server where it was deployed, so the base portion of the URL can be omitted. Note that in `get<Product>()`, we use the type assertion `<Product>` (an equivalent to `as Product`) to specify the type of data expected in the body of the HTTP response. This type assertion tells the static type analyzer something like this: "Dear TypeScript, you're having a hard time inferring the type of the data returned by the server. Let me help you - it's `Product`."

The returned result is always an RxJS `Observable` object, which has the method `subscribe()`. We specified two callbacks as its arguments:

- The first one will be invoked if the data is received; it prints the data on the browser's console.
- If the request returns an error, the second callback is invoked.

The `post()`, `put()`, and `delete()` methods are used in a similar fashion by invoking one of these methods and subscribing to the results.

#### NOTE

We stated earlier that every injectable service requires a provider declaration, but the providers for `HttpClient` are declared inside the `HttpClientModule`, which is included in the imports of `@NgModule`, so you don't need to explicitly declare them in your app.

By default, `HttpClient` expects the data in JSON format, and the data is automatically converted into JavaScript objects. If you expect non-JSON data, use the `responseType` option. For example, you can read arbitrary text from a file as shown in listing 11.15.

#### **Listing 11.15 Specifying string as a returned data type**

```
let someData: string;

this.httpClient
    .get<string>('/my_data_file.txt', {responseType: 'text'})    ①
    .subscribe(
        data => someData = data,    ②
        (err: HttpErrorResponse) => console.log(`Got error: ${err}`)  ③
    );
}
```

- ① Specifying string as a response body type
- ② Assigning the received data to a variable
- ③ Logging errors, if any

Now let's see how to read some data from a JSON file using `HttpClient`. This chapter comes

with a project read-file, which illustrate using `HttpClient.get()` for reading a file containing JSON-formatted product data. This app has the directory data that contains the file `products.json` as seen in listing 11.16.

### **Listing 11.16 The file data/products.json**

```
[  
  { "id": 0, "title": "First Product", "price": 24.99 },  
  { "id": 1, "title": "Second Product", "price": 64.99 },  
  { "id": 2, "title": "Third Product", "price": 74.99 }  
]
```

Now the directory data contains project assets (the file `products.json`) and needs to be included in the project bundles, so we'll add this directory to the app's `assets` property in the file `angular.json` as shown in listing 11.17.

### **Listing 11.17 A fragment from angular.json**

```
"assets": [  
  "src/favicon.ico",      ①  
  "src/assets",          ②  
  "src/data"             ③  
]
```

- ① Default assets generated by Angular CLI
- ② The name of the assets directory that we've added to the project

Typically, you'll be specifying the server's URL while using the `HttpClient` service, but in our sample app the URL will point at the local file `data/products.json`. Our app will read this file and will render the products as shown in figure 11.8.



## **Products**

- First Product: \$24.99
- Second Product: \$64.99
- Third Product: \$74.99

**Figure 11.8 Rendering the content of products.json**

The `AppComponent` will use `HttpClient.get()` to issue an HTTP GET request, and we'll declare an interface `Product` defining the structure of the expected product data. as in listing 11.18

## Listing 11.18 src/product.ts

```
export interface Product {
  id: string;
  title: string;
  price: number;
}
```

The file app.component.ts is shown in listing 11.19, where we'll implement a simpler way of subscribing to the the `HttpClient` responses. This time you won't see the explicit `subscribe()` but we'll use the `async` pipe instead.

### NOTE

Angular pipes are special converter functions that can be used in the component's template and are represented by the vertical bar followed by the pipe name. For example, the `currency` pipe converts a number into the currency, e.g. `123.5521 | currency` will render \$123.55 (dollar is a default currency sign).

## Listing 11.19 app.component.ts

```
import {HttpClient} from '@angular/common/http';
import {Observable} from 'rxjs';      ①
import {Component, OnInit} from "@angular/core";
import {Product} from "./product";

@Component({
  selector: 'app-root',
  template: `<h1>Products</h1>
<ul>
  <li *ngFor="let product of products$ | async"> ②
    {{product.title }}: {{product.price | currency}} ③
  </li>
</ul>
`)
export class AppComponent implements OnInit{
  products$: Observable<Product[]>; ④
  constructor(private httpClient: HttpClient) {} ⑤

  ngOnInit() {
    this.products$ = this.httpClient
      .get<Product[]>('/data/products.json'); ⑥
  }
}
```

- ① Importing Observable from the RxJS library
- ② Iterating through the observable products and auto-subscribing to them with the `async` pipe
- ③ Render the product title and the price formatted as currency.
- ④ Declare a typed observable for products.
- ⑤ Injecting the `HttpClient` service
- ⑥ Making an HTTP GET request specifying the type of the expected data

The observable returned by `get()` will be unwrapped in the template by the `async` pipe, and each product's title and price will be rendered by the following code:

```
<li *ngFor="let product of products$ | async">
  {{product.title }}: {{product.price | currency}} ①
</li>
```

`*ngFor` is an Angular structural directive that iterates through every item emitted by the observable `products$` and renders the `<li>` element. Each element will show the product title and price using binding. The dollar sign at the end of `products$` is just a naming convention for variables that represent observables.

To see this app in action, run `npm install` in the directory client and then run the following command:

```
ng serve -o
```

## 11.7 Getting started with forms

HTML provides basic features for displaying forms, validating entered values, and submitting the data to the server. But HTML forms may not be good enough for real-world applications, which need a way to programmatically process the entered data, apply custom validation rules, display user-friendly error messages, transform the format of the entered data, and choose the way data is submitted to the server.

Angular offers two APIs for handling forms: *template-driven* and *reactive*. With the template-driven API, forms are fully programmed in the component's template using directives, and the model object is created implicitly by Angular. Because you're limited to the HTML syntax while defining the form, the template-driven approach suits only simple forms.

With the reactive API, you explicitly create the model object in the TypeScript code and then link the HTML template elements to that model's properties using special directives. You construct the form model object explicitly using the classes `FormControl`, `FormGroup`, and `FormArray`. For non-trivial forms, the reactive approach is a better option. In this section, we'll give you a brief overview of using reactive forms, which will be also used in our blockchain app.

To enable reactive forms, you need to add `ReactiveFormsModule` from `@angular/forms` to the `imports` list of the `@NgModule()` decorator as in listing 11.20.

## Listing 11.20 Adding support for reactive forms

```
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  ...
  imports: [
    ...
    ReactiveFormsModule     ②
  ],
  ...
})
```

- ① Importing the module that supports reactive forms

Now let's talk about how to create a form model, which is a data structure that holds the form's data. It can be constructed out of `FormControl`, `FormGroup`, and `FormArray` classes. For example, listing 11.21 declares a class property of type `FormGroup` and initializes it with a new object that will contain instances of the form controls for your form.

## Listing 11.21 Creating a form model object

```
myFormModel: FormGroup;

constructor() {
  this.myFormModel = new FormGroup({
    username: new FormControl(''),      ①
    ssn: new FormControl('')           ②
  });
}
```

- ① Creating an instance of form model
- ② Adding form controls to the form model

`FormControl` is an atomic form unit, which typically corresponds to a single `<input>` element, but it can also represent a more complex UI component like a calendar or a slider. A `FormControl` instance stores the current value of the HTML element it corresponds to, the element's validity status, and whether it's been modified. Here's how you can create a control passing its initial value as the first argument of the constructor:

```
city = new FormControl('New York');
```

You can also create a `FormControl` attaching one or more built-in or custom validators, which can be attached to a form control or to the entire form. Listing 11.22 shows how to add two built-in Angular validators to a form control.

## Listing 11.22 Adding validators to a form control

```
city = new FormControl('New York',          ①
  [Validators.required,                   ②
  Validators.minLength(2)]);            ③
```

- ① Creating a form control with the initial value New York
- ② Adding a required validator to a form control
- ③ Adding a minLength validator to a form control

`FormGroup` is a collection of `FormControl` objects and represents either the entire form or its part. `FormGroup` aggregates the values and validity of each `FormControl` in the group. If one of the controls in a group is invalid, the entire group becomes invalid.

The injectable service `FormBuilder` is one of the way creation of form models. Its API is terser and saves you from the repetitive instantiation of objects shown in listing 11.21. In listing 11.23, Angular injects the `FormBuilder` object that's used for declaring the form model.

### Listing 11.23 Creating a `formModel` with `FormBuilder`

```
constructor(fb: FormBuilder) {      ①
  this.myFormModel = fb.group({      ②
    username: ['', Validators.required], ③
    ssn: ['', Validators.minLength(9)],
    passwordsGroup: fb.group({        ④
      password: ['', Validators.required],
      pconfirm: ['', Validators.required]
    })
  });
}
```

- ① Inject the `FormBuilder` service.
- ② `FormBuilder.group()` creates a `FormGroup` using a configuration object passed to it.
- ③ Each `FormControl` is instantiated using the array that may contain an initial control's value and its validators.
- ④ Like `FormGroup`, `FormBuilder` allows you to create nested groups.

The method `FormBuilder.group()` accepts an object with extra configuration parameters as the last argument. You can use it to specify group-level validators there if needed.

The reactive approach requires you to use directives in the component templates. These directives are prefixed with `form`—for example, `formGroup` (note the small `f`) as shown in listing 11.24.

## Listing 11.24 Binding the FormGroup to HTML <form>

```

@Component({
  selector: 'app-root',
  template: `
    <form [formGroup]="myFormModel" >           ①
      </form>
    `
})
class AppComponent {
  myFormModel = new FormGroup({            ②
    username: new FormControl(''),
    ssn: new FormControl('')
  });
}

```

- ① Bind the instance of the form model to the `FormGroup` directive of the `<form>`
- ② Create an instance of the form model.

The reactive directives `FormGroup` and `FormControl` bind DOM element like `<form>` and `<input>` to the model object (e.g. `myFormModel`) using the property-binding syntax with square brackets:

```

<form [formGroup]="myFormModel">
  ...
</form>

```

The directives that link DOM elements to the TypeScript model's properties by name are `formGroupName`, `formControlName`, and `formArrayName`. They can only be used inside the HTML element marked with the `FormGroup` directive.

The `FormGroup` directive binds an instance of the `FormGroup` class that represents the entire form model to a top-level form's DOM element, usually a `<form>`. In the component template, use `formGroup` with a lowercase `f`, and in TypeScript, create an instance of the class `FormGroup` with a capital `F`. To use the `FormGroup` directive in a template, you need to first create an instance of the `FormGroup` in the TypeScript code of a component.

`formControlName` must be used in the scope of the `FormGroup` directive. It links an individual `FormControl` instance to a DOM element. Let's continue adding code to the example of the `dateRange` model from the previous section. The component and form model remain the same. You only need to add HTML elements with the `formControlName` directive to complete the template, as shown in listing 11.25.

## Listing 11.25 Completed form template

```
<form [FormGroup]="myFormModel">
  <div formGroupName="dateRange">
    <input type="date" FormControlName="from"> ①
    <input type="date" FormControlName="to"> ②
  </div>
</form>
```

- ① from is a property name in the model's nested group dateRange.
- ② to is a property name in the model's nested group dateRange.

As in the `formGroupName` directive, you specify the name of a `FormControl` you want to link to the DOM element. Again, these are the names you chose while defining the form model.

The `FormControl` directive is used with individual form controls or single-control forms, when you don't want to create a form model with `FormGroup` but still want to use Forms API features like validation and the reactive behavior provided by the `FormControl.valueChanges` property, which is of type `observable`, i.e. you can subscribe to `valueChanges` and receive the data item each time the user enters a character in the form control. The code fragment in listing 11.26 looks up the weather in the entered city and then prints it on the console.

## Listing 11.26 FormControl

```
@Component({
  ...
  template: `<input type="text" [FormControl]="weatherControl">` ①
})
class FormComponent {
  weatherControl: FormControl = new FormControl(); ②

  constructor() {
    this.weatherControl.valueChanges
      .pipe(
        switchMap(city => this.getWeather(city)) ③
      )
      .subscribe(weather => console.log(weather)); ④
  }
}
```

- ① Using `FormControl` with the property binding
- ② Instead of defining a form model, create a standalone instance of a `FormControl`
- ③ Use the `valueChanges` observable to get the value from the form
- ④ Use the RxJS operator to switch to another observable returned by `getWeather()`
- ⑤ Subscribe to `valueChanges` and print the weather received from this observable

RxJS comes with dozens of operators that can be applied to the data item emitted by the observable before it's given to the method `subscribe()`. Without going into details, we'll just say that in the above code sample, is a fragment of a program where the method `getWeather()`

makes an HTTP request to the weather server and also returns an observable. The `switchMap` operator takes the data from the observable `valueChanges` and passes them to `getWeather()`, which also returns an observable.

In chapter 12, we'll use reactive Forms API in the class `AppComponent` for handling the form with the blockchain transactions:

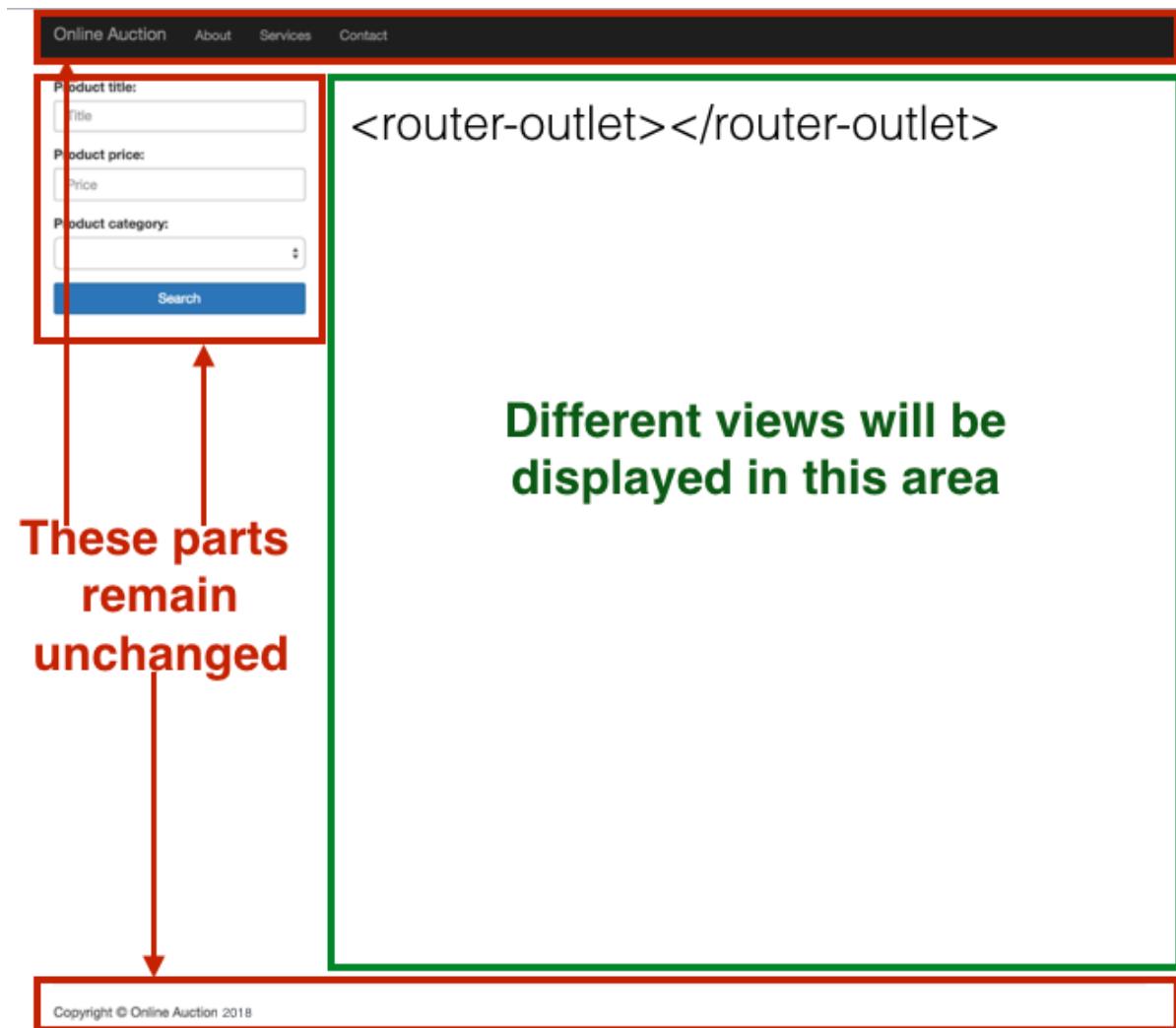
```
this.transactionForm = fb.group({
  sender : ['', Validators.required],
  recipient: ['', Validators.required],
  amount : ['', Validators.required]
});
```

This code will render a form with three input controls: `sender`, `recipient`, and `amount`. Each of these controls gets an empty string as initial value and has the `Validator.required` attached to it.

## 11.8 Router basics

In a single-page application (SPA), the web page won't be reloaded, but its parts may change. Now we want to add navigation to this application so it'll change the content area of the page (known as *router outlet*) based on the user's actions. The Angular router allows you to configure and implement such navigation without performing a full page reload.

The landing page of a SPA will have some parts that will always stay on the page, while some of them will render different components based on the user actions or other events. Figure 11.9 shows a sample web page where the navigation bar on top, the search panel on the left, and the footer will always be rendered on the page. But the large area marked with the tag `<router-outlet>` is where different components can be rendered one at a time.



**Figure 11.9 The router-outlet area**

Initially, the router outlet can display  `HomeComponent`, and when the user clicks on some link, the router will show  `ProductComponent` there.

Every application has one router object, and to arrange navigation, you need to configure the routes of your app. Angular includes many classes supporting navigation — for example, `Router`, `Route`, `Routes`, `ActivatedRoute`, and others. You configure routes in an array of objects of type `Route`, as in listing 11.27:

### Listing 11.27 A sample routes configuration

```
const routes: Routes = [
  {path: '', component: HomeComponent}, ①
  {path: 'product', component: ProductDetailComponent} ②
];
```

- ① An empty path indicates that the  `HomeComponent` is rendered by default
- ② If the URL contains the product fragment, render  `ProductDetailComponent`

Because routes configuration is done on the module level, you need to let the app module know about the routes in the `@NgModule()` decorator. If you declare routes for the root module, use the `forRoot()` method as seen in listing 11.28:

### **Listing 11.28 Letting the root module know about the routes**

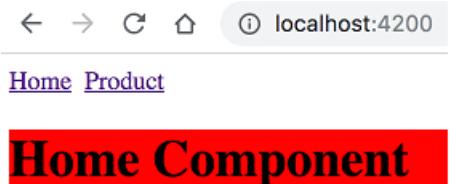
```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { ProductDetailComponent } from './product/product-detail.component';
import { AppRoutingModule } from './app-routing.module';

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ProductDetailComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: []
})
export class AppModule {}
```

- ① Create a router module and a service for the app root module.

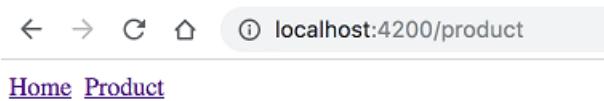
Let's review a simple app located in the directory called `router`. We generated it using the command `ng new router --minimal`. When it asked, "Would you like to add Angular routing?", we selected the Yes option, and CLI generated the file `app-routing.module.ts`.

The `AppComponent` has two links, `Home` and `Product Details`, at the top of the page. The application renders either `HomeComponent` or `ProductDetailComponent`, depending on which link the user clicks. `HomeComponent` renders the text "Home Component", and `ProductDetailComponent` render "Product Detail Component". Initially the web page displays `HomeComponent` as shown in figure 11.10.



**Figure 11.10 Rendering HomeComponent**

After the user clicks the `Product Details` link, the router should display `ProductDetailComponent`, as shown in figure 11.11. See how the URLs for these routes look in figures 11.10 and 11.11.



**Figure 11.11 Rendering ProductDetailComponent**

The main goal of this basic app is to become familiar with the router, so the components will be

very simple, as in listing 11.29 and 11.30.

### Listing 11.29 home.component.ts

```
@Component({
  selector: 'home',
  template: '<h1 class="home">Home Component</h1>',
  styles: ['.home {background: red;}']) ①
export class HomeComponent {}
```

- ① Render this component with red background.

The code of ProductDetailComponent looks similar, as you can see in the following listing, but it uses a cyan background.

### Listing 11.30 product-detail.component.ts

```
@Component({
  selector: 'product',
  template: '<h1 class="product">Product Detail Component</h1>',
  styles: ['.product {background: cyan;}']) ①
export class ProductDetailComponent {}
```

- ① Render this component with cyan background.

CLI generated a separate module for routing in the file app-routing.module.ts. The root module will import the configured RouterModule from this file shown in listing 11.31. We pass to the method `forRoot()` a config object with route. In this use just two properties defined in the Routes interface: path and component.

### Listing 11.31 app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home.component';
import { ProductDetailComponent } from './product-detail.component';

const routes: Routes = [
  { path: '', component: HomeComponent }, ①
  { path: 'product', component: ProductDetailComponent }
]; ②

@NgModule({
  imports: [RouterModule.forRoot(routes)], ③
  exports: [RouterModule] ④
})
export class AppRoutingModule { }
```

- ① HomeComponent is mapped to a path containing an empty string, which makes it a default route
- ② If the URL has the product segment, render ProductDetailComponent in the router outlet
- ③ Make routes available in RouterModule

- ④ Export the configured RouterModule so it can be imported by the root module

The next step is to create a root component that will contain the links for navigating between the Home and Product Details views as seen in listing 11.32.

### **Listing 11.32 app.component.ts**

```
@Component({
  selector: 'app-root',
  template: `
    <a [routerLink]=[['']]>Home</a>
    <a [routerLink]=[['/product']]>Product Details</a>
    <router-outlet></router-outlet>
  `
})
export class AppComponent {}
```

①  
②  
③

- ① Creates a link that binds routerLink to the empty path
- ② Creates a link that binds routerLink to the path /product
- ③ The <router-outlet> specifies the area on the page where the router will render the components (one at a time).

The square brackets around `routerLink` denote property binding, while the brackets on the right represent an array with one element (for example, `[ '' ]`). The second anchor tag has the `routerLink` property bound to the component configured for the `/product` path.

The path is provided as an array because it may include parameters that are being passed during the navigation. For example, `['/product', 123]` could instruct the router to go to the component that will render information about the product with the ID 123. The matched components will be rendered in the area marked with `<router-outlet>`, which in this app is located below the anchor tags.

None of the components is aware of the router configuration, because it's done at the module level, as shown in listing 11.33.

### **Listing 11.33 app.module.ts**

```
...
@NgModule({
  declarations: [
    AppComponent, HomeComponent, ProductDetailComponent ①
  ],
  imports: [
    BrowserModule,
    AppRoutingModule ②
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

- ① Declare components that belong to this module

② Import the module with preconfigured routes

To run the app described in this section, install dependencies by running `npm install` in the directory router, and then the `ng serve -o` command will start the server and open the browser at `localhost:4200`. The browser will render the window as seen in figure 11.10.

We just showed you a very basic app that uses Angular router, but it offers a lot more features, e.g.

- Passing parameters during the navigation
- Subscribing to changing parameters of the parent component
- Guarding routes to apply business logic that may prevent the user from navigation
- Lazy loading of the modules during navigation
- The ability to define more than one router outlet in a component

Angular is a very solid solution for developing single-page apps, and the router plays the main role in the client-side navigation.

This concludes our brief introduction to the Angular framework. It won't make you an Angular expert, but you'll be ready to read and understand the code of the new version of the blockchain client presented in chapter 12.

## 11.9 Summary

In this chapter you learned:

- How to generate your first Angular app without knowing anything about this framework
- What are the roles of Angular modules, components and services
- What are the benefits of dependency injection and how Angular implements it
- How to arrange basic user navigation using Angular router
- How to start working with forms
- How to use TypeScript decorators in Angular



# 12

## *Developing the blockchain client in Angular*

### **This chapter covers**

- The code review of the blockchain web client in Angular
- How to run the Angular client that communicates with the WebSocket server

In this chapter, we'll review a new version of the blockchain app where the client portion is written in Angular. The source code is located in two directories: client and server. But if in chapter 10 these directories were part of the same project, now they are two different projects with separate package.json files. In real-world apps, the front and back end apps typically are separate projects.

The code of the messaging server is the same as in chapter 10, and the functionality of this version of the blockchain app is the same as well, but the implementation of the front end was completely re-written in Angular. Let's see this app in action.

**TIP**

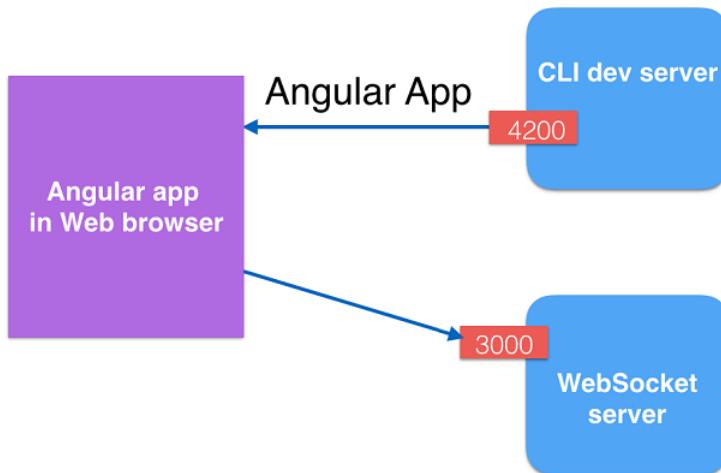
You may want to re-read chapter 10 to refresh in memory the functionality of the blockchain client and messaging server.

To start the server, open the Terminal window in the server directory, run `npm install` to install the server's dependencies and then run the `npm start` command. You'll see the message *Listening on localhost:3000*. Keep the server running while we're starting the client.

To start the Angular client, open another Terminal window in the client directory, run `npm install` to install Angular and its dependencies and then run the `npm start` command. In the

client's package.json, the `start` command is an alias to a familiar `ng serve` command. It'll build the bundles the same way as in every app reviewed earlier in this chapter, and you can open the browser at localhost:4200.

At this point, we have two servers running: the dev server that was installed by Angular CLI and the WebSocket messaging server that runs under Node.js as seen in figure 12.1



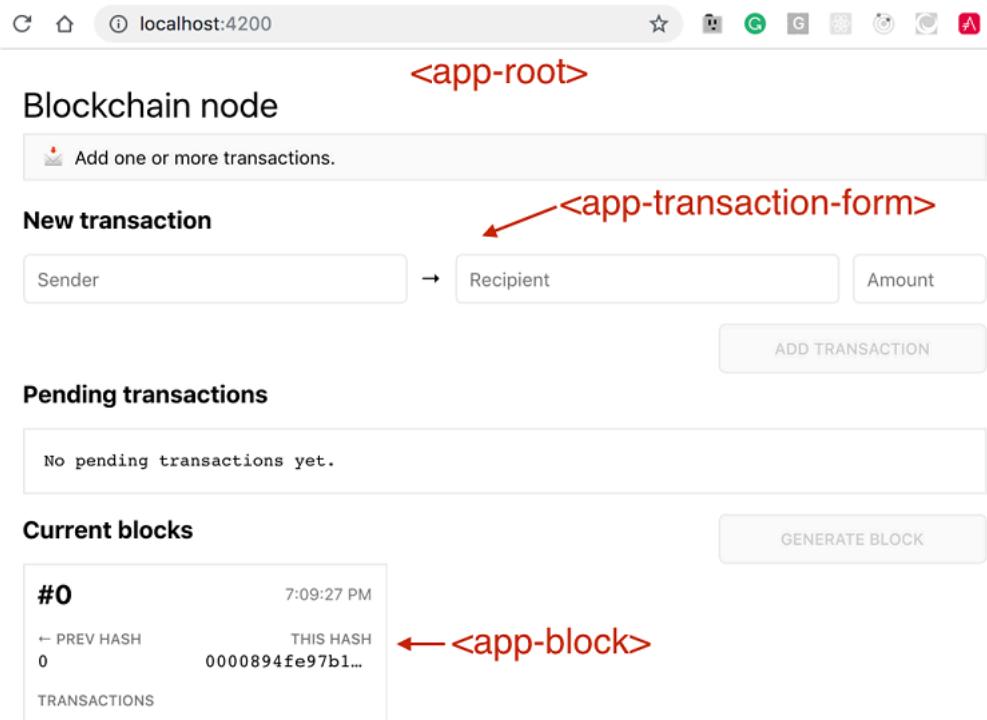
**Figure 12.1 One app, two servers**

If instead of WebSocket server we'd be running an HTTP server, we'd have to configure a proxy to overcome the restrictions imposed by the same origin policy. You can read more about proxying to a back-end server in Angular documentation at [angular.io/guide/build#proxying-to-a-backend-server](https://angular.io/guide/build#proxying-to-a-backend-server).

**NOTE**

If we had to deploy this app in production, we'd need a WebServer that will host the bundles of our blockchain client and we'd run the WebSocket server on the same port too. You can build the optimized bundles for deployment by running the command `ng build --prod`. The process of deployment is described in the Angular documentation at [angular.io/guide/deployment](https://angular.io/guide/deployment).

The app will spend some time generating the genesis block, and then you'll see a familiar window as shown in figure 12.2.

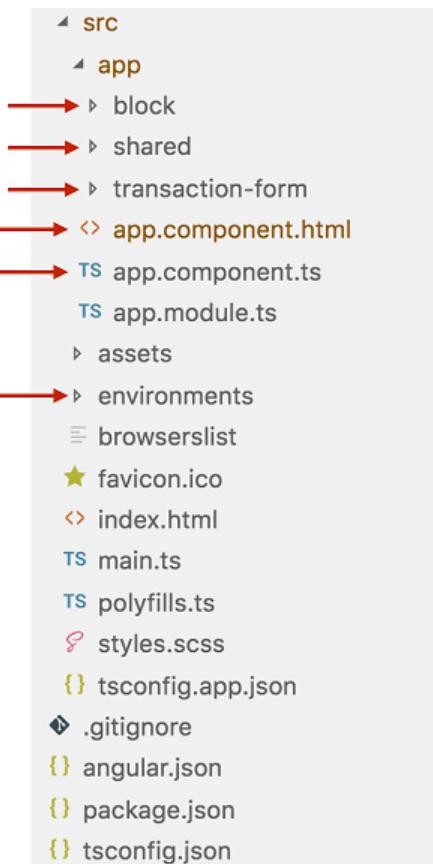


**Figure 12.2 Running the blockchain client written in Angular**

This app has three components:

- AppComponent - the top level component; its selector is `app-root` .
- TransactionFormComponent - the component to render transactions. Its selector is `app-transaction-form`, and this component is a form containing three input fields and the button ADD TRANSACTION.
- BlockComponent - the component for rendering the block data; its selector is `app-block`

The structure of the Angular project is shown in figure 12.3, and the arrows point at the files or directories where you find the source code of our blockchain client.



**Figure 12.3 Angular project structure**

The files representing the `BlockComponent` are located in the directory `block`. The files of the `TransactionFormComponent` are in the directory `transaction-form`. The `shared` directory contains reusable services `BlockchainNodeService`, `CryptoService`, and `WebSocketService`.

## 12.1 Reviewing AppComponent

We won't be reviewing the code specific to the blockchain functionality, because we already did it in chapters 8, 9, and 10. Here, we'll review only the code fragments that show how things done in Angular. The code for the root component is located in two files: `app.component.html` and `app.component.ts`. Listing 12.1 shows the template of the top-level component.

## Listing 12.1 The template of the AppComponent: app.component.html

```

<main>
  <h1>Blockchain node</h1>
  <aside><p>{{ statusLine }}</p></aside>
  <section>
    <app-transaction-form></app-transaction-form> ①
  </section>
  <section>
    <h2>Pending transactions</h2>
    <pre class="pending-transactions__list">{{ formattedTransactions }}</pre>
    <div class="pending-transactions__form">
      <button type="button"
        class="ripple"
        (click)="generateBlock()"
        [disabled]="node.noPendingTransactions || node.isMining"> ②
        GENERATE BLOCK ③
      </button>
    </div>
    <div class="clear"></div>
  </section>
  <section>
    <h2>Current blocks</h2>
    <div class="blocks">
      <div class="blocks__ribbon">
        <app-block ④
          *ngFor="let blk of node.chain; let i = index" ⑤
          [block]="blk" ⑥
          [index]="i">
        </app-block>
      </div>
      <div class="blocks__overlay"></div>
    </div>
  </section>
</main>

```

- ① The child component to enter transactions
- ② Adding the click event handler for the button that generated blocks
- ③ Binding the button's disabled attribute
- ④ The child component for a block rendering
- ⑤ Iterating through the existing blocks in the chain
- ⑥ Binding the block object to the block component's property
- ⑦ Binding the iterator's index to the block component's input property

In this template, we use data binding three times:

- The disable property of the button GENERATE BLOCK will be either true or false depending on the value of the expression on the right.
- The block property of the component <app-block> gets the value from the current blk as the directive ngFor iterates through the array node.chain, which is a property of the AppComponent class shown in listings 12.2, 12.3, and 12.5.
- The index property of the component <app-block> gets the value from the variable index offered by the directive ngFor, which represents the current value of the loop iterator.

The line `(click)="generateBlock()"` declares the event handler for the click event. In Angular templates you use parentheses to specify an event handler.

The template from listing 12.1 is included in the decorator `@Component()` of the TypeScript class `AppComponent`, and its first part is shown in listing 12.2.

### **Listing 12.2 The part 1 of app.component.ts**

```
import {Component} from '@angular/core';
import {Message, MessageTypes} from './shared/messages';
import {Block, BlockchainNodeService, formatTransactions, Transaction, WebsocketService}
      from './shared/services'; ①

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {
  constructor(private readonly server: WebsocketService, ②
              readonly node: BlockchainNodeService) { ②
    this.server.messageReceived.subscribe(message =>
      this.handleServerMessages(message)); ③
    this.initializeBlockchain(); ④
  }

  private async initializeBlockchain() {
    const blocks = await this.server.requestLongestChain();
    if (blocks.length > 0) {
      this.node.initializeWith(blocks);
    } else {
      await this.node.initializeWithGenesisBlock();
    }
  }
}
```

- ① Using the directory (not a file) in the import statement
- ② Injecting a service
- ③ Subscribing for service messages
- ④ Create an instance of the blockchain

Revisit the template of the app component in listing 12.1; if the user clicks on the button GENERATE BLOCK, it'll invoke the method `generateBlock()` declared in the TypeScript class `AppComponent`. Listing 12.3 shows several methods from the class `AppComponent`.

### Listing 12.3 The part 2 of app.component.ts

```

get statusLine(): string {
    return (
        this.node.chainIsEmpty      ? 'Initializing the blockchain...' :
        this.node.isMining          ? 'Mining a new block...' :
        this.node.noPendingTransactions ? 'Add one or more transactions.' :
   'Ready to mine a new block.'
    );
}

get formattedTransactions() {
    return this.node.hasPendingTransactions
        ? formatTransactions(this.node.pendingTransactions)
        : 'No pending transactions yet.';
}

async generateBlock(): Promise<void> { ❶
    this.server.requestNewBlock(this.node.pendingTransactions);
    const miningProcessIsDone = this.node.mineBlockWith(this.node.pendingTransactions);

    const newBlock = await miningProcessIsDone;
    this.addBlock(newBlock);
};

private async addBlock(block: Block, notifyOthers = true): Promise<void> { ❷
    try {
        await this.node.addBlock(block);
        if (notifyOthers) { ❸
            this.server.announceNewBlock(block);
        }
    } catch (error) { ❹
        console.log(error.message);
    }
}

```

- ❶ The click event handler for the button GENERATE BLOCK
- ❷ An attempt to add the new block to the blockchain
- ❸ The new block was accepted by the blockchain
- ❹ The new block was rejected by the blockchain

Note how using the `async` and `await` keywords allows us to write the code that looks as if it's being executed synchronously even though every function called prepended with `await` is an asynchronous execution.

**SIDE BAR** **Organizing imports with index.ts**

Note that the class `AppComponent` imports several classes specifying the name of the directory instead of having several import statements pointing at different files. This is possible because we introduced a special TypeScript file `index.ts` in the directory `shared/services` as seen in listing 12.4.

**Listing 12.4 shared/services/index.ts**

```
export * from './blockchain-node.service';
export * from './crypto.service';
export * from './websocket.service';
```

In this file, we re-export all the members exported from the three files listed in this file. If a directory includes the file named `index.ts`, you can simplify your import statements by just using the directory name, and `tsc` will find the members for imports in the files included in `index.ts`, for example:

```
import {Block, BlockchainNodeService, formatTransactions, Transaction,
        WebsocketService}
      from './shared/services';
```

Without this `index.ts` file, we'd need to write five import statements pointing at different files.

The third part of the code in `app.component.ts` presented in listing 12.5 shows the methods of the class `AppComponent` that handle server messages pushed over the WebSocket; they handle the longest chain requests and the new block requests. This functionality was explained in chapter 10.

### Listing 12.5 The part 3 of app.component.ts

```

handleServerMessages(message: Message) {
  switch (message.type) {
    case MessageType.GetLongestChainRequest: return this.handleGetLongestChainRequest(message);
    case MessageType.NewBlockRequest : return this.handleNewBlockRequest(message);
    case MessageType.NewBlockAnnouncement : return this.handleNewBlockAnnouncement(message);
    default: {
      console.log(`Received message of unknown type: "${message.type}"`);
    }
  }
}

private handleGetLongestChainRequest(message: Message): void { ❶
  this.server.send({
    type: MessageType.GetLongestChainResponse,
    correlationId: message.correlationId,
    payload: this.node.chain
  });
}

private async handleNewBlockRequest(message: Message): Promise<void> {
  const transactions = message.payload as Transaction[];
  const newBlock = await this.node.mineBlockWith(transactions);
  this.addBlock(newBlock);
}

private async handleNewBlockAnnouncement(message: Message): Promise<void> { ❷
  const newBlock = message.payload as Block;
  this.addBlock(newBlock, false);
}
}

```

- ❶ Handle WebSocket server's messages
- ❷ Handle the longest chain requests
- ❸ Handle the new block announcement

Revisit the template of the `AppComponent` shown in listing 12.1, and you'll find there the reference to the child component `<app-transaction-form>`, which we'll review next.

## 12.2 Reviewing `TransactionFormComponent`

The template of the `AppComponent` hosts two child components: `TransactionFormComponent` and `BlockComponent`. The template of the `TransactionFormComponent` is a 3-control form with a button ADD TRANSACTION as seen in listing 12.6. We took a regular HTML tag `<form>` and added to it the `[formGroup]="transactionForm"` directive to enable the reactive forms API offered by Angular.

## Listing 12.6 transaction-form.component.html

```

<h2>New transaction</h2>
<form class="add-transaction-form"
      [formGroup]="transactionForm"
      (ngSubmit)="enqueueTransaction()">
  <input type="text"
        name="sender"
        autocomplete="off"
        placeholder="Sender"
        formControlName="sender"> ①

  <span class="hidden-xs"></span>

  <input type="text"
        name="recipient"
        autocomplete="off"
        placeholder="Recipient"
        formControlName="recipient"> ②

  <input type="number"
        name="amount"
        autocomplete="off"
        placeholder="Amount"
        formControlName="amount"> ③

  <button type="submit"
          class="ripple"
          [disabled]="transactionForm.invalid || node.isMining"> ④
    ADD TRANSACTION
  </button>
</form>

```

- ① Binding the class property `transactionForm` to Angular directive `formGroup`
- ② Invoke `enqueueTransaction()` when the submit button is clicked
- ③ The name of the corresponding property in the form model
- ④ Using property binding to conditionally disable the submit button

In section 11.7 in chapter 11, we gave you a brief intro to Angular reactive forms, and listing 12.6 uses the directives of this API as well. Note that we start with binding the model object `transactionForm` (defined in the class `BlockComponent`) to the `formGroup` attribute. Also, each form control has an attribute `formControlName`, which corresponds to the property with the same name of the `transactionForm` object. The code of the `TransactionFormComponent` is shown in listing 12.7.

## Listing 12.7 transaction-form.component.ts

```

import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { BlockchainNodeService } from '../shared/services';

@Component({
  selector: 'app-transaction-form',
  templateUrl: './transaction-form.component.html'
})
export class TransactionFormComponent {
  readonly transactionForm: FormGroup;

  constructor(readonly node: BlockchainNodeService,      ①
              fb: FormBuilder) {                         ①
    this.transactionForm = fb.group({                   ②
      sender : ['', Validators.required],            ③
      recipient: ['', Validators.required],          ③
      amount : ['', Validators.required]             ③
    });
  }

  enqueueTransaction() {                            ④
    if (this.transactionForm.valid) {
      this.node.addTransaction(this.transactionForm.value);
      this.transactionForm.reset();
    }
  }
}

```

- ① Injecting services
- ② Declaring the model object transactionForm
- ③ Each form control has required validator attached and no initial value
- ④ This method is invoked when you add a new transaction to the list of pending ones

Once again, revisit the template of the `AppComponent` shown in listing 12.1, and you'll find there the `*ngFor` loop that renders the child component(s) `<app-block>`, which we'll review next.

### 12.3 Reviewing the `BlockComponent`

The `BlockComponent` is responsible for rendering one block, and its template is shown in listing 12.8. This template is pretty straightforward: a bunch of `<div>` and `<span>` tags, and the `Block` properties are inserted using binding represented by double curly braces.

## Listing 12.8 block.component.html

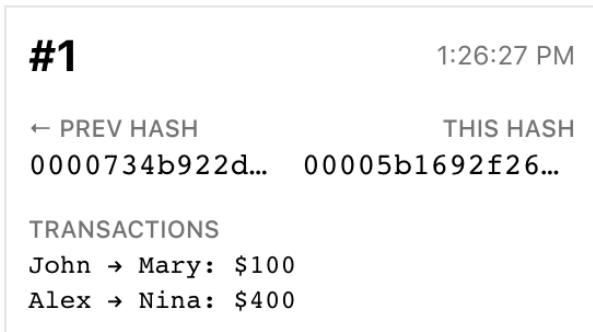
```

<div class="block">
  <div class="block__header">
    <span class="block__index">#{{ index }}</span> ①
    <span class="block__timestamp">{{ block.timestamp | date:'mediumTime' }}</span> ①
  </div>
  <div class="block__hashes">
    <div class="block__hash">
      <div class="block__label"> PREV HASH</div>
      <div class="block__hash-value">{{ block.previousHash }}</div> ①
    </div>
    <div class="block__hash">
      <div class="block__label">THIS HASH</div>
      <div class="block__hash-value">{{ block.hash }}</div> ①
    </div>
  </div>
  <div>
    <div class="block__label">TRANSACTIONS</div>
    <pre class="block__transactions">{{ formattedTransactions }}</pre> ②
  </div>
</div>

```

- ① Inserting the values of the Block property in the the template
- ② Inserting the formatted transactions

On top of the template, we used `date` pipe to format the `date: block.timestamp | date: 'mediumTime'`, which will render the date like in the form `h:mm:ss a`, and the rendered block is shown in figure 12.4. You can read about the `date` pipe at [angular.io/api/common/DatePipe](https://angular.io/api/common/DatePipe).



**Figure 12.4 A block rendered in the browser**

The TypeScript class `BlockComponent` is shown in listing 12.9. It's a presentation component that just receives the values from its parent and displays them. No app logic is applied here.

## Listing 12.9 block.component.ts

```
import { Component, Input } from '@angular/core';
import { Block, formatTransactions } from '../shared/services';

@Component({
  selector: 'app-block',
  templateUrl: './block.component.html'
})
export class BlockComponent {
  @Input() index: number;           ①
  @Input() block: Block;            ②

  get formattedTransactions(): string {
    return formatTransactions(this.block.transactions); ③
  }
}
```

- ① This input property will get the index from the parent component
- ② This input property will get the Block from the parent component
- ③ Formatting transactions using the function from the file blockchain-node.service.ts

One more time, revisit the listing 12.1 where the parent component's template uses the directive `*ngFor` to loop through all blocks in the blockchain and pass the data to the each `BlockComponent` instance as seen in listing 12.10:

## Listing 12.10 A fragment from app.component.html

```
<app-block
  *ngFor="let blk of node.chain; let i = index"
  [block]="blk" [index]="i">
</app-block>
```

The instance of the block object (`blk`) and the current index (`i`) are passed to the `BlockComponent` instance via bindings through the `@Input()` properties, and the user sees the values of `block` and `index` as seen in figure 12.4.

Any Angular component can receive data from its parent using the properties marked with the `@Input()` decorator, and our `BlockComponent` has two of such properties. The code snippet in listing 12.10 may look a little confusing, so let's imagine that we have a parent component that needs to display just one block. Listing 12.11 shows how such a parent could have passed data to the `BlockComponent`.

## Listing 12.11 A parent passing data to a child

```

@Component({
  selector: 'app-parent',
  template: `Meet my child
<app-block
  [block]="blk"          ①
  [index]="blockNumber">  ②
</app-block>

`)

export class ParentComponent {           ③
  blk: Block =
    { hash: "00005b1692f26",
      nonce: 2634,
      previousHash: "0000734b922d",
      timestamp: 25342683;
      transactions: ["John to Mary $100",
                      "Alex to Nina $400"];
    };
  blockNumber: 123;                    ③
}

```

- ① Binding the value to the block property of the child
- ② Binding the value 123 to the index property of the child
- ③ Initializing the values in the parent

**TIP**

The child can pass the data to the parent via the property marked with the `@Output()` decorator. You can read about it in the Yakov Fain's blog "Angular 2: Component communication with events vs callbacks" at [angular-2-component-communication-with-events-vs-callbacks](https://angular-2-component-communication-with-events-vs-callbacks).

So far we've been reviewing components (the classes with UI) of our blockchain app. Now let's review services (the classes with the app logic).

## 12.4 Reviewing services

In chapter 10, the blockchain app came with the class `WebSocketController`, which was instantiated with the `new` keyword. Here, the same functionality is wrapped into a service instantiated and injected by Angular. In this project, services are located in the directory `shared/services`, and listing 12.12 shows a fragment of the `websocketService` responsible for all communications with our WebSocket server.

## Listing 12.12 A fragment from websocket.service.ts

```

interface PromiseExecutor<T> {
    resolve: (value?: T | PromiseLike<T>) => void;
    reject: (reason?: any) => void;
}

@Injectable({
    providedIn: 'root'
})
export class WebsocketService {
    private websocket: Promise<WebSocket>;
    private readonly messagesAwaitingReply = new Map<UUID, PromiseExecutor<Message>>();
    private readonly _messageReceived = new Subject<Message>(); ③

    get messageReceived(): Observable<Message> {
        return this._messageReceived.asObservable(); ④
    }

    constructor(private readonly crypto: CryptoService) {
        this.websocket = this.connect(); ⑤
    }

    private get url(): string {
        const protocol = window.location.protocol === 'https:' ? 'wss' : 'ws';
        const hostname = environment.wsHostname; ⑥
        return `${protocol}://${hostname}`;
    }

    private connect(): Promise<WebSocket> {
        return new Promise((resolve, reject) => {
            const ws = new WebSocket(this.url);
            ws.addEventListener('open', () => resolve(ws));
            ws.addEventListener('error', err => reject(err));
            ws.addEventListener('message', this.onMessageReceived);
        });
    }
}

```

- ① PromiseExecutor knows which client waits for the response
- ② This service is a singleton available to all components and other services
- ③ Creating an instance of the RxJS Subject
- ④ Get the observable portion of the Subject
- ⑤ Connect to the WebSocket server
- ⑥ Get the URL of the server from the environment variable

**TIP**

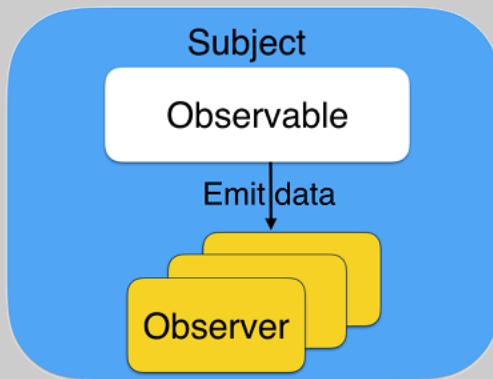
The type `PromiseExecutor` was explained in listing 10.29 in chapter 10.

The `WebsocketService` object is injected into the `AppComponent` (see listing 12.2), which can subscribe and handle messages coming from the server or send messages to the server, e.g. requesting the longest chain or announcing the new block.

**SIDE BAR**    **RxJS: Observable, Observer, and Subject**

The library RxJS offers different ways of handling streams of data. If you have an instance of an `Observable` object, you can invoke `subscribe()` on it providing the `Observer` instance that knows what to do with the data. Every time the `Observable` emits new data, the `Observer` will process it.

RxJS `Subject` encapsulates and `Observable` and `Observer`. One `Subject` can have multiple observers as seen in figure 12.5, and each of them represents one subscriber.



**Figure 12.5 Broadcasting with the RxJS Subject**

We use `Subject`, for broadcasting the data to all subscribers, which is done by invoking the method `next(someData)` on the `Subject`. To subscribe to the data, invoke `subscribe()` on the `Subject`. Listing 12.13 shows a code fragment that creates one `Subject` instance and two subscribers. In the last line, the code emits 123, and each subscriber will get this value.

### Listing 12.13 One subject two subscribers

```

const mySubject = new Subject();          ①
...
const subscription1 = mySubject.subscribe(...); ②
const subscription2 = mySubject.subscribe(...); ③
...
mySubject.next(123);                   ④
  
```

- ① Creating a Subject
- ② Creating the first subscriber
- ③ Creating the second subscriber
- ④ Broadcasting 123 to subscribers

If you want to restrict a piece of code so it can only subscribe (but no emit) to the subject, give this code only the observable portion of the `Subject` using the method `asObservable()` as seen in the getter `messageReceived` in listing 12.12.

The service `WebsocketService` gets the URL of the WebSocket server from the environment variable `environment.wsHostname`, which is defined in each of the files located in the environments directory of the project. Since our client starts in the dev mode (`ng serve`), it uses the setting from the file `environments.ts` shown in listing 12.14.

### **Listing 12.14 The file environments/environment.ts**

```
export const environment = {
  production: false,          ①
  wsHostname: 'localhost:3000' ②
};
```

- ① The code runs in the dev mode
- ② This is the url for the dev WebSocket server

If you'd start the client with `ng serve --prod` or build the files with `ng build --prod`, your app would use the file `environment.prod.ts`, which can have different value of `wsHostname`.

The directory `shared/services` also has the following files:

- `blockchain-node.service.ts` - it's the code has the functionality for creating a block. In chapter 10, this functionality was implemented in the file `blockchain-node.ts`, and you can find the code review starting from listing 10.25.
- `crypto.service.ts` has the method `sha256()` that knows how to calculate the hash value.

This concludes the code review of the blockchain app where the front end was developed in Angular, which is a large piece of software, and in this chapter, we just scratched its surface, so you could see the forest for the trees. Due to space limitations, we didn't explain the principles of reactive programming supported by the library RxJS, which is included in Angular. We didn't show you Angular Material - a set of modern-looking UI component. These topics are explained in our 500-page book "Angular Development with TypeScript", Second Edition.

**NOTE**

If you like Angular's syntax for developing the client side of a web app, consider getting familiar with the server-side framework called `nest.js` (see [github.com/nestjs](https://github.com/nestjs)). It runs under Node.js and its syntax will look very familiar for any Angular developer. Nest.js supports TypeScript, comes with the CLI tool, and even has a module TypeORM, which allows you to work with relational databases from the server-side TypeScript.

## **12.5 Summary**

In this chapter you learned:

- How to run the Angular app in dev mode and use two servers
- How to organize TypeScript imports with `index.ts`

- How to create an Angular web client for our blockchain app.
- How to run the Angular client that communicates to the WebSocket server

In the next chapter, we'll show you how to use TypeScript with the React.js framework.

# 13

## *Developing React.js apps with TypeScript*

### ***This chapter covers:***

- A quick intro to the React.js library
- How React components use props and state
- How React components can communicate with each other.

The React.js library (a.k.a. React) was created by a Facebook engineer Jordan Walke in 2013, and today it has 1300 contributors and 130K stars on GitHub! According to Stack Overflow Developer Survey of 2019, it's the second most popular JavaScript library (jQuery remains the most broadly used library), and in this chapter, we'll show how to start developing web apps in React using TypeScript. React is not a framework but a library responsible for rendering views in the browser (think of the letter V in the MVC design pattern).

The main player of React is a component, and the UI of a web app consists of components having parent-child relations. But if Angular takes control of the entire root element of the Web page, React allows you to control a smaller page element (e.g. a `<div>`) even if the rest of the page was implemented with any other framework or in pure JavaScript.

You can develop React apps either in JavaScript or in TypeScript and deploy them using tools like Babel and Webpack described in chapter 6, and without further ado let's start by writing the simplest version of the Hello World app using React and JavaScript; we'll switch to TypeScript in section 13.2.

## 13.1 Developing the simplest web page with React

In this section, we'll show you two versions of a very simple web page written with React and JavaScript. Each of these pages renders Hello World, but if the first version uses React with no additional tooling, the second version will engage Babel.

In the real-world apps, a React app is a project with configured dependencies, tools and the build process, but to keep things simple, our first web page will have just a single HTML file, which loads the React library from CDN. The code of the first version of the Hello World page is located in the file `hello-world-simplest/index.html`, and its content is shown in listing 13.1.

### Listing 13.1 `hello-world-simplest/index.html`

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"> ①
    </script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"> ②
    </script>
  </head>
  <body>
    <div id="root"></div> ③

    <script >
      const element = React.createElement('h1',   ④
                                      null,        ⑤
                                      'Hello World'); ⑥
      ReactDOM.render(element,
                        document.getElementById('root')); ⑦
    </script>

  </body>
</html>
```

- ① Loading the React package from CDN
- ② Loading the ReactDOM package from CDN
- ③ Adding the `<div>` with the id "root"
- ④ Creating the `<h1>` element using `createElement` function
- ⑤ We don't pass any data (props) to the `<h1>` element
- ⑥ The text of the `<h1>` element
- ⑦ Rendering the `<h1>` inside the `<div>`

The processes of declaring the page content (`React.createElement()`) and rendering it to the browser's DOM (`ReactDOM.render()`) are decoupled, and the former is supported by the API offered by the `React` object, while the latter is done by `ReactDOM`. Accordingly, we loaded these two packages in the `<head>` section of the page.

In React, UI elements are represented as a tree of React components which always has a single

root element, and this web page has a `<div>` with the ID `root` that serves as such element for the content rendered by React. In the script, we prepare the element to render using `React.createElement()`, and then invoke `ReactDOM.render()` that finds the element with the `root` ID and renders it there.

**TIP**

In Chrome, right-click on Hello World and select the menu Inspect. It'll open the Dev Tools showing the `<div>` with the `<h1>` element inside.

The method `createElement()` has three arguments: the name of HTML element, its *props* (data to be passed to the element) and content. In this case, we didn't need to provide any props (think attributes) and used `null` here; we'll explain what props are for in section 13.6. The content of `h1` is "Hello World", but it can contain child elements (e.g. `ul` with nested `li` elements), which could be created with the nested `createElement()` calls.

Open the file `index.html` in your browser, and it'll render the text Hello World as seen in figure 13.1.



## Hello World

**Figure 13.1** Rendering `hello-world-simplest/index.html`

Invoking `createElement()` on the page that has only one element is fine, but for page had dozens of elements this would become tedious and annoying. React allows you to embed the UI markup into the JavaScript code, which looks like HTML, but is JSX, which we'll discuss in the sidebar in section 13.2. Meanwhile let's see how our Hello World page could look like if we used JSX (note `const myElement`) instead of invoking `createElement()` as seen in listing 13.2.

## Listing 13.2 hello-world-simplest/index\_jsx.html

```

<!DOCTYPE html>
<head>
  <meta charset="utf-8">
  <script
    src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script
    src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>

  <script src="https://unpkg.com/babel-standalone/babel.min.js"></script> ①
</head>
<body>
  <div id="root"></div>
  <script type="text/babel"> ②
    const myElement = <h1>Hello World</h1>; ③

    ReactDOM.render( ④
      myElement,
      document.getElementById('root')
    );

    console.log(myElement); ⑤
  </script>
</body>
</html>

```

- ① Adding Babel from CDN
- ② The type of the script is text/babel
- ③ Assigning a JSX value to a variable
- ④ Initiating the rendering of myElement to the <div>
- ⑤ Monitoring the JavaScript object that was rendered

This app renders the same page as seen in figure 13.1, but it's written differently. The JavaScript code has embedded string `<h1>Hello World!</h1>` that looks like HTML to you, but it's actually JSX. Browsers can't parse this, so we need a tool to turn JSX into a valid JavaScript. Babel to the rescue!

The `<head>` section in listing 13.2 has an additional `<script>` tag that loads Babel from the CDN. Also, we changed the type of our script to `text/babel`, which makes browsers to ignore it, but tells Babel to transform the content of this `<script>` tag into JavaScript.

**NOTE**

We wouldn't use CDN for adding Babel to a Node-based project as we did in listing 13.2, but for demo purposes, it suffices. In the Node-based apps, Babel would be installed locally in the project and it would be a part of the build process.

Figure 13.2 shows a screenshot with the browser's console open. Babel converted the JSX value to a JavaScript object that was rendered inside the `<div>`, and we printed this object in the

console.

# Hello World

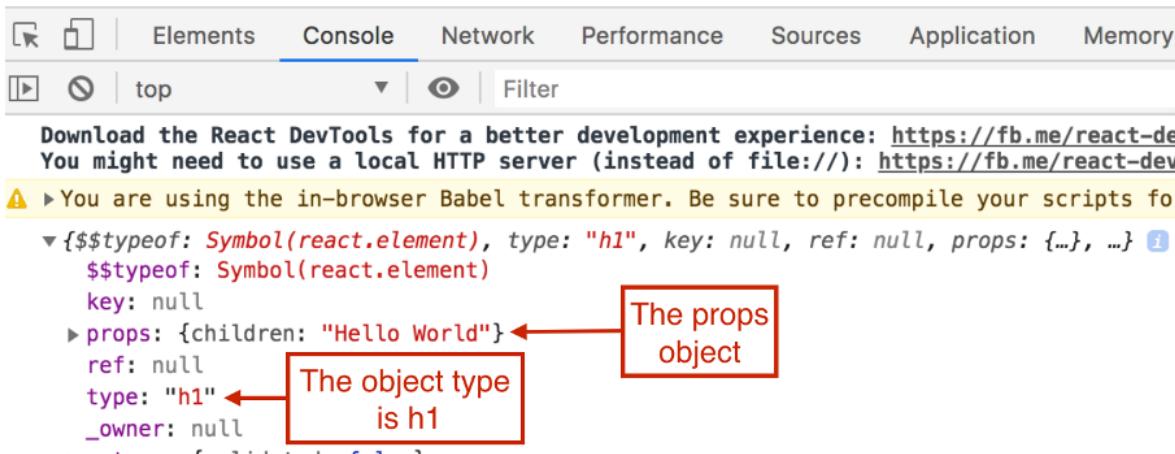


Figure 13.2 The rendered JavaScript object

Now that you have a fair understanding of how the very basic pages use React, let's switch to Node-based projects and component-based apps and see some tooling that React developers use in the real world.

## 13.2 Generating and running a new app with `create-react-app`

If you want to create a React app that includes a transpiler and a bundler, you'd need to add configuration files to your app, and this process is automated by the command-line interface (CLI) called `create-react-app` (see [www.npmjs.com/package/create-react-app](http://www.npmjs.com/package/create-react-app)). This tool generates all required configuration files for Babel and Webpack, so you can concentrate on writing your app instead of wasting time configuring tooling. To install the package `create-react-app` globally on your computer, run the following command in the Terminal window:

```
npm install create-react-app -g
```

Now you can generate either a JavaScript or a TypeScript version of the app. To generate the TypeScript app, run the command `create-react-app` followed by the app name and the option `--typescript`:

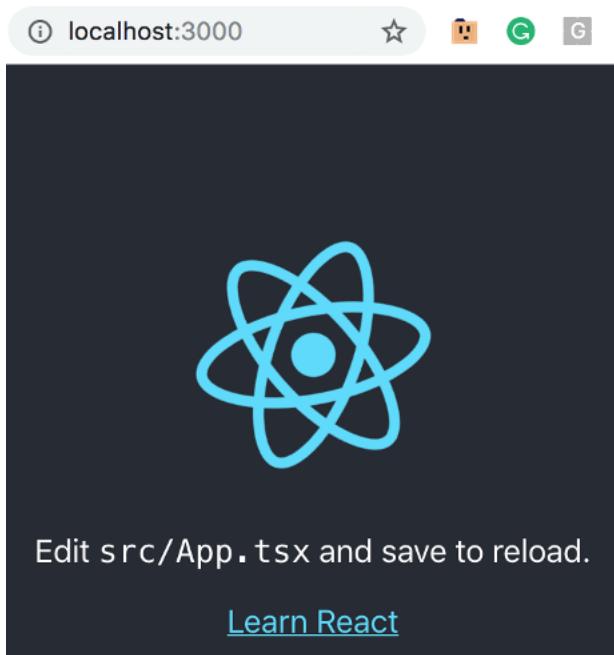
```
create-react-app hello-world --typescript
```

In a minute or so, all required files will be generated in the directory `hello-world` and the project

dependencies will be installed. In particular, it installs the following React packages:

- react - a JavaScript library for creating user interfaces
- react-dom - React package for working with the DOM
- react-scripts - scripts and configurations used by create-react-app; for TypeScript support, you need react-scripts of version 2.1 or higher

Besides the above packages, the CLI installs Webpack, Babel, TypeScript, their type definition files, and other dependencies. To launch the generated web app, switch to the directory hello-world and run `npm start`, which in turn runs `react-scripts start`. Webpack will bundle the app and webpack-dev-server will serve the app on localhost:3000 as shown in figure 13.3.



**Figure 13.3** Running the hello-world app

**TIP**

For bundling, Webpack uses the file `webpack.config.js` located in the directory `node_modules/react-scripts/config`.

The good part is that if you generated the project with `create-react-app`, it'll recompile the code, rebuild the bundles, and re-render the UI. This functionality is provided by the Webpack Dev Server. The UI of this app tells us to edit the file `src/App.tsx`, which is the main TypeScript file of the generated app. Open the directory in VS Code, and you'll see the project files as shown in figure 13.4.



**Figure 13.4 The generated files and directories**

The source code of your app is located in the src directory, and the public directory is for the assets of your app that shouldn't be included in the app bundles. For example, your app has thousands of images and needs to dynamically reference their paths, and the directory public is for files that don't require any processing before deployment.

The file index.html contains an element `<div id="root"></div>`, which serves as a container of the generated React app. You won't find any `<script>` tags for loading the React library code there; they'll be added during the build process when the app's bundles are ready.

**TIP** Run the app and open the Chrome Dev Tools under the Elements tab to see the runtime content of index.html.

**NOTE** The file serviceWorker.ts is generated just in case you want to develop a Progressive Web App (PWA) that can be started offline using cached assets. We are not going to use it in our sample apps.

As you see, some of the files have an unusual extension .tsx. If we'd be writing the code in JavaScript, the CLI would generate the app file with the extension .jsx (not .tsx) and the meanings of JSX and TSX are explained in the sidebar.

**SIDEBAR** **JSX and TSX**

The draft of the JSX specification (see [facebook.github.io/jsx](https://facebook.github.io/jsx)) offers the following definition: "*JSX is an XML-like syntax extension to ECMAScript without any defined semantics. It's NOT intended to be implemented by engines or browsers*".

JSX stands for JavaScript XML. It defines a set of XML tags that can be embedded inside the JavaScript code. These tags can be parsed and turned into regular HTML tags for rendering by the browser, and React includes such a parser. In chapter 6, we demonstrated Babel's REPL (see [babeljs.io/repl](https://babeljs.io/repl)), and figure 13.5 shows a screenshot of this REPL with a sample JSX.

On the left, we selected the preset `react` and pasted a sample code from the JSX spec. This preset means that we want to turn each JSX tag into `React.createElement()` invocation. The sample code should render a dropdown with a menu containing three menu items. On the right, you see how the JSX was parsed into JavaScript.

```

1 var dropdown =
2   <Dropdown>
3     A dropdown list
4     <Menu>
5       <MenuItem>Do Something</MenuItem>
6       <MenuItem>Do Something Fun!</MenuItem>
7       <MenuItem>Do Something Else</MenuItem>
8     </Menu>
9   </Dropdown>;
10
11 render(dropdown);

```

```

1 var dropdown = React.createElement(Dropdown, null,
2   "A dropdown list", React.createElement(Menu, null,
3     React.createElement(MenuItem, null, "Do
4       Something"), React.createElement(MenuItem, null,
5         "Do Something Fun!"))
6     React.createElement(MenuItem, null, "Do Something
7       Else")));
8   render(dropdown);

```

Turning each JSX tag into createElement()

Figure 13.5 Parsing JSX in Babel

**SIDE BAR** Every React app has at least one component, which is called a *root component*, and our generated app has only the root component `App`. The file with the code of the function `App` has the name extension `.tsx`, which tells the TypeScript compiler that it contains JSX. But just having the extension `.tsx` is not enough for `tsc` to handle it: you need to enable JSX by adding the `jsx` compiler option. Open the file `tsconfig.json`, and you'll find there the following line:

```
"jsx": "preserve"
```

The `jsx` option only affects the emit stage - type checking is unaffected. The value `preserve` tells `tsc` to copy the JSX portion into the output file changing its extension to `.jsx`, because there will be another process (e.g. Babel) that will be parsing it. If the value would be `react` `tsc` would turn the JSX tags to `React.createElement()` invocations as seen in figure 13.5 on the right.

A React component can be declared either as a function or as a class. A functional component is implemented as a function and is structured as shown in listing 13.3 (types are omitted).

### Listing 13.3 A functional component

```
const MyComponent = (props) => {    ①
  return (
    <div>...</div>    ②
  )
  // other functions may go here
}
export default MyComponent;
```

- ① props is used to pass data to the components
- ② Return the component's JSX

Developers who prefer working with classes can create a class-based component, which is implemented as a subclass of `React.Component` and is structured as in listing 13.4.

## Listing 13.4 A class-based component

```
class MyComponent extends React.Component { ①

  render() { ②
    return (
      ③
      <div>...</div>
    );
  }

  // other methods may go here
}

export default MyComponent;
```

- ① The class must be inherited from `React.Component`
- ② The `render()` method is invoked by React
- ③ Return JSX for rendering

If a functional component would simply return JSX, the class-based one has to include the method `render()` that returns JSX. We prefer using functional components, which have several benefits over the class-based ones:

- A function requires less code to write; there is no need to inherit the component's code from any class either
- A functional component generates less code during Babel transpiling, and code minifiers better infer unused code and can shorten variables names more aggressively since all of them are local to the function, unlike class members that considered public API and cannot be renamed
- No need for the `this` reference
- Functions are easier to test than classes; assertions simply map props to the returned JSX

**NOTE**

You should use class-based components only if you have to use the React version older than 16.8. In old versions, only the class-based components would support the component's state and lifecycle methods.

If you use the current version of `create-react-app` with the `--typescript` option, the generated file `App.tsx` file will already have the boilerplate code of a functional component (i.e. a function of type `React.FC`) shown in listing 13.5.

### Listing 13.5 The file App.tsx

```

import React from 'react';      ①
import logo from './logo.svg';
import './App.css';

const App: React.FC = () => {    ②
  return (
    ③
    <div className="App">        ④
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.tsx</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;      ⑤

```

- ① Import React library
- ② This is a functional component
- ③ Return the component's template as a JSX expression ( it's not a string)
- ④ In JSX, use className instead of the CSS selector class to avoid conflicts with the JavaScript keyword class
- ⑤ Export the component declaration so it can be used in other modules

**NOTE** We use version 3.0 of `create-react-app`. The older versions of this tool would generate a class-based component `App`.

The generated `App` function returns the markup (or template), which React uses for rendering this component's UI as seen earlier in figure 13.3. During the build process, Babel will convert it into a pure JavaScript object `JSX.element` with the `<div>` container that will update the Virtual DOM (explained in section 13.5) and the browser's DOM. This `App` component didn't have a separate place for storing its data (a.k.a. state), and we'll add it in the next section.

## 13.3 Managing the component's state

A component's state is the datastore that contains the data that should be rendered by the component. The data in the component's state is preserved even if React re-renders the component. Whenever the code updates the component's state, React updates the component's UI to reflect changes caused by the user's actions (e.g. button clicks or typing in the input fields) or other events. If you have a Search component, its state can store the last search criteria and the last search result.

**NOTE**

Do not confuse the component's state with the application's state. The former is the storage of an individual component's state while the latter stores the app's data that may come from multiple components, functions, or classes.

And how would you define and update the component's state? This depends on how the component was created in the first place. We're going to move back to class-based components for a minute so that you can understand the difference in dealing with state in class-based and functional approaches. Then we'll return to functional components, which we recommend using.

### 13.3.1 Adding state to a class-based component

If you have to work with a class-based component, you could define a type representing the state, create and initialize an object of this type, and then update it as needed by invoking `this.setState(...)`.

Let's consider a simple class-based component that has a state object with two properties: the user name and the image to be displayed. For serving images we'll use the web site called Lorem Picsum, which returns random images of the specified size. For example, if you enter the URL [picsum.photos/600/150](https://picsum.photos/600/150), the browser will show a random image having the width 600px and height 150px. Listing 13.6 shows such a class-based component with a two-property state object.

## Listing 13.6 An app component with state

```

interface State { ①
  userName: string;
  imageUrl: string;
}

export default class App extends Component {

  state: State = { userName: 'John', ②
    imageUrl: 'https://picsum.photos/600/150' };

  render() {
    return (
      <div>
        <h1>{this.state.userName}</h1> ③
        <img src={this.state.imageUrl} alt="" /> ④
      </div>
    );
  }
}

```

- ① Defining the type for the component's state
- ② Initializing the State object
- ③ Rendering the userName here
- ④ Rendering the imageUrl here

By looking at the code of the `render()` method, you can guess that this component would render John and the image. Note that we embedded the values of the state properties into JSX by placing them inside the curly braces, e.g. `{this.state.userName}`.

Any class-based component is inherited from the class `Component`, which has a property `state` and the method `setState()`. If you need to change the value of any state property, you must do it using this method, for example:

```
this.setState({userName: "Mary"});
```

By invoking `setState()` you let React know that the UI update may be required. If you update the state directly (e.g. `this.state.userName='Mary'`), React won't call the method `render()` to update the UI. As you might have guessed, the `state` property is declared on the base class `Component`.

In section 13.2, we listed the benefits of functional components over the class-based ones, and we won't use class-based components any longer. In functional components, we manage state by using *hooks* introduced in React 16.8.

### 13.3.2 Using hooks for managing state in functional components

In general, hooks allow to "attach" behavior to a functional component without the need to write classes, create wrappers or use inheritance. It's as if you say to a functional component, "I want you to have additional functionality while remaining a plain old function".

Hooks must have their names started with the word `use` - this is how Babel detects them and distinguishes them from regular functions. For example, `useState()` is the name of the hook for managing the component's state, while `useEffect()` is used for adding a side-effect behavior (e.g. fetching data from a server). In this section, we'll focus on the `useState()` hook using the same example as in the previous section: a component whose state is represented by the user name and the image URL, but this time it'll be a functional component.

The `useState()` hook can create a primitive value or a complex object and preserve it between the functional component invocations. The following line shows you how to define a state for the user name.

```
const [userName, setUserName] = useState('John');
```

The function `useState()` returns a pair: the current state value and a function that lets you update it. Do you remember the syntax of array destructuring introduced in ECMAScript 6? If not, read section A.8.2 in Appendix A. The above line means that the hook `useState()` takes the string 'John' as an initial value and returns an array, and we use destructuring to get the two elements of this array into two variables: `userName` and `setUserName`. The syntax of array destructuring allows you to give any names to these variables. If you need to update the value of `userName` from John to Mary and make React to update the UI (if needed), do it as follows:

```
setUserName('Mary');
```

**TIP**

In your IDE, do CMD-Click or Ctrl-Click on the `useState()`, and it'll open the type definition of this function, which will declare that this function *returns a stateful value and a function to update it*. The function `useState()` is not a pure function because it stores the component's state somewhere inside React. It's a *function with side effects*.

Listing 13.7 shows a functional component that stores the state in two primitives: `userName` and `imageUrl` and displays their values using JSX.

## Listing 13.7 Using primitives for storing state

```
import React, {useState} from 'react';      ①

const App: React.FC = () => {

  const [userName, setUserName] = useState('John');    ②
  const [imageUrl, setImageUrl] = useState('https://picsum.photos/600/150');  ③

  return (
    <div>
      <h1>{userName}</h1>  ④
      <img src={imageUrl} alt="" />  ⑤
    </div>
  );
}

export default App;
```

- ① Importing the useState hook
- ② Defining the userName state
- ③ Defining the imageUrl state
- ④ Rendering the value of the state variable userName
- ⑤ Rendering the value of the state variable imageUrl

Now let's re-write the component from listing 13.7 so instead of two primitives, it'll declare its state as an object with two properties: `userName` and `imageUrl`. Listing 13.8 declares an interface `State` and uses the `useState()` hook to work with the object of type `State`.

## Listing 13.8 Using an object for storing state

```
import React, {useState} from 'react';

interface State {      ①
  userName: string;
  imageUrl: string;
}

const App: React.FC = () => {

  const [state, setState] = useState<State>({      ②
    userName: 'John',
    imageUrl: 'https://picsum.photos/600/150'
  });

  return (
    <div>
      <h1>{state.userName}</h1>  ③
      <img src={state.imageUrl} alt="" />  ④
    </div>
  );
}

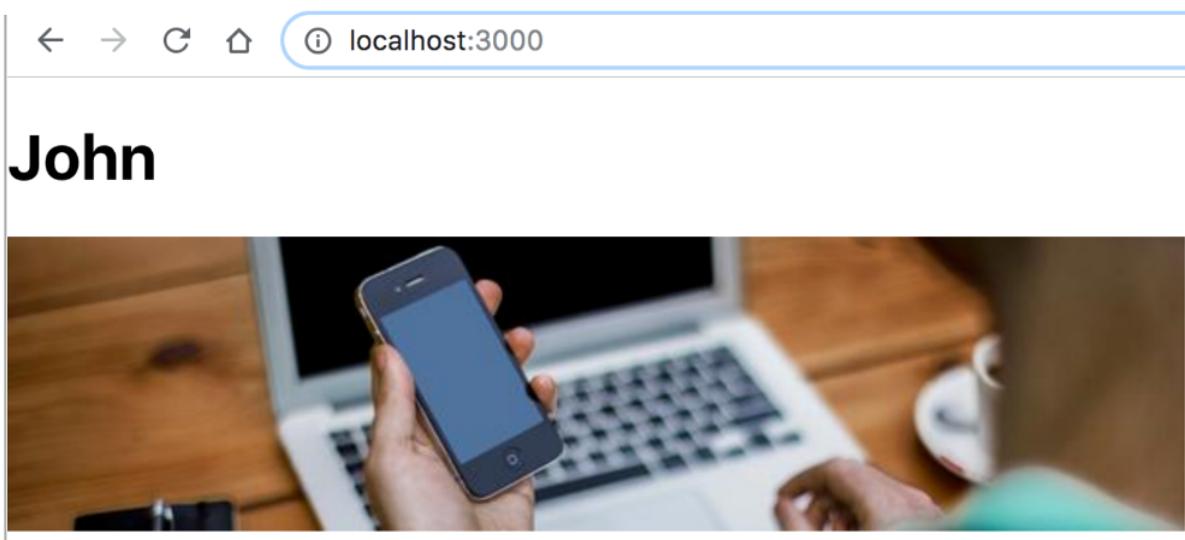
export default App;
```

- ① Defining the type for the component state

- ② Defining and initializing the state object
- ③ Rendering the value of the state property `userName`
- ④ Rendering the value of the state property `imageUrl`

Note that the `useState()` is a generic function and during its invocation, we provided the concrete type `State`.

The source code of this sample app is located in the directory `hello-world`. Run the command `npm start` and the browser will render the window that look similar to figure 13.6 (the image may be different though).



**Figure 13.6** Rendering user name and image

The user name and image are too close to the left border of the window, which is easy to fix with CSS. The generated app shown in listing 13.5 had a separate file `app.css` with CSS selectors applied in the component with the `className` attribute; you can't use `class` to avoid a conflict with the reserved JavaScript keyword `class`. This time, we'll add the `margin` by declaring a JavaScript object with styles and using it in JSX. In listing 13.9, we added the variable `myStyles` and used it in the component's JSX.

### Listing 13.9 Adding styles

```
const App: React.FC = () => {
  const [state, setState] = useState<State>({
    userName: 'John',
    imageUrl: 'https://picsum.photos/600/150'
  });

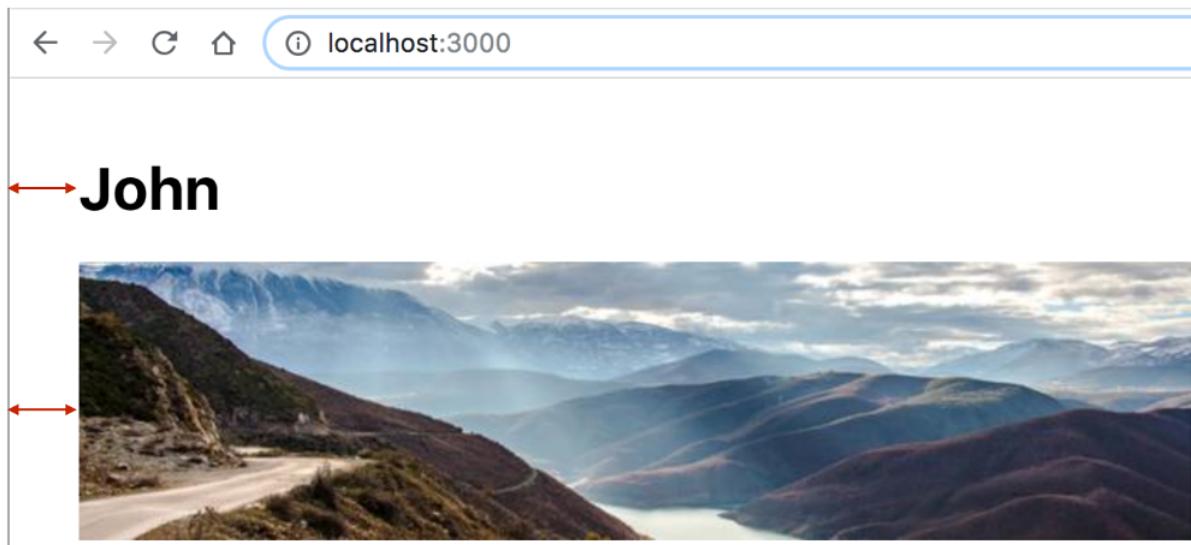
  const myStyles = {margin: 40}; ①

  return (
    <div style ={myStyles}> ②
      <h1>{state.userName}</h1>
      <img src={state.imageUrl} alt="" />
    </div>
  );
}
```

- ① Declaring the styles
- ② Applying the styles

Note that the `style` property is strongly typed, which helps in validating CSS properties. This is one of the JSX advantages over plain HTML - since it's JavaScript, HTML and CSS elements are strongly typed via the TypeScript type definition files.

With this margin, the browser will render the `<div>` with additional 40px of space around it as shown in figure 13.7.



**Figure 13.7 Adding the margin**

Our first React app works and looks nice! It has one functional component that stores hard-coded data in the state object and renders them using JSX. It's a good start, and in the next section, we'll start writing a new app that will have more functionality.

## 13.4 Developing a weather app

In this section, we'll develop an app that will let the user enter the name of the city and get the current weather there. We'll develop this app gradually, and its first version won't be fetching data from the weather server just yet. First, we'll add a little HTML form to the `App` component, where the user will be entering the name of the city.

Second, we'll add the code to fetch the real weather data from the weather server and the `App` component will display the weather.

Finally, we'll create another component `WeatherInfo`, which will be used as a child of the `App` component. The `App` component will retrieve the weather data and will pass it to `WeatherInfo`, which will display the weather.

We'll be getting real weather data from the weather service at [openweathermap.org](https://openweathermap.org), which provides an API for making weather requests for many cities around the world. This service returns the weather information as a JSON-formatted string. For example, to get the current temperature in London in Fahrenheit (`units=imperial`), the URL could look like this:

[api.openweathermap.org/data/2.5/find?q=London&units=imperial&appid=12345](https://api.openweathermap.org/data/2.5/find?q=London&units=imperial&appid=12345)

Creators of this service require you to receive the application ID, which is a simple process. If you want to run our weather app, do the same and replace 12345 in the above URL with your APPID.

The sample code for this chapter includes the weather app located in the directory `weather`, which was initially generated by the following command:

```
create-react-app weather --typescript
```

Then, we replaced the JSX code in the `app.tsx` with a simple HTML form where the user could enter the name of the city and press the button Get Weather. Also, the entered city represents the state of this component, and the `App` component will be updating its state as the user is entering the city name.

### 13.4.1 Adding a state hook to the App component

The first version of our `App` component defines its state with the `useState()` hook as follows:

```
const [city, setCity] = useState('');
```

Now the value in the variable `city` has to be updated using the function `setCity()`. Our hook `useState()` initializes the variable `city` with an empty string, so TypeScript will infer the type

of city as a string. Listing 13.10 shows the App component with the declared state, and the form defined in the JSX section. This code also has an event handler handleChange(), which is invoked each time the user enters or updates any character in the input field.

### **Listing 13.10 The file App.tsx in the weather app**

```
import React, { useState, ChangeEvent } from 'react';

const App: React.FC = () => {
    const [city, setCity] = useState(''); ①

    const handleChange = (event: ChangeEvent<HTMLInputElement>) => { ②
        setCity(event.target.value); ③
    }

    return (
        <div>
            <form>
                <input type="text" placeholder="Enter city"
                    onChange = {handleChange} /> ④
                <button type="submit">Get weather</button>
                <h2>City: {city}</h2> ⑤
            </form>
        </div>
    );
}

export default App;
```

- ① Declaring the state city
- ② Declaring the function to handle the input field events
- ③ Updating the state by invoking setCity()
- ④ Assigning the handler to the onChange attribute
- ⑤ Displaying the current state value

The input field defines the event handler `onChange = {handleChange}`. Note that we didn't invoke `handleClick()` here; we just provided the name of this function. React's `onChange` behaves as `onInput` and is fired as soon as the content of the input field changes. As soon as the user enters (or changes) a character in the input field, the function `handleChange()` is invoked; it updates the state, which causes the UI update.

**TIP**

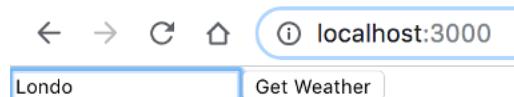
There is no documentation on which types of the React events to use with specific JSX elements, and to avoid using `event: any` as the argument in event handler functions, open the file `index.d.ts` in the directory `node_modules/@types/react` and search for "Event Handler Types". This should help you to figure out that the proper type for the `onChange` event is a generic `ChangeEvent<T>` that takes the type of a specific element as a parameter, i.e. `ChangeEvent<HTMLInputElement>`.

To illustrate the state updates, we've added the `<h2>` element that displays the current value of the state: `<h2>Entered city: {city}</h2>`. Note that for re-rendering the current value of `city`, we didn't need to write a jQuery-like code finding the reference to this `<h2>` element and changing its value directly. The invocation of `setCity(event.target.value)` forces React to update the corresponding node in the DOM.

In general, if you need to update the functional component's state, do it only using the appropriate `setXXX()` function that's returned by the hook `useState()`:

1. By invoking `setXXX()` you let React know that the UI update may be required. If you update the state directly (e.g. `city = "London"`), React won't update the UI.
2. React may batch the UI updates before reconciliating the Virtual DOM with the browser's one.

Figure 13.8 shows a screenshot taken after the user entered *Londo* in the input field.



## City: Londo

**Figure 13.8 After the user entered Londo**

**TIP**

To see that React updates only the `<h2>` node in the DOM, run this app (`npm start`) with the Chrome Dev Tools open in the tab Elements. Expand the DOM tree so the content of the `<h2>` element is visible, and start typing in the input field. You'll see that the browser changes only the content of the `<h2>` element, while all other elements remain unchanged.

## SIDE BAR Redux and app state management

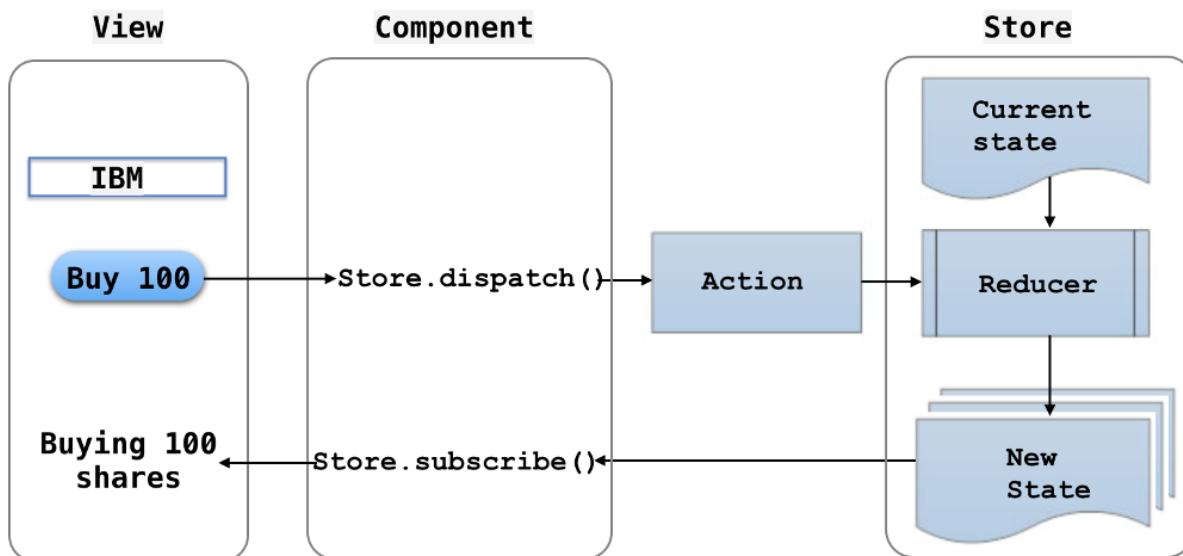
The `useState()` hook in functional components (or the method `setState()` in class-based components) is used to store the internal component's data and synchronizing the data with UI, but the entire app may also need to store and maintain some data used by multiple components or the current state of UI (e.g. the user selected the product X in the component Y). In React apps, the most popular state management JavaScript library is Redux, which is based on the following three principles:

- **Single source of truth.** There is a single *data store*, which contains the state of your app.
- **State is read-only.** When an action is emitted, the *reducer* function clones the current state and updates the cloned object based on the action.
- **State changes are made with pure functions.** You write the reducer function(s) that takes an action and the current state object, and returns a new state.

In Redux, the data flow is unidirectional:

1. The app component dispatches the action on the store
2. The reducer (a pure function) takes the current state object and then clones, updates, and returns it.
3. The app component subscribes to the store, receives the new state object and updates the UI accordingly.

Figure 13.9 shows the unidirectional Redux data flow.



**Figure 13.9 The Redux data flow**

**SIDE BAR** The user clicks on the button to buy 100 shares of the IBM stock. The click handler function invokes the method `dispatch()` emitting an *action*, which is a JavaScript object that has a property `type` describing what happened in your app, e.g. the user wants to buy the IBM stock. Besides the property `type`, an *action* object can optionally have another property with the payload of data for storing in the app state as seen in listing 13.11:

### Listing 13.11 An action with a payload

```
{
  type: 'BUY_STOCK',    ①
  stock: {symbol: 'IBM', quantity: 100}  ②
}
```

- ① The type of action
- ② The action payload

This object only describes the action and provides the payload, but it doesn't know how the state should be changed. Who does? The reducer, which is a *pure function* that specifies how the app state should be changed. The reducer never changes the current state, but creates its new version and returns a new reference to it. As seen in figure 13.9, the component subscribes to the state changes and updates the UI accordingly.

The reducer function doesn't implement the functionality, which requires work with external services (e.g. placing an order), because reducers are meant for updating and returning the app state based on the action and its payload, if any. For example, the stock to buy is "IBM". Besides, implementing the app logic would require interaction with the environment external to the reducer, i.e. it would cause *side effects*, but pure functions can't have side effects. For more details, read Redux documentation at [github.com/reactjs/redux/tree/master/docs](https://github.com/reactjs/redux/tree/master/docs).

MobX is another popular state management library for React.

Working with the component's state is an internal function of the component. But at some point, the component may need to start working with external data, and this is where the `useEffect()` hook comes in.

### 13.4.2 Fetching data with `useEffect` hook in the App component

You learned how to store the city in the state of the `App` component, but our ultimate goal is finding the weather in the given city by fetching the data from an external server. Using the terminology from functional programming, we need to write a function with *side effects*. As opposed to *pure functions*, the functions with side effects use the external data and every invocation may produce different results even if the function arguments remain the same.

In React's functional components, we'll be using the `useEffect()` hook for implementing the functionality with side effects. By default, React automatically invokes the callback function passed to `useEffect()` after every DOM rendering. Let's add the following function to the `App` component from listing 13.10:

```
useEffect(() => console.log("useEffect() was invoked"));
```

If you run the app with the browser console open, you'll see the message "useEffect() was invoked" each time when you enter a character in the input field and the UI is refreshed. Every React component goes through a set of lifecycle events, and if you need your code to be executed after the component was added to the DOM or each time it was re-rendered, the `useEffect()` is the right place for such a code. But if you want the code in `useEffect()` to be executed only once after initial rendering, specify an empty array as the second argument:

```
useEffect(() => console.log("useEffect() was invoked"), []);
```

The code in this hook will be executed only once, which makes it a good place for performing the initial data fetch right after the component has been rendered.

Let's assume you live in London and would like to see the weather in London as soon as this app is launched. Start by initializing the city state with "London":

```
const [city, setCity] = useState('London');
```

Now you'd need to write a function that would fetch the data for the specified city. The URL will include the following static parts (replace 12345 with your APPID).

```
const baseUrl = 'http://api.openweathermap.org/data/2.5/weather?q=';
const suffix = "&units=imperial&appid=12345";
```

In between, you need to place the name of the city, so complete URL may look like this:

```
baseUrl + 'London' + suffix
```

For making Ajax requests, we'll use the browser's Fetch API (see [developer.mozilla.org/en-US/docs/Web/API/Fetch\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)). The function `fetch()` returns a Promise, and we'll use the `async/await` keywords (see section A.10.4 in Appendix A) in our

method `getWeather()` that can look as in listing 13.12.

### **Listing 13.12 Fetching the weather data**

```
const getWeather = async (city: string) => {
  const response = await fetch(baseUrl + city + suffix); ①
  const jsonWeather = await response.json(); ②
  console.log(jsonWeather); ③
}
```

- ① Make an async call to the weather server
- ② Convert the response to the JSON format
- ③ Print the weather JSON on the console

**NOTE**

We prefer using the `async/await` keywords for asynchronous code, but using promises with chained `.then()` invocations would also work here.

When you use a standard browser's `fetch()` method, getting the data is a two-step process: You get the response first, and then you need to call the `json()` function on the response object to get to the actual data.

**TIP**

JavaScript developers often use third-party libraries for handling HTTP requests. One of the most popular ones is a promise-based library called Axios (see [www.npmjs.com/package/axios](http://www.npmjs.com/package/axios)).

Now you can use this function for the initial data fetch in the `useEffect()`:

```
useEffect(() => getWeather(city), []);
```

If you want the code in `useEffect()` to be executed selectively only if a specific state variable changed, you can attach the hook to such a state variable. For example, you can specify that the `useEffect()` has to run only if the `city` gets updated as follows:

```
useEffect(() => console.log("useEffect() was invoked"),
  ['city']);
```

The current version of the `App` component is shown in listing 13.13.

### Listing 13.13 Fetching the London weather in `useEffect()`

```
import React, { useState, useEffect, ChangeEvent } from 'react';

const baseUrl = 'http://api.openweathermap.org/data/2.5/weather?q=';
const suffix = "&units=imperial&appid=12345";

const App: React.FC = () => {

  const [city, setCity] = useState('London');

  const getWeather = async (city: string) => { ①
    const response = await fetch(baseUrl + city + suffix);
    const jsonWeather = await response.json();
    console.log(jsonWeather);
  }

  useEffect( { () => getWeather(city) }, []); ②

  const handleChange = (event: ChangeEvent<HTMLInputElement>) => {
    setCity( event.target.value ); ③
  }

  return (
    <div>
      <form>
        <input type="text" placeholder="Enter city"
              onInput = {handleChange} />
        <button type="submit">Get Weather</button>
        <h2>City: {city}</h2>
      </form>
    </div>
  );
}

export default App;
```

- ① Asynchronously fetch the weather data for the specified city
- ② An empty array means run this hook once
- ③ Updating the state

The second argument of the `useEffect()` is an empty array, so `getWeather()` will be invoked only once when the `App` component is initially rendered.

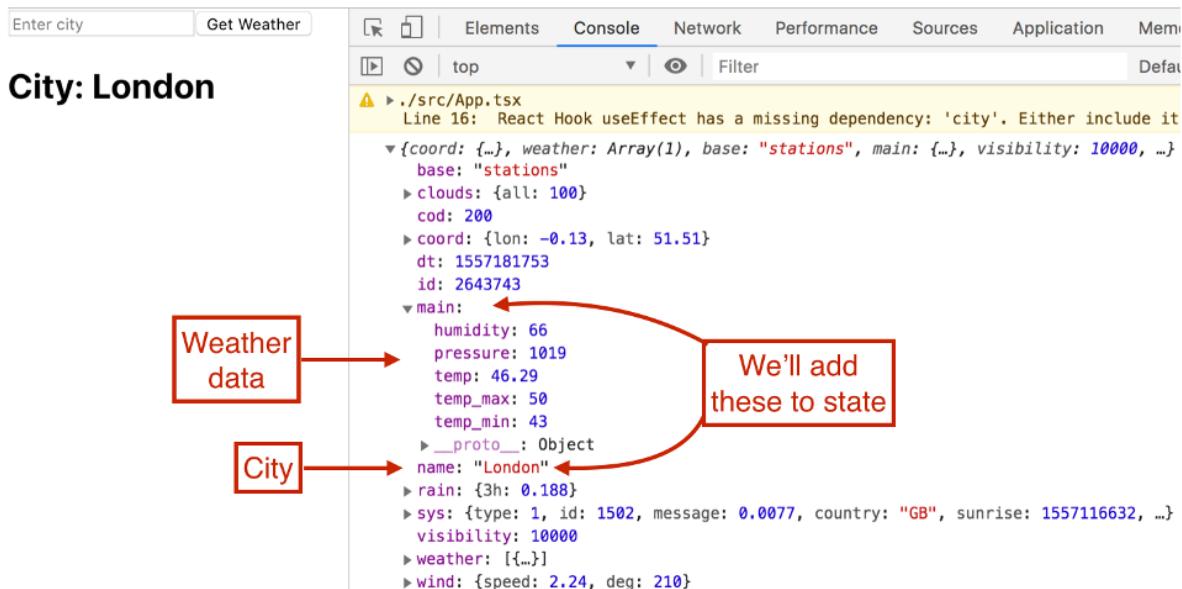
#### NOTE

If you run this app, the browser's console will show the following warning:  
*React Hook `useEffect` has a missing dependency: 'city'. Either include it or remove the dependency array `react-hooks/exhaustive-deps`.* The reason is that inside this hook we use the state variable `city`, which is a dependency and should be listed in the array. This is not an error and for simplicity we'll keep this code as is, but you should keep this in mind while designing your hooks.

**TIP**

For the in-depth coverage of the `useEffect()` hook, read the article by Dan Abramov "A complete guide to `useEffect`" available at [overreacted.io/a-complete-guide-to-useeffect](https://overreacted.io/a-complete-guide-to-useeffect).

Run this app with the browser console open, and it'll print the retrieved JSON with the London weather as shown in figure 13.10.



**Figure 13.10 The London's weather in the console**

The initial data fetch is complete for the default city, and it looks like a good idea to store the retrieved weather data in the component's state. Let's define a new type `Weather` as shown in listing 13.14 for storing the content of the properties `name` and `main` marked in figure 13.10.

#### **Listing 13.14 The file weather.ts**

```

export interface Weather {
  city: string; ①
  humidity: number; ②
  pressure: number; ②
  temp: number; ②
  temp_max: number; ②
  temp_min: number; ②
}
  
```

- ① This property corresponds to the `name` property shown in figure 13.10
- ② This value comes from the `main` property shown in figure 13.10

In the `App` component, we'll add a new state variable `weather` and the function to update it as follows:

```
const [weather, setWeather] = useState<Weather | null>(null);
```

Note that the hook `useState()` allows you to use a generic parameter for better type safety. Now we need update the function `getWeather()` so it saves the retrieved weather and city name in the component's state as shown in listing 13.15.

### Listing 13.15 Saving state in getWeather

```
async function getWeather(location: string) {
  const response = await fetch(baseUrl + location + suffix);
  if (response.status === 200){
    const jsonWeather = await response.json();
    const cityTemp: Weather = jsonWeather.main;      ①
    cityTemp.city=jsonWeather.name;      ②
    setWeather(cityTemp);      ③
  } else {
    setWeather(null);      ④
  }
}
```

- ① Storing the content of the main property
- ② Storing the city name
- ③ Saving the weather in the component's state
- ④ The weather retrieval failed

This code takes the object `jsonWeather.main` and the city name from `jsonWeather.name`, and saves them in the state variable `weather`.

So far, our function `getWeather()` was invoked for the initial retrieval of the London weather by the hook `useEffect()`. The next step is to add the code to invoke `getWeather()` when the user enters any other city and clicks on the button Get Weather. As per listing 13.13, this button is a part of the form (its type is `submit`), so we'll add the event handler to the `<form>` tag. The function `handleSubmit()` and the first version of the JSX are shown in listing 13.16.

### Listing 13.16 Handling the button click

```
const handleSubmit = (event: FormEvent) => {      ①
  event.preventDefault();      ②
  getWeather(city);      ③
}

return (
  <div>
    <form onSubmit = {handleSubmit}>      ④
      <input type="text" placeholder="Enter city"
        onChange = {handleChange} />
      <button type="submit">Get Weather</button>
      <h2>City: {city}</h2>
      {weather && <h2>Temperature: {weather.temp}F</h2>}      ⑤
    </form>
  </div>
);
```

- ① When the Submit button is clicked the FormEvent is dispatched

- ② Prevent default behavior of the form's submit button
- ③ Invoke `getWeather()` for the entered city
- ④ Attach the event handler to the form
- ⑤ Display the retrieved temperature

In React, event handlers get instances of `SyntheticEvent`, which is an enhanced version of browser's native events (see [reactjs.org/docs/events.html](https://reactjs.org/docs/events.html) for details). `SyntheticEvent` has the same interface as the browser native events (e.g. `preventDefault()`) but events work identically across all browsers.

To pass the argument to `getWeather(city)`, we didn't have to find the reference to the `<input>` field on the UI. The component's state `city` was updated as the user typed the name of the city, so the variable `city` already has the value displayed in the `<input>` field. Figure 13.11 shows a screenshot of the page after the user entered Miami and clicked on the button Get Weather.

## City: Miami

## Temerature: 85.28F

**Figure 13.11 It's hot in Miami**

**NOTE**

In our book "Angular Development with TypeScript", we also used this weather service, and you can find the Angular version of this app at [github.com/Farata/angulartypescript/..//chapter6/observables/src/app/weather](https://github.com/Farata/angulartypescript/blob/master/src/app/weather)

What if the user enters the city that doesn't exist or not supported by [openweathermap.org](http://openweathermap.org)? The server returns 404 and we should add the appropriate error handling. So far we have the following line to prevent displaying the temperature if the `weather` state is falsy:

```
{weather && <h2>Temperature: {weather.temp}F</h2> }
```

In the next version of this app, we'll create a type guard to check if the weather for the provided city was received or not. For now, let's take a breather and recap what we did so far while developing the weather app:

1. Applied for the APPID at [openweathermap.org](http://openweathermap.org)
2. Generated a new app and replaced the JSX with a simple `<form>`

3. Declared the state `city` using the hook `useState()`
4. Added the function `handleChange()` that updated `city` on each change in the input field
5. Added the hook `useEffect()` that would be invoked only once on the app startup
6. Ensured that `useEffect()` invokes the function `getWeather()` that uses the `fetch()` API to retrieve the weather in London
7. Declared the state `weather` to store the retrieved temperature and humidity
8. Added the event handler `handleSubmit()` to invoke `getWeather()` after the user entered the city name and clicked on the button Get Weather
9. Modified the function `getWeather()` to save the retrieved weather in the state `weather`
10. Displayed the retrieved temperature on the web page under the form

This is all good, but you shouldn't program all the app logic in one `App` component, and in the next section, we'll create a separate component to be responsible for displaying the weather data.

### 13.4.3 Using props

Any React app is a tree of components, and you need to decide which are going to be the *container* components, and which the *presentation* ones. A container (a.k.a. smart) component contains the application logic, communicates with external data providers and passes the data to its child component(s). Typically, container components are stateful and have little or no markup.

A presentation (a.k.a. dumb) component just receives data from its parent and displays the data. A typical presentation component is stateless and has lots of markup. A presentation component gets the data to display is via its *props*.

**TIP**

In chapter 14 in section 14.4, we'll review the code of the UI components of the React version of the blockchain app. There, you'll see one container component and three presentation ones.

**TIP**

If you use a library for managing the state of entire app (e.g. Redux), only the container components would be communicating with such a library.

In our weather app, `App` is a container component that knows how to receive the weather data from the external server. So far our `App` component also displayed the received temperature as seen in figure 13.11, but we should delegate the weather rendering functionality to a separate presentation component e.g. `WeatherInfo`.

The `App` component (the parent) would contain the `WeatherInfo` component (the child), and the parent would need to pass the received weather data to the child. Passing data to a React component works similarly to passing data to HTML elements.

We'll start getting familiar with the props role by using JSX elements as an example. Any JSX

element can be rendered differently depending on the data it gets. For example, the JSX of a red disabled button can look as follows:

```
<button className="red" disabled />
```

This code instructs React to create a `button` element and pass to it certain values via the attributes `className` and `disabled`. React will start by transforming the above JSX into the invocation of `createElement()`:

```
React.createElement("button", {
  className: "red",
  disabled: true
});
```

Then, the above code will produce the JavaScript object that will look like this:

```
{
  type: 'button',
  props: { className: "red", disabled: true }
}
```

As you see, `props` contains the data being passed to a React element. React uses `props` for data exchange between the parent and child components (the above button was a part of some parent element too, right?).

Say you created a custom component `Order` and added it to the parent's JSX. You can pass the data to it via `props` as well. For example, an `Order` component may need to receive the values of such `props` as `operation`, `product`, and `price`:

```
<Order operation="buy" product="Bicycle" price={187.50} />
```

Similarly, we'll add the `WeatherInfo` component to the `App` component's JSX passing the received weather data. Also, we promised to add a user-defined type guard to ensure that the `WeatherInfo` component won't render anything if the weather for the city is not available. Listing 13.17 shows the code fragment from the `App` component that defines and uses the type guard called `has`.

### Listing 13.17 Adding the type guard has

```
const has = (value: any): value is boolean => !!value; ①

...
return (
  <> ②
  <form onSubmit={handleSubmit}>
    <input type="text" placeholder="Enter city"
      onInput={handleChange} />
    <button type="submit">Get Weather</button>
  </form>
  {has(weather) ? ( ③
    <WeatherInfo weather={weather} /> ④
  ) : (
    <h2>No weather available</h2> ⑤
  )}
  </> ⑥
);
```

- ① Declaring the `has` type guard
- ② An empty JSX tag can be used as a container
- ③ Applying the type guard `has`
- ④ Pass the weather to `WeatherInfo` and render it
- ⑤ Render the text message instead of `WeatherInfo`
- ⑥ Closing the empty JSX tag

As you see, the `App` component hosts the form and the `WeatherInfo` component that should do the rendering of the weather data. All JSX tags has to be wrapped into a single container tag. Earlier, we'd use `<div>` as a parent tag. In listing 13.17, we use an empty tag instead, which is a shortcut for a special tag container `<React.Fragment>` that doesn't add an extra node to the DOM.

In section 2.3 in chapter 2, we introduced the TypeScript user-defined type guards. In listing 13.17 we declared the type guard `has` as a function whose return type is a type predicate:

```
const has = (value: any): value is boolean => !!value;
```

It takes a value of `any` type and applies to it the JavaScript double bang operator to check if the provided value is truthy. Now, the expression `has(weather)` will check if the weather is received, and we do this in the JSX portion in listing 13.17. The received weather is given to the `WeatherInfo` component via its `weather` props.

```
<WeatherInfo weather={weather} />
```

Now let's discuss how we created the `WeatherInfo` component that can receive and render the weather data. In VS Code, we created a new file `weather-info.tsx` and started declaring the

`WeatherInfo` component there. Similarly to the `App` functional component, we used the arrow function notation for `WeatherInfo`, but this time our component would accept an explicit argument `props`. Hover the mouse over FC, and you'll see its declaration as shown in figure 13.12.

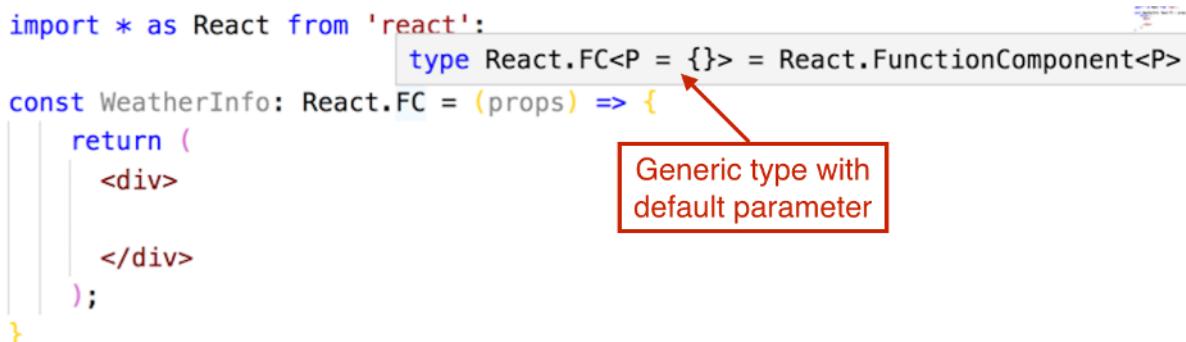


Figure 13.12 A generic type with default parameter

`React.FC` is a generic type that takes `P` (for `props`) as a type parameter. Then why the TypeScript compiler didn't complain when we declared the `App` component without using the generic notation and a concrete type? The part `P = {}` did the trick. This is how you can declare a generic type with a default value (see the sidebar in section 4.2.2 in chapter 4). Our `App` component didn't use `props`, and by default, React assumed that `props` was an empty object.

Each component has a property called `props`, which can be an arbitrary JavaScript object with the properties specific to your app. In JavaScript, you can't specify the type of the `props` content, but TypeScript generics allow you to let the component know that it's going to get the `props` containing the `Weather` as seen in listing 13.18.

### Listing 13.18 WeatherInfo.tsx

```

import * as React from 'react';
import {Weather} from './weather';

const WeatherInfo: React.FC<{weather: Weather}> = ❶
  ({ weather }) => { ❷
    const {city, humidity, pressure, temp, temp_max, temp_min} = weather; ❸
    return (
      <div>
        <h2>City: {city}</h2> ❹
        <h2>Temperature: {temp}</h2> ❺
        <h2>Max temperature: {temp_max}</h2> ❺
        <h2>Min temperature: {temp_min}</h2> ❺
        <h2>Humidity: {humidity}</h2> ❺
        <h2>Pressure: {pressure}</h2> ❺
      </div>
    );
}

export default WeatherInfo;

```

- ❶ Our component is a generic function with the argument of type `Weather`

- ② The fat arrow function has one argument - the weather object
- ③ Destructuring the weather object
- ④ Rendering the city
- ⑤ Rendering the weather data

The `WeatherInfo` component is a generic function with one parameter `<P>` as seen in figure 13.12, and we used the type `{weather: Weather}` as its argument. For rendering the data, we could access each property of the `weather` object by using the dot notation (e.g. `weather.city`), but the fastest way to extract the values of these properties into local variables is destructuring (see Appendix A in section A.8) as shown in the following line:

```
const {city, humidity, pressure, temp, temp_max, temp_min} = weather;
```

Now all these variables can be used in the JSX returned by this component, e.g. `<h2>City: {city}</h2>`.

#### SIDE BAR    Passing a markup to the child component

`props` can be used not only for passing data to a child component, but also for passing JSX fragments. If you place any JSX fragment between the opening and closing tags of the component as shown in listing 13.19, this content will be stored in the property `props.children` and you can render it as needed.

#### Listing 13.19 Passing JSX via props

```
<WeatherInfo weather = {weather} >
  <strong>Hello from the parent!</strong> ①
</WeatherInfo>
```

- ① Passing this markup to `WeatherInfo`

Here, the `App` component passes the HTML element `<strong>Hello from the parent!</strong>` to the child component `WeatherInfo`, and it could have passed any other React component the same way. Accordingly, the `WeatherInfo` component has to declare its interest in receiving not only in the `Weather` object, but also the content of `props.children` from `React.FC` as follows:

```
const WeatherInfo: React.FC<{weather: Weather} >= ({ weather, children }) => ...
```

This line "tells" to React, "We'll give your React.FC component the object with the property `weather`, but this time, we'd like to use the property `children` as well". Now the local variable `children` contains the markup provided by the parent, which can be rendered along with the weather data as shown in listing 13.20.

#### Listing 13.20 Using `props.children`

```
return (
  <div>
    {children} ①
    <h2>City: {city}</h2>
    <h2>Temperature: {temp}</h2>
    { /* The rest of the JSX is omitted */ }
  </div>
);
```

- ① Rendering the markup received from parent

After embedding the `{children}` expression, the `WeatherInfo` component will be rendered as shown in figure 13.13.

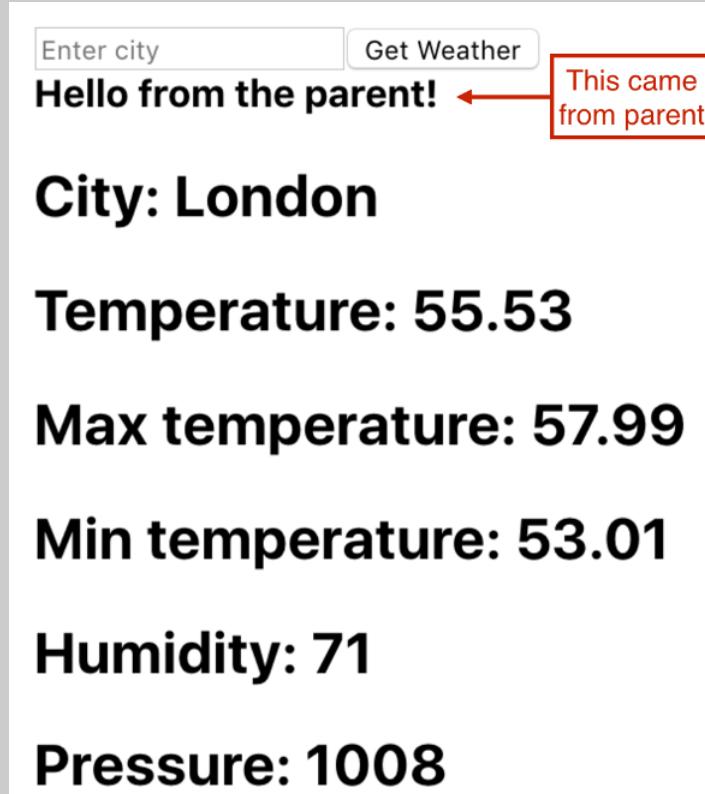


Figure 13.13 Adding the content received via `props.children`

To visualize the props and state of each component, install the Chrome extension called React Developer Tools and run a React app with the Chrome Dev Tools open. You'll see an extra tab

React that shows rendered elements on the left and props and state (if any) of every component on the right as seen in figure 13.14. Our `WeatherInfo` component is stateless, otherwise you'd see the content of the state as well.

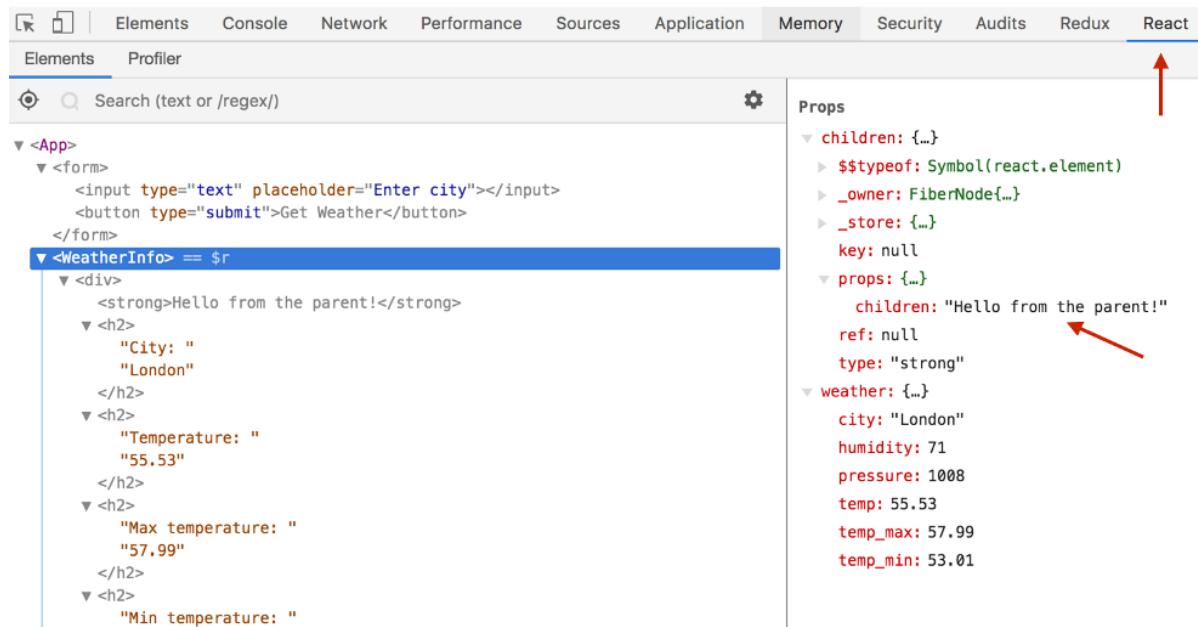


Figure 13.14 React Dev Tools

It's great that a parent component passes data to its child via props, but what about sending data in the opposite direction?

#### 13.4.4 How a child component can pass data to its parent

There are scenarios when a child component needs to send some data to its parent, and it's also done via props. Imagine a child component that's connected to a stock exchange, and every second it receives the latest stock prices. If its parent is responsible for handling the data, the child need to be able to pass the data to such a parent.

For simplicity reasons, we'll just illustrate how the `WeatherInfo` component can send some text to its parent. This will require some coding on both parent and the child's sides. Let's start with the parent.

If a parent expects to get some data from the child, it has to declare a function that knows how to handle this data. Then, in the parent's JSX where it embeds the child component you add the attribute to the child's tag containing the reference to this function. In the weather app, the `App` component is the parent, and we want it to be able to receive text messages from its child `WeatherInfo`. Listing 13.21 shows the additions to the code of the `App` component developed in the previous section.

## Listing 13.21 Adding code to the App component for receiving data

```

const [msgFromChild, setMsgFromChild] = useState(''); ①

const getMsgFromChild = (msg: string) => setMsgFromChild(msg); ②

return (
<>
/* The rest of the JSX is omitted */

{msgFromChild} ③
{has(weather) ? (
<WeatherInfo weather = {weather} parentChannel = {getMsgFromChild}> ④
</>
)

```

- ① Declaring the state variable msgFromChild to store the child's message
- ② Declaring the function to store the child's message in the state msgFromChild
- ③ Displaying the child's message
- ④ Adding the parentChannel property to the child

Here, we added the state variable `msgFromChild` to store the message received from the child. The function `getMsgFromChild()` gets the message and updates the state using the `setMsgFromChild()` function, which results in re-rendering `{msgFromChild}` in the UI of the `App` component.

Finally, we need to give the child a reference for invoking the message handler. We decided to call this reference `parentChannel` and passed it to `WeatherInfo` as follows:

```
parentChannel = {getMsgFromChild}
```

`parentChannel` is an arbitrary name that the child will use as a reference to invoke the message handler `getMsgFromChild()`. The modifications of the parent's code are complete, and now we'll take care of the child component `WeatherInfo`. Listing 13.22 shows additions to the child's code.

## Listing 13.22 Adding the code to WeatherInfo for sending data to parent

```

const WeatherInfo: React.FC<{weather: Weather, parentChannel: (msg: string) => void}> = ①
({weather, children, parentChannel}) => { ②

/* The rest of the WebInfo code is omitted */

return (
<div>
  <button
    onClick =={() => parentChannel ("Hello from child!")}> ③
    Say hello to parent
  </button>

/* The rest of the JSX code is omitted */
</div>
);

```

- ① Adding parentChannel to the type of the generic function
- ② Adding parentChannel to the function argument
- ③ Invoking the parent's function on the button click using parentChannel as a reference

Since the parent gives to the child the reference `parentChannel` to the message handler-function, we need to modify the type parameter of the component to include this reference:

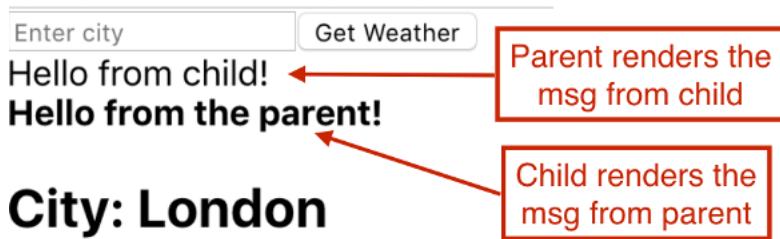
```
<{weather: Weather, parentChannel: (msg: string) => void}>
```

The type of the `parentChannel` is a function that takes a string parameter and returns no value. Our `WeatherInfo` component will process props, which is now the object with three properties: `weather`, `children`, and `parentChannel`.

The `onClick` button handler returns a function that would invoke `parentChannel()` passing the message text to it:

```
() => parentChannel ("Hello from child!")
```

Run this version of the weather app, and if you click on the button at bottom of the `WeatherInfo` component, the parent receives the message and renders it as shown in figure 13.15.



**Temperature: 54.48**

**Max temperature: 57**

**Min temperature: 52**

**Humidity: 76**

**Pressure: 1007**



**Figure 13.15 Message from child**

**NOTE**

The code in listing 13.22 can be optimized, and this is going to be your homework. This code works, but it recreates the function `() parentChannel("Hello from child!")` on each re-render of the UI. To avoid this from happening, read about `useCallback()` at [reactjs.org/docs/hooks-reference.html#usecallback](https://reactjs.org/docs/hooks-reference.html#usecallback) and wrap the function `parentChannel()` in this hook.

Now that we introduced both `state` and `props`, we'd like to stress that they serve different purposes:

- A `state` stores private component's data, while `props` is used for passing the data to child components or back.
- A component may use the received `props` to initialize its `state` if any.
- Direct modifications of the `state` properties' values is prohibited; use the function returned by the hook `useState()` in functional components or the method `setState()` in the class-based ones to ensure that the changes are reflected on the UI.
- `props` are immutable, and a component can't modify the original data received via `props`.

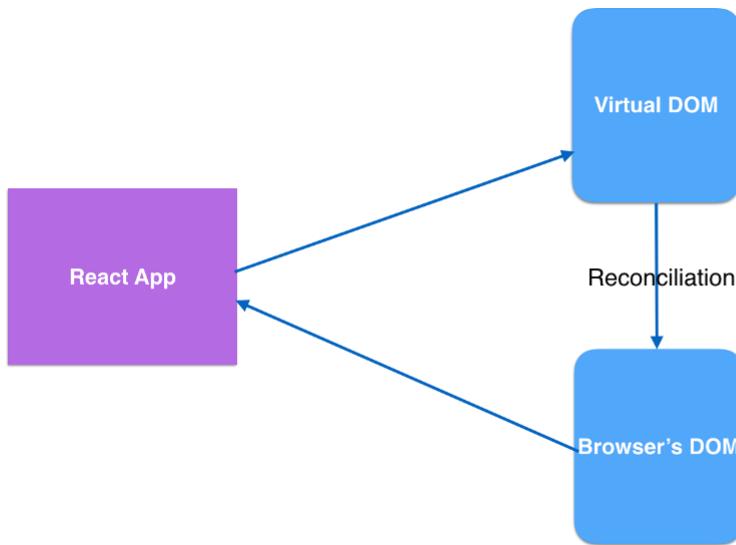
**NOTE**

State and Props serve different purposes, but UI in React is a function of state and props.

In our brief overview of the React framework, we discussed components, state, props, but even the shortest overview of React must include one more topic: Virtual DOM.

## 13.5 What's Virtual DOM

A unique to React feature is *Virtual DOM*, a layer between the component and the actual browser's DOM. Each component consists of UI elements, and Virtual DOM optimizes the process of rendering these elements to the browser's DOM as seen in figure 13.16.



**Figure 13.16 React's virtual DOM**

When you start the app, React creates a tree of UI components in its own Virtual DOM, and renders this tree to the browser's DOM. As the user works with the React app, the browser's DOM triggers events to the app. If the event is handled by JavaScript, and if the handler code updates the component's state, then React re-creates the Virtual DOM, diffs its new version with the previous one and syncs differences with the browser's DOM. Applying this *diffing* algorithm is called reconciliation (see more at [reactjs.org/docs/reconciliation.html](https://reactjs.org/docs/reconciliation.html)).

**NOTE** The term "Virtual DOM" is a bit of a misnomer, because the library React Native uses same principles for rendering the iOS and Android UI that don't have DOM.

In general, rendering of UI from the browser's DOM is a slow operation, and to make it faster, React doesn't re-render all elements of the browser's DOM every time when an element changes. You may not see much of a difference in the speed of rendering in a web page that has a handful of HTML elements, but in a page that has thousands elements, the difference in rendering performed by React vs regular JavaScript will be noticeable.

If you can't imagine a web page with thousands of HTML elements, think of a financial portal showing the latest trading activities in a tabular form. If such a table contains 300 rows and 40 columns, it has 12000 cells, and each cell consists of several HTML elements, and a portal may need to render several of such tables.

The Virtual DOM spares the developers from working with the browser DOM API like jQuery does - just update the component's state and React will update the corresponding DOM elements in the most efficient manner. This is basically all that React library does, which means that you'd need other libraries for implementing such functionality as making HTTP requests, routing, or working with forms.

This concludes our brief introduction to developing web apps using the React.js library and TypeScript. In chapter 14, we'll review the code of the new version of the blockchain client written in React.js.

## 13.6 Summary

In this chapter you learned:

- How to generate your first React app that uses TypeScript
- The two types of React components: class-based and functional
- What's the role of the component state and props
- How a parent and child component can exchange data
- What's the role of React's Virtual DOM

# 14

## *Developing a blockchain client in React.js*

### **This chapter covers:**

- The code review of the blockchain web client written using React.js
- How the React.js web client can communicate with the WebSocket server
- How to run a React app that works with two servers in dev mode
- How to split the UI of the blockchain client into components and arrange their communications

In the previous chapter, you've learned the basics of React, and now we'll review a new version of the blockchain app where the client portion is written in React. The source code of the web client is located in the directory `blockchain/client`, and the messaging server located in the directory `blockchain/server`.

The code of the server remains the same as in chapters 10 and in 12, and the functionality of this version of the blockchain app is the same as well, but the UI portion of the app was completely re-written in React.

In this chapter, we won't be reviewing the blockchain functionality, because it was already covered in the previous chapters, but we will review the code that's specific to the React.js library. You may want to re-read chapter 10 to refresh in memory the functionality of the blockchain client and messaging server.

First, we'll show how to start the messaging server and the React client of the blockchain app. Then, we'll introduce the code of the UI components highlighting the difference between smart and presentation components. You'll also see multiple code samples of arranging the inter-component communications via props.

## 14.1 Starting the client and the messaging server

To start the server, open the Terminal window in the blockchain/server directory, run `npm install` to install the server's dependencies and then run the `npm start` command. You'll see the message *Listening on localhost:3000*; keep the server running.

To start the React client, open another Terminal window in the directory blockchain/client, run `npm install` to install React and its dependencies and then run the `npm start` command.

The blockchain client will start on port 3001, and after a short delay, the genesis block will be generated. Figure 14.1 shows the screenshot of the React client and the names of the files of the UI portion of the app, where the file `App.tsx` contains the code of the root component called `App`.

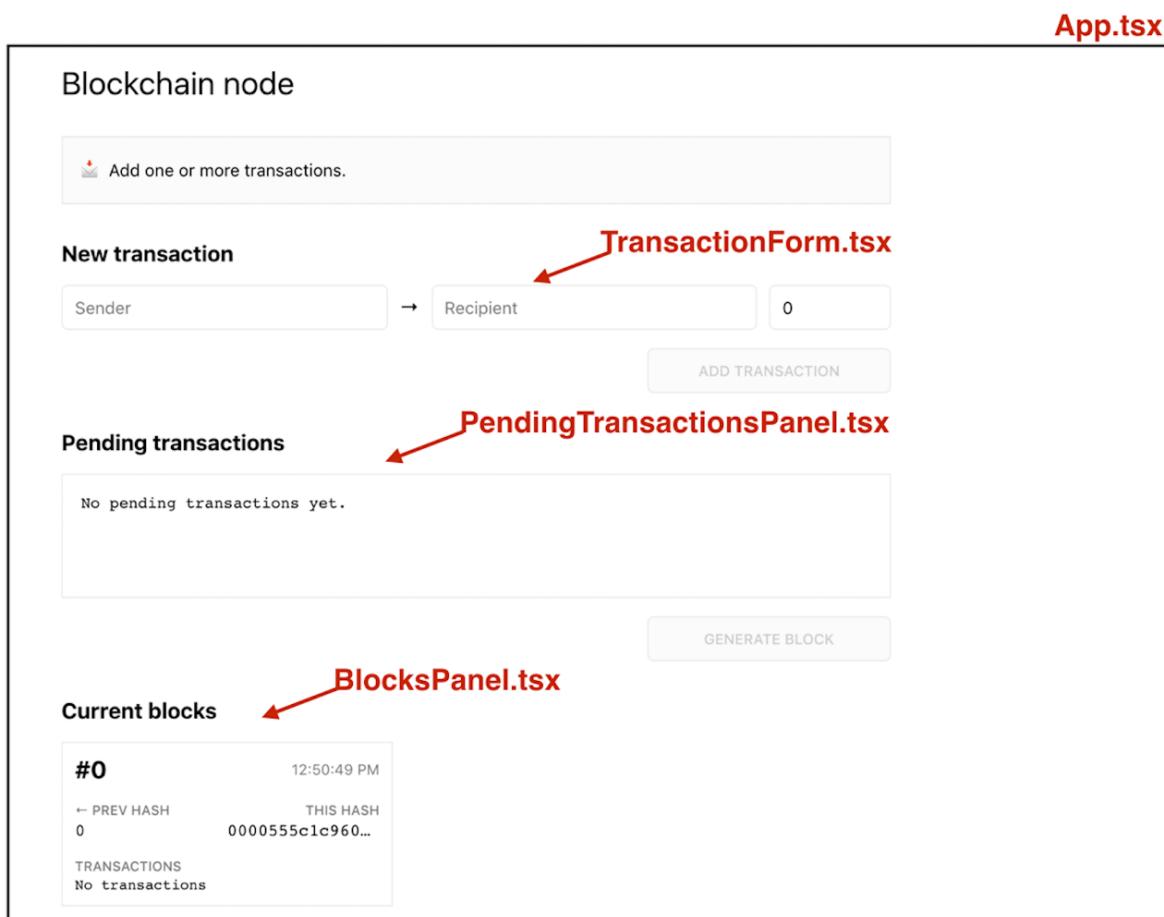


Figure 14.1 The blockchain client is launched

The `start` command is defined in `package.json` as an alias to the `react-scripts start` command, and it would engage Webpack to build the bundles and use the `webpack-dev-server` to launch the app. We went through a similar workflow while starting the Angular blockchain client in chapter 12, but apps generated by the `create-react-app` CLI are pre-configured to start the

webpack-dev-server on the port 3000, so we had to deal with the ports conflict as the port 3000 was already taken by our messaging server. We had to find a way to configure a different port for the client, and we'll explain you how we did it shortly.

Projects that were generated by the `create-react-app` tool are preconfigured to read custom environment variables from the files `.env.development` (shown in listing 14.1) in the dev mode or `.env.production` in prod.

**TIP**

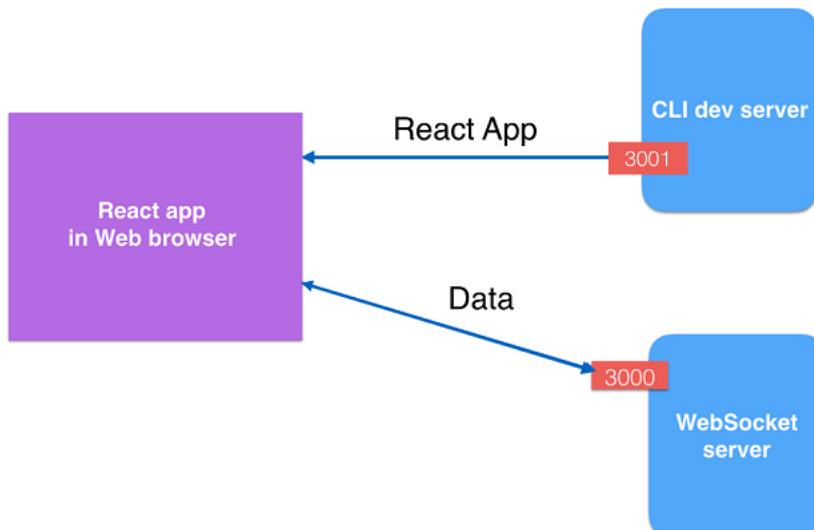
To prepare the optimized production build, run the command `npm run build` and you'll find `index.html` and the app bundles in the directory `build`. This command would use the environment variables from the file `.env.production`.

### **Listing 14.1 `env.development`**

```
PORT=3001
REACT_APP_WS_PROXY_HOSTNAME=localhost:3000
```

First of all, we declared the port 3001 to be used when webpack-dev-server starts the app, and this will resolve the conflict with the server that runs on port 3000. Then, we declared a custom environment variable `REACT_APP_WS_PROXY_HOSTNAME` that can be used to proxy the client's requests to the server running on `localhost:3000`. This variable will be used in the script `websocket-controller.ts`. You can read more about adding custom environment variables in the `create-react-app` projects at [facebook.github.io/create-react-app/docs/adding-custom-environment-variables](https://facebook.github.io/create-react-app/docs/adding-custom-environment-variables).

Figure 14.3 shows how our blockchain client runs in dev mode. The CLI dev server (currently, it's webpack-dev-server) serves the React app on port 3001, which then connects to another server that runs on port 3000.



**Figure 14.2 One app, two servers**

**NOTE**

The `create-react-app` CLI allows you to add the property `proxy` to `package.json` and specify a URL to be used if the dev server doesn't find the requested resource, e.g. `"proxy": "http://localhost:3000"`. But this property doesn't work with WebSocket protocol, so we had to use the custom environment variable for specifying the URL of our messaging server.

Re-read the first two pages of chapter 12 and compare the figures 14.3 and 12.2, and you'll see that both Angular and React projects have similar file structures and launching procedures. But since our React app uses Babel in its workflow, the configuration of the supported browsers is done differently compared to `tsc`. Here, we list specific browsers' versions that our app should support.

In section 6.4 in chapter 6, we showed you how Babel can use presets to specify which versions of the browsers the app have to support. `create-react-app` adds the default `browserslist` section to `package.json`. Open this file and you'll find the default browsers configuration shown in listing 14.2.

### **Listing 14.2 A fragment from `package.json`**

```
"browserslist": {
  "production": [    ①
    ">0.2%",
    "not dead",
    "not op_mini all"
  ],
  "development": [    ②
    "last 1 chrome version",
    "last 1 firefox version",
    "last 1 safari version"
  ]
}
```

- ① Which browsers have to be supported in prod builds
- ② Which browsers have to be supported in dev builds

**TIP**

Even though we don't use `tsc` to transpile the code and emit JavaScript, we still specified the option `"target": "es5"` in `tsconfig.json` to ensure that TypeScript doesn't complain about the TypeScript syntax from our blockchain app. Remove the `target` option or change its value to `es3`, and the getter `get url()` from listing 14.5 will be underlined by a red squiggly line with the error message "Accessors are only available when targeting ECMAScript 5 and higher."

In chapter 6 in section 6.4, we explained the use of `browserlist` (see [browserlist](#)) that allows you to configure the emitted JavaScript to run in specific browsers. The `create-react-app` supports

it as well, and the setting from package.json are being used when you either create a production build by running the `npm build` script or create dev bundles by running the `start` script. Copy-paste each entry from your app's `browserlist` to the web page [browserlist](#), and it'll show you which browsers are covered by this entry.

As in previous versions of the blockchain app, the code is split into the UI part and the supporting scripts that implement the blockchain's algorithms. In the React's version, the UI components are located in the directory `components`, and the supporting scripts are in `lib`.

## 14.2 What changed in the `lib` directory

As a reminder, the `lib` directory has the code that generates new blocks, requests the longest chain, notifies the other nodes about the newly generated blocks and invites other members of the blockchain to start generating new blocks for the specified transactions. These processes were described in chapter 10 in sections 10.1 and 10.2.

We slightly modified only the file `websocket-controller.ts`, which contains the script communicating with the WebSocket server. In chapter 10, we didn't use JavaScript frameworks and simply instantiated the class `websocketController` using the `new` operator. We'd pass `messageCallback` to the constructor to handle messages coming from the server.

In chapter 12, we used Angular's Dependency Injection, and this framework would instantiate and inject the object `websocketService` into the `AppComponent`. We could rely on Angular to instantiate the service first and only after create the `App` component.

### TIP

If you have Angular background and are accustomed to creating singleton services that can be injected into component, learn about React's Context (see [reactjs.org/docs/context.html](https://reactjs.org/docs/context.html)), which can be used as a common storage for data that should be passed from one component to another. In other words, `props` is not the only way to pass data between components.

In the React version of the blockchain app, we'll manually instantiate `websocketController`, and the script `App.tsx` starts as shown in listing 14.3.

### Listing 14.3 Instantiating classes before the component

```
const server = new WebsocketController(); ①
const node = new BlockchainNode(); ②

const App: React.FC = () => { ③
  // The code of the App component is omitted
  // We'll review it later in this chapter
}
```

- ① First, instantiate `WebsocketController`

- ② Second, instantiate the BlockchainNode
- ③ Third, declare the root UI component

To ensure that the `WebsocketController` and `BlockchainNode` are global objects, we start the script starts with instantiating them. But `WebsocketController` needs a callback method from the `App` component to handle server's messages changing the component's state. The problem is that the `App` component is not instantiated yet, so we can't provide such callback to the component's constructor.

That's why we made the method `connect()` in the the `WebsocketController`, and this method takes the callback as its parameter. The complete code of the method `connect()` is shown in listing 14.4.

#### **Listing 14.4 The method connect() from WebsocketController**

```
connect(messagesCallback: (messages: Message) => void): ①
    Promise<WebSocket> {
  this.messagesCallback = messagesCallback;
  return this.websocket = new Promise((resolve, reject) => { ②
    const ws = new WebSocket(this.url);
    ws.addEventListener('open', () => resolve(ws));
    ws.addEventListener('error', err => reject(err));
    ws.addEventListener('message', this.onMessageReceived);
  });
}
```

- ① Passing the callback to the controller
- ② Wrapping the socket creation into a Promise

In the next section, we'll review the `App` component, which will invoke `connect()` (see listing 14.10) passing the proper callback. What's the value of `this.url` that supposed to point at the WebSocket server? In section 14.1, we stated that the server's domain name and port will be taken from the environment variables, and listing 14.4 shows the code of the getter `url` in `WebsocketController`.

#### **Listing 14.5 The getter url**

```
private get url(): string {
  const protocol = window.location.protocol === 'https:' ? 'wss' : 'ws';
  const hostname = process.env.REACT_APP_WS_PROXY_HOSTNAME ①
    || window.location.host; ②
  return `${protocol}://${hostname}`;
```

- ① Getting the host name and port from the environment variable
- ② If the host name is not found in `process.env`, use the current app host

The variable `REACT_APP_WS_PROXY_HOSTNAME` was defined in the file `env.development` shown

in listing 14.1. The property `env` of the Node.js global variable `process` is the place where your code can access all available environment variables. But you can say that our app runs in the browser and not in the Node.js runtime! This is correct, but during the bundling, Webpack reads the values of the variables available in `process.env` and inlines them into the bundles of the web app.

**TIP**

The values of the custom environment variables are inlined into the bundles. Run the command `npm run build` and open the main bundle from the build/static directory. Then search for the value of one of the variables from the file `.env.production`, e.g. `localhost:3002`.

After adding the method `connect()`, we decided to add the method `disconnect()` that would close the socket connection:

```
disconnect() {
  this.websocket.then(ws => ws.close());
}
```

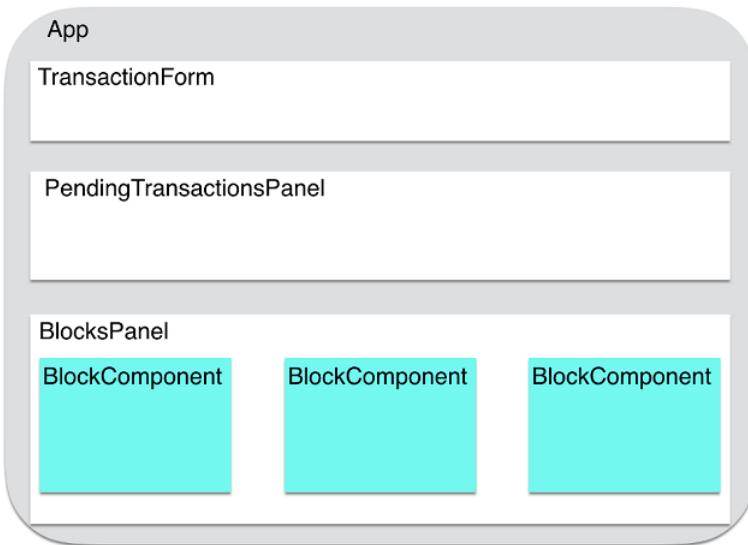
In our version of the blockchain client, the WebSocket connection is established by the `App` component, so when the `App` component is destroyed, the entire app is destroyed including the socket connection. But this may not always be the case, so it's nice to have a separate method that would close the socket, if other components would need to establish and destroy the connection when need be.

### **14.3 The smart component App**

The UI of this version of blockchain consists of five React components located in the `.tsx` files in the `components` directory:

- `App`
- `BlocksPanel`
- `BlockComponent`
- `PendingTransactionsPanel`
- `TransactionForm`

The file `App.tsx` contains the root `App` component, and the other `.tsx` files contain the code of its children `TransactionForm`, `PendingTransactionsPanel`, and `BlocksPanel`, which is also a parent of one or more `BlockComponent` instances as shown in figure 14.4.



**Figure 14.3 Parents and child components**

In section 13.4.3 in chapter 13, we introduced the concept of the container (smart) and presentation (dumb) components, and `App` is a smart component that contains a reference to the blockchain node instance and all related algorithms. The `App` component also performs all communications with the messaging server.

A typical presentation component either displays the data or, based on the user's actions, sends its data to other components. Presentation components don't implement complex application logic. For example, if a `PendingTransactionsPanel` component needs to initiate the new block creation, it simply invokes the proper callback on the `App` component that will start the block generation process. In our blockchain client, the presentation components are `TransactionForm`, `PendingTransactionPanel`, and `BlockPanel`.

The smart `App` component is implemented in the file `App.tsx`, which also creates instances of `BlockchainNode` and `WebsocketController`. To give you a big picture of how the `App` component communicates with its children, listing 14.6 just shows the JSX part of the `App` component.

## Listing 14.6 The App component's JSX

```
const App: React.FC = () => {
    // Other code is omitted for brevity

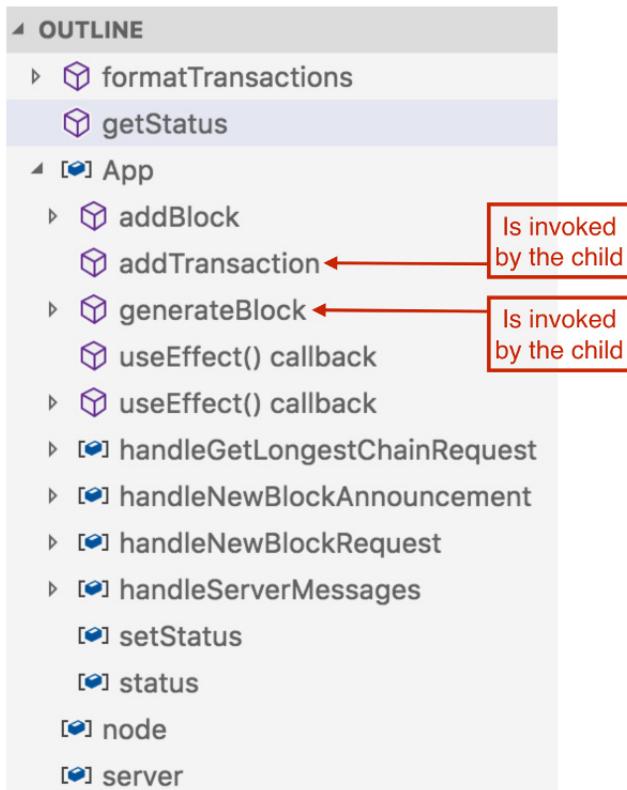
    return (
        <main>
            <h1>Blockchain node</h1>
            <aside><p>{status}</p></aside>
            <section>
                <TransactionForm
                    onAddTransaction={addTransaction} ①
                    disabled={node.isMining || node.chainIsEmpty}
                />
            </section>
            <section>
                <PendingTransactionsPanel ②
                    formattedTransactions={formatTransactions(node.pendingTransactions)}
                    onGenerateBlock={generateBlock} ③
                    disabled={node.isMining || node.noPendingTransactions}
                />
            </section>
            <section>
                <BlocksPanel blocks={node.chain} /> ④
            </section>
        </main>
    );
}
```

- ① The first child component
- ② This child can invoke addTransaction() on App component
- ③ The second child
- ④ This child can invoke generateBlock() on App component
- ⑤ The third child

### 14.3.1 The user wants to add a transaction

The JSX of the `App` component includes child components, which can invoke callback methods on `App`. We discussed how a React child component can interact with its parent in section 13.4.4 in chapter 13. In listing 14.6, you can see that the `App` component passes the callback `onAddTransaction` to the `TransactionForm` via props.

Accordingly, when the user clicks on the button ADD TRANSACTION on the `TransactionForm` component, it'll invoke its method `onAddTransaction()`, which will result in the invocation of the method `addTransaction()` on the `App` component. This is what the word "implicitly" means in figure 14.5, which shows the screen shot from VS Code with the outline of the `App` component.



**Figure 14.4. The App component has methods that will be invoked by its children**

To complete discussing the add transaction workflow, let's see what triggers the UI change when a new transaction is being added. In React, to update the UI, we change the state of the component, and we do this by invoking the function returned by the `useState()` hook. In the `App` component, the name of this function is `setStatus()`.

#### Listing 14.7. The App's code that adds a transaction

```

const node = new BlockchainNode();                                ①

const App: React.FC = () => {
  const [status, setStatus] = useState<string>('');            ②

  function addTransaction(transaction: Transaction): void {
    node.addTransaction(transaction);                            ③
    setStatus(getStatus(node));                                ④
  }

  // the rest of the code is omitted for brevity
}

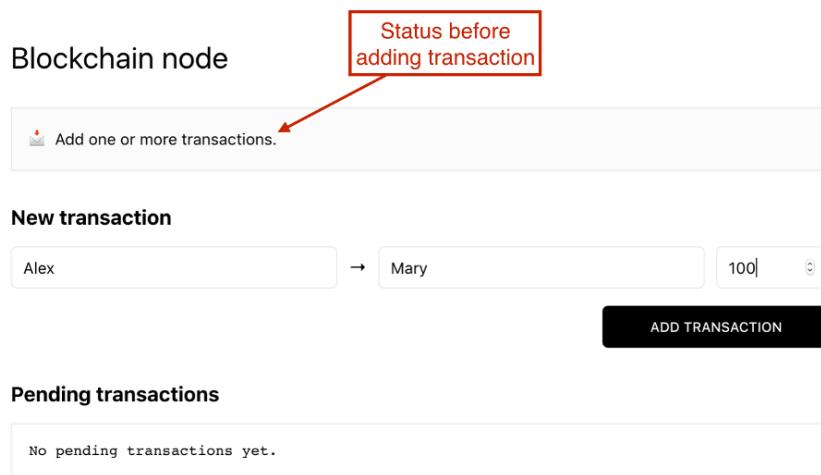
function getStatus(node: BlockchainNode): string {             ⑤
  return node.isEmpty ? 'Initializing the blockchain...' :
    node.isMining ? 'Mining a new block...' :
    node.noPendingTransactions ? 'Add one or more transactions.' :
      'Ready to mine a new block.';
}
  
```

- ① The node instance
- ② Declaring the status state of the App component
- ③ Adding the transaction received from the child component
- ④ Updating the state
- ⑤ This function is implemented outside of the App component

But what can we use in the `App` component to trigger the invocation of `setStatus()`? We reviewed the blockchain node implementation and identified operations that change the internal state of the node. We added helper properties `chainIsEmpty`, `isMining` and `noPendingTransactions` for probing node's internal state. After each operation that may change the internal node's state we verify the UI state against node's internal state, and React applies required changes if need be. If any of these values changes, we need to update the UI of the `App` component.

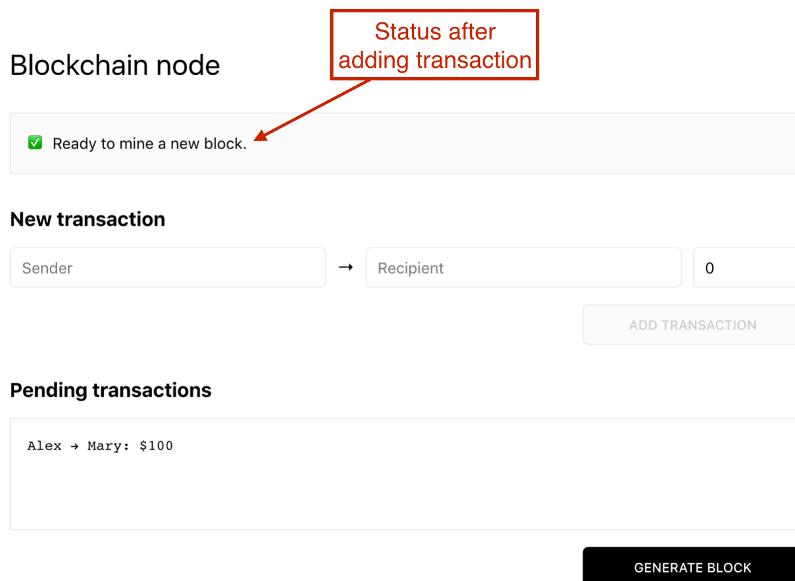
The function `getStatus()` returns the text describing the current blockchain node's status, and the `App` component will render the corresponding messages (see listing 14.7). Initially, the status value is "Initializing the blockchain..." If the user adds a new transaction, `getStatus()` will return "Add one or more transaction", and the line `setStatus(getStatus(node));` will change the component's state resulting in re-rendering of the UI.

Figure 14.6. shows the app status (or state) "Add one or more transaction".



**Figure 14.5 The user is ready to add a transaction**

Figure 14.7 shows the screen shot taken after the user clicked on the button ADD TRANSACTION. The app component's state is "Ready to mine new block", and the button GENERATE BLOCK became enabled.



**Figure 14.6 The user is ready to mine a new block**

While testing this app, we noticed an issue: If the user would add more than one transaction, it would still show only the first one in the Pending transactions field. But after clicking the GENERATE BLOCK button, the new block would include all pending transaction. For some reason, React didn't re-render the UI for each new transaction except the first one.

The problem was that after the adding the first transaction, the state of the `App` component would contain "Ready to mine new block", and this value wouldn't change after adding other transactions. Accordingly, the line `setStatus(getStatus(node));` wouldn't change the component's state and React didn't see the reason for UI re-rendering!

This issue was easy to fix. We just slightly modified the function `getStatus()` by adding the transaction counter to the status line. Now instead of keeping the status "Ready to mine new block" it would include a changing part:

```
Ready to mine a new block (transactions: ${node.pendingTransactions.length}).`;
```

Now every invocation of `addTransaction()` would change the component state.

**TIP**

Actually, making the node immutable would be a better solution to this issue. For an immutable object, whenever its state changes, the new instance (and the reference) would be created, and the hack with the transaction counter wouldn't be necessary.

### 14.3.2 The user wants to generate a new block

Let's take another look at listing 14.6. The `App` component passes the callback `onGenerateBlock()` to `PendingTransactionPanel` using props. When the user will click on the button GENERATE BLOCK, the component `PendingTransactionPanel` invokes `onGenerateBlock()`, which in turn invokes the method `generateBlock()` on the `App` component. This method is shown in listing 14.8.

#### Listing 14.8 The method `generateBlock()`

```
async function generateBlock() {
    server.requestNewBlock(node.pendingTransactions); ①
    const miningProcessIsDone = node.mineBlockWith(node.pendingTransactions); ②

    setStatus(getStatus(node)); ③

    const newBlock = await miningProcessIsDone; ④
    addBlock(newBlock);
}
```

- ① Invite other nodes to start generating a new block for the pending transactions
- ② Declare an expression for block mining
- ③ Change the component's state
- ④ Start the block mining the block and wait for completion
- ⑤ Add the new block to the blockchain

#### NOTE

By providing parent's method references to child component, we control what the child can access in the parent. In this app, neither `TransactionForm` nor `PendingTransactionPanel` component have access to the `BlockchainNode` and `WebSocketController` objects. These children are strictly presentational components, and they can display data or notify the parent about some events.

### 14.3.3 Explaining the `useEffect()` hooks

In the code of the `App` component, you can find two `useEffect()` hooks. As a reminder, `useEffect()` hooks can be automatically invoked when a specified variable have changed. The `App` component has two of such hooks. Listing 14.9 shows the first `useEffect()` that works only once on the app startup.

#### Listing 14.9 The first `useEffect`

```
useEffect(() => {
    setStatus(getStatus(node));
}, []);
```

The goal of this effect is to initialize the component state with the message "Initializing the blockchain..." If you would comment out this hook, the app would still work, but there wouldn't be any status message on the top when the app starts.

Listing 14.10 shows the second `useEffect()`, which connects to the WebSocket server passing it the callback `handleServerMessages` that will handle the messages pushed by the server.

### **Listing 14.10. The useEffect hook that attached to handleServerMessages**

```
useEffect(() => {
  async function initializeBlockchainNode() {
    await server.connect(handleServerMessages); ①
    const blocks = await server.requestLongestChain(); ②
    if (blocks.length > 0) {
      node.initializeWith(blocks); ③
    } else {
      await node.initializeWithGenesisBlock(); ④
    }
    setStatus(getStatus(node)); ⑤
  }

  initializeBlockchainNode(); ⑥
  return () => server.disconnect(); ⑦
}, [handleServerMessages]); ⑧
```

- ① Declare the function `initializeBlockchainNode()`
- ② Connect to the WebSocket server providing the callback
- ③ Request the longest chain
- ④ The blockchain already has some blocks
- ⑤ No blocks exist yet; create the genesis block
- ⑥ Update the App state
- ⑦ Invoke the function `initializeBlockchainNode()`
- ⑧ Disconnect from WebScocket when the App component gets destroyed
- ⑨ This hook is attached to `handleServerMessages`

If a function is used only inside the effect, it's recommended to declare it inside the effect. Since `initializeBlockchainNode()` is used only by the above `useEffect()`, we declared it inside this hook.

This hook performs the initial connection to the server and initialization of the blockchain, and we want it to be invoked only once. For this, we tried to use an empty array as the second argument as the documentation prescribes. An empty array means that this effect doesn't use any values that could participate in the React data flow, so it's safe to invoke only once.

But React noticed that this hook uses a component-scoped function `handleServerMessages()`

that being a closure, could potentially capture a component's state variable that may become outdated on the next rendering, but our effect will still keep the reference to the `handleServerMessages()` that captured the old state. Because of this, React forced us to replace the empty array with `[handleServerMessages]`, but since we won't be changing state inside this callback, the above `useEffect()` will be invoked only once.

Note the `return` statement at the end of the `useEffect()` shown in listing 14.10. Returning a function from `useEffect()` is optional, but if it's there, React guarantees to call that function when the component is about to be destroyed. If we'd established the WebSocket connection in other components (not in the root one), it would be a good practice to have a return statement in the `useEffect()` to avoid memory leaks.

While wrapping the asynchronous function inside the effect from listing 14.10, initially we tried to just add the `async` keyword as follows:

```
useEffect(async () => { await ...})
```

But since any `async` function returns a `Promise`, TypeScript started complaining that "Type '`Promise<void>`' is not assignable to type '`() void | undefined`' ". The `useEffect()` didn't like a function that would return a `Promise`. That's why we changed the signature a little bit:

```
useEffect(() => {
  async function initializeBlockchainNode() {...}
  initializeBlockchainNode();
})
```

First, we declared the asynchronous function and then invoked it. This made TypeScript happy and we were able to reuse the same code as in chapters 10 and 12 inside the React's `useEffect()`.

#### 14.3.4 Memoization with the `useCallback()` hook

Now let's talk about yet another React hook `useCallback()`, which returns a *memoized* callback. Memoization is the optimization technique that stores the results of function calls and returns the cached result when the same inputs occur again. Imagine there is a function `doSomething(a,b)` that performs some long-running calculations on the supplied arguments. Let's say, the calculations take 30 seconds, and this is a pure function that always return the same result if the arguments are the same. The following code snippet should run 90 second, right?

```
let result: number;
result = doSomething(2, 3); // 30 sec
result = doSomething(10, 15); // 30 sec
result = doSomething(2, 3); // 30 sec
```

But if we were saving the results for each pair of arguments in some table, we wouldn't need to

invoke `doSomething(2, 3)` for the second time because we already have the result for this pair and just need to do a quick lookup in our table with results. This is an example where memoization could optimize the code in the above snippet so it would run for a little more than 60 seconds instead of 90.

In React components, you don't need to implement memoization for each function manually, and can use the provided hook `useCallback()`. Listing 14.11 shows the hook `useCallback()` that returns a memoized version of the function `doSomething()`.

### Listing 14.11 Wrapping a function into the `useCallback()` hook

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);      ②
  },
  [a, b],    ③
);
```

- ① The `useCallback()` hook
- ② Memoized function `doSomething()`
- ③ Dependencies of `doSomething()`

If the function `doSomething()` is a part of the React component, memoization will prevent unnecessary re-creations of this function during each UI rendering unless its dependencies `a` or `b` changed.

In our `App` component, we wrapped inside the `useCallback()` all the functions that handle messages from the WebSocket server, e.g `handleServerMessages()`. Figure 14.8 shows a screen shot of the code from `App` component with the collapsed bodies of the functions wrapped inside `useCallback()`.

```
12 const App: React.FC = () => {
13   const [status, setStatus] = useState<string>('');
14
15 ④ const handleGetLongestChainRequest = useCallback((message: Message) => { ... }, []);
16
17 ⑤ const handleNewBlockRequest = useCallback(async (message: Message) => { ... }, []);
18
19 ⑥ const handleNewBlockAnnouncement = useCallback(async (message: Message) => { ... }, []);
20
21 ⑦ const handleServerMessages = useCallback((message: Message) => { ... }, [
22   handleGetLongestChainRequest,
23   handleNewBlockAnnouncement,
24   handleNewBlockRequest
25 ]);
```

**Figure 14.7 Memoized functions from App component**

In figure 14.8, each of the variables declared in lines 15, 23, 33, and 38 is local to the `App`

component, hence React assumes that its value (i.e. the function expression) can change. By wrapping the body of these functions in `useCallback()`, we instruct React to re-use the same function instance on each rendering, which makes it more efficient.

Look at the last line of the listing 14.10 – `handleServerMessages` is a dependency of `useEffect()`. Technically, if instead of the function expression `const handleNewBlockRequest = useCallback()` we'd use function `handleNewBlockRequest()`, the app would still work, but each function would be recreated on each rendering.

In lines 21, 31, and 36 in figure 14.8, the array of dependencies is empty, which tells us that these callbacks can't have any stale values, and there's no need for any dependencies.

In lines 48 - 49, we listed variables `handleGetLongestChainRequest`, `handleNewBlockAnnouncement`, and `handleNewBlockRequest` as dependencies, which are used inside the callback `handleServerMessages()` as seen in listing 14.12. Didn't we state in the previous paragraph that these callbacks won't create a stale state situation? We did, but React can't peek inside these callback to see this.

### **Listing 14.12. The callback `handleServerMessages()`**

```
const handleServerMessages = useCallback((message: Message) => {
  switch (message.type) {
    case MessageType.GetLongestChainRequest:
      return handleGetLongestChainRequest(message); ①
    case MessageType.NewBlockRequest :
      return handleNewBlockRequest(message); ①
    case MessageType.NewBlockAnnouncement :
      return handleNewBlockAnnouncement(message); ①
    default: {
      console.log(`Received message of unknown type: "${message.type}"`);
    }
  }
}, [
  handleGetLongestChainRequest, ②
  handleNewBlockAnnouncement, ②
  handleNewBlockRequest ②
]);
```

- ① Using a dependency inside the `useCallback()`
- ② Declaring a dependency of the `useCallback()`

Besides the functions that communicate with the WebSocket server, the `App` component has three functions that communicate with the instance of the `BlockchainNode`. Listing 14.13 shows the functions `addTransaction()`, `generateBlock()`, and `addBlock()`. We didn't change the app logic of these operations, but each of these functions ends with the invocations of the React-specific `setState()`, which is done to request re-rendering.

### Listing 14.13 Three more functions from the App component

```

function addTransaction(transaction: Transaction): void {    ①
  node.addTransaction(transaction);
  setStatus(getStatus(node));    ②
}

async function generateBlock() {    ①

  server.requestNewBlock(node.pendingTransactions);
  const miningProcessIsDone = node.mineBlockWith(node.pendingTransactions);

  setStatus(getStatus(node));    ②

  const newBlock = await miningProcessIsDone;
  addBlock(newBlock);
}

async function addBlock(block: Block, notifyOthers = true): Promise<void> {
  try {
    await node.addBlock(block);
    if (notifyOthers) {
      server.announceNewBlock(block);
    }
  } catch (error) {
    console.log(error.message);
  }

  setStatus(getStatus(node));    ②
}

```

- ① The invocation of this function is initiated by the child component
- ② Update the component's status

**NOTE**

The invocation of the functions `addTransaction()` and `generateBlock()` is driven by the child components `TransactionForm` and `PendingTransactionPanel` respectively. We'll review the relevant code in the next section.

The function `addTransaction()` accumulates pending transactions, which are handled by the function `generateBlock()`, and when one of the nodes completes the mining first, the function `addBlock()` attempts to add it to the blockchain. If our node was the first in mining, this function adds the new block and notifies others; otherwise, the new block will arrive from the server via the callback `handleNewBlockAnnouncement()`.

The function `getStatus()` is located in the file `App.tsx`, but it's implemented outside of the `App` component, and its body is shown in listing 14.14.

#### **Listing 14.14. The function getStatus() from App.tsx**

```
function getStatus(node: BlockchainNode): string {
  return node.isEmpty ? 'Initializing the blockchain...' :
    node.isMining ? 'Mining a new block...' :
    node.noPendingTransactions ? 'Add one or more transactions.' :
      'Ready to mine a new block.';
}
```

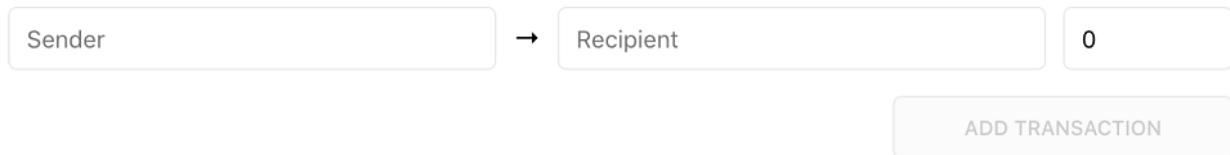
When the `App` component invokes `setStatus(getStatus(node))`, there are two possible outcomes: either `getStatus()` returns the same status as before or a new one. If the status didn't change, calling `setStatus()` won't result in re-rendering of the UI, and vice versa.

We've covered the React specifics in handling the smart `App` component, now let's get familiar with the code of the presentation ones.

#### **14.4 The presentation component `TransactionForm`**

Figure 14.9 shows the UI of the `TransactionForm` component, which allows the user to enter the names of the sender and recipient as well as the transaction amount. When the user clicks on the button `ADD TRANSACTION`, this information has to be sent to the parent `App` component that's a smart component, because it knows how to process this data.

**New transaction**



**Figure 14.8. The UI of the `TransactionForm` component**

The JSX of the `App` component that communicates with `TransactionForm` is shown in listing 14.14.

#### **Listing 14.15. The App's JSX for rendering the `TransactionForm` component**

```
<TransactionForm
  onAddTransaction={addTransaction} ①
  disabled={node.isMining || node.isEmpty} ②
/>
```

- ① Child's `onAddTransaction()` results in parent's `addTransaction()`
- ② When the child has to be disabled

From this JSX, we can guess that when the `TransactionForm` component invokes its function `onAddTransaction()`, the `App` component will invoke its `addTransaction()` shown earlier in listing 14.13. We can also see that the child component has the prop `disabled` driven by

status of the variable `node`, which holds the reference to the `BlockchainNode` instance.

Listing 14.15 shows the first half of the code located in the file `TransactionForm.tsx`.

### **Listing 14.16 The first part of TransactionForm.tsx**

```
import React, { ChangeEvent, FormEvent, useState } from 'react';
import { Transaction } from '../lib/blockchain-node';

type TransactionFormProps = {
  onAddTransaction: (transaction: Transaction) => void,      ①
  disabled: boolean    ②
};

const defaultValue = {recipient: '', sender: '', amount: 0};  ③

const TransactionForm: React.FC<TransactionFormProps> = ({onAddTransaction, disabled}) => {  ④
  const [formValue, setFormValue] = useState<Transaction>(defaultValue);  ⑤
  const isValid = formValue.sender && formValue.recipient && formValue.amount > 0;  ⑥

  function handleInputChange({ target }: ChangeEvent<HTMLInputElement>) {  ⑦
    setFormValue({
      ...formValue,
      [target.name]: target.value
    });
  }

  function handleSubmit(event: FormEvent<HTMLFormElement>) {
    event.preventDefault();
    onAddTransaction(formValue);  ⑧
    setFormValue(defaultValue);  ⑨
  }

  return (
    // The JSX is shown in listing 14.16
  );
}
```

- ① A prop for sending data to the parent
- ② A prop for getting data from the parent
- ③ The object with the default values for the form
- ④ This component accepts two props
- ⑤ The component's state
- ⑥ The `isValid` flag defines if the button can be enabled
- ⑦ One event handler for all input fields
- ⑧ Pass the `formValue` object to the parent
- ⑨ Reset the form

When the user clicks on the button ADD TRANSACTION, the `TransactionForm` component has to invoke some function on the parent. Since React doesn't want the child to know the internals of the parent, the child just gets the prop `onAddTransaction`, but it has to know the correct signature of the parent's function that corresponds to `onAddTransaction`. The following

line maps the name of the prop `onAddTransaction` to the signature of the function to be called on the parent:

```
onAddTransaction: (transaction: Transaction) => void,
```

Revisit listing 14.13, and you'll see that the parent's function `addTransaction()` indeed has the signature `(transaction: Transaction) void`. In listing 14.6, you can easily find the line that maps the parent's `addTransaction` to the child's `onAddTransaction`.

The `TransactionForm` component renders a simple form and defines only one state variable `formValue`, which is the object containing the current form's values. When the user types in the input fields, the event handler `handleInputChange()` is invoked and saves the entered value in `formValue`. In listing 14.16, you'll see that this event handler is assigned to each input field of the form.

In this handler, using destructuring we extract the `target` object, which point at the input field that triggered this event. The name and value of the DOM element we get dynamically from the object `target`. The property `target.name` will contain the name of the field and `target.value` – its value. In Chrome Dev Tools, put a breakpoint in the method `handleInputChange()` to see how this works. By invoking `setFormValue()`, we change the component's state to reflect the current values of the input fields.

**TIP**

While invoking `setState()` we use object cloning with the spread operator.  
This technique is described in appendix A in section A.7.

The default values for the transaction form are stored in the variable `defaultFormValue` and they are used for the initial form rendering as well as resetting the form after the button ADD TRANSACTION is clicked. When the user clicks on this button, the function `handleFormSubmit()` invokes `onAddTransaction()`, passing the `formValue` object to the parent (i.e. the `App` component).

Listing 14.16 shows the JSX of the `TransactionForm` component. It's a form with three input fields and a submit button.

## Listing 14.17 The second part of TransactionForm.tsx

```

return (
  <>
  <h2>New transaction</h2>
  <form className="add-transaction-form" onSubmit={handleFormSubmit}>
    <input
      type="text"
      name="sender"
      placeholder="Sender"
      autoComplete="off"
      disabled={disabled}    ①
      value={formValue.sender} ②
      onChange={handleInputChange} ③
    />
    <span className="hidden-xs"></span>
    <input
      type="text"
      name="recipient"
      placeholder="Recipient"
      autoComplete="off"
      disabled={disabled}    ①
      value={formValue.recipient} ②
      onChange={handleInputChange} ③
    />
    <input
      type="number"
      name="amount"
      placeholder="Amount"
      disabled={disabled}    ①
      value={formValue.amount} ②
      onChange={handleInputChange} ③
    />
    <button type="submit"
      disabled={!isValid || disabled} ④
      className="ripple">ADD TRANSACTION</button>
  </form>
</>
);

```

- ① Bind the disable attribute to the disabled prop
- ② Bind the value from the corresponding state property
- ③ Invoke handleInputChange on every state mutation
- ④ Enable the button only when the form is valid

React handles HTML forms differently compared to other elements, because a form has an internal state, i.e. an object with all form fields values. In React, you can turn a regular form field into a *controlled components* by binding the state object's property (e.g. `formValue.sender`) to its attribute `value` and adding an `onChange` event handler.

Our form has three controlled components (i.e. input fields), and every state mutation will have an associated handler function. In the `TransactionForm` component, `handleInputChange()` is such a handler function. As you can see in listing 14.15, we're just cloning the state object in `handleInputChange()`, but you can put any app logic in such a handler.

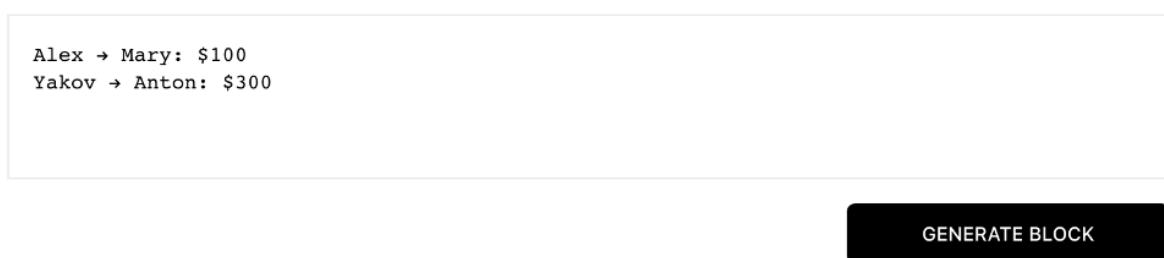
We'd like to stress again, that `TransactionForm` is a presentation component that just knows

how to present the values and which function to invoke on form submit. It has no knowledge about its parent and doesn't communicate with any external services, which makes it 100% reusable.

## 14.5 The presentation component PendingTransactionPanel

Each time the user clicks the button ADD TRANSACTION in the TransactionForm component, the entered transaction should somehow be passed to the PendingTransactionsPanel, and figure 14.10 shows the rendering of this component with two pending transactions. These two components don't know about each other so the App component can play the role of a mediator in passing the data from one component to another.

### Pending transactions



**Figure 14.9 The UI of the PendingTransactionsPanel component**

Listing 14.18 shows the fragment of the App component's JSX that renders the PendingTransactionsPanel component. The App component communicates with PendingTransactionsPanel similarly to how it does it with TransactionForm, and this component gets three props from App.

### Listing 14.18 The App's JSX for rendering the PendingTransactionPanel component

```
<PendingTransactionsPanel
  formattedTransactions={formatTransactions(node.pendingTransactions)}      ①
  onGenerateBlock={generateBlock}    ②
  disabled={node.isMining || node.noPendingTransactions}    ③
/>
```

- ① Format the transaction and pass it to this component
- ② Child's onGenerateBlock() results in parent's generateBlock()
- ③ This child has to be disabled initially

The first prop is `formattedTransactions`, and the App component passes them in to `PendingTransactionsPanel` for rendering. While reviewing the code from the file `App.tsx`, we

left out its utility function `formatTransactions()`, which simply creates a nicely formatted message about this transaction. Listing 14.19 shows the code of the self-explanatory function `formatTransactions()` located in the file `App.tsx` outside of the `App` component.

### **Listing 14.19 The function `formatTransactions()`**

```
function formatTransactions(transactions: Transaction[]): string {
    return transactions.map(t => `${t.sender} ${t.recipient}: ${t.amount}`)
        .join('\n');
}
```

Figure 14.10 shows how the formatted transactions look like. The second prop `onGeneratedBlock` is a reference to the function to call on its parent when the user clicks on the button `GENERATE BLOCK`.

Listing 14.20 shows the code of the component `PendingTransactionsPanel`, which is pretty straightforward because it doesn't contain any forms and doesn't need to handle user input except the click on the button `GENERATE BLOCK`.

### **Listing 14.20 The file `PendingTransactionPanel.tsx`**

```
import React from 'react';

type PendingTransactionsPanelProps = {
    formattedTransactions: string;      ①
    onGenerateBlock: () => void;        ②
    disabled: boolean;
}

const PendingTransactionsPanel: React.FC<PendingTransactionsPanelProps> =
    ({formattedTransactions, onGenerateBlock, disabled}) => {
    return (
        <>
            <h2>Pending transactions</h2>
            <pre className="pending-transactions__list">      ③
                {formattedTransactions || 'No pending transactions yet.'}  ④
            </pre>
            <div className="pending-transactions__form">
                <button disabled={disabled}
                    onClick={() => onGenerateBlock()}           ⑤
                    className="ripple"
                    type="button">GENERATE BLOCK</button>
            </div>
            <div className="clear"></div>          ⑥
        </>
    );
}

export default PendingTransactionsPanel;
```

- ① The prop for formatted transactions
- ② The prop `onGenerateBlock` must use this method signature
- ③ Align everything (including subsequent sibling elements) to the right side of the parent container
- ④ Display either provided transaction or the default text

- ⑤ Invoke the prop `onGenerateBlock()`
- ⑥ Clear the right-alignment

When the user clicks on the button GENERATE BLOCK, we invoke the prop `onGenerateBlock()`, which in turn invokes the function `generateBlock()` on the `App` component.

In the style selector `.pending-transactions__form` (see `index.css`), we use `float: right`, which forces everything to be aligned to the right side of the parent container, including subsequent sibling elements. The style `clear` is defined as `clear: both`, and it stops the right-alignment rule so we do not break the following Current blocks section.

The last component to review is the one that shows the blockchain at the bottom of the window.

## **14.6 The presentation components `BlockPanel` and `BlockComponent`**

When the user clicks on the button GENERATE BLOCK in the `PendingTransactionPanel` component, all active blocks in the blockchain start the mining process, and after the consensus, a new block will be added to the blockchain and rendered in the `BlockPanel` component, which can parent one or more `BlockComponent`. Figure 14.11 shows the rendering of the `BlockPanel` of a two-block blockchain.

### **Current blocks**

<b>#0</b>	8:37:49 AM
← PREV HASH	THIS HASH
0	000044ce35f9a...
TRANSACTIONS	
No transactions	
<b>#1</b>	9:06:35 AM
← PREV HASH	THIS HASH
	000044ce35f9... 0000e8a2330a4...
TRANSACTIONS	
Alex → Mary: \$100	
Yakov → Anton: \$300	

**Figure 14.10 The UI of the `BlockPanel` component**

During block mining and getting the consensus, the instances of `BlockchainNode` and `WebSocketController` are involved, but being a presentation component, `BlockPanel` doesn't directly communicate with either of these objects, as this work is delegated to the smart `App` component. The `BlockPanel` component doesn't send any data to its parent, and its goal is to render the blockchain provided via the prop `block`:

```
<BlocksPanel blocks={node.chain} />
```

The file `BlocksPanel.tsx` contains the code of two components: `BlocksPanel` and

`BlockComponent`. Listing 14.21 shows the code of the `BlockComponent`, which renders a single block in the blockchain. Figure 14.11 shows two instances of the `BlockComponent`.

### Listing 14.21 The BlockComponent

```
const BlockComponent: React.FC<{ index: number, block: Block }> = ({ index, block }) => {
  const formattedTransactions = formatTransactions(block.transactions); ①
  const timestamp = new Date(block.timestamp).toLocaleTimeString();

  return (
    <div className="block">
      <div className="block__header">
        <span className="block__index">#{index}</span> ②
        <span className="block__timestamp">{timestamp}</span>
      </div>
      <div className="block__hashes">
        <div className="block__hash">
          <div className="block__label"> PREV HASH</div>
          <div className="block__hash-value">{block.previousHash}</div> ③
        </div>
        <div className="block__hash">
          <div className="block__label">THIS HASH</div>
          <div className="block__hash-value">{block.hash}</div> ④
        </div>
      </div>
      <div>
        <div className="block__label">TRANSACTIONS</div>
        <pre className="block__transactions">{formattedTransactions
          || 'No transactions' }</pre> ⑤
      </div>
    </div>
  );
}
```

- ① The function `formattedTransaction()` is the same as in `App` component
- ② Block number
- ③ Previous block's hash
- ④ This block's hash
- ⑤ Block's transactions

Listing 14.22 shows the `BlocksPanel` component, which serves as a container of all `BlockComponent` components.

## Listing 14.22 The BlocksPanel component

```

import React from 'react';
import { Block, Transaction } from '../lib/blockchain-node';

type BlocksPanelProps = {
  blocks: Block[] ①
};

const BlocksPanel: React.FC<BlocksPanelProps> = ({blocks}) => {
  return (
    <>
      <h2>Current blocks</h2>
      <div className="blocks">
        <div className="blocks__ribbon">
          {blocks.map((b, i) => ②
            <BlockComponent key={b.hash} index={i} block={b}></BlockComponent>) } ③
        </div>
        <div className="blocks__overlay"></div>
      </div>
    </>
  );
}

```

- ① An array of `Block` instances is the only prop here
- ② Using `Array.map()` to turn the data into components
- ③ Passing `key`, `index`, and `block` props to `BlockComponent`

The `BlocksPanel` gets an array of `Block` instances from the `App` component and applies the method `Array.map()` to convert each `Block` object into the `BlockComponent`. The method `map()` passes the `key` (the hash code) and the unique index of the block and the `Block` object to each instance of the `BlockComponent`.

The props of the `BlockComponent` are `index` and `block`. Note that we're assigning the block hash as a `key` prop to each instance of the `BlockComponent` even though the `key` prop was never mentioned in its code in listing 14.20. The reason is that when you have a collection of rendered objects (e.g. list items or multiple instances of the same component), React needs a way to uniquely identify each component during its reconciliation with Virtual DOM to keep track of data associated with each DOM element.

If you won't use the unique value for the `key` prop on each `BlockComponent`, React will print a warning in the browser console that "Each child in array or iterator should have a unique key prop". In our app this won't mess up the data because we only adding new blocks to the end of the array, but if the user could add or remove arbitrary elements from a collection of UI components, without having the unique `key` prop, it could create a situation when a UI element and the underlying data wouldn't match.

This concludes the code review of the React version of our blockchain app.

## 14.7 Summary

- In dev, our React web app was deployed under a Webpack’s dev server, but it was communicating with another (messaging) server as well. For that, we declare custom environment variables with the messaging server’s URL. For a WebSocket server, this was enough, but if you’d use other HTTP servers, you’d have to proxy such HTTP requests as described at [facebook.github.io/create-react-app/docs/proxying-api-requests-in-development](https://facebook.github.io/create-react-app/docs/proxying-api-requests-in-development).
- Typically, the UI of a React app consists of smart and presentation components. Don’t place the application logic in the presentation components, which are meant for presenting the data received from other components or for providing the interaction with the user with further sending the user’s input to other components.
- A child component should never call an API from its parent directly. Using props, the parent component should give to the child a name that’s mapped to its function that have to be called as a result of some child’s action. The child would invoke the provided function reference without knowing the real name of the parent’s function.
- To prevent unnecessary re-creations of function expressions located in the React components, consider using the memoization technique with the `useCallback()` hook.

In the next chapter, we’ll show you how to use TypeScript with the Vue.js framework.

# 15

## *Developing Vue.js apps with TypeScript*

### **This chapter covers:**

- A quick intro to the Vue.js framework
- How to jumpstart a new project with Vue CLI
- How to work with the class-based components
- How to arrange the client-side navigation using the Vue Router

Angular is a framework, React.js is a library, and Vue.js (a.k.a. Vue) feels like a "library plus plus". Vue (see [vuejs.org](https://vuejs.org)) was created by Evan You in 2014 as an attempt to create a lighter version of Angular. At the time of this writing, Vue.js has 145K stars and 275+ contributors on GitHub. The numbers are high, but Vue is not backed by any large corporation like Angular (Google) and React.js (Facebook).

Vue is a progressive, incrementally-adoptable JavaScript framework for building UI on the web, so if you already have a web app written with or without any JavaScript libraries, you can introduce Vue to just a small portion of your app and then keep adding Vue to other parts of the app as needed. Similarly to React, you can attach a Vue instance to any HTML element (e.g. a `<div>`), and only this element will be controlled by Vue.

Vue is a component-based library that's focused on the app's *View* part (remember MVC?). The core Vue library is about declarative rendering of UI components, and similarly to React.js, Vue uses the virtual DOM under the hood. Beside the code library, Vue comes with other modules for the client-side routing, state management, et al.

**NOTE****Note**

The first two versions of Vue were written in JavaScript, but Vue 3.0 is being re-written in TypeScript. We write this chapter in the Summer of 2019, and the creators of Vue announced that the upcoming Vue 3 will have a number of major changes - built-in reactivity API, hooks-like API, and improved TypeScript integration. The new function-based composition API is in the works (see [github.com/vuejs/rfcs/pull/78](https://github.com/vuejs/rfcs/pull/78)). The creators of the new version of Vue state that the new API will be 100% compatible with current syntax and purely additive, but for breaking changes, they will offer the version-updater tool that should automatically upgrade your existing code base to Vue 3.

After getting familiar with Angular and React, you should be comfortable with the concept of web components, which can be represented by custom tags like `<transaction-form-component>` or `<BlocksPanel>`. Such components can have their state, can take the data in or send them out, so components can communicate with each other. A component can have its child components, and in that sense, Vue works the same way as Angular or React although Vue uses different syntax for declaring components.

Similarly to React and Angular, you can scaffold the project using the CLI tool, but we'd like to start in a simplest possible way, and in the next section, you'll see how to create a Hello World web app without using any tooling.

## 15.1 Developing the simplest web page with Vue

In this section, we'll show you a very simple web page written with Vue and JavaScript. This page renders the message "Hello World" inside the HTML element `<div>`, and we'll add the Vue library to this HTML page by using a `<script>` tag pointing at the URL of a Vue CDN as seen in listing 15.1.

### **Listing 15.1 Adding Vue to index.html**

```
<!DOCTYPE html>
<body>
  <div id="one"></div>
  <div id="two"></div>

  <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script> ①
</body>
</html>
```

#### ① Adding Vue from a CDN

This web page contains two empty `<div>` tags and the `<script>` tag to load the code of the Vue library. We purposely added two `<div>` tags to illustrate how to attach the Vue instance just to a

specific HTML element.

By loading Vue in a web page, we make all its API available to the scripts of this page, and the next step is to create an instance of the `Vue` object and attach it to a specific HTML element. Note that our `<div>` elements have different IDs, and we can "say" to Vue, "Please start controlling the `<div>` that has the ID `one`." Imagine, that the second `<div>` contains the content of the existing app written using different technology, and we don't want Vue to control it.

The constructor of the `Vue` object, requires the argument of type `ComponentOptions`, and you can find the names of all optional properties of `ComponentOptions` in the type definition file `options.d.ts`. We'll just specify the property `el` (for element) containing the `id` of the HTML element to be controlled by Vue and the property `data` that will store the data to render. Listing 15.2 shows the script that creates and attaches the `Vue` instance to the first `div` passing the greeting `Hello World` as `data`.

### **Listing 15.2 Attaching the Vue instance to the first div**

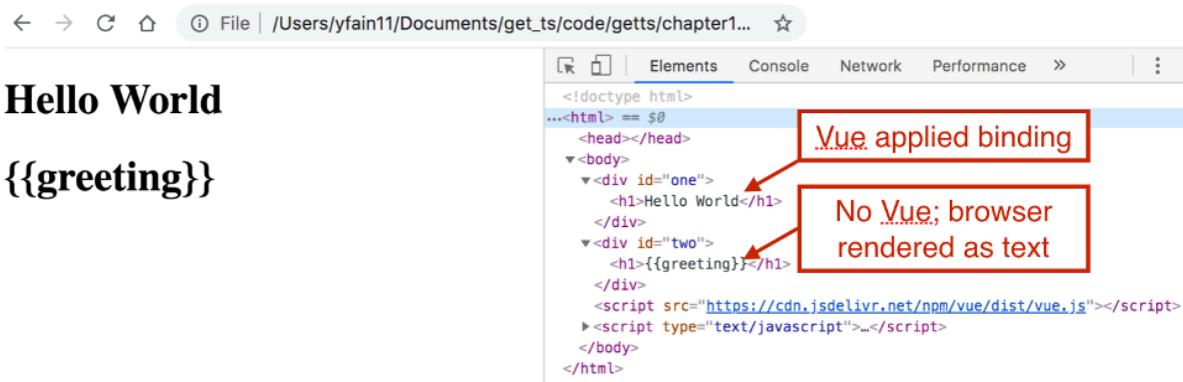
```
<!DOCTYPE html>
<body>
  <div id="one">      ①
    <h1>{{greeting}}</h1>    ②
  </div>
  <div id="two">      ③
    <h1>{{greeting}}</h1>    ④
  </div>

  <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>

  <script type="text/javascript">
    const myApp = new Vue({      ⑤
      el: "#one",               ⑥
      data: {                  ⑦
        greeting: "Hello World"
      }
    })
  </script>
</body>
</html>
```

- ① This `div` will be controlled by Vue
- ② Data binding will show the value of the `greeting` variable
- ③ This `div` is not controlled by Vue
- ④ No data binding here; the browser will render the text `{{greeting}}`
- ⑤ Creating the `Vue` instance
- ⑥ Attaching the `Vue` instance to the element with the ID `one`
- ⑦ Passing the `data` to the element

Open the file `index.html` in Chrome browser with its Dev Tools showing the Elements tab, and you'll see a web page as shown in figure 15.1.



**Figure 15.1 Rendering divs with and without Vue**

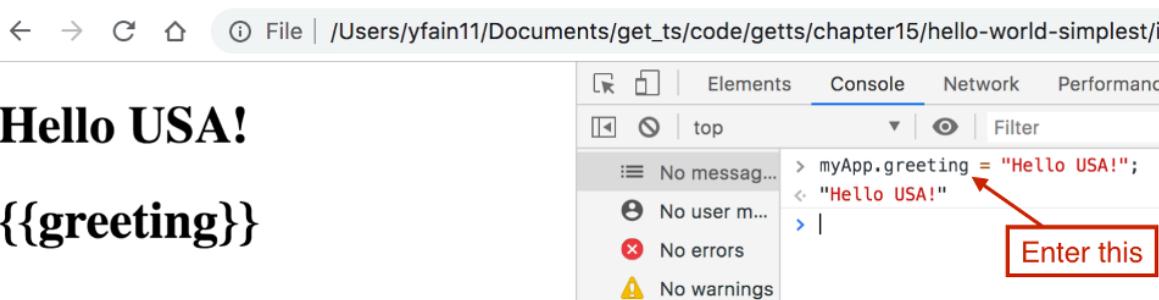
In this web page, the bottom `<div>` is just a regular HTML element and the browser renders `{{expression}}` as text. Note that while instantiating Vue, we passed in a JavaScript object providing the ID of the top `<div>` element so the Vue instance started control it and applied binding to the `greeting` variable rendering its value Hello World. When an HTML element uses the "double mustache" notation (`{{expression}}`), Vue understands that it needs to render the evaluation of this expression. The UI of this app becomes *reactive*, and as soon as the value of the variable `greeting` changes, the new value is rendered inside the top `<div>` that has the `id="one"`. Since the `Vue` instance is scoped to a particular DOM element, nothing stops you from creating multiple `Vue` instances bound to different DOM elements.

#### TIP

#### Tip

Enter the URL of the CDN in the browser, and you'll see the version of the Vue library it hosts. At the time of this writing, the version was 2.6.10.

You can access all the properties defined in the `data` object through the reference variable `myApp`. Figure 15.2 shows a screen shot taken after we entered `myApp.greeting = "Hello USA!"`. The new value is rendered in the top `div`.



**Figure 15.2 Changing the greeting's value in the browser's console**

A `Vue` instance contains one or more UI components, and if in listing 15.2 we just passed to the `Vue` instance an object literal with two properties, we could provide an object with a `render()`

function to render a top-level component:

```
new Vue({
  render: h => h(App) // App is a top-level component
})
```

You'll see this syntax starting from the next section in the apps generated by Vue CLI. Here, the letter `h` just stands for a script that generates HTML structures. You can also think of `h` as a `createElement()` function, which is described in the product documentation at [vuejs.org/v2/guide/render-function.html#createElement-Arguments](https://vuejs.org/v2/guide/render-function.html#createElement-Arguments).

In our very simple app shown in listing 15.2, we just use the DOM element `<div>` to play a role of a UI component, but in the next section, you'll see how to declare a Vue component that will have three sections:

- a declarative template
- the script
- the styles

The `data` property in listing 15.2 played the role of the component state. If we added an input element so the user could enter data (e.g. her name), then the Vue instance would update the component's state and the function `render()` would re-render the component with the new state.

Let's switch to the Node-based project setup to see how a Vue app can be split into UI components in the real world.

## 15.2 Generating and running a new app with Vue CLI

The command-line interface called Vue CLI (see [cli.vuejs.org/](https://cli.vuejs.org/)) automates the process of creating a Vue project that has a transpiler, a bundler, scripts for reproducible builds, and config files. This tool generates all required configuration files for Webpack, so you can concentrate on writing your app instead of wasting time configuring tooling. To install the Vue CLI package globally on your computer, run the following command in the Terminal window:

```
npm install @vue/cli -g
```

Now you can run the `vue` command in your Terminal window for generating a new JavaScript or TypeScript version of the project.

### TIP

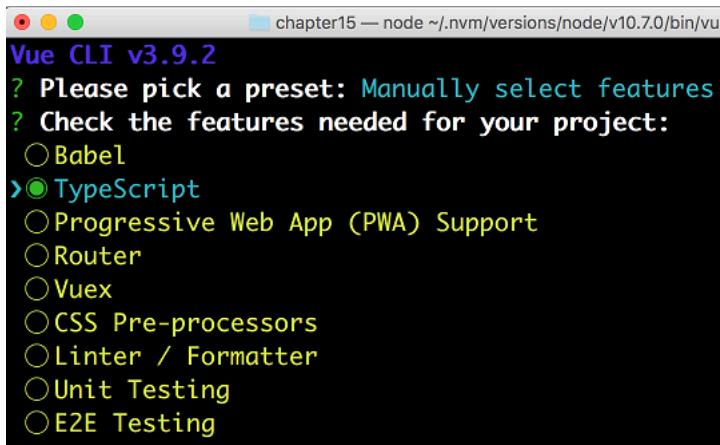
### Tip

To see which version of the Vue CLI was installed, enter the `vue --version` command. We used the CLI of version 3.9.2.

To generate the TypeScript app, run the command `vue create` followed by the app name:

```
vue create hello-world
```

This command will start a dialog asking you to select options for your project. The default configuration would be Babel and ESLint, but to work with the TypeScript compiler, select "Manually select features." Then you'll see a list of options shown in figure 15.3, and you can select/unselect the project options by using the up and down arrows and the space bar.



**Figure 15.3 Manually selecting the project features**

For our hello-world project, we'll select only TypeScript. Hit the Enter key, and the next question will be shown, "Use class-style component syntax?" Agree to this. The next question was if we wanted to use Babel alongside with TypeScript (we rejected this option). In remaining questions, we selected keeping separate config files for Babel and other tools, we didn't want to save these answers for future projects, and selected npm as a default package manager.

#### TIP

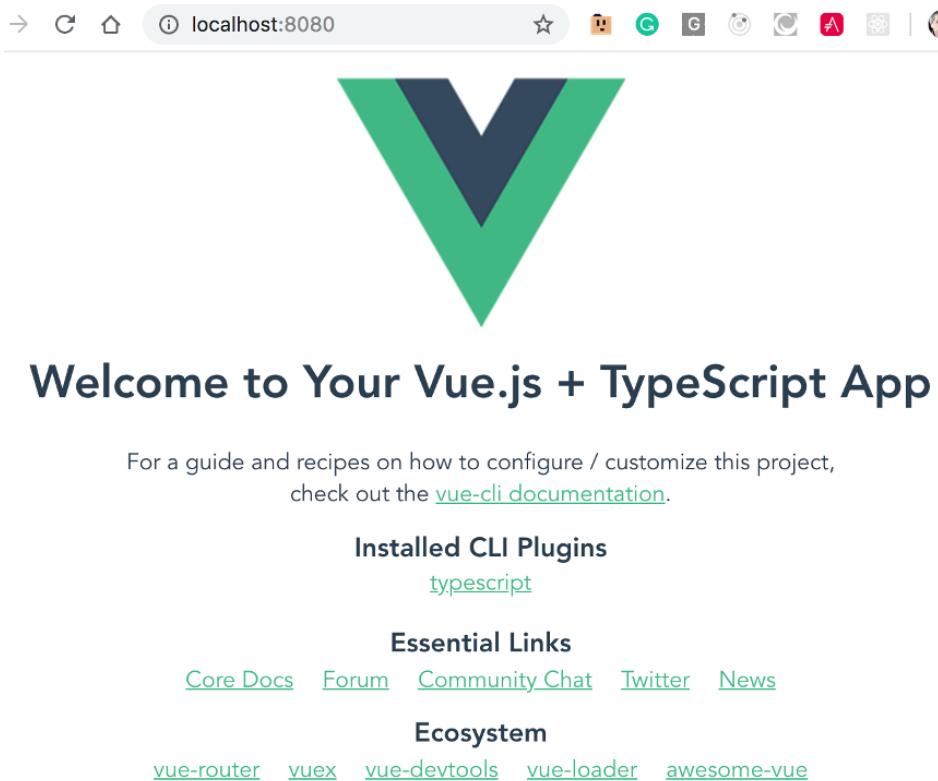
#### Tip

Vuex is a state management library for Vue apps. Consider using it for real-world projects.

The Vue CLI generated a new Node-based project in the hello-world directory and installed all required dependencies. Ready to run this app? Just enter the following commands in your Terminal window:

```
cd hello-world
npm run serve
```

The code of the generated project will be compiled, and the dev server will serve the app at localhost:8080 as shown in figure 15.4.



**Figure 15.4** Running the initially generated project

The procedure of generating and running a Vue project is similar to using CLI for generating Angular and React projects. Under the hood, Vue CLI also uses the Webpack for bundling and its webpack-dev-server for serving the app in the dev mode, and listing 15.3 includes npm script commands `serve` and `build` to start the dev server or to build the bundles with Webpack.

### Listing 15.3 The generated package.json file

```
{
  "name": "hello-world",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "serve": "vue-cli-service serve",      ①
    "build": "vue-cli-service build"      ②
  },
  "dependencies": {
    "vue": "^2.6.10",
    "vue-class-component": "^7.0.2",
    "vue-property-decorator": "^8.1.0"
  },
  "devDependencies": {
    "@vue/cli-plugin-typescript": "^3.9.0", ③
    "@vue/cli-service": "^3.9.0",             ④
    "typescript": "^3.4.3",
    "vue-template-compiler": "^2.6.10"
  }
}
```

- ① Starting the app using Webpack's dev server
- ② Building the app bundles with Webpack

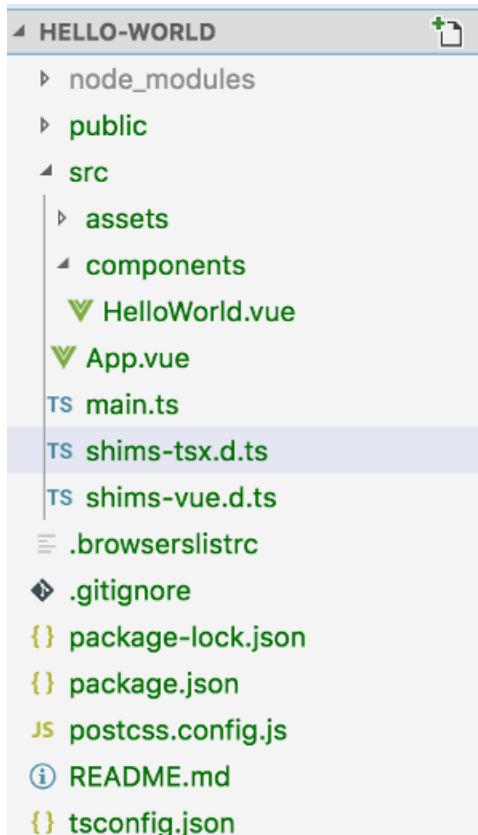
- ③ The CLI TypeScript plugin
- ④ The TypeScript compiler

**TIP****Tip**

Installing Vue using npm gives you TypeScript type declaration files, and IDE will offer your auto-complete and help with static types without the need to use any additional tooling.

Let's open the generated project in VS code and get familiar with the structure of the generated project shown in figure 15.5. It's structured as a typical Node.js project with all dependencies listed in package.json and installed under node\_modules. Since we write in TypeScript, the compiler's options are listed in the file tsconfig.json. The file main.ts loads the top-level component from the file App.vue. All UI components are located in the directory components.

The folder public has the file index.html, which contains the markup including the HTML element that will be controlled by Vue. The build process will modify this file to include the scripts with the app bundles. You're free to add more directories and files containing app logic, and we'll do this in chapter 16 while working on the Vue version of the blockchain client.



**Figure 15.5 The project structure of the CLI-generated Hello World**

Listing 15.4 shows the content of the file main.ts, that creates the `Vue` instance and bootstraps the app. This time, the script uses the `options` object with the `render` property that is a function to create an instance of the `App` component and render it in the DOM element with the ID `app`.

### Listing 15.4 main.ts

```
import Vue from 'vue'
import App from './App.vue'

Vue.config.productionTip = false

new Vue({    ①
  render: h => h(App),    ②
}).$mount('#app')    ③
```

- ① Instantiating `Vue` and passing the options object
- ② This function starts the rendering of the components tree
- ③ Attaching the `Vue` instance to the DOM element with the id `app`

#### TIP

#### Tip

If VS Code is your IDE, install the extension called Vetur (see [vuejs.github.io/vetur](https://vuejs.github.io/vetur)), which offers Vue-specific syntax highlighting, linting, auto-completion, formatting and more.

In listing 15.2, we *mounted* `Vue` on a specific HTML element via the configuration object with the `el` property:

```
const myApp = new Vue({
  el: "#one"
  ...
})
```

In listing 15.4, the `Vue` instance didn't receive the `el` property, and invoking the method `$mount('#app')` starts the mounting process and attaches the `Vue` instance to the DOM element with the id `app`. If you open the generated file `public/index.html`, you'll find the element `<div id="app"></div>` there.

Now let's review the code in the file `App.vue`. It consists of three sections `<template>`, `<script>`, and `<style>`. Figure 15.6 shows these sections after we collapsed their contents in VS Code. Note that the `<script>` section has an attribute `lang = "ts"`, which means TypeScript.

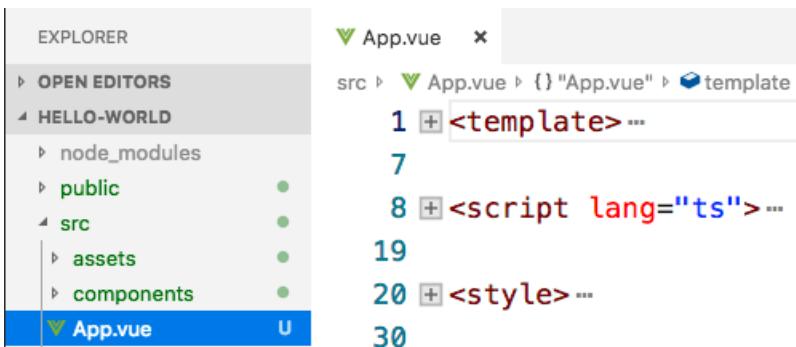


Figure 15.6 Three sections of the App.vue

When Webpack builds the bundles for deployment, it uses a special Vue plugin to transform the code of each component to JavaScript so web browsers can parse and render it.

Listing 15.5 shows the content of the CLI-generated `<template>` section. After seeing how Angular and React represent custom web components, you should easily spot the child component's tag `<HelloWorld>` there.

### Listing 15.5 The template section of the App component

```
<template>
  <div id="app">
    
    <HelloWorld msg="Welcome to Your Vue.js + TypeScript App"/> ①
  </div>
</template>
```

① The child HelloWorld component

From this template, you might also have guessed that the `HelloWorld` component takes the property `msg`, and the `App` component passes the welcome message to it. Most of the content of the generated app was rendered by the `HelloWorld` component.

Listing 15.6 shows the `<script>` section of the `App` component. Similarly to Angular, when you write Vue apps in TypeScript, you can use decorators, e.g. `@Component()`, which takes an optional argument of the type `ComponentOptions` that has such properties as `el`, `data`, `template`, `props`, `components` and more.

### Listing 15.6 The template section of the App component

```
<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import HelloWorld from './components/HelloWorld.vue';

@Component({ ①
  components: { ②
    HelloWorld,
  },
})
export default class App extends Vue {} ③
</script>
```

- ① Applying the Component decorator to the class App
- ② Passing the ComponentOptions argument with the property components
- ③ The App component is a class that extends Vue

**TIP****Tip**

To support TypeScript decorators, the compiler's option `experimentalDecorators` must be set to `true` in `tsconfig.json`.

During the code generation, the CLI added two dependencies in `package.json`: `vue-class-component` and `vue-property-decorator`. The package `vue-class-component` allows us to write a Vue component as a class that extends `Vue`, but starting from Vue 3.0, class-based components will be supported natively.

The package `vue-property-decorator` allows us to use a variety of decorators like `@Component()`, `@Prop()` and others. Without these packages, we could have exported the object instead of a class in the `<script>` section of the file `App.vue`:

```
import HelloWorld from './components/HelloWorld.vue';

export default {
  name: 'app',
  components: {
    HelloWorld
  }
}
```

The child component `HelloWorld` has a large `<template>` section with multiple `<a>` tags, but on top of the template you'll see the bound value `{ { msg } }` as shown in listing 15.7.

### **Listing 15.7 The fragment of the HelloWorld component's template**

```
<template>
  <div class="hello">
    <h1>{ { msg } }</h1> ①
    <p>
      <!-- The rest of the content is omitted for brevity-->
  </template>
```

- ① Binding the value of the `msg` property to the view

The `<script>` section of the `HelloWorld` component is shown in listing 15.8. You can see two TypeScript decorators there: `Component()` and `Props()`. In chapter 13, we've introduced React.js props, in Vue, they play the same role - to pass data from parent to child.

## Listing 15.8 The <script> section of HelloWorld.vue

```
<script lang="ts">
import { Component, Prop, Vue } from 'vue-property-decorator';

@Component ①
export default class HelloWorld extends Vue {
  @Prop() private msg!: string; ②
}
</script>
```

- ① Using the class decorator `@Component()` without any arguments
- ② Using the property decorator `@Prop()`

Have you noticed the exclamation point after `msg`? It's a non-null assertion operator. By adding the exclamation point to the property name you say to the TypeScript's type checker, "Don't complain about the possibility of `msg` being `null` or `undefined`, because it won't be. Take my word for it!" You can also provide a default value for `msg` as follows:

```
@Prop({default: "The message will go here"}) private msg: string;
```

Vue CLI generated the code with a property-level decorator `@Prop()` to declare that the `HelloWorld` component takes one property `msg`. The other way to do this is by using the `props` property of the `@Component()` decorator. The following code snippet shows an alternative way of passing the `msg` prop via the property of `@Component()`.

```
@Component({
  props: {
    msg: {
      default: "The message will go here"
    }
  }
})
export default class HelloWorld extends Vue {}
```

The default property value will be rendered if the parent component won't assign a value to the `msg` attribute, e.g. `<HelloWorld />`.

Using `props` you can send the data from parent to child, but To pass the data from child to parent, use the method `$emit()`. For example, the child component `<order-component>` can send an event `place-order` with some payload `orderData` to the parent as follows:

```
this.$emit("place-order", orderData);
```

The parent can receive this event like this:

```
<order-component @place-order = "processOrder">

...
processOrder(payload) {
```

```
// handle the payload, i.e. the orderData received from the order component
}
```

**TIP****Tip**

You'll see an example of using `$emit()` in listing 16.4 in chapter 16. There, the `PendingTransaction` component sends the `generate-block` even to its parent.

Now that you understand how the basic Vue app works, we'll introduce you to the navigation with the client-side router offered by Vue.

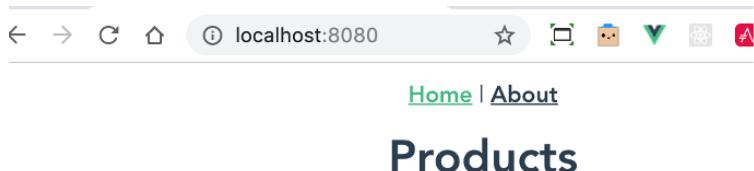
## 15.3 Developing single-page apps with the router support

In chapter 11, we developed a simple Angular app that read and rendered the data from the file `products.json`. In this section, we'll create a single-page Vue app that will also read this file and display the list of products. In this app, we'll introduce the Vue Router and will show how to use some of the Vue directives to render the list of products. Then we'll go over another app illustrating how to pass parameters while navigating to the route that shows the product details view. The first app is located in the directory `router-product-list` and the second one in `router-product-details`.

**NOTE****Note**

In chapter 11, we introduced the Angular router, and the `vue-router` package implements the client-side navigation using similar concepts.

The landing page of the `router-product-list` SPA will show the list of products in the `Home` component as seen in figure 15.7. The user will be able to click on the selected product so the app can process it as needed. The `About` link will navigate to the `About` view without making any requests to the server.



- First Product
- Second Product
- Third Product

**Figure 15.7 Product list**

The point of using the Vue router is to support the user's navigation on the client side and persist

state in the address bar. It creates a bookmarkable location that can be shared and opened directly without a need to go through multiple steps to see a desired view. Also route allows to avoid loading separate web pages from the server - the page remains the same, but the user can navigate *on the client* from one view to another without asking the server to load a different page. The Vue router is implemented in the package called `vue-router` and you can find it in the list of dependencies in the file `package.json`.

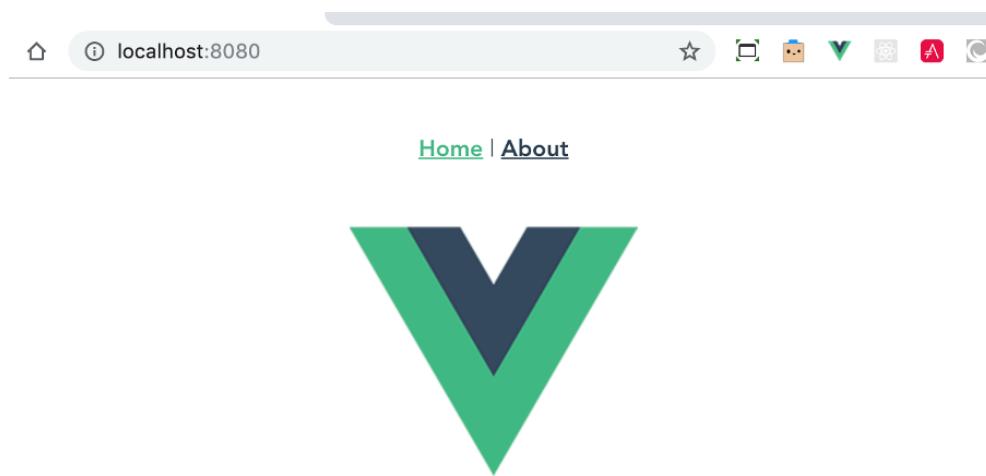
### 15.3.1 Generating a new app with the Vue router

We started again by using CLI to generate the new project called `router-product-list`, but this time, we also selected the Router in the list of CLI options. The CLI also asked if this app should use the history mode for router, and we agreed to this.

The History API is implemented by the browsers that support HTML5 API, but if your app has to support really old browsers, don't select the history mode, and all URLs in your app will include the hash sign to separate the server and client portion of the URL.

For example, without the history mode, the URL of the client's resource could look like `localhost:8080/#about` and the segment to the left of the hash-sign is handled by the server. The URL segment to the right of the hash-sign is handled by the client's app. If you select the history mode, the URL to the same resource would look like `localhost:8080/about`. You can read more about the HTML5 history mode at [developer.mozilla.org/en-US/docs/Web/API/History\\_API](https://developer.mozilla.org/en-US/docs/Web/API/History_API).

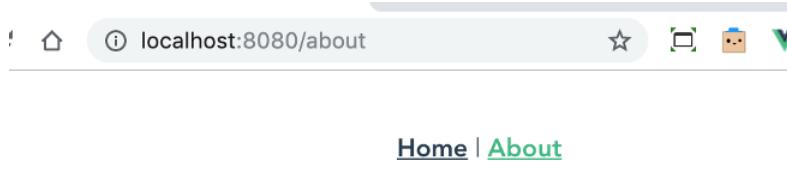
Change the directory to `router-product-list` and run the command `npm run serve`, and you'll see the landing page of the generated app that looks similar to figure 15.4 but with a small addition: there will be two links Home and About at the top of the window as shown in figure 15.8.



**Figure 15.8 Two links on top**

CLI generated the `src/views` directory with two files: `Home.vue` and `About.vue`. These

components are controlled by the Vue router. Note the URL: it is just the protocol (not shown), domain name and port. There is no client's segment in this URL. We can guess, that the router was configured to render the `Home` component by default, if the URL doesn't contain the client's segment. If the user clicks on the About link, the browser renders the `About` component as shown in figure 15.9.



**Figure 15.9 The generated About component**

This time, the URL contains the client's segment `/about`, and again, we can guess that the router was configured to render the `About` component for this segment. Pretty soon you'll see that our guesses were correct.

The generated file `main.ts` imports the `Router` object as seen in listing 15.9 and adds it to the `Vue` instance.

### Listing 15.9 Configuring the router

```
import Vue from 'vue';
import App from './App.vue';
import router from './router';      ①

Vue.config.productionTip = false;

new Vue({
  router,          ②
  render: (h) => h(App),
}).$mount('#app');
```

- ① Importing the routes configuration from `router.ts`
- ② Adding the `Router` object with configured routes to the `Vue` instance

#### TIP

#### Tip

We pass an object literal to the `Vue` instance, and ES6 allows us to use a shortcut syntax, and we can just specify the property name if it's the same as the variable name that contains the value. That's why instead of writing `router: router` we just wrote `router` here.

The `Vue` instance includes the reference to the `Router` object that contains configured routes, i.e.

which component to render if the user clicks on Home or About. For that, the Vue CLI has generated the file router.ts, and its content is shown in listing 15.10. In general, while designing a single-page app you need to think of the user's navigation on the client and create an array that maps the URL segments to the UI components. In listing 15.10 such an array is called `routes`.

### Listing 15.10 The CLI-generated router.ts

```
import Vue from 'vue';
import Router from 'vue-router';
import Home from './views/Home.vue';

Vue.use(Router); ①

export default new Router({ ②
  mode: 'history', ③
  base: process.env.BASE_URL, ④
  routes: [ ⑤
    {
      path: '/',
      name: 'home', ⑥
      component: Home,
    },
    {
      path: '/about',
      name: 'about',
      component: () => import(/* webpackChunkName: "about" */ './views/About.vue'), ⑦
    },
  ],
});
```

- ① Enable the use of the Router package
- ② Creating the Router object
- ③ Support the HTML5 History API (no # in the URL)
- ④ Use the server's URL as a base
- ⑤ Configuring an array of routes
- ⑥ Render the Home component for the default path /
- ⑦ Render the About component for the path /about

When you create an instance of the `Router` object you are passing the object of type `RouterOptions` to its constructor. We didn't use the property `linkActiveClass` here, but if you don't like the green color of the active link, you can change it using this property.

By enabling the Router package, we'll get the access to a special variable `$route`, and we'll use it in the next section to get the parameters passed during the navigation. Note the property `mode`: its value is `history` because we selected the history mode during the app generation. The path `'/'` is mapped to the `Home` component, and CLI didn't forget to import this component from the file `Home.vue`. But instead of mapping the `/about` path to the 'About component, it's mapped to the following fat arrow expression:

```
() => import(/* webpackChunkName: "about" */ './views/About.vue')
```

This line states that the router has to lazy-load the About component when the route is visited. For this to happen, the code instructs Webpack that when it generates production bundles to split the code and generate a separate chunk (`about.[hash].js`) for this route. The import is done dynamically only if the user decides to navigate to the About view.

**TIP****Tip**

To see that Webpack does split the code, run the production build with `npm run build` and check the content of the `dist` directory. There, you'll find a separate file with the name similar to `about.8027d92e.js`. In prod, this file won't be loaded unless the router navigates to the About view.

Our top-level component has links that will display one view or another, and listing 15.11 shows the template section of the `App.vue`, which contains the tags `<router-link>` and `<router-view>`. The `<router-view>` tag defines the area where the changing content (i.e. Home or About component) has to be rendered. Each `<router-link>` has the `to` attribute telling Vue which component to render based on the configured routes. The attribute `to` in the tag `<router-link>` is where you specify where to navigate. For example, `to="/" tells where to navigate if the URL has no client-side segments.`

### **Listing 15.11 The <template> section of the App.vue**

```
<template>
  <div id="app">
    <div id="nav">
      <router-link to="/">Home</router-link> ①
      <router-link to="/about">About</router-link> ②
    </div>
    <router-view> ③
  </div>
</template>
```

- ① Render the default component
- ② Render the component configured to the /about URL segment
- ③ The router has to render Home or About here

In single-page apps, we don't use the original `<a href="...">` HTML tags for links as they would result in server requests and page reloads. A framework that supports client-side routing produces the anchor tags that include click handlers that invoke functions on the client and update the address bar. In Vue, the router offers the `<router-link>` tag, which doesn't send the request to the server. For the `about` route, the Vue router will form the URL `localhost:8080/about`, and then it'll read the mapping for the `/about` segment and renders the `About` component in the `<router-view>` area. If the user clicks on `About` for the very first time, Vue will lazily-load the `About` component before rendering it. On all subsequent clicks on this link the `About` component will be just rendered.

**NOTE****Note**

In listing 15.11, both `<router-link>` tags have static values in the `to` attribute. This doesn't have to be the case, and you can bind the variable to the `:to` attribute (note the colon in front of `to`).

In the next section, we'll replace the code generated in the `Home.vue` to read and render the list of products. We'll also replace the code in `About.view` to display the details of the selected product. While doing this, we'll see how to pass data while navigating to the product details view.

### 15.3.2 Displaying the list of products in the Home view

In this section, we'll read the file `products.json` located in the directory `public`, and it has the following content:

```
[  
  { "id":0, "title": "First Product", "price": 24.99 },  
  { "id":1, "title": "Second Product", "price": 64.99 },  
  { "id":2, "title": "Third Product", "price": 74.99 }  
]
```

This file contains JSON-formatted product data of a rather simple structure and it's easy to write a TypeScript interface to represent a product. But you can also generate the corresponding TypeScript interface by using one of the third-party tools like MakeTypes (see [jvilk.com/MakeTypes](http://jvilk.com/MakeTypes)). Figure 15.10 shows a screen shot taken from the MakeTypes web site. Just paste the JSON data in the text area on the left, and it'll generate the corresponding TypeScript interface in the field on the right.

The screenshot shows the MakeTypes web application interface. At the top, there is a dark header bar with the text "MakeTypes". Below it, the interface is divided into two main sections: "Input: JSON Examples" on the left and "Output: TypeScript Interfaces" on the right.

**Input: JSON Examples:** A text input field labeled "Product" containing the following JSON array:

```
1 [  
2   { "id":0, "title": "First Product", "price": 24.99 },  
3   { "id":1, "title": "Second Product", "price": 64.99 },  
4   { "id":2, "title": "Third Product", "price": 74.99 }  
5 ]
```

**Output: TypeScript Interfaces:** A text output field containing the generated TypeScript interface code:

```
1 export interface Product {  
2   id: number;  
3   title: string;  
4   price: number;  
5 }
```

Below the output field, there are two buttons: "Copy to clipboard" and "Save interface file...".

**Figure 15.10 Generating the TypeScript interface from JSON using MakeTypes**

We used MakeTypes to generate the `Product` interface, which is located in the file `product.ts`.

We want to read this file as soon as our app navigates to the Home route, which is a default route. Where in the Home component should we put the code for fetching data? Do we know the exact moment when the creation of the `Home` component is complete? Yes, we do. Vue offers a number of callbacks that are invoked at different stages of the component's lifecycle. Check the TypeScript file declaration file `options.d.ts` located in `node_modules/vue/types` and look for the declarations of the interface `ComponentOptions`. You'll find there the declarations of all the lifecycle hooks as seen in listing 15.12.

### **Listing 15.12 Component lifecycle hooks**

```
beforeCreate?(this: V): void;
created(): void;
beforeDestroy(): void;
destroyed(): void;
beforeMount(): void;
mounted(): void;
beforeUpdate(): void;
updated(): void;
activated(): void;
deactivated(): void;
errorCaptured?(err: Error, vm: Vue, info: string): boolean | void;
serverPrefetch?(this: V): Promise<void>;
```

You can find the description of each of these methods in product documentation at [vuejs.org/v2/guide-instance.html#Lifecycle-Diagram](https://vuejs.org/v2/guide-instance.html#Lifecycle-Diagram), but we'll just say that the method `created()` fits our needs. It's invoked when the component is initialized and is ready to receive data and handle events. Lifecycle hooks are invoked by Vue, so we just need to put the data fetching code inside the method `created()`. Listing 15.13 shows the first modified version of the `Home` component.

### **Listing 15.13 Adding the created() lifecycle hook component.**

```
<template>
  <div class="home">
    <h1>I'm the Home component</h1>
  </div>
</template>

<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';

@Component
export default class Home extends Vue {

  created() { ①
    console.log("Home created!");
  }
}

</script>
```

- ① This lifecycle hook is invoked by Vue

**TIP****Tip**

The Vue router has its own lifecycle hooks and guards that allow to intercept important events during the navigation to the route. They are described in the vue-router documentation at [router.vuejs.org/guide/](https://router.vuejs.org/guide/).

Run our app, and you'll see the message "Home created!" on the browser's console. Now that we're sure that the `created()` hook is invoked, we'll make it fetching data from products as shown in listing 15.14.

### **Listing 15.14 Fetching products**

```
<template>
<div class="home">
  <h1>I'm the Home component</h1>
</div>
</template>

<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import { Product } from '@/product';      ①

@Component
export default class Home extends Vue {

  products: Product[] = [];

  created() {
    fetch("/products.json")    ②
      .then(response => response.json())      ③
      .then(json => {
        this.products = json;      ④

        console.log(this.products);    ⑤
      },
      error => {
        console.log('Error loading products.json:', error);
      });
  }
</script>
```

- ① Importing the interface Product;
- ② Initiating the data fetch using Promise
- ③ Convert the response to JSON format
- ④ Populating the products array with the data
- ⑤ Print the retrieved data on the browser console

In this version of the `Home` component we used the browser's Fetch API for reading the file `products.json` and simply print the retrieved data on the browser's console. Here, we used the Promise-based syntax, and in the `router-product-detail` app, we'll used `async` and `await` keywords, so you can compare the two.

In listing 15.14, there is an `import` statement that uses the `@`-sign as a shortcut for `./src`. This is possible because the `tsconfig.json` file specifies the `paths` option as follows:

```
"paths": {
  "@/*": [
    "src/*"
  ]
}
```

The `@`-sign can be also as a shortcut for the Vue directive `v-on` that's used for handling events. For example, instead of `<button v-on:click="doSomething()">` you can write `<button @click="doSomething()">`.

The next step is to add the `<ul>` tag to display the list of products in the `Home` component. Vue comes with a number of directives that tell the `Vue` instance what to do with the DOM element. Directives can be used in the template and look like an prefixed HTML attribute: `v-if`, `v-show`, `v-for`, `v-bind`, `v-on` and others.

We use the directive `v-for` to iterate through the `products` array rendering `<li>` for each element of the array. Vue needs to be able to track each element of the list, so you need to provide a unique key attribute for each item, and we use the `v-bind:key` directive specifying the product id as a unique key. Listing 15.15 shows the next version of the `Home` component that renders the list of products.

## Listing 15.15 Displaying the list of products in the Home component

```

<template>
  <div class="home">
    <h1>Products</h1>
    <ul id="prod">
      <li v-for="product in products"          ①
          v-bind:key="product.id"             ②
          {{ product.title }}               ③
        </li>
    </ul>
  </div>
</template>

<style>
  ul {
    text-align: left;           ④
  }
</style>

<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import { Product } from '@/product';

@Component
export default class Home extends Vue {

  products: Product[] = [];

  created() {
    fetch('/products.json')
      .then(response => response.json())
      .then(json => {
        this.products = json;
      },
      error => {
        console.log('Error loading products.json:', error);
      });
  }
}
</script>

```

- ① Iterating products with the v-for directive
- ② Assigning a unique key to each <li> element
- ③ Rendering only the product title
- ④ Aligning the text of the list elements

Running the app will render the Home component that will look as seen in figure 15.11.

## Products

- First Product
- Second Product
- Third Product

**Figure 15.11 Rendering products**

We'll implement one more feature in this app - the user should be able to select a product from the list and the app should know which one was selected. In the next version of the `Home` component we'll handle the click event and highlight the selected product with the light blue background. Listing 15.16 shows the template with added directive `v-on:click`, and we used the shortcut `@` for `v-on`.

### Listing 15.16 The new template of the Home component

```
<template>
<div class="home">
  <h1>Products</h1>
  <ul id="prod">
    <li v-for="product in products"
        v-bind:key="product.id"
        v-bind:class="{selected: product === selectedProduct}" ①
        @click = "onSelect(product)"> ②
      {{ product.title }}
    </li>
  </ul>
</div>
</template>
```

- ① Using binding, dynamically apply a different style to the selected item
- ② Call the method `onSelect` passing the selected product's data

The template in listing 15.16 has two additions. First, we've added the `v-bind` directive to bind the CSS selector `selected` to the `<li>` element that has the same value as the `class` property `selectedProduct`. Second, we've added the `click` event handler to call the method `onSelect()`, where we'll set the value for `selectedProduct`, so the binding mechanism could highlight the corresponding list item. Listing 15.17 shows the `<style>` section of the `Home` component, where the `selected` class is defined.

## Listing 15.17 The new style of the Home component

```
<style>

.home {
  display: flex;
  flex-direction: column;
}
ul {
  text-align: left;
  display: inline-block;
  align-self: start;
}
.selected {    ①
  background-color: lightblue
}
</style>
```

- ① Declaring the style for highlighting the selected product

Listing 15.18 shows the content of the `<script>` section of the `Home` component, which has the new property `selectedProduct`.

## Listing 15.18 The script section of the Home component

```
<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import { Product } from '@/product';

@Component
export default class Home extends Vue {

  products: Product[] = [];
  selectedProduct: Product | null = null;      ①

  created() {
    fetch('/products.json')
      .then(response => response.json())
      .then(json => {
        this.products = json;
      },
      error => {
        console.log('Error loading products.json:', error);
      });
  }

  onSelect(prod: Product): void {      ②
    this.selectedProduct = prod;      ③
  }
}
</script>
```

- ① The `selectedProduct` property stores the selected product
- ② The handler function for the click event
- ③ Set the value of the `selectedProduct`

We'd like you to note that type of the `selectedProduct` property of the `Home` class. We had to initialize this property, otherwise TypeScript would complain that *Property 'selectedProduct' has no initializer and is not definitely assigned in the constructor*. This check can be either disabled for the entire project in `tsconfig.json` with `strictPropertyInitialization: false`, or suppressed on the property level with the exclamation mark right after the property name:

```
`selectedProduct!: Product;`
```

Declaring this property as `selectedProduct: Product = null` wouldn't work either because TypeScript would complain that you can't assign `null` to type `Product`. That's why we explicitly allowed `selectedProduct` to be `null` by applying the union type `selectedProduct: Product | null = null`; We need to initialize `selectedProduct` with a value, because otherwise property won't exist in the generated code hence Vue won't make it reactive, and we won't be able to use it in the component's template.

Now when the user clicks on the product, the method `onSelect()` is invoked setting the value of the property `selectedProduct`, which is used in the `v-bind:class` directive for changing the CSS selector of the selected list item. Figure 15.12 shows the rendered product list with the second product selected. So, when we set the value of the `selectedProperty` the contents of the entire UL is rendered (this happens when the value of any class property changes), clearing the style on the previously selected item.

- First Product
- Second Product
- Third Product

**Figure 15.12 Second product selected**

Earlier in this chapter, you saw how a parent component can pass data to its child by using `props`, and in the next section, we'll show you how to pass data while navigating to the route.

### 15.3.3 Passing data with the Vue router

When the user navigates to a route, your app can pass the data to the destination component using the route parameters. In this section, we'll review yet another version of the app that renders the list of products, and when the user selects one, it navigates to the product details view showing the info about the selected product.

This app is located in the directory `router-product-detail`. Run `npm install` and then `npm run serve`, and the list of products will be shown on top of the window. Click on the product, and the app will navigate to the product detail. Figure 15.13 shows the screen shot taken when the user clicked on the Second Product in the list. If you read the black-and-white version of the book, the text "Second Product" is shown in green.



**Figure 15.13 Showing details for the second product**

This app has only two components: `App` and `ProductDetail`. The top part of the image is the UI of the `App` component, and the bottom part is `ProductDetails`. Note the URL's segment `/products/1`. The route that navigates to the product details view is configured as the path `'/products/:productId'` as shown in listing 15.19.

### Listing 15.19 The file router.ts

```
import Vue from 'vue';
import Router from 'vue-router';
import ProductDetails from './views/ProductDetails.vue';

Vue.use(Router);

export default new Router({
  base: process.env.BASE_URL,
  mode: 'history',
  routes: [
    {
      path: '/products/:productId',      ①
      component: ProductDetails,        ②
    },
  ],
});
```

- ① If the URL has a the word product followed by the value...
- ② ...navigate to ProductDetail passing the value as productId

It's not obvious from figure 15.13, but the product list items are represented by the HTML anchor tags, and each links has the URL that includes the selected product ID. Listing 18.20 shows the template of the `App` component that renders a link for each product.

## Listing 15.20 The template of the App component

```
<template>
  <div id="app">
    <div id="nav">
      <ul>
        <li v-for="product in products"
            v-bind:key="product.id">
          <router-link v-bind:to="/products/" + product.id"> ①
            {{ product.title }}
          </router-link>
        </li>
      </ul>
      <p>Click on a product to see details</p>
    </div>
    <router-view> ②
    </div>
  </template>
```

- ① Form a link that includes the selected product ID
- ② The ProductDetail component will be rendered here

Compare the content of the dynamically generated `<li>` element here with the version from listing 15.16. There, we would just render the text `product.title`, but now we render the following:

```
<router-link v-bind:to="/products/" + product.id">
  {{ product.title }}
</router-link>
```

The code displays the title, but add the product ID to the URL. During the compilation, Vue will replace the `<router-link>` with the regular anchor tag `<a>` and the entire list will be shown in the area identified by the `<router-view>` tag. The `<script>` section in `App.vue` just has the code to read the file `products.json` as shown in listing 15.21. We already explained this code earlier.

## Listing 15.21 The script section in App.vue

```
<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import { Product } from '@/product';

@Component
export default class App extends Vue {
  private products: Product[] = [];

  private created() {
    fetch('/products.json')
      .then((response) => response.json())
      .then(
        (data) => this.products = data,
        (error) => console.log('Error loading products.json:', error),
      );
  }
}
</script>
```

The fact that the property `product` is declared as `private` but can be used in the template anyway, shows that Vue need to improve TypeScript support. Angular wouldn't let you access private class variables from the template.

Now let's see review the code of the `ProductDetail` component that needs to extract the value of the `productId` from the router and render the product details. Listing 15.22 shows the template section of the `ProductDetails` component.

### **Listing 15.22 The template section from `ProductDetails.vue`**

```
<template>
  <div>
    <h1>Product details</h1>
    <ul v-if="product"> ①
      <li>ID: {{ product.id }}</li>
      <li>Title: {{ product.title }}</li>
      <li>Price: {{ product.price }}</li>
    </ul>
  </div>
</template>
```

- ① Conditional rendering of the `<ul>`

Here we use the directive `v-if` that allows to control rendering of the DOM element based on some condition. Here, the expression `v-if="product"` means render this `<ul>` only if the variable `product` is has a truthy value. The property `product` is declared in the class `ProductDetails` and it'll have the value only after the data for the product are fetched. For this to happen, the user has to select the product in the `App` component, and the router will navigate to `ProductDetails` passing the product id as a router parameter. Then the method `fetchProductByID()` will populate the property `product` and its info will be rendered using the component template.

Listing 15.23 shows the code of the class `ProductDetails`, which has the property `product` and three methods: `beforeRouteEnter()`, `beforeRouteUpdate()`, and `fetchProductByID()`. The first two are the router's hooks (a.k.a. navigation guards) and the last one finds the product by ID.

## Listing 15.23 The script section of ProductDetails.vue

```

<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import { Route } from 'vue-router';
import { Product } from '@/product';

@Component({
  async beforeRouteEnter(to: Route, from: Route, next: Function) { ①
    const product = await fetchProductByID(to.params.productId); ②
    next((component) => component.product = product); ③
  },
  async beforeRouteUpdate(to: Route, from: Route, next: Function) { ④
    this.product = await fetchProductByID(to.params.productId);
    next();
  },
})
export default class ProductDetails extends Vue {
  private product: Product | null = null; ⑤
}

async function fetchProductByID(id: string): Promise<Product> { ⑥
  const productId = parseInt(id, 10);
  const response = await fetch('/products.json');
  const products = await response.json();
  return products.find((p) => p.id === productId);
}
</script>

```

- ① The navigation guard beforeRouterEnter
- ② Fetch the data for the provided product id
- ③ Resolve the navigation guard
- ④ The navigation guard beforeRouterUpdate
- ⑤ Declaring the property product
- ⑥ The method to fetch the product details

The `beforeRouteEnter()` hook is called before the route that renders this component is confirmed. Vue provides three values as argument of this hook:

- `to` - the target `Route` object being navigated to
- `from` - the current `Route` being navigated from
- `next` - this function must be called to continue navigation

The argument `to` contains the `params` property that store the value of the parameter passed to the route. In our case, we used the name `productId` in the file `router.ts` hence we have to use the same name to get the value of this parameter in the destination route.

The hook `beforeRouteEnter()` does NOT have access to `this` component instance, because it has not been created yet when this guard is called. However, you can access the instance by passing a callback to `next()`. The callback will be invoked when the navigation is confirmed,

and the component instance will be passed to the callback as the argument:

```
next((component) => component.product = product);
```

Here, we initialize the property `product` of the component's instance. The hook `beforeRouteUpdate()` is invoked when the route that renders this component changes. In our app, this happens when the `ProductDetails` component is already rendered, but the user clicks on another product in the list. This hook has access to `this` component instance so we can simply assign the product's value to `this.product`, and the callback `next()` doesn't need any arguments.

For simplicity, to find the product details info, the method `fetchProductByID()` uses the Fetch API to read the entire file `products.json` and then finds just one object with the matching product ID. Here, we use `async` and `await` keywords, and you can compare this syntax with the Promise-based one shown in listing 15.14.

This concludes our introduction to the Vue library/framework, and in the next chapter, we'll create yet another version of the blockchain UI using Vue.

## 15.4 Summary

- Vue is a library that not only allows you to create UI components for rendering but also includes the client-side router for arranging user's navigation and the tooling to generate a new project as well as create dev or prod bundles for deployment.
- If you prefer developing UI components that have HTML, styles, and the code in one file, Vue will fit the bill. A single file contains three sections: `<template>` for markup, `<script>` for code, and `<style>` for CSS.
- While learning Vue is easier than React or Angular, Vue offers similar features as the other two counterparts.
- While JavaScript developers work with Vue using the object-based API, for TypeScript developers using class-based components seems more natural.
- Vue v3 is in the works and it will offer new Composition API that will allow developing functional UI components. At the time of this writing Composition API is at the Request For Comments stage but the core Vue team promises that upgrade to Vue v3 from v2 will be a simple and mostly automated process. The Composition API will be an addition to the existing object-based API.
- Similarly to React.js, Vue doesn't force you to turn an existing app into a SPA, and you can gradually introduce Vue to an exiting front-end code without the need to re-write the entire code at once.

# 16

## *Developing the blockchain client in Vue.js*

### **This chapter covers**

- The code review of the blockchain web client written using Vue.js
- How to run a Vue app that works with two servers in dev mode
- The workflow of the data starting from entering a transaction to the block generation
- How arrange the communications between the blockchain's client components

In the previous chapter, you've learned the basics of Vue, and now we'll review a new version of the blockchain app where the client portion is written in Vue. The source code of the web client is located in the directory blockchain/client, and the messaging server located in the directory blockchain/server.

The code of the server side remains the same as in chapter 14 and the functionality of this version of the blockchain app is the same as well, but the UI portion of the app was completely re-written in Vue and TypeScript.

In this chapter, we won't be reviewing the functionality of our blockchain, because it was already covered in the previous chapters, but we will review the code that's specific to the Vue library. You may want to re-read chapter 10 to refresh in memory the functionality of the blockchain client and messaging server.

Just to recap, when the user of any node clicks on the button GRNERATE BLOCK, the client's code announces the start of the mining process, which doesn't means that this node will be the first to finish the mining. Other nodes may also start mining of the the block with the same transactions, and all these the nodes will use the messaging server to exchange their longest chains to come to a consensus as to which node is the winner.

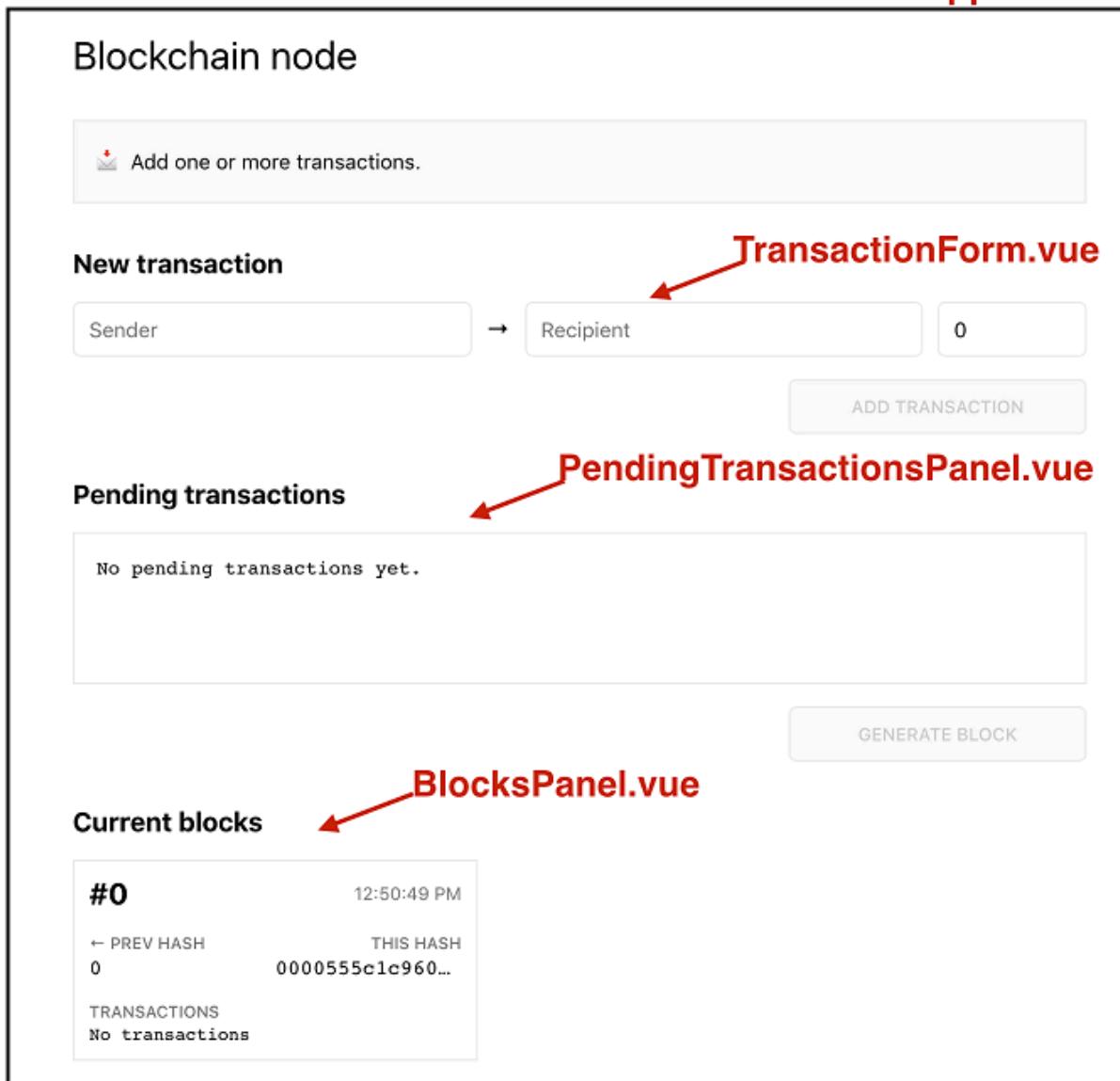
First, we'll show you how to start the messaging server and the Vue client of the blockchain app.

Then, we'll introduce the code of the class-based Vue components.

## 16.1 Starting the client and the messaging server

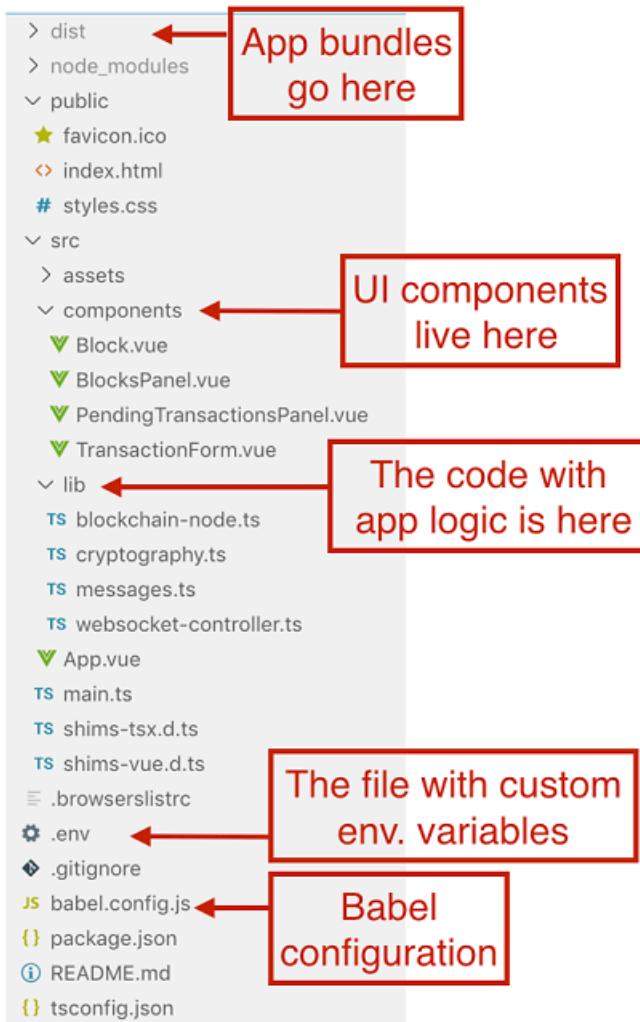
To start the server, open the Terminal window in the server directory, run `npm install` to install the server's dependencies, and then run the `npm start` command. You'll see the message *Listening on localhost:3000*. Keep the server running.

To start the Vue client, open another Terminal window in the directory client, run `npm install` to install Vue and its dependencies, and then run the `npm run serve` command. Open the browser at `localhost:8080` and you'll see a familiar blockchain web page as shown in figure 16.1. The file `App.vue` contains the code of the top-level `App` component, and the other `*.vue` files contain child components `TransactionForm`, `PendingTransactionsPanel`, and `BlocksPanel`.



**Figure 16.1** The blockchain client is launched

The client portion of this app was generated by Vue CLI, and we selected Babel, TypeScript, and class-based components from the list of the CLI options. Figure 16.2 shows the file structure of the directory *client*. UI components are located in the subdirectory *components*, and the subdirectory *lib* contains other scripts that create blockchain nodes and communicate with the messaging server. The directory *public* contains an incomplete index.html (it'll be updated during the build process) and the file styles.css that contains all the styles for this app. We didn't use the Vue router in this app.



**Figure 16.2 The project structure**

During this project generation, we selected both TypeScript and Babel. You can see that the configuration file includes the preset `@vue/app`. Also, Babel comes with a TypeScript plugin, so our app has a single compile process controlled by Babel.

You may want to consider using Babel for the following reasons:

1. **Modern mode** Basically this is feature identical to Angular's differential loading - two sets of bundles are generated: one in the ES5 format and the other - in ES2015. If the user's web browser supports the ES2015 syntax, only the corresponding bundles will be loaded.
2. **Auto-detecting polyfills** This is related to the reason 1, but it's not the same. The modern mode is about the JavaScript language features while this one is about the browser's APIs.
3. **JSX support** Without Babel we can use only HTML-based templates.

The lib directory has the code that generates new blocks, requests the longest chain, notifies the other nodes about the newly generated blocks and invites other members of the blockchain to

start generating new blocks for the specified transactions. These processes were described in chapter 10 in sections 10.1 and 10.2. Since the code in the lib directory doesn't have UI components, it remains exactly the same as in the React blockchain client from chapter 14.

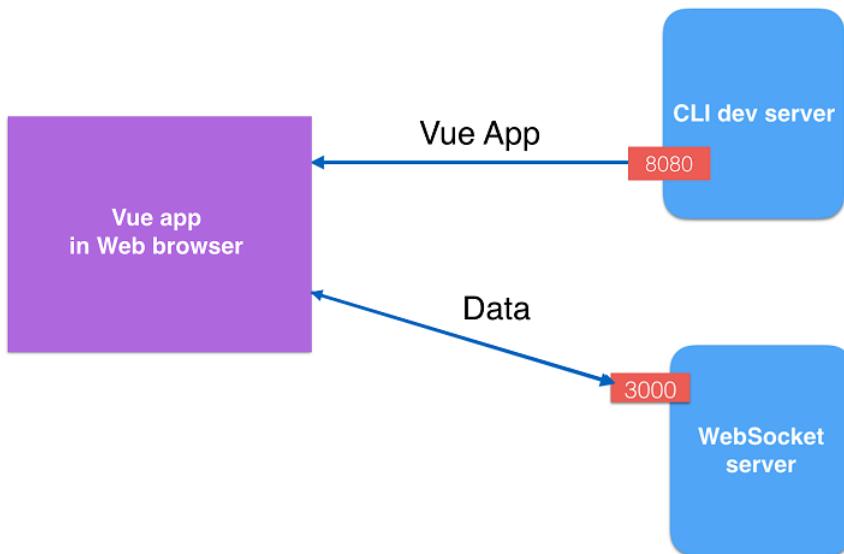
**TIP**

Compare figures 16.1 and 16.2 with figures 14.1 and 14.2 from chapter 14. You'll see that splitting the landing page of the blockchain client into UI components is done the same way in Vue and React.

When you run the CLI-generated app, it uses the script vue-cli-service, and you can find these commands in the file package.json:

```
"scripts": {
  "serve": "vue-cli-service serve",
  "build": "vue-cli-service build"
}
```

vue-cli-service always reads the file .env, which can be used for configuring custom environment variables, e.g. the host names and port numbers. Figure 16.3 illustrates proxying requests by the Webpack dev server (port 8080) to the messaging server (port 3000), and it's done in a similar way as in React and Angular. The proxy is configured in the file .env as VUE\_APP\_WS\_PROXY\_HOSTNAME=localhost:3000.



**Figure 16.3 One app, two servers**

The entry point to the blockchain client is the file main.ts, which mounts the `Vue` instance at the DOM element with the id `app`.

```
new Vue({
  render: h => h(App),
}).$mount('#app')
```

Similarly to React, Vue uses a virtual DOM and the function `render()` returns an instance of the

virtual node `vNode` that's actually a tree of `vNode` elements with a root element having the id `app`. This is where the top-level component of our app will be rendered, and we'll review its code in the next section.

## 16.2 The App component

The file `App.vue` contains the code of the class-based `App` component, and its `<template>` section contains three child components `TransactionForm`, `PendingTransactionsPanel`, and `BlocksPanel` as shown in listing 16.1.

### Listing 16.1 The script section of the `App.vue` file

```
<template>
  <main id="app">
    <h1>Blockchain node</h1>
    <aside><p>{{ status }}</p></aside>
    <section>
      <transaction-form ❶
        :disabled="shouldDisableForm()" ❷
        @add-transaction="addTransaction"> ❸
      </transaction-form>
    </section>
    <section>
      <pending-transactions-panel ❹
        :transactions="transactions()"
        :disabled="shouldDisableGeneration()"
        @generate-block="generateBlock">
      </pending-transactions-panel>
    </section>
    <section>
      <blocks-panel :blocks="blocks()"></blocks-panel> ❺
    </section>
  </main>
</template>
```

- ❶ The `TransactionForm` component
- ❷ Binding the `disabled` property
- ❸ Handling the `add-transaction` event
- ❹ The `PendingTransactionsPanel` component
- ❺ The `BlocksPanel` component

The expression `:disabled="shouldDisableForm()` is a shortcut for `v-bind:disabled="shouldDisableForm()"`, and in this context, it controls the property `disabled` of the `TransactionForm` component. The parentheses after the method name mean that we invoke the method `shouldDisableForm()` here. This method is declared in the class `App` (see listing 16.2). All methods of the component's instance can be invoked from the template.

The expression `@add-transaction="addTransaction"` tells us that the `TransactionForm` component may dispatch an event `add-transaction`, and when it happens, the `App`

component's method `addTransaction()` has to be invoked. In this case, there were no parentheses after the method name - it's just a reference to the method that might be invoked later.

**TIP**

If a component class is named using a camel-case notation, you can use it as as in the template of other component or use a dash as a separator. In listing 16.1, we used the tag `<transaction-form>` to represent the `TransactionForm` component, but we could have used the tag with a camel-case name `<TransactionForm>` as well.

The expression `@generate-block="generateBlock"` means that the `PendingTransactionsPanel` component may emit the event `generate-block`, and when it happens, the `App` component will invoke its method `generateBlock()`.

In the expression `:blocks="blocks()`, we invoke the method `block()` and whatever it returns is assigned to the property `blocks` of the `BlocksPanel` component, which is marked with the `@Props` decorator (see listing 16.7).

Listing 16.2 shows the partial code of the `<script>` section of the `App` component. We replaced the code of most of the methods with ellipses (dot-dot-dot). We didn't want to take the book space repeating the blockchain-specific code that we already explained in the previous chapters. We'll just comment some Vue-specific code.

## Listing 16.2 The script section (partial) from the App.vue file

```

<script lang="ts">
// imports are omitted for brevity
const node = new BlockchainNode();
const server = new WebsocketController();

@Component({
  components: {    ①
    BlocksPanel, PendingTransactionsPanel, TransactionForm
  }
})
export default class App extends Vue {
  status: string = '';

  blocks(): Block[] {    ②
    return node.chain;
  }

  transactions(): Transaction[] {    ②
    return node.pendingTransactions;
  }

  shouldDisableForm(): boolean {    ②
    return node.isMining || node.chainIsEmpty;
  }

  shouldDisableGeneration(): boolean {    ②
    return node.isMining || node.noPendingTransactions;
  }

  created() {    ③
    this.updateStatus();
    server
      .connect(this.handleServerMessages.bind(this))
      .then(this.initializeBlockchainNode.bind(this));
  }

  destroyed() {    ④
    server.disconnect();
  }

  updateStatus() {    ⑤
    this.status = node.chainIsEmpty ? 'Initializing the blockchain...' :
      node.isMining ? 'Mining a new block...' :
        node.noPendingTransactions ? 'Add one or more transactions.' :
          `Ready to mine a new block (transactions: ${node.pendin
  }

  async initializeBlockchainNode(): Promise<void> {...}
  addTransaction(transaction: Transaction): void {...}
  async generateBlock(): Promise<void> {...}
  async addBlock(block: Block, notifyOthers = true): Promise<void> {...}
  handleServerMessages(message: Message) {...}
  handleGetLongestChainRequest(message: Message): void {...}
  async handleNewBlockRequest(message: Message): Promise<void> {...}
  handleNewBlockAnnouncement(message: Message): void {...}
}
</script>

```

- ① Listing all child components in the decorator @Component
- ② This function is invoked from the template
- ③ The component's lifecycle callback created()

- ④ The component's lifecycle callback `destroyed()`
- ⑤ Updating the status property

The parameter of the `@Component()` decorator is an object literal, and here we use the shorthand syntax introduced by ES6. If the property value in the object literal has the same name as the property identifier, you don't have to repeat them. The long version of the object representing the child component would look like this:

```
{
  BlocksPanel: BlocksPanel,
  PendingTransactionsPanel: PendingTransactionsPanel,
  TransactionForm: TransactionForm
}
```

**TIP** In listing 16.7 in the `BlocksPanel` component, we'll use the long notation and will explain why.

Rather than declaring `node` and `server` as class properties, we keep them outside of the Vue component class to prevent Vue from augmenting objects with getters and setters required for the Vue's change detection process. We wanted to write a method that returns all nodes from the blockchain as a getter, e.g. `get blocks() { return node.chain; }`, but Vue didn't allow component templates to work with getters, so we wrote it as a class method. The same applies to several other methods of the class `App`.

The component's lifecycle hook `created()` is invoked by Vue when the data and events are ready to use, but the template is not rendered yet. In this method we connect to the messaging server providing the callback `handleServerMessages()`, and when the WebSocket connection is established, the code initializes the blockchain node, i.e. requests the longest chain (it's explained in chapter 10) and either initializes the node with existing blocks or with a genesis one.

**TIP** There is another lifecycle hook `mounted()`, which is invoked after the component's template has been rendered.

The component's lifecycle hook `destroyed()` is invoked by Vue when all internals of the component were destroyed and you just need to do some final cleanups. In our case, we disconnect from the WebSocket server to avoid having an orphan connection in memory that continues receiving the messages from other blocks. The callback `beforeDestroy()` could be an alternative place for performing some data cleanup. When `beforeDestroy()` is invoked, the component is still fully functional, and you can apply some business logic regarding the cleanup procedure.

**TIP**

Vue documentation includes a diagram illustrating all lifecycle hooks at [vuejs.org/v2/guide/instance.html#Lifecycle-Diagram](https://vuejs.org/v2/guide/instance.html#Lifecycle-Diagram).

The method `updateStatus()` is called from several other methods like `generateBlock()` or `addBlock()`. It updates the `status` property, which causes the UI re-rendering because `status` is a property of the component. Vue wraps each component property in getter/setter, which is how it knows that it's time to re-render the UI. The Vue documentation calls the properties of a component *reactive* because all of them become setters/getters and can react to changes.

**SIDE BAR****Once again about programming to interfaces**

In chapter 3, we spent some time explaining the benefits of programming to interfaces, and now we'd like to illustrate what happens when you don't. Vue has a hook called `created()`, which is a callback invoked by the `Vue` object. Try to misspell the name of this hook by adding an extra `t` as in `createtted()`. Your app won't work correctly, because the method `created()` which communicates with the messaging server and updates the class variable `status` won't exist.

If such errors only show up during the runtime, there is no benefit of using TypeScript. In this particular case, this clearly shows that the TypeScript support was added to Vue as an afterthought. What can be done differently? Let's see how the component lifecycle hooks are designed in Angular, where TypeScript was considered as a primary language from the very beginning.

Angular declares an interface for each lifecycle hook. For example, there is a interface `OnInit`, which declares one method `ngOnInit()`. If you want your component to implement this hook, you start with declaring that your class implements `OnInit`, and then write the implementation of `ngOnInit()` in that class:

**Listing 16.3 Angular does it right**

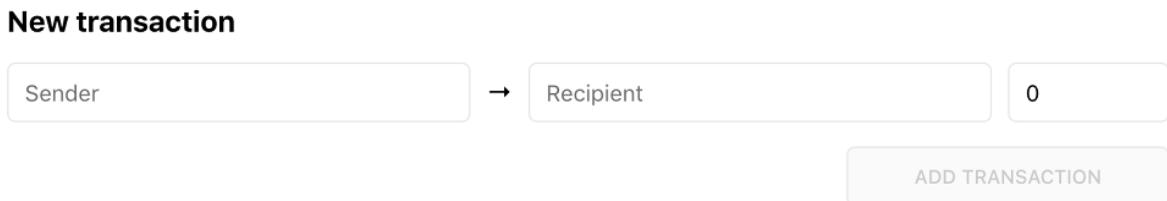
```
export class App implements OnInit {
    ngOnInit() { ... }
}
```

Try to misspell the name of the hook by adding an extra `t` as in `ngOnInitt()`. The TypeScript static code analyzer will highlight it with an error stating that you promised to implement all the methods declared in the interface `OnInit`, so where's `ngOnInit()`? Wouldn't you agree that programming to interfaces eliminates such bugs?

Now let's review the code of the child components starting from `TransactionForm`.

## 16.3 The presentation component `TransactionForm`

Figure 16.4 shows the UI of the `TransactionForm` component, which allows the user to enter the names of the sender and recipient as well as the transaction amount. When the user clicks on the button ADD TRANSACTION, this information has to be sent to the parent `App` component that's a smart component, because it knows how to process this data. This button will become enabled when the form will be filled out.



**Figure 16.4 The UI of the `TransactionForm` component**

The template of the top-level `App` component uses `TransactionForm` as follows:

```
<transaction-form
  :disabled="shouldDisableForm()"
  @add-transaction="addTransaction">
</transaction-form>
```

Listing 16.3 shows the template of the `TransactionForm`, which is an HTML form where every input field uses the `disabled` property controlled by the parent's method `shouldDisableForm()`. Re-visit listing 16.2 and you'll see that `shouldDisableForm()` returns `true` if either the node is being mined or there were no blocks in the blockchain yet.

Behind this form, there is a data model object that stores all the values entered by the user. Vue comes with the `v-model` directive that's used to create two-way data bindings between the forms elements `input`, `textarea`, and `select`. "Two-way" means that if the user enters or changes the data in the form field, the new value will be assigned to the variable, specified in `v-model` directive of this field; if the value of that variable is changed programmatically, the form field will be updated as well.

## Listing 16.4 The template of the TransactionForm component

```

<template>
  <div>
    <h2>New transaction</h2>
    <form class="add-transaction-form"
          @submit.prevent="handleFormSubmit" > ①
      <input
        type="text"
        name="sender"
        placeholder="Sender"
        autoComplete="off"
        v-model.trim="formValue.sender" ②
        :disabled="disabled"> ③

      <span class="hidden-xs"></span>

      <input
        type="text"
        name="recipient"
        placeholder="Recipient"
        autoComplete="off"
        :disabled="disabled" ④
        v-model.trim="formValue.recipient"> ④

      <input
        type="number"
        name="amount"
        placeholder="Amount"
        :disabled="disabled" ⑤
        v-model.number="formValue.amount"> ⑤

      <button type="submit"
              class="ripple"
              :disabled="!isValid() || disabled"> ⑥
        ADD TRANSACTION
      </button>
    </form>
  </div>
</template>

```

- ① Prevent the default page reload of the form's submit event
- ② `formValue.sender` is bound to this form field
- ③ The class variable `disabled` controls this field
- ④ `formValue.recipient` is bound to this form field
- ⑤ `formValue.amount` is bound to this form field
- ⑥ Conditionally enable the form's submit button

Vue offers several event modifiers, and here we use the one called `.prevent`. In Vue, the expression `@submit.prevent="handleFormSubmit"` means "Prevent the default handling of the form's button Submit. Invoke the method `handleFormSubmit()` instead."

Each input field is bound to one of the properties of the object `formValue`, which plays the role of the form model and is defined in the script section of this component as `formValue: Transaction`. The type `Transaction` is defined like this:

```
export interface Transaction {
  readonly sender: string;
  readonly recipient: string;
  readonly amount: number;
}
```

For example, the following line uses the Vue directive `v-model` to map the `sender` field of the form to the property `sender` of the `formValue` object:

```
v-model="formValue.sender"
```

But the `v-model` directive supports *modifiers*, and we wrote it as follows:

```
v-model.trim="formValue.sender"
```

The `trim` modifier automatically trims the whitespaces from the user's input. We also used the `number` modifier in `v-model.number="formValue.amount"` to ensure that the input value is automatically typecast as a number while synchronizing the value from the field `sender` with the property `formValue.sender`.

Listing 16.5 shows the `<script>` section of the file `TransactionForm.vue`. It defines and initializes the object the object `formValue` and has the `isValid()` method to check if the form is valid as well as the method `handleFormSubmit()` that is invoked when the user clicks on the button ADD TRANSACTION.

## Listing 16.5 The script section of the file TransactionForm.vue

```

<script lang="ts">
import { Component, Prop, Vue } from 'vue-property-decorator';
import { Transaction } from '../lib/blockchain-node';

@Component
export default class TransactionForm extends Vue {

    @Prop(Boolean) readonly disabled: boolean;      ①

    formValue: Transaction = this.defaultFormValue();  ②

    isValid(): boolean { ③
        return (
            this.formValue.sender &&
            this.formValue.recipient &&
            this.formValue.amount > 0
        );
    }

    handleFormSubmit(): void { ④
        this.$emit('add-transaction', { ...this.formValue });  ⑤

        this.formValue = this.defaultFormValue();  ⑥
    }

    private defaultFormValue(): Transaction { ⑦
        return {
            sender: '',
            recipient: '',
            amount: 0
        };
    }
}
</script>

```

- ① The prop value is given by the parent
- ② Initialize the form model with default values
- ③ Is the form valid?
- ④ Handle the click on the button ADD TRANSACTION
- ⑤ Emit the event on the parent
- ⑥ Reset the form
- ⑦ The default value of the form model

Here we use the decorator `@Prop` with the argument `Boolean` telling Vue to typecast the provided value (HTML data are strings) to this type.

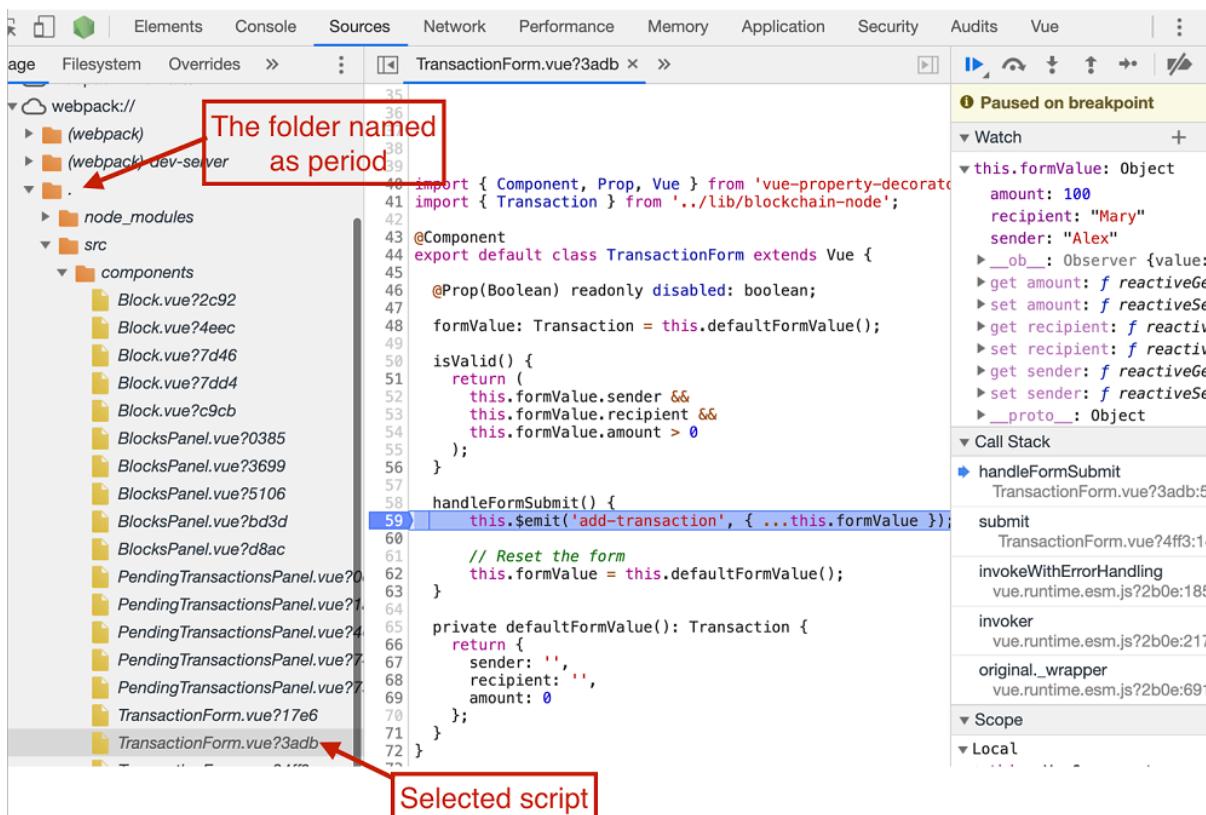
The method `isValid()` returns `true` only if the user entered all three values in the form. This will enable the form's submit button ADD TRANSACTION, and if the user clicks on it, the method `handleFormSubmit()` emits the event `add-transaction` to the parent component `App`, which will invoke its method `addTransaction()`.

A child component can send the data to its parent using the method `$emit()`, and we invoke it

with the payload `{...this.formValue}`. Here we clone the object `formValue` using the JavaScript spread operator. The method `addTransaction()` in the `App` component will receive the object of type `Transaction` and will add it to the list of pending transactions maintained by the component `PendingTransactionsPanel`.

We encourage you to run this app through the browser's debugger placing the breakpoint in the method `handleFormSubmit()` of `TransactionForm`. Figure 16.5 shows a screenshot taken after we entered Alex, Mary, and 100 as sender, recipient, and the amount and clicked on ADD TRANSACTION in the form. The Chrome's debugger stopped at the breakpoint in the method `handleFormSubmit()`. By default, the file `tsconfig.json` has the source map option turned on, so you can debug TypeScript.

To find the TypeScript sources in the debugger, open the Source panel of the Chrome Dev Tools and find the Webpack section in the left panel. Then find the image of the folder named as a period and then find the `src` subfolder there. You may see multiple files with the same name ending with the different number as shown in figure 16.5. It's because of the hot module replacement: whenever you modify a file the Webpack pushes a new version of the file but with a different name suffix, so you may need to spend a couple of seconds finding the one with TypeScript.



**Figure 16.5 Debugging TransactionForm component**

The middle section of figure 16.5 shows the breakpoint at the line 59. On the right panel, we've

added `this.formValue` to the Watch section, and you see the values 100, Mary and Alex there. Click on the Step over icon, and the debugger will take you to the method `addTransaction()` in the `App` component, and you'll see that the object with these values is received. After that, this transaction is added to the list of the Node's pending transactions as seen in listing 16.6.

### **Listing 16.6 The method addTransaction of the App component**

```
addTransaction(transaction: Transaction): void {
  node.addTransaction(transaction);
  this.updateStatus();
}
```

Invoking the method `this.updateStatus()` modifies the class variable `status`, which causes re-rendering of the component `PendingTransactionsPanel` reviewed next.

## **16.4 The presentation component PendingTransactionsPanel**

`PendingTransactionsPanel` is a presentation component, which has the prop `transactions`, and its parent `App` component provides the array of transactions as follows:

```
<pending-transactions-panel
  :transactions="transactions()" ①
  :disabled="shouldDisableGeneration()"
  @generate-block="generateBlock">
```

- ① Passing the transactions array

In the template of the `PendingTransactionsPanel`, we invoke `formattedTransactions()` (see listing 16.7), which iterates over the array `Transactions[]` and formats and renders its elements as strings in a `<pre>` element.

Actually, the `PendingTransactionsPanel` component can do one more thing - initiate the block generation when the user clicks on the button GENERATE BLOCK. Since this is a presentation component, it doesn't know how to generate block, but it can send the event `generate-block` to the parent, so this smart guy will decide what to do with it. Listing 16.7 shows the code of the `PendingTransactionsPanel` component.

## Listing 16.7 The file PendingTransactionsPanel.vue

```

<template>
  <div>
    <h2>Pending transactions</h2>
    <pre class="pending-transactions__list">{{ formattedTransactions() || 'No pending transactions yet.' }</pre> ①
    <div class="pending-transactions__form">
      <button class="ripple"
        type="button"
        :disabled="disabled"
        @click="generateBlock()"> ②
        GENERATE BLOCK
      </button>
    </div>
    <div class="clear"></div>
  </div>
</template>

<script lang="ts">
import { Component, Prop, Vue } from 'vue-property-decorator';
import { Transaction } from '@/lib/blockchain-node';

@Component
export default class PendingTransactionsPanel extends Vue {

  @Prop(Boolean) readonly disabled: boolean; ③
  @Prop({ type: Array, required: true }) readonly transactions: Transaction[]; ③

  formattedTransactions(): string { ④
    return this.transactions
      .map((t: any) => `${t.sender} ${t.recipient}: ${t.amount}`)
      .join('\n');
  }

  generateBlock(): void { ⑤
    this.$emit('generate-block');
  }
}
</script>

```

- ① Formatted transactions are shown here
- ② Click on the button GENERATE BLOCK invokes generateBlock()
- ③ The value for this prop as boolean
- ④ Formatting pending transactions
- ⑤ Emitting the generate-block event

One of the `@Prop` decorators has the following parameter: `{ type: Array, required: true }`. This is how we tell Vue to parse the provided value as an array, and the value of this prop is required.

Figure 16.6 shows the rendering of the `PendingTransactionsPanel` component with two pending transactions that came from the `TransactionForm` component.

## Pending transactions

```
Alex → Mary: $100
Yakov → Anton: $300
```

**GENERATE BLOCK**

**Figure 16.6 The UI of the PendingTransactionsPanel component**

Clicking on the button GENERATE BLOCK in the PendingTransactionsPanel has to start the process of the block generation. Since the App component has access to the array Transactions[], the method generateBlock() simply emits the event. The App component hosts PendingTransactionsPanel as follows:

```
<pending-transactions-panel
  :transactions="transactions()"
  :disabled="shouldDisableGeneration()"
  @generate-block="generateBlock">
</pending-transactions-panel>
```

When the generate-block even is dispatched the method generateBlock() is invoked, and when once again the this.updateStatus() re-renders UI and the component BlocksPanel gets the all blocks via its prop blocks. Now let's see what's going on in BlocksPanel.

## 16.5 The presentation components BlockPanel and Block

When the user clicks on the button GENERATE BLOCK in the PendingTransactionPanel component, all active nodes in the blockchain start the mining process, and after the consensus, a new block will be added to the blockchain and rendered in the BlocksPanel component, which serves as a container of child components Block. Figure 16.7 shows the rendering of the BlockPanel of a two-block blockchain.

### Current blocks

<b>#0</b> ← PREV HASH 0 THIS HASH 000044ce35f9a...	8:37:49 AM  <b>#1</b> ← PREV HASH 000044ce35f9... THIS HASH 0000e8a2330a4...
TRANSACTIONS No transactions	TRANSACTIONS Alex → Mary: \$100 Yakov → Anton: \$300

**Figure 16.7 The UI of the BlockPanel component**

During block mining and getting the consensus, the instances of `BlockchainNode` and `WebSocketController` are involved, but being a presentation component, `BlockPanel` doesn't directly communicate with either of these objects, as this work is delegated to the smart `App` component. The `BlockPanel` component doesn't send any data to its parent, and its goal is to render the blockchain provided via the prop `blocks`. The `App` component invokes its method `blocks()` and binds the returned value (i.e. the collection of existing blocks in blockchain) to the property `blocks` of the `BlocksPanel` component (the colon is for binding):

```
<blocks-panel :blocks="blocks()"></blocks-panel>
```

Listing 16.8 shows the code of the `BlocksPanel` component. Note that the declaration of the property `blocks` is decorated with the `@Props` decorator, which means that the values are coming from the parent.

### Listing 16.8 BlocksPanel.vue

```
<template>
  <div>
    <h2>Current blocks</h2>
    <div class="blocks">
      <div class="blocks__ribbon">
        <block v-for="(b, i) in blocks" ①
          :key="b.hash" ②
          :index="i" ③
          :block="b">
        </block>
      </div>
      <div class="blocks__overlay"></div>
    </div>
  </div>
</template>

<script lang="ts">
import { Component, Prop, Vue } from 'vue-property-decorator';
import { Block } from '@/lib/blockchain-node'; ④
import BlockComponent from './Block.vue'; ⑤

@Component({
  components: {
    Block: BlockComponent ⑥
  }
})
export default class BlocksPanel extends Vue {
  @Prop({ type: Array, required: true }) readonly blocks: Block[]; ⑦
}
</script>
```

- ① Iterating over the `blocks` array and rendering `BlockComponents`
- ② Assigning a unique key to each rendered block
- ③ Passing the value to the prop of the `Block` component
- ④ Importing the interface `Block`
- ⑤ Importing the component `Block`
- ⑥ Register the child `BlockComponent` under the name `Block`

## ⑦ Declaring the decorated property blocks

The presentation component `BlocksPanel` uses the Vue directive `v-for` to iterate over the `blocks` array rendering one `Block` component for each element of the array. In the React version of the app, we used the method `Array.map()` for rendering the blocks (see listing 14.22 in chapter 14). Why here we used this special attribute `v-for` of the HTML element to render blocks? The reason is that in React we used JSX, which allowed us to use the full power of JavaScript, but here the HTML-based templates only allowed using special tag attributes. In other words in React you can use JavaScript for rendering, while in Vue it's a static string.

### TIP

Vue documentation makes recommendation on how to use JSX at [vuejs.org/v2/guide/render-function.html#JSX](https://vuejs.org/v2/guide/render-function.html#JSX). Consider using JSX if you prefer JavaScript to HTML in templates.

Note that we didn't use the short object literal syntax for specifying the child component here:

```
components: {
  Block: BlockComponent
}
```

The property name on the left defines the name of the component you can use in the template, i.e. `<block>`. We had to use the long syntax here because of the name conflict. On one hand, we declared an interface named `Block` in the `lib/blockchain-node` directory, and on the other - the file `Block.vue` declares the component, which is also named `Block` as seen in listing 16.8. First, we tried the shorthand ES6 syntax for object literals:

```
components: {
  Block
}
```

Vue started complaining about not recognizing the tag `<block>`, but since we used the `default` keyword to export the class `Block`, we could import it under any arbitrary name we gave it a name `BlockComponent` as seen earlier in listing 16.8. We're saying to Vue, "We have a component called `Block`, but we imported its code under the name `BlockComponent`. Actually, a simpler solution would be changing the template tag from `<block>` to `<block-component>`, but we wanted to use this naming conflict to present a use-case when the shorthand syntax for object literals wouldn't work. Listing 16.9 shows the code of the `Block` component.

## Listing 16.9 Block.vue

```

<template>
  <div class="block">
    <div class="block__header">
      <span class="block__index">{{ index }}</span>
      <span class="block__timestamp">{{ timestamp() }}</span>
    </div>
    <div class="block__hashes">
      <div class="block__hash">
        <div class="block__label"> PREV HASH</div>
        <div class="block__hash-value">{{ block.previousHash }}</div>
      </div>
      <div class="block__hash">
        <div class="block__label">THIS HASH</div>
        <div class="block__hash-value">{{ block.hash }}</div>
      </div>
    </div>
    <div>
      <div class="block__label">TRANSACTIONS</div>
      <pre class="block__transactions">{{ formattedTransactions() || 'No transactions' }}</pre>
    </div>
  </div>
</template>

<script lang="ts">
import { Component, Prop, Vue } from 'vue-property-decorator';
import { Block as ChainBlock, Transaction } from '@/lib/blockchain-node'; ①

@Component
export default class Block extends Vue {
  @Prop(Number) readonly index: number; ②

  @Prop({ type: Object, required: true }) readonly block: ChainBlock; ③

  timestamp() {
    return new Date(this.block.timestamp).toLocaleTimeString();
  }

  formattedTransactions(): string {
    return this.block.transactions
      .map((t: Transaction) => `${t.sender} ${t.recipient}: ${t.amount}`)
      .join('\n');
  }
}
</script>

```

- ① Import with giving an alias name to Block
- ② The block's sequential number
- ③ The block's data

In listing 16.9, we also had to resolve a naming conflict between the component class `Block` and the interface with the same name. Here, we use different syntax:

```
import { Block as ChainBlock} from '@/lib/blockchain-node';
```

In this case, the interface `Block` was exported as a named export in the file `blockchain-node.ts`, so we couldn't just use any arbitrary name but had to write `import { Block as ChainBlock}` to introduce the alias name `ChainBlock`. Note the curly braces - you have to use them while

importing named exports.

The `Block` component is the simplest component in this app, and it just renders the data of one block as seen in figure 16.7. This concludes the review of the blockchain client written in Vue and TypeScript.

## 16.6 Summary

- Since Vue generates setters and getters for each component property, the process of change detection is greatly simplified. Changing a value of the component's property serves as a signal for the UI re-rendering.
- Similarly to React and Angular, the UI of a Vue app consists of smart and presentation components. Don't place the application logic in the presentation components, which are meant for presenting the data received from other components or for providing the interaction with the user with further sending the user's input to other components.
- In Vue, a parent component passes the data to its child via props. The child sends the data to its parent by emitting events with or without the payload.
- Vue CLI generates projects that use Webpack under the hood for bundling. In development, the Webpack Dev Server supports auto-recompilation and hot module replacement, which pushes the new code to the browser without reloading the page.

## 16.7 Epilogue

In this book, we showed you the main syntax constructs of TypeScript as well as multiple applications that use this language. Major web frameworks support TypeScript, and you don't need to wait for a new project to start using it as you can gradually introduce this language in the existing JavaScript projects.

As an extra bonus, we have explained the basics of the blockchain technology with multiple versions of TypeScript apps that use it.

We hope that after reading this book you understand why TypeScript is gaining popularity leaps and bounds. We believe that TypeScript's ascendancy is here to stay. Enjoy TypeScripting!

# *Modern JavaScript*



ECMAScript is a standard for scripting languages, and the evolution of ECMAScript is governed by the TC39 committee. ECMAScript syntax is implemented in several languages, and the most popular implementation is JavaScript. Starting from the sixth edition (a.k.a. ES6 or ES2015), TC39 releases the new specification of ECMAScript.

At the time of writing, the latest version of the spec is ES2018 (see [www.ecma-international.org/publications/standards/Ecma-262.htm](http://www.ecma-international.org/publications/standards/Ecma-262.htm)), but the major additions to JavaScript were introduced in ES2015 compared to its predecessor ES5, and most of the syntax covered in this appendix was introduced in the ES2015 spec.

At the time of writing, most web browsers fully support the ES2015 specification (see [mng.bz/ao59](http://mng.bz/ao59)). Even if the users of your app have older browsers, you can develop in ES6/7/8/9 today and use transpilers like TypeScript or Babel to turn the code that uses latest ECMAScript syntax into its ES5 version.

We assume that the reader know the ES5 syntax of JavaScript, and we'll cover only selected new features introduced in the ECMAScript starting from 2015.

## A.1 How to run the code samples

The code samples for this appendix come as JavaScript files with extension .js, and we'll use a Web site called CodePen (see [codepen.io](http://codepen.io)) to run code samples.

This site allows you to quickly write, test, and share apps that use HTML, CSS, and JavaScript. We'll provide CodePen links to most of code samples so you can just follow the link, see the selected code sample in action, and modify it if you choose to do so. If a code sample produces output on the console, just click the Console at the bottom of the CodePen window to see it.

Let's review some of the features of ECMAScript as they are implemented in JavaScript.

## A.2 The keywords `let` and `const`

The keywords `let` or `const` should be used as a replacement for the `var` keyword. Let's start with reviewing the issues with the `var` keyword first.

### A.2.1 The `var` keyword and hoisting

In ES5 and older versions of JavaScript you'd use the keyword `var` to declare a variable, and JavaScript engine would move the declaration to the top of the execution context (for example, a function). This is called *hoisting* (see more on hoisting at [mng.bz/3x9w](https://mng.bz/3x9w)).

Because of hoisting, if you'd declare a variable inside the code block (e.g. inside the curly braces in the `if`-statement), this variable would be visible outside of the block as well. Look at the following example where we declare the variable `i` inside the `for` loop but use it outside as well:

```
function foo() {
    for (var i=0; i<10; i++) {
    }
    console.log("i=" + i);
}
foo();
```

Running this code will print `i=10`. The variable `i` is still available outside the loop, even though it seems like it was meant to be used only inside the loop. JavaScript automatically hoists the variable declaration to the top of the function.

In the preceding example hoisting didn't cause any harm, because there was only one variable named `i`. If two variables with the same name are declared inside and outside the function, however, this may result in confusing behavior. Consider listing A.1, which declares the variable `customer` on the global scope. A bit later we'll introduce another `customer` variable in the local scope, but for now let's keep it commented out.

#### **Listing A.1 Hoisting a variable declaration**

```
var customer = "Joe";
(function () {
    console.log("The name of the customer inside the function is " + customer);
    /* if (true) {
        var customer = "Mary";
    }*/
})();
console.log("The name of the customer outside the function is " + customer);
```

The global variable `customer` is visible inside and outside the function, and running this code will print the following:

```
The name of the customer inside the function is Joe
The name of the customer outside the function is Joe
```

Uncomment the `if` statement that declares and initializes the `customer` variable inside the curly braces. Now we have two variables with the same name — one on the global scope and another on the function scope. The console output is different now:

```
The name of the customer inside the function is undefined
The name of the customer outside the function is Joe
```

The reason is that in ES5, the variable declarations are hoisted to the top of the scope (the expression within the topmost parentheses), but the variable initializations with values aren't. When a variable is created, its initial value is `undefined`. The declaration of the second `undefined` `customer` variable was hoisted to the top of the function declaration, and `console.log()` printed the value of the variable declared inside the function, which has shadowed the value of the global variable `customer`.

**NOTE** See it on CodePen: [mng.bz/cK9y](https://mng.bz/cK9y).

Function declarations are hoisted as well, so you can invoke a function before it's declared:

```
doSomething();

function doSomething() {
  console.log("I'm doing something");
}
```

On the other hand, function expressions are considered variable initializations, so they aren't hoisted. The following code snippet will produce `undefined` for the `doSomething` variable:

```
doSomething();

var doSomething = function() {
  console.log("I'm doing something");
}
```

Now let's see how the `let` or `const` keyword can help you with scoping.

## A.2.2 Block scoping with `let` and `const`

ES6 eliminates the hoisting confusion by introducing the keywords `let` and `const`. The `let` keyword is used when you need to declare a variable that may be initialized with one value and then get another value(s) assigned. The `const` keyword is used if you can assign a value to the identifier only once and it can't be reassigned afterwards.

Don't assume that `const` represents immutable values. The `const` qualifier just means that it can be initialized only once. But this doesn't mean that the property of an object assigned to a `const`

identifier can't be changed. For example, the following `const products` represents an array of objects, and you can change individual properties of these objects after initializing the `const products`:

```
const products = [
  { id: 1, description: 'Product 1' },
  {id: 2, description: 'Product 2'}
]

products[0].id = 111;
products[1].description = 'Product 222';
```

Declaring variables with the keywords `let` or `const` instead of `var` allows variables to have the block scoping. The next listing shows an example.

### **Listing A.2 Variables with block scoping**

```
let customer = "Joe";
(function () {
  console.log("The name of the customer inside the function is " + customer);
  if (true) {
    let customer = "Mary";
    console.log("The name of the customer inside the block is " + customer);
  }
})();

console.log("The name of the customer in the global scope is " + customer);
```

Now two `customer` variables have different scopes and values, and this program will print the following:

```
The name of the customer inside the function is Joe
The name of the customer inside the block is Mary
The name of the customer in the global scope is Joe
```

To put it simply, if you're developing a new application, don't use `var`. Use `let` or `const` instead.

In the preceding code sample we should have used `const` instead of `let` since we never reassigned the values for both `customer` identifiers. See it on CodePen: [mng.bz/fkJd](https://mng.bz/fkJd).

**TIP**

If you need to declare an identifier, make it a `const`. It's never too late to change it to `let` if there is a need to assign a new value to it.

## **A.3 Template literals**

Now the string literals can contain embedded expressions. This feature is known as *string interpolation*. In ES5 you'd use concatenation to create a string that contains string literals combined with the values of variables:

```
const customerName = "John Smith";
```

```
console.log("Hello" + customerName);
```

Now, you can use the template literals, which are strings surrounded with backtick symbols. You can embed expressions right inside the literal by placing them between the curly braces prefixed with a dollar sign. In the next code snippet, the value of the variable `customerName` is embedded in the string literal:

```
const customerName = "John Smith";
console.log(`Hello ${customerName}`);

function getCustomer() {
    return "Allan Lou";
}
console.log(`Hello ${getCustomer()}`);
```

The output of this code is shown here:

```
Hello John Smith
Hello Allan Lou
```

**NOTE** See it in CodePen at [mng.bz/Ey30](#).

In the preceding example we embedded the value of the variable `customerName` into the template literal and then embedded the value returned by the `getCustomer()` function. You can use any valid JavaScript expression between the curly braces.

Strings can span multiple lines in your code. Using backticks you can write multi-line strings without the need to concatenate them:

```
const message = `Please enter a password that
    has at least 8 characters and
    includes a capital letter`;

console.log(message);
```

The resulting string will treat all spaces as part of the string, so the output will look like this:

```
Please enter a password that
    has at least 8 characters and
    includes a capital letter
```

**NOTE** See it in CodePen: [mng.bz/1SSP](#).

### A.3.1 Tagged template strings

If a template string is preceded with a function name, the string is evaluated first and then passed to the function for further processing. The string parts of a template are given to the function as an array, and all the expressions that were evaluated in the template are passed as separate arguments. The syntax looks a little unusual, because you don't use parentheses as in regular function calls. In the following code snippet, the tag function `mytag` is followed by the template string:

```
mytag`Hello ${name}`;
```

The value of the variable `name` would be evaluated and provided to the function `mytag`.

Let's write a simple tagged template that would print an amount with a currency sign that depends on the `region` variable. If the value of the `region` is 1, we keep the amount unchanged and prepend it with a dollar sign. If the value of the `region` is 2, we need to convert the amount, applying 0.9 as an exchange rate, and prepend it with a euro sign. Our template string will look like this:

```
`You've earned ${region} ${amount}!`
```

Let's call the tag function `currencyAdjustment`. The tagged template string will look like this:

```
currencyAdjustment`You've earned ${region} ${amount}!`
```

Our `currencyAdjustment` function will take three arguments: the first will represent all string parts from our template string, the second will get the `region`, and the third is for the `amount`. You can add any number of arguments after the first one. The complete example follows.

```
function currencyAdjustment(stringParts, region, amount) {
    console.log( stringParts );
    console.log( region );
    console.log( amount );

    let sign;
    if (region === 1){
        sign="$"
    } else{
        sign='\u20AC'; // the euro sign
        amount=0.9*amount; // convert to euros using 0.9 as exchange rate
    }
    return `${stringParts[0]}${sign}${amount}${stringParts[2]}`;
}

const amount = 100;
const region = 2; // Europe: 2, USA: 1

const message = currencyAdjustment`You've earned ${region} ${amount}!`;
console.log(message);
```

The `currencyAdjustment` function will get a string with embedded `region` and `amount`, and it will parse the template, separating the string parts from these values (blank spaces are also

considered string parts). We'll print these values first for illustration. Then this function will check the region, apply the conversion, and return a new string template. Running the preceding code will produce the following output:

```
[ "You've earned ", " ", "!" ]
2
100
You've earned €90!
```

You can see this example in action at [codepen.io/yfain/pen/yZzOjP?editors=0011](https://codepen.io/yfain/pen/yZzOjP?editors=0011). In chapter 10 in section 10.6.2, we'll discuss the code of the web client that uses a library lit-html, which uses the tagged template strings.

## A.4 Optional parameters and default values

You can specify default values for function parameters (arguments) that will be used if no value is provided during function invocation. Say you're writing a function to calculate tax that takes two arguments: the annual income and the state where the person lives. If the state value isn't provided, we want to use Florida as a default.

In ES5 we'd need to start the function body by checking whether the state value was provided, otherwise we'd use Florida:

```
function calcTaxES5(income, state) {
    state = state || "Florida";
    console.log("ES5. Calculating tax for the resident of " + state +
               " with the income " + income);
}
calcTaxES5(50000);
```

Here's what this code prints:

```
"ES5. Calculating tax for the resident of Florida with the income 50000"
```

Starting from ES6 you can specify the default value right in the function signature:

```
function calcTaxES6(income, state = "Florida") {
    console.log("ES6. Calculating tax for the resident of " + state +
               " with the income " + income);
}
calcTaxES6(50000);
```

### NOTE

See it in CodePen: [mng.bz/U51z](https://mng.bz/U51z).

## A.5 Arrow function expressions

Arrow function expressions provide a shorter notation for anonymous functions and add lexical scope for the `this` variable. The syntax of arrow function expressions consists of arguments, the fat arrow sign `()`, and the function body. If the function body is just one expression, you don't even need curly braces. If a single-expression function returns a value, there's no need to write the `return` statement — the result is returned implicitly:

```
let sum = (arg1, arg2) => arg1 + arg2;
```

The body of a multi-line arrow function expression has to be enclosed in curly braces and use the explicit `return` statement:

```
(arg1, arg2) => {
  // do something
  return someResult;
}
```

If an arrow function doesn't have any arguments, use empty parentheses:

```
() => {
  // do something
  return someResult;
}
```

If the function has just one argument, parentheses are not mandatory:

```
arg1 => {
  // do something
}
```

In the following code snippet, we pass arrow function expressions as arguments to array's `reduce()` method to calculate a sum, and `filter()` to print even numbers:

```
const myArray = [1, 2, 3, 4, 5];

console.log( "The sum of myArray elements is " +
            myArray.reduce((a,b) => a+b)); // prints 15

console.log( "The even numbers in myArray are " +
            myArray.filter( value => value % 2 === 0)); // prints 2 4
```

Now that you're familiar with the syntax of arrow functions, let's see how they streamline working with the `this` object reference.

In ES5, figuring out which object is referred to by the `this` keyword isn't always a simple task. Search online for “JavaScript this and that” and you'll find multiple posts where people complain about `this` pointing to the “wrong” object. The `this` reference can have different values

depending on how the function is invoked and on whether strict mode was used (see the documentation for “Strict Mode” on the Mozilla Developer Network at [mng.bz/VNVL](https://mng.bz/VNVL)). We’ll illustrate the problem first, and then we’ll show you the solution offered by ES6.

Consider the code in the following listing that invokes the anonymous function every second. The function prints random generated prices for the stock symbol provided to the `StockQuoteGenerator()` constructor function.

### **Listing A.3 this points at different objects**

```
function StockQuoteGenerator(symbol){
    this.symbol = symbol;      ①
    console.log(`this.symbol=${this.symbol}`);

    setInterval( function () {
        console.log(`The price of ${this.symbol}      ②
                    is ${Math.random()}`);
    }, 1000);
}
const stockQuoteGenerator = new StockQuoteGenerator("IBM");
```

- ① `this.symbol` is a property of `StockQuoteGenerator()`
- ② `this.symbol` is undefined here

In the first occurrence, `this` was pointing at the function object and `this.symbol` had a value of IBM. In the second occurrence, because of `setInterval()` the value of `this.symbol` is undefined. You’ll see the same behavior not only if a function is invoked inside `setInterval()`, but if a function is invoked in any callback. Inside the callback, if strict mode (see [developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict\\_mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode)) is off, `this` would either point at the global object, which is not the same as `this` defined by the `StockQuoteGenerator()` constructor function. In the strict mode is on, the `this` object would be undefined.

**NOTE**

Actually, in the preceding code sample we could have just used `symbol` instead of `this.symbol`. But our goal was to show you how the `this` variable points at different object, and you can see it in CodePen: [codepen.io/yfain/pen/LreVgz?editors=0012](https://codepen.io/yfain/pen/LreVgz?editors=0012).

The other solution for ensuring that a function runs in a particular `this` object is to use the JavaScript `call()`, `apply()`, or `bind()` functions.

**NOTE**

If you’re not familiar with the `this` problem in JavaScript, check out Richard Bovell’s article, “Understand JavaScript’s ‘this’ with Clarity and Master It” at [mng.bz/ZQfz](https://mng.bz/ZQfz).

The following listing illustrates an arrow function solution that offers an unambiguous `this`. We just replaced the anonymous function given to `setInterval()` with a fat arrow function.

### **Listing A.4 fat arrow function**

```
function StockQuoteGenerator(symbol){
    this.symbol = symbol;      // this.symbol is undefined inside getQuote()
    console.log("this.symbol=" + this.symbol);

    setInterval(() =>
        console.log(`The price of ${this.symbol} is ${Math.random()}`)
    , 1000);
}
const stockQuoteGenerator = new StockQuoteGenerator("IBM");
```

The preceding code sample will properly resolve the `this` reference. An arrow function that's given as an argument to `setInterval()` uses the `this` value of the enclosing context, so it will recognize IBM as the value of `this.symbol`, and you can see it in action at [codepen.io/yfain/pen/BVJoKX?editors=0012](https://codepen.io/yfain/pen/BVJoKX?editors=0012).

## **A.6 The rest operator**

In ES5, writing a function with a variable number of parameters required using a special `arguments` object. This object is *similar* to an array, and it contains values corresponding to the arguments passed to a function.

Starting from ES6, you can use the rest operator for a variable number of arguments in a function. The ES6 rest operator is represented by three dots (...), and it has to be the last one in the arguments list. If the name of the function argument starts with the three dots, the function will get the rest of the arguments in an array.

For example, you can pass multiple customers to a function using a single variable name with a rest operator:

```
function processCustomers(...customers) {
    // implementation of the function goes here
}
```

Inside this function, you can handle the `customers` data the same way you'd handle any array. Imagine that you need to write a function to calculate taxes that must be invoked with the first argument, `income`, followed by any number of arguments representing the names of the customers. The following listing shows how you could process a variable number of arguments using first ES5 and then ES6 syntax. The `calcTaxES5()` function uses the object named `arguments`, and the function `calcTaxES6()` uses the ES6 rest operator.

## Listing A.5 rest operator

```
// ES5 and arguments object
function calcTaxES5() {

    console.log("ES5. Calculating tax for customers with the income ",
                arguments[0]); // income is the first element

    // extract an array starting from 2nd element
    var customers = [].slice.call(arguments, 1);

    customers.forEach(function (customer) {
        console.log("Processing ", customer);
    });
}

calcTaxES5(50000, "Smith", "Johnson", "McDonald");
calcTaxES5(750000, "Olson", "Clinton");

// ES6 and rest operator
function calcTaxES6(income, ...customers) {
    console.log(`ES6. Calculating tax for customers with the income ${income}`);

    customers.forEach( (customer) => console.log(`Processing ${customer}`));
}

calcTaxES6(50000, "Smith", "Johnson", "McDonald");
calcTaxES6(750000, "Olson", "Clinton");
```

Both functions, `calcTaxES5()` and `calcTaxES6()`, produce the same results:

```
ES5. Calculating tax for customers with the income 50000
Processing Smith
Processing Johnson
Processing McDonald
ES5. Calculating tax for customers with the income 750000
Processing Olson
Processing Clinton
ES6. Calculating tax for customers with the income 50000
Processing Smith
Processing Johnson
Processing McDonald
ES6. Calculating tax for customers with the income 750000
Processing Olson
Processing Clinton
```

**NOTE**

See it on CodePen: [mng.bz/l2zq](https://mng.bz/l2zq).

There's a difference in handling customers, though. Because the `arguments` object isn't a real array, we had to create an array in the ES5 version by using the `slice()` and `call()` methods to extract the names of the customers starting from the second element in `arguments`. The ES6 version doesn't require us to use these tricks because the rest operator gives you a regular array of customers. Using the rest arguments made the code simpler and more readable.

## A.7 The spread operator

The ES6 spread operator is also represented by three dots (...) like the rest operator, but whereas the rest operator can turn a variable number of parameters into an array, the spread operator can do the opposite: turn an array into a list of values or function parameters.

Say you have two arrays and you need to add the elements of the second array to the end of the first one. With the spread operator it's one line of code:

```
let array1= [...array2];
```

Here, the spread operator extracts each element of myArray and adds to the new array (the square brackets mean "create a new array" here). You can also create a copy of an array as follows:

```
array1.push(...array2);
```

Finding a maximum value in the array is also easy with the spread operator:

```
const maxValue = Math.max(...myArray);
```

In some cases, you want to clone an object. For example, you have an object that stores the state of your app and want to create a new object when one of the state properties changes. You don't want to mutate the original object but want to clone it with modification of one or more properties. One way to implement immutable objects is by using the `Object.assign()` function. The following code listing creates a clone of the object first and then creates another clone with changing the `lastName` at the same time.

### **Listing A.6 Clone with `assign()`**

```
// Clone with Object.assign()
const myObject = {name: "Mary" , lastName: "Smith"};
const clone = Object.assign({}, myObject);
console.log(clone);

// Clone with modifying the `lastName` property
const cloneModified = Object.assign({}, myObject, {lastName: "Lee"});
console.log(cloneModified);
```

The spread operator offers a more concise syntax for achieving the same goals, as you can see in the following listing.

## Listing A.7 Clone with spread

```
// Clone with spread
const myObject = { name: "Mary" , lastName: "Smith"};
const cloneSpread = {...myObject};
console.log(cloneSpread);

// Clone with modifying the `lastName`
const cloneSpreadModified = {...myObject, lastName: "Lee"};
console.log(cloneSpreadModified);
```

Our `myObject` has two properties, `name` and `lastName`. The line that clones `myObject` with modification of the `lastName` will still work even if you or someone else will add more properties to `myObject`.

**NOTE**

See it in CodePen: [mng.bz/X2pL](https://mng.bz/X2pL).

Cloning with `Object.assign()` as well as with the spread operator create a shallow copy of the object. It copies all the properties values that the object has at the time of cloning. But if some of the properties of an object are also objects, only the references to the nested ones are going to be copied. If after a shallow cloning the values of the nested properties will change in the original object, the clone will get the same changes.

The following listing shows an object that has a nested object `birth`. Initially, the birth date is 18 Jan 2019. After cloning, the clone object will have the same birth date. But if you change the birth date on the original object, the clone will get the new value as well. This proves that the only the reference to the nested object was copied, but not its values.

## Listing A.8 Shallow cloning

```
const myObject = { name: 'Mary' , lastName: 'Smith', birth: { date: '18 Jan 2019' } };
const clone = {...myObject};      ①
console.log(clone.birth.date);    ②
myObject.birth.date = '20 Jan 2019'; ③
console.log(clone.birth.date);    ④
```

- ① Cloning `myObject`
- ② The clone's birth date is 18 Jan 2019
- ③ Changing the birth date on the original object
- ④ The clone's birth date changed to 20 Jan 2019

## A.8 Destructuring

Creating instances of objects means constructing them in memory. The term *destructuring* means changing the structure or taking objects apart. In ES5 you could deconstruct any object or a collection by writing a function to do it. ES6 introduced the destructuring assignment syntax that allowed you to extract data from an object's properties or an array in a simple expression by specifying a *matching pattern*. It's easier to explain by example, which we'll do next.

### A.8.1 Destructuring objects

Let's say that a `getStock()` function returns a `Stock` object that has the attributes `symbol` and `price`. In ES5, if you wanted to assign the values of these attributes to separate variables, you'd need to create a variable to store the `Stock` object first and then write two statements assigning the object attributes to corresponding variables:

```
var stock = getStock();
var symbol = stock.symbol;
var price = stock.price;
```

Starting in ES6 you just need to write a matching pattern on the left and assign the `Stock` object to it:

```
let {symbol, price} = getStock();
```

It's a little unusual to see curly braces on the left of the equal sign, but this is part of the syntax of a matching expression. When you see curly braces on the left side, think of them as a block of code and not the object literal.

The following listing demonstrates getting the `Stock` object from the `getStock()` function and destructuring it into two variables.

#### **Listing A.9 Destructuring an object**

```
function getStock() {
  return {
    symbol: "IBM",
    price: 100.00
  };
}

let {symbol, price} = getStock();

console.log(`The price of ${symbol} is ${price}`);
```

Running that script will print the following:

```
The price of IBM is 100
```

In other words, we bind a set of data (object properties, in this case) to a set of variables (`symbol`

and `price`) in one assignment expression. Even if the `stock` object had had more than two properties, the preceding destructuring expression would still work because `symbol` and `price` would have matched the pattern. The matching expression lists only the variables for the object attributes you’re interested in.

**NOTE**

See in in CodePen: [mng.bz/C147](https://mng.bz/C147).

You can also destructure nested objects. The next code listing creates a nested object that represents Microsoft stock and passes it to the `printStockInfo()` function, which pulls the stock symbol and name of the stock exchange from this object.

### **Listing A.10 Destructuring a nested object**

```
const msft = {
  symbol: "MSFT",
  lastPrice: 50.00,
  exchange: {①
    name: "NASDAQ",
    tradingHours: "9:30am-4pm"
  }
};

function printStockInfo(stock) {
  let {symbol, exchange: {name}} = stock; ②
  console.log(`The ${symbol} stock is traded at ${name}`);
}

printStockInfo(msft);
```

<sup>①</sup> The nested object

<sup>②</sup> Destructuring a nested object to get the name of the stock exchange

Running the preceding script will print the following:

```
The MSFT stock is traded at NASDAQ
```

**NOTE**

See it in CodePen: [mng.bz/Xauq](https://mng.bz/Xauq).

Say you’re writing a function to handle a browser DOM event. In the HTML part, you invoke this function, passing the event object as an argument. The event object has multiple properties, but your handler function only needs the `target` property to identify the component that dispatched this event. The destructuring syntax makes it easy:

```
<button id="myButton">Click me</button>
...
document
  .getElementById("myButton")
  .addEventListener("click", ({target}) =>
    console.log(target));
```

Note the destructuring syntax `{target}` in the function argument.

**NOTE** See it on CodePen: [mng.bz/Dj24](https://mng.bz/Dj24).

Starting from ES2018, you can use the syntax similar to rest and spread operators while destructuring objects. For example, the following code will assign the value of 50 to the variable `lastPrice` and the rest of the `msft` object properties will be placed in the object `otherInfo`.

### Listing A.11 Combining destructuring and the rest operator

```
const msft = {
    symbol: "MSFT",
    lastPrice: 50.00,
    exchange: {
        name: "NASDAQ",
        tradingHours: "9:30am-4pm"
    }
};
const { lastPrice, ...otherInfo } = msft; ①

console.log(`lastPrice= ${lastPrice}`);
console.log(`otherInfo=`, otherInfo);
```

① Destructuring and the rest operator

You can see it in action at [codepen.io/yfain/pen/rZJGeK?editors=0011](https://codepen.io/yfain/pen/rZJGeK?editors=0011)

## A.8.2 Destructuring arrays

Array destructuring works much like object destructuring, but instead of curly brackets you'll need to use square ones. Whereas in destructuring objects you need to specify variables that match object properties, with arrays you specify variables that match arrays' indexes. The following code extracts the values of two array elements into two variables:

```
let [name1, name2] = ["Smith", "Clinton"];
console.log(`name1 = ${name1}, name2 = ${name2}`);
```

The output will look like this:

```
name1 = Smith, name2 = Clinton
```

If you just wanted to extract the second element of this array, the matching pattern would look like this:

```
let [, name2] = ["Smith", "Clinton"];
```

If a function returns an array, the destructuring syntax turns it into a function with a multiple-value return, as shown in the `getCustomers()` function:

```
function getCustomers() {
    return ["Smith", "Clinton", "Lou", "Gonzales"];
}

let [firstCustomer, secondCustomer, ...lastCustomer] = getCustomers();
console.log(`The first customer is ${firstCustomer} and the last one is ${lastCustomer}`);
```

Now let's combine array destructuring with rest parameters. Let's say we have an array of multiple customers but we want to process only the first two. The following code snippet shows how to do it:

```
let customers = ["Smith", "Clinton", "Lou", "Gonzales"];

let [firstCust, secondCust, ...otherCust] = customers;

console.log(`The first customer is ${firstCust} and the second one is ${secondCust}`);
console.log(`Other customers are ${otherCust}`);
```

Here's the console output produced by that code:

```
The first customer is Smith and the second one is Clinton
Other customers are Lou, Gonzales
```

On a similar note, you can pass the matching pattern with a rest parameter to a function:

```
var customers = ["Smith", "Clinton", "Lou", "Gonzales"];

function processFirstTwoCustomers([firstCust, secondCust, ...otherCust]) {

    console.log(`The first customer is ${firstCust} and the second one is ${secondCust}`);
    console.log(`Other customers are ${otherCust}`);
}

processFirstTwoCustomers(customers);
```

The output will be the same:

```
The first customer is Smith and the second one is Clinton
Other customers are Lou, Gonzales
```

To summarize, the benefit of destructuring is that you can write less code when you need to initialize some variables with data that's located in object properties or arrays.

## A.9 Classes and inheritance

Although ES5 supports object-oriented programming and inheritance, with ES6 classes the code is easier to read and write.

In ES5, objects can be created either from scratch or by inheriting from other objects. By default, all JavaScript objects are inherited from `Object`. This object inheritance is implemented via a

special property called `prototype`, which points at this object's ancestor. This is called *prototypal inheritance*. In ES5, to create an `NJTax` object that inherits from the object `Tax`, you can write something like this:

```
function Tax() {
    // The code of the tax object goes here
}

function NJTax() {
    // The code of New Jersey tax object goes here
}

NJTax.prototype = new Tax(); ①

var njTax = new NJTax();
```

### ① Inherits NJTax from Tax

ES6 introduced the keywords `class` and `extends` to bring the syntax in line with other object-oriented languages such as Java and C#. The ES6 equivalent of the preceding code is shown next:

```
class Tax {
    // The code of the tax class goes here
}

class NJTax extends Tax {
    // The code of New Jersey tax object goes here
}

let njTax = new NJTax();
```

The `Tax` class is an ancestor or *superclass*, and `NJTax` is a descendant or *subclass*. You can also say that the `NJTax` class has the “is a” relation with the class `Tax`. In other words, `NJTax` is a `Tax`. You can implement additional functionality in `NJTax`, but `NJTax` still “is a” or “is a kind of” `Tax`. Similarly, if you create an `Employee` class that inherits from `Person`, you can say that `Employee` is a `Person`.

You can create one or more instances of the objects, like this:

```
var tax1 = new Tax(); ①
var tax2 = new Tax(); ②
```

- ① First instance of the `Tax` object
- ② Second instance of the `Tax` object

**NOTE** In contrast to function declarations, class declarations aren't hoisted. You need to declare the class before you use it otherwise you'll get a `ReferenceError`.

Each of these objects will have properties and methods that exist in the `Tax` class, but they will have different *state*; for example, the first instance could be created for a customer with an annual income of \$50,000, and the second for a customer who earned \$75,000. Each instance would share the same copy of the methods declared in the `Tax` class, so there's no duplication of code.

In ES5, you can also avoid code duplication by declaring methods not inside the objects but on their prototypes:

```
function Tax() {
    // The code of the tax object goes here
}

Tax.prototype = {
    calcTax: function() {
        // code to calculate tax goes here
    }
}
```

JavaScript remains a language with prototypal inheritance, but ES6 allows you to write more elegant code:

```
class Tax() {

    calcTax() {
        // code to calculate tax goes here
    }
}
```

#### SIDE BAR Class member variables aren't supported

JavaScript doesn't allow you to declare class member variables (a.k.a. class properties), as you can in Java, C#, or TypeScript. You'll see how to declare member variables in TypeScript classes in section "Using classes as custom types" in chapter 1.

### A.9.1 Constructors

During instantiation, classes execute the code placed in special methods called *constructors*. In languages like Java and C#, the name of the constructor must be the same as the name of the class; but in JavaScript, you specify the class's constructor by using the `constructor` keyword:

```
class Tax {

    constructor(income) {
        this.income = income;
    }
}

const myTax = new Tax(50000);
```

A constructor is a special method that's executed only once: when the object is created. The class

Tax doesn't declare a separate class-level `income` variable, but creates it dynamically on the `this` object, initializing `this.income` with the values of the constructor's argument. The `this` variable points at the instance of the current object.

The next example shows how you can create an instance of an `NJTax` subclass, providing the income of 50,000 to its constructor:

```
class Tax {
    constructor(income) {
        this.income = income;
    }
}

class NJTax extends Tax {
    // The code specific to New Jersey tax goes here
}

const njTax = new NJTax(50000);

console.log(`The income in njTax instance is ${njTax.income}`);
```

The output of this code snippet is as follows:

```
The income in njTax instance is 50000
```

Because the `NJTax` subclass doesn't define its own constructor, the one from the `Tax` superclass is automatically invoked during the instantiation of `NJTax`. This wouldn't be the case if a subclass defined its own constructor. You'll see such an example in the next section.

#### NOTE

JavaScript classes are just syntactic sugar that increases code readability. Under the hood, JavaScript still uses prototypal inheritance, which allows you to replace the ancestor dynamically at runtime, whereas a class can have only one direct ancestor. Try to avoid creating deep inheritance hierarchies, because they reduce the flexibility of your code and complicate refactoring if it's needed.

## A.9.2 The super keyword and the super function

The `super()` function allows a subclass (descendant) to invoke a constructor from a superclass (ancestor). The `super` keyword is used to call a method defined in a superclass. The following listing illustrates both `super()` and `super`. The `Tax` class has a `calculateFederalTax()` method, and its `NJTax` subclass adds the `calculateStateTax()` method. Both of these classes have their own versions of the `calcMinTax()` method.

## Listing A.12 `super()` and `super`

```

class Tax {
    constructor(income) {
        this.income = income;
    }

    calculateFederalTax() {
        console.log(`Calculating federal tax for income ${this.income}`);
    }

    calcMinTax() {
        console.log("In Tax. Calculating min tax");
        return 123;
    }
}

class NJTax extends Tax {
    constructor(income, stateTaxPercent) {
        super(income);
        this.stateTaxPercent = stateTaxPercent;
    }

    calculateStateTax() {
        console.log(`Calculating state tax for income ${this.income}`);
    }

    calcMinTax() {
        let minTax = super.calcMinTax();
        console.log(`In NJTax. Will adjust min tax of ${minTax}`);
    }
}

const theTax = new NJTax(50000, 6);

theTax.calculateFederalTax();
theTax.calculateStateTax();

theTax.calcMinTax();

```

Running this code produces the following output:

```

Calculating federal tax for income 50000
Calculating state tax for income 50000
In Tax. Calculating min tax
In NJTax. Will adjust min tax of 123

```

**NOTE**

See it in CodePen: [mng.bz/6e9S](https://mng.bz/6e9S).

The `NJTax` class has its own explicitly defined constructor with two arguments, `income` and `stateTaxPercent`, which you provide while instantiating `NJTax`. To make sure the constructor of `Tax` is invoked (it sets the `income` attribute on the object), you explicitly call it from the subclass's constructor: `super(income)`. Without this line, the preceding script would report an error; you must call the constructor of a superclass from the derived constructor by calling the function `super()`.

The other way of invoking code in superclasses is by using the `super` keyword. Both `Tax` and

NJTax have the `calcMinTax()` methods. The one in the `Tax` superclass calculates the base minimum amount according to federal tax laws, whereas the subclass's version of this method uses the base value and adjusts it. Both methods have the same signature, so you have a case for *method overriding*.

By calling `super.calcMinTax()`, you ensure that the base federal tax is taken into account for calculating state tax. If you didn't call `super.calcMinTax()`, the subclass's version of the `calcMinTax()` method would apply. Method overriding is often used to replace the functionality of the method in the superclass without changing its code.

### A.9.3 Static class members

If you need a class property that's shared by multiple class instances, you have to create using the `static` keyword. Such a property will be created once on any particular instance, but on the class itself.

In the following listing, the static variable `counter` is visible from both instances of the object `A` by invoking the method `printCounter()`. But if you try to access the variable `counter` on the instance level, it'll be undefined.

#### **Listing A.13 Sharing a class property**

```
class A {
    static counter = 0;      ①

    printCounter(){
        console.log("static counter=" + A.counter); ②
    };
}

const a1 = new A();      ③
A.counter++;            ④
a1.printCounter();     // prints 1

A.counter++;            ⑤

const a2 = new A();      ⑥
a2.printCounter();     // prints 2

console.log("On the a1 instance, counter is " + a1.counter);
console.log("On the a2 instance, counter is " + a2.counter);
```

- ① Declaring a static property
- ② Referring to a static property by class name
- ③ Creating the first instance of the class A
- ④ Incrementing the static counter
- ⑤ Incrementing the static counter
- ⑥ Creating the second instance of the class A

In this code sample, we increment the counter outside the class instances by using the class names as a reference, i.e. `A.counter`. Both instances of the class `A` see the same value of the counter.

Note that even if we invoke the method `printCounter` on a particular instance, it still refers to the static property using the class name.

That code produces this output:

```
static counter=1
static counter=2
On the a1 instance, counter is undefined
On the a2 instance, counter is undefined
```

In the last two lines of this sample we try to access the property `counter` using the instance references `a1` and `a2`, but there is no such a property on any of the instances hence they are undefined.

**NOTE**

See it in CodePen: [codepen.io/yfain/pen/VGBjvR?editors=0012](https://codepen.io/yfain/pen/VGBjvR?editors=0012).

You can also create a static method by using the keyword `static`. Static methods are also invoked on not on the instance of the class, but on the class itself. We often use static methods in a class that serves as a collection of utility functions and no instantiation is needed.

```
class Helper {

    static convertDollarsToEuros() { ①
        console.log("Converting dollars to euros");
    }

    static convertCelciusToFahrenheit() { ②
        console.log("Converting Celcius to Fahrenheit");
    }
}

Helper.convertDollarsToEuros(); ③
Helper.convertCelciusToFahrenheit(); ③
```

- ① Declare the first static method
- ② Declare the second static method
- ③ Invoke the static method without instantiating the class

See it in CodePen: [codepen.io/yfain/pen/VGBjRL?editors=0012](https://codepen.io/yfain/pen/VGBjRL?editors=0012)

In chapter 2, you'll see a practical use of the static class members in listing 2.2 where we'll implement the singleton design pattern.

## A.10 Asynchronous processing

To arrange asynchronous processing in ES5, you had to use *callbacks*, functions that are given as arguments to another function for invocation. Callbacks can be called synchronously or asynchronously.

For example, you can pass a callback to the array's method `forEach()` for synchronous invocation. In making AJAX requests to the server, you pass a callback function to be invoked asynchronously when the result arrives from the server.

### A.10.1 A callback hell

Let's consider an example of getting data about some ordered products from the server. It starts with an asynchronous call to the server to get the information about the customers, and then for each customer you'll need to make another call to get the orders. For each order, you need to get products, and the final call will get the product details.

In asynchronous processing, you don't know when each of these operations will complete, so you need to write callback functions that are invoked when the previous one is complete. Let's use the `setTimeout()` function to emulate delays, as if each operation requires one second to complete. Figure A.1 shows how this code may look like.

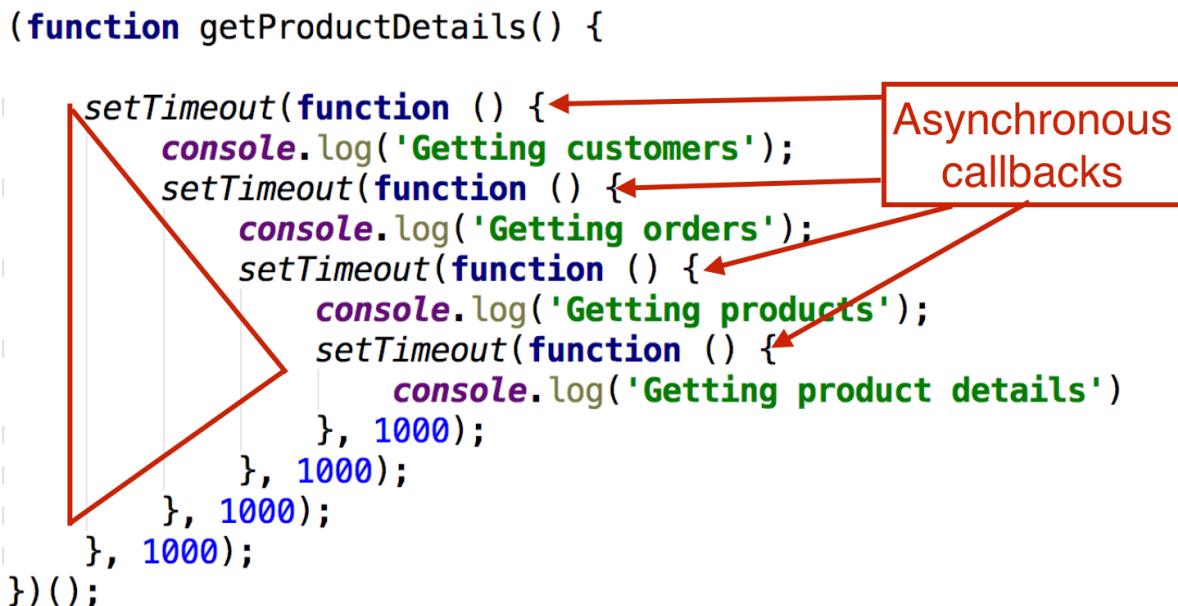


Figure A.1 Callback hell or pyramid of doom

**NOTE** Using callbacks is considered an anti-pattern, also known as Pyramid of Doom, as seen in figure A.1 on the left. In our code sample we had four callbacks, and this level of nesting makes the code hard to read. In real-world apps, the pyramid may quickly grow, making the code very hard to read and debug.

Running this code will print the following messages with one-second delays:

```
Getting customers
Getting orders
Getting products
Getting product details
```

**NOTE** See it in CodePen: [mng.bz/DAX5](https://mng.bz/DAX5).

## A.10.2 Promises

When you press the button on your coffee machine, you don't get a cup of coffee that very second. You get a promise that you'll get a cup of coffee sometime later. If you didn't forget to provide the water and the ground coffee, the promise will be *resolved*, and you can enjoy the coffee in a minute or so. If your coffee machine is out of water or coffee, the promise will be *rejected*. The entire process is asynchronous, and you can do other things while your coffee is being brewed.

JavaScript promises allow you to avoid nested calls and make the async code more readable. The `Promise` object represents an eventual completion or failure of an async operation. After the `Promise` object is created, it waits and listens for the result of an asynchronous operation and lets you know if it succeeded or failed so you can proceed with the next steps accordingly. The `Promise` object represents the future result of an operation, and it can be in one of these states:

- *Fulfilled*—The operation successfully completed.
- *Rejected*—The operation failed and returned an error.
- *Pending*—The operation is in progress, neither fulfilled nor rejected.

You can instantiate a `Promise` object by providing two functions to its constructor: the function to call if the operation is fulfilled, and the function to call if the operation is rejected. Consider a script with a `getCustomers()` function, shown in the following listing.

## Listing A.14 Using a promise

```

function getCustomers() {

    return new Promise(
        function (resolve, reject) {

            console.log("Getting customers");
            // Emulate an async server call here
            setTimeout(function() {
                var success = true;
                if (success) {
                    resolve("John Smith");      ①
                } else {
                    reject("Can't get customers"); ②
                }
            }, 1000);

        }
    );
}

getCustomers()
    .then((cust) => console.log(cust))      ③
    .catch((err) => console.log(err));       ④
console.log("Invoked getCustomers. Waiting for results");

```

- ① Got the customer
- ② Is invoked if an error occurs
- ③ Is invoked when the promise is fulfilled
- ④ Is invoked if the promise is rejected

The `getCustomers()` function returns a `Promise` object, which is instantiated with a function that has `resolve` and `reject` as the constructor's arguments. In the code, you invoke `resolve()` if you receive the customer information. For simplicity, `setTimeout()` emulates an asynchronous call that lasts one second. You also hard-code the `success` flag to be true. In a real-world scenario, you could make a request with the `XMLHttpRequest` object and invoke `resolve()` if the result was successfully retrieved or `reject()` if an error occurred.

At the bottom of the preceding listing, you attach `then()` and `catch()` methods to the `Promise()` instance. Only one of these two will be invoked. When you call `resolve("John Smith")` from inside the function, it results in the invocation of the `then()` that received "John Smith" as its argument. If you changed the value of `success` to `false`, the method `catch()` would be called with the argument containing "Can't get customers":

```

Getting customers
Invoked getCustomers. Waiting for results
John Smith

```

Note that the message "Invoked `getCustomers`. Waiting for results" is printed before "John Smith". This proves that the `getCustomers()` function worked asynchronously.

**NOTE**

See it in CodePen: [mng.bz/5rf3](https://mng.bz/5rf3).

Each promise represents one asynchronous operation, and you can chain them to guarantee a particular order of execution. Let's add a `getOrders()` function in the following listing that can find the orders for a provided customer, and chain `getOrders()` with `getCustomers()`.

### **Listing A.15 Chaining promises**

```
function getCustomers() {
    return new Promise(
        function (resolve, reject) {
            console.log("Getting customers");
            // Emulate an async server call here
            setTimeout(function() {
                const success = true;
                if (success){
                    resolve("John Smith");          ①
                }else{
                    reject("Can't get customers");
                }
            }, 1000);
        }
    );
    return promise;
}

function getOrders(customer) {
    return new Promise(
        function (resolve, reject) {
            // Emulate an async server call here
            setTimeout(function() {
                const success = true;
                if (success) {
                    resolve(`Found the order 123 for ${customer}`);      ②
                } else {
                    reject("Can't get orders");
                }
            }, 1000);
        }
    );
}
getCustomers()
    .then((cust) => {
        console.log(cust);
        return cust;
    })
    .then((cust) => getOrders(cust))      ③
    .then((order) => console.log(order))
    .catch((err) => console.error(err));  ④
console.log("Chained getCustomers and getOrders. Waiting for results");
```

- ① Invoke when the customer is successfully obtained.
- ② Invoke when the order for a customer is successful.
- ③ Chain with `getOrders()`.

④ Handle errors.

This code not only declares and chains two functions but also demonstrates how you can print intermediate results on the console. The output of listing A.15 follows (note that the customer returned from `getCustomers()` was properly passed to `getOrders()`):

```
Getting customers
Chained getCustomers and getOrders. Waiting for results
John Smith
Found the order 123 for John Smith
```

**NOTE** See it in CodePen: [mng.bz/6z5k](https://mng.bz/6z5k).

You can chain multiple function calls using `then()` and have just one error-handling script for all chained invocations. If an error occurs, it will be propagated through the entire chain of thens until it finds an error handler. No thens will be invoked after the error.

Changing the value of the `success` variable to `false` in the preceding listing will result in printing the message “Can’t get customers”, and the `getOrders()` method won’t be called. If you remove these console prints, the code that retrieves customers and orders looks clean and is easy to understand:

```
getCustomers()
  .then((cust) => getOrders(cust))
  .catch((err) => console.error(err));
```

Adding more thens doesn’t make this code less readable (compare it with the pyramid of doom shown in figure A.1).

### A.10.3 Resolving several promises at once

Another case to consider is asynchronous functions that don’t depend on each other. Say you need to invoke two functions in no particular order, but you need to perform some action only after both of them are complete. The `Promise` object has an `all()` method that takes an iterable collection of promises and executes (resolves) all of them. Because the `all()` method returns a `Promise` object, you can add `then()` or `catch()` (or both) to the result.

Imagine a web portal that needs to make several asynchronous calls to get the weather, stock market news, and traffic information. If you want to display the portal page only after all of these calls have completed, `Promise.all()` is what you need:

```
Promise.all([getWeather(),
            getStockMarketNews(),
            getTraffic()])
  .then( (results) => { /* render the portal GUI here */ })
  .catch(err => console.error(err)) ;
```

Keep in mind that `Promise.all()` resolves only after all of the promises resolve. If one of them rejects, the control goes to the `catch()` handler.

Compared to callback functions, promises make your code more linear and easier to read, and they represent multiple states of an application. On the negative side, promises can't be cancelled. Imagine an impatient user who clicks a button several times to get some data from the server. Each click creates a promise and initiates an HTTP request. There's no way to keep only the last request and cancel the uncompleted ones.

The JavaScript code with promises is easier to read, but if you look at the `then()` function carefully, you still have to provide a callback function that will be called some time later. The keyword `async` and `await` are the next step in the evolution of the JavaScript syntax for asynchronous programming.

#### A.10.4 `async-await`

The keywords `async` and `await` were introduced in ES8 (a.k.a. ES2017). They allow you to treat functions returning promises as if they're synchronous. The next line of code is executed only when the previous one completes. It's important to note that the waiting for the asynchronous code to complete happens in the background and doesn't block the execution of other parts of the program:

- `async` is a keyword that marks an asynchronous function
- `await` is a keyword that you place right before the invocation of the `async` function. This instructs the JavaScript engine to not proceed to the next line until the asynchronous function either returns the result or throws an error. The JavaScript engine will internally wrap the expression on the right of the `await` keyword into a promise and the rest of the method into a `then()` callback.

To illustrate the use of `async` and `await` keywords, the following listing reuses the functions `getCustomers()` and `getOrders()` that use promises inside to emulate asynchronous processing.

## Listing A.16 Declaring two functions that use promises

```

function getCustomers() {

    return new Promise(
        function (resolve, reject) {

            console.log("Getting customers");
            // Emulate an async call that takes 1 second to complete
            setTimeout(function() {
                const success = true;
                if (success){
                    resolve("John Smith");
                } else {
                    reject("Can't get customers");
                }
            }, 1000);
        );
    }
}

function getOrders(customer) {

    return new Promise(
        function (resolve, reject) {

            // Emulate an async call that takes 1 second
            setTimeout(function() {
                const success = true; // change it to false

                if (success){
                    resolve(`Found the order 123 for ${customer}`);
                } else {
                    reject(`getOrders() has thrown an error for ${customer}`);
                }
            }, 1000);
        );
}
}

```

We want to chain these function calls, but this time we won't be using the `then()` calls as we did with promises. We'll create a new function `getCustomerOrders()` that internally invokes `getCustomers()`, and when it completes, `getOrders()`.

We'll use the keyword `await` in the lines where we invoke `getCustomers()` and `getOrders()` so the code would wait for each of these functions to complete before continuing execution. We'll mark the function `getCustomerOrders()` with the keyword `async` because it'll use `await` inside. The following listing declares and invokes the function `getCustomerOrders()`.

### Listing A.17 Declaring and invoking an `async` function

```
(async function getCustomersOrders() {  
    try {  
        const customer = await getCustomers();  
        console.log(`Got customer ${customer}`);  
        const orders = await getOrders(customer);  
        console.log(orders);  
    } catch(err) {  
        console.log(err);  
    }  
}());  
  
console.log("This is the last line in the app. Chained getCustomers() and getOrders() are still running");  
⑤
```

- ① Declaring the function with the `async` keyword
- ② Invoking the asynchronous function `getCustomers()` with `await` so the code below won't be executed until the function completes
- ③ Invoking the asynchronous function `getOrders()` with `await` so the code below won't be executed until the function completes
- ④ Handling errors
- ⑤ This code runs outside of the `async` function.

As you see, this code looks as if it's synchronous. It has no callbacks and is executed line by line. Error processing is done in a standard way using the `try/catch` block.

Running this code will produce the following output:

```
Getting customers  
This is the last line in the app. Chained getCustomers() and getOrders() are still running without block  
Got customer John Smith  
Found the order 123 for John Smith
```

Note the message about the last line of code is printed before the name of the customer and the order number. Even though these values are retrieved asynchronously a bit later, the execution of this small app was not blocked, and the script reached the last line before the `async` functions `getCustomers()` and `getOrders()` finished their execution.

**NOTE**

See it in CodePen: [mng.bz/pSV8](https://mng.bz/pSV8).

## A.11 Modules

In any programming language, splitting code into modules helps organize the application into logical and possibly reusable units. Modularized applications allow programming tasks to be split between software developers more efficiently. Developers get to decide which API should be exposed by the module for external use and which should be used internally.

ES5 doesn't have language constructs for creating modules, so we have to resort to one of these options:

- Manually implement a module design pattern as an immediately initialized function.
- Use third-party modules implementations, e.g. AMD ([mng.bz/JKVc](#)) or CommonJS ([mng.bz/7Lld](#)) standard.

CommonJS was created for modularizing JavaScript applications that run outside the web browser (such as those written in Node.js and deployed under Google's V8 engine). AMD is primarily used for applications that run in a web browser.

You should split your app into modules to make your code more maintainable. Besides that, you should minimize the amount of JavaScript code loaded to the client on app startup. Imagine a typical online store. Do you need to load the code for processing payments when users open the application's home page? What if they never click the Place Order button? It would be nice to modularize the application so the code is loaded on an as-needed basis. RequireJS is probably the most popular third-party library that implements the AMD standard; it lets you define dependencies between modules and load them into the browser on demand.

Starting with ES6, modules have become part of the language. A script becomes a module if it uses `import` and/or `export` keywords. For example the script `shipping.js` exports the function `ship()`, which can be imported by other scripts. The function `calculateShippingCost()` remains invisible for external scripts.

### **Listing A.18 shipping.js**

```
export function ship() {
  console.log("Shipping products...");
}

function calculateShippingCost(){
  console.log("Calculating shipping cost");
}
```

The script `main.js` imports and uses the function `ship()` from `shipping.js`.

### **Listing A.19 main.js**

```
import {ship} from './shipping.js';

ship();
```

Syntax-wise it looks pretty clean and simple, but having the `import` statement doesn't load the module, and ES6 didn't standardize load modules, and developers were using third-party loaders like SystemJS or Webpack (see chapter 6 for details).

But all modern web browsers support `module` as a valid type in the `<script>` tag, so you can tell the browser to load the script as ES6 module.

## Listing A.20 index.html

```
<!DOCTYPE html>
<head>
  <title>My modules</title>
</head>
<body>
  <h1>Hello modules!</h1>
  <script type="module" src=".main.js"></script> ①
  <script type="module" src=".shipping.js"></script> ②
</body>
</html>
```

- ① Loading the first module
- ② Loading the second module

For the older browsers, you can use the `nomodule` attribute and provide the fallback script, for example:

```
<script type="module" src=".main.js"></script>
<script nomodule src=".main_fallback.js"></script>
```

If a browser supports the `module` type, it'll ignore the line with `nomodule`.

**TIP**

In chapter 9, you'll see the HTML file that uses the `<script>` tag with the attribute `type="module"`.

## SIDE BAR JavaScript modules and global scope

Say you have a multi-file project, and one of the files has the following content:

```
class Person {}
```

Because we didn't export anything from this file, it's not an ES6 module, and the instance of the class `Person` would be created in the global scope. If you already have another script in the same project that also declares the class `Person`, the TypeScript compiler will give you an error in the preceding code stating that you're trying to declare a duplicate of what already exists.

Adding the `export` statement to the preceding code changes the situation, and this script becomes a module:

```
export class Person {}
```

Now the objects of type `Person` won't be created on the global scope, and their scope will be limited to only those scripts (other ES6 modules) that import `Person`.

### NOTE

ES6 modules allow you to avoid polluting the global scope and restrict the visibility of the script and its members (classes, functions, variables, and constants) to those modules that import them.

### A.11.1 Imports and exports

A *module* is just a JavaScript file that implements certain functionality and exports (or imports) a public API so other JavaScript programs can use it. There's no special keyword to declare that the code in a particular file is a module. Just by using the keywords `import` and `export`, you turn the script into an ES6 module.

The `import` keyword enables one script to declare that it needs to use exported members from another script. Similarly, the `export` keyword lets you declare variables, functions, or classes that the module should be exposed to other scripts. In other words, by using the `export` keyword, you can make selected APIs available to other modules. The module's functions, variables, and classes that aren't explicitly exported remain private to the module.

ES6 offers two types of `export` usage: named and default. With named exports, you can use the `export` keyword in front of multiple members of the module (such as classes, functions, and variables). The code in the following file (`tax.js`) exports the variable `taxCode` and the functions

`calcTaxes()` and `fileTaxes()`, but the `doSomethingElse()` function remains hidden to external scripts:

```
export let taxCode = 1;

export function calcTaxes() { }

function doSomethingElse() { }

export function fileTaxes() { }
```

**NOTE** It's important to note that ES6 modules are statically resolved, which is a serious advantage compared to Node's modules that use a function `require()`.

When a script imports named exported module members, their names must be placed in curly braces. The following `main.js` file illustrates this:

```
import {taxCode, calcTaxes} from 'tax';

if (taxCode === 1) { // do something }

calcTaxes();
```

Here, `tax` refers to the filename of the module, minus the file extension.

The curly braces represent destructuring. The module from the file `tax.js` exports three members, but we're interested in importing only `taxCode` and `calcTaxes`.

One of the exported module members can be marked as `default`, which means this is an anonymous export, and another module can give it any name in its `import` statement. The `my_module.js` file that exports a function may look like this:

```
export default function() { // do something } ①

export let taxCode;
```

① No semicolon

The `main.js` file imports both named and default exports while assigning the name `coolFunction` to the default one:

```
import coolFunction, {taxCode} from 'my_module';

coolFunction();
```

Note that you don't use curly braces around `coolFunction` (default export) but you do around `taxCode` (named export). The script that imports a class, variable, or function that was exported with the `default` keyword can give them new names without using any special keywords:

```
import aVeryCoolFunction, {taxCode} from 'my_module';
aVeryCoolFunction();
```

But to give an alias name to a named export, you need to write something like this:

```
import coolFunction, {taxCode as taxCode2016} from 'my_module';
```

The module `import` statements don't result in copying the exported code. Imports serve as references. The script that imports modules or members can't modify them, and if the values in the imported modules change, the new values are immediately reflected in all places where they were imported.

## A.12 Transpilers

If you're about to start a new JavaScript project, don't use a ten year old syntax; use the syntax from the latest ECMAScript specs. But if the users of your apps have to work with older browsers that don't support the latest ECMAScript, you need to *transpile* your code down to ES5 or other supported syntax.

Transpilers convert the source code from one language to the source code of another. In the context of this appendix, you may need to transpile the code from ES6 (or later) to ES5 before deploying your app in production.

In the JavaScript ecosystem, the most popular transpiler is called Babel, and it's available at [babeljs.io](https://babeljs.io). You can try any of the code samples from this appendix in Babel's REPL utility (see [babeljs.io/repl](https://babeljs.io/repl)) that allows you to enter a code fragment in one of the newer version of ECMAScript, and compile it down to ES5. Figure A.2 shows the screenshot taken from the Babel "Try it out" menu that shows how the ES2015 code from listing A.12 (on the left) would be transpiled into ES5 (on the right). You can see it in action at [bit.ly/2tk1OJ2](https://bit.ly/2tk1OJ2).

The screenshot shows the Babel REPL interface. On the left, there's a sidebar with settings like Evaluate, Line Wrap, Minify, Pretty, File Size, Time Travel, Source Type (Module selected), Presets (es2015 selected), Options (Decorators mode set to Current Proposal), Env Preset, and Plugins. The main area shows a block of JavaScript code being transpiled:

```

1 class Tax {
2   constructor(income) {
3     this.income = income;
4   }
5
6   calculateFederalTax() {
7     console.log(`Calculating federal tax for income
8 ${this.income}`);
9   }
10
11   calcMinTax() {
12     console.log("In Tax. Calculating min tax");
13     return 123;
14   }
15
16   class NJTax extends Tax {
17     constructor(income, stateTaxPercent) {
18       super(income);
19       this.stateTaxPercent=stateTaxPercent;
20     }
21
22     calculateStateTax() {
23       console.log(`Calculating state tax for income
24 ${this.income}`);
25
26       calcMinTax() {
27         let minTax = super.calcMinTax();
28         console.log(`In NJTax. Will adjust min tax of ${minTax}`);
29       }
30     }
31
32   const theTax = new NJTax(50000, 6);
33
34   theTax.calculateFederalTax();

```

The right side shows the transpiled code, which includes polyfills for modern features like `Symbol` and `Reflect`.

**Figure A.2 Using Babel’s REPL**

You can use Babel not only for transpiling the newer JavaScript syntax into the older one. In section 6.4 in chapter 6, we explain how to use Babel with TypeScript if need be. But typically you’ll be transpiling TypeScript (to any version of JavaScript) using its own compiler introduced in section 1.4 in chapter 1).

This concludes our overview of some of the most important features introduced by the recent ECMAScript specs. The good news is that you can use all of these features in your TypeScript programs without waiting until all the browsers will start supporting them.

## **Index Terms**

all() method  
arrays  
classes (subclasses)  
class keyword  
destructuring (arrays)  
fulfilled  
importing and exporting  
modules (importing and exporting)  
pending  
promises (fulfilled)  
promises (rejected)  
promises (pending)  
promises (resolve() method)  
promises (all() method)  
prototypal inheritance  
providers  
rejected  
resolve() method  
state  
state  
subclasses  
then() method  
tokens