appendix B
TypeScript essentials

TypeScript was released in 2012 by Microsoft, and its core developer was Anders Hejlsberg. He's also one of the authors of Turbo Pascal and Delphi, and is a lead architect of C#. In this appendix, we'll cover main elements of the TypeScript syntax.

We'll also show you how to turn TypeScript code into JavaScript (ES5) so it can be executed by any web browser or a standalone JavaScript engine. This appendix doesn't offer a complete coverage of TypeScript. Refer to the TypeScript documentation at www.typescriptlang.org/docs/home.html for complete coverage. Also, TypeScript supports all syntax constructs described in appendix A, so we don't repeat those here.

B.1 The role of transpilers

Web browsers don't understand any language but JavaScript. If the source code is written in TypeScript, it has to be *transpiled* into JavaScript before you can run it in a JavaScript engine, whether browser or standalone.

Transpiling means converting the source code of a program in one language into source code in another language. Many developers prefer to use the word *compiling*, so phrases like "TypeScript compiler" and "compile TypeScript into JavaScript" are also valid.

Figure B.1 shows TypeScript code on the left and its equivalent in an ES5 version of JavaScript generated by the TypeScript transpiler on the right. In TypeScript, we declared a variable `foo` of type `string`, but the transpiled version doesn't have the type information. In TypeScript, we declared a class `Bar`, which was transpiled in a class-like pattern in the ES5 syntax.

You can try it for yourself by visiting the TypeScript playground at www.typescriptlang.org/play. If we had specified ES6 as a target for transpiling, the generated JavaScript code would look different; you'd see the `let` and `class` keywords on the right side as well.

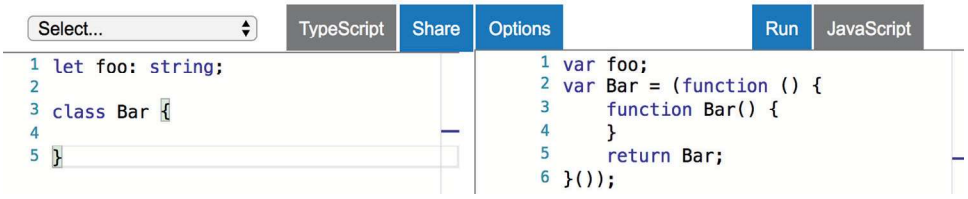


Figure B.1 Transpiling TypeScript into ES5

A combination of Angular with statically typed TypeScript simplifies the development of web applications. Good tooling and a static type analyzer substantially decrease the number of runtime errors and shorten the time to market. When complete, your Angular application will have lots of JavaScript code; and although developing in TypeScript may require you to write a little more code, you'll reap benefits by saving time on testing and refactoring and minimizing the number of runtime errors.

B.2 Getting started with TypeScript

Microsoft has open sourced TypeScript and hosts the TypeScript repository on GitHub at https://github.com/Microsoft/TypeScript/wiki/Roadmap. You can install the TypeScript compiler using npm. The TypeScript site www.typescriptlang.org has the language documentation and has a web-hosted TypeScript compiler (under the Playground menu), where you can enter TypeScript code and compile it to JavaScript interactively, as shown in figure B.1. Enter TypeScript code on the left, and its JavaScript version (ES5) is displayed on the right. Click the Run button to execute the transpiled code (open the browser console to see the output produced by your code, if any).

Such interactive tools will suffice for learning the language syntax, but for real-world development, you'll need better tooling to be productive. You may decide to use an IDE or a text editor, but having the TypeScript compiler installed locally is a must for development. We'll show you how to install the TypeScript compiler and run code samples in this appendix, using the Node JavaScript engine.

We assume that you have Node.js and npm installed on your computer. If you don't have them yet, refer to appendix C.

B.2.1 Installing and using the TypeScript compiler

We'll use Node.js's npm package manager to install the TypeScript compiler. To install it globally, run the following npm command in the Terminal window:

```
npm install -g typescript
```

The `-g` option installs the TypeScript compiler globally on your computer, so it's available from the Terminal window in all your projects. To check the version of your TypeScript compiler, run the following command:

```
tsc
 --version
```

As mentioned earlier, code written in TypeScript has to be transpiled into JavaScript so web browsers can execute it. TypeScript code is saved in files with the .ts extension. Say you write a script and save it in the main.ts file. The following command will transpile main.ts into main.js:

```
tsc main.ts
```

You can also generate source map files that map lines in the TypeScript program to corresponding lines in the generated JavaScript. With source maps, you can place breakpoints in your TypeScript code while running it in the browser, even though it executes JavaScript. To compile main.ts into main.js while also generating the source map file main.js.map, run the following command:

```
tsc --sourcemap main.ts
```

If a 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.

During compilation, TypeScript's compiler removes from the generated code all TypeScript types, interfaces, and keywords not supported by JavaScript. By providing compiler options, you can generate JavaScript compliant with ES3, ES5, ES6, or newer syntax.

Here's how to transpile the code to ES5-compatible syntax (the `--t` option specifies the target syntax):

```
tsc --t ES5 main.ts
```

You can start your TypeScript compiler in watch mode by providing the `-w` option. In this mode, whenever you modify and save your code, it's automatically transpiled into corresponding JavaScript files. To compile and watch all .ts files from the current directory, run the following command:

```
tsc -w *.ts
```

The compiler will compile all the TypeScript files, print error messages (if any) on the console, and continue watching the files for changes. As soon as a file changes, `tsc` will immediately recompile it.

> **NOTE** Typically, we turn off TypeScript autocompilation in the IDE. With Angular apps, we use Angular CLI to compile and bundle the entire project. The IDEs use the TypeScript code analyzer to highlight errors even without compilation.

`tsc` offers dozens of compilation options described at http://mng.bz/rf14. You can preconfigure the process of compilation (specifying the source and destination directories,

source map generation, and so on). The presence of the tsconfig.json file in the project directory means you can enter `tsc` on the command line, and the compiler will read all the options from tsconfig.json. A sample tsconfig.json file from one of the Angular projects is shown in the following listing.

Listing B.1 tsconfig.json

```
{
    "compilerOptions": {
        "baseUrl": "src",
         "outDir": "./dist",
         "sourceMap": true,
         "moduleResolution": "node",
         "noEmitOnError": true,
         "target": "es5",
         "experimentalDecorators": true
     }
}
```

Transpiles .ts files located in the src directory

Saves the generated .js files in the dist directory

Generates sourcemaps

Looks for modules according to the structure of Node-based projects

Transpiles the .ts file into ES5 syntax

Required to support decorators

If any of the files has compilation errors, doesn't generate JavaScript files

Every Angular/TypeScript app uses *decorators* with classes or class members (such as `@Component()` and `@Input()`). We'll discuss decorators later in this appendix.

If you want to exclude some of your project files from compilation, add the `exclude` property to tsconfig.json. This is how you can exclude the entire content of the node_modules directory:

```
"exclude": [
    "node_modules"
  ]
```

B.2.2 TypeScript as a superset of JavaScript

TypeScript supports ES5, ES6, and newer ECMAScript syntax. Just change the name extension of a file with JavaScript code from .js to .ts, and it'll become valid TypeScript code. Being a superset of JavaScript, TypeScript adds a number of useful features to JavaScript. We'll review them next.

B.3 How to run the code samples

To run the code samples from this appendix locally on your computer, perform the following steps:

1. Install Node.js from https://nodejs.org/en/download/ (use the current version).
2. Clone or download the https://github.com/Farata/angulartypescript repository into any directory.
3. In the command window, change into this directory, and then go to the code-samples/appendixB subdirectory.

4. Install the project dependencies (the TypeScript compiler) *locally* by running `npm install`.
5. Use the locally installed TypeScript compiler to compile all code samples into the dist directory by running `npm run tsc`, which will transpile all code samples from the src directory into the dist directory.
6. To run a particular code sample (such as fatArrow.js) use the following command: `node dist/fatArrow`.

B.4 Optional types

You can declare variables and provide types for all or some of them. The following two lines are valid TypeScript syntax:

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

TIP In the second line, specifying the type `string` is unnecessary. Since the variable is initialized with the string, TypeScript will guess (infer) that the type of `name2` is `string`.

If you use types, TypeScript's transpiler can detect mismatched types during development, and IDEs will offer code completion and refactoring support. This will increase your productivity on any decent-sized project. Even if you don't use types in declarations, TypeScript will guess the type based on the assigned value and will still do type checking afterward. This is called type *inference*.

The following fragment of TypeScript code shows that you can't assign a numeric value to a `name1` variable that was meant to be a `string`, even though it was initially declared without a type (JavaScript syntax). After initializing this variable with a `string` value, the inferred typing won't let you assign the numeric value to `name1`:

```
let name1 = 'John Smith';
name1 = 123;    <-- Assigning a value of a different type to a variable is valid in JavaScript but invalid in TypeScript because of the inferred type.
```

In TypeScript, you can declare typed variables, function parameters, and return values. There are four keywords for declaring basic types: `number`, `boolean`, `string`, and `void`. The last one indicates the absence of a return value in a function declaration. A variable can have a value of type `null` or `undefined`, similar to JavaScript.

Here are some examples of variables declared with explicit types:

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

NOTE Starting from TypeScript 2.7, you need to either initialize variables during declaration or initialize (member variables) in the constructor.

All of these types are subtypes of the `any` type. You may as well explicitly declare a variable, specifying `any` as its type. In this case, inferred typing isn't applied. Both of these declarations are valid:

```
let name2: any = 'John Smith';
name2 = 123;
```

If variables are declared with explicit types, the compiler will check their values to ensure that they match the declarations. TypeScript includes other types that are used in interactions with the web browser, such as `HTMLElement` and `Document`. If you define a class or an interface, it can be used as a custom type in variable declarations. We'll introduce classes and interfaces later, but first let's get familiar with TypeScript functions, which are the most-used constructs in JavaScript.

B.5 Functions

TypeScript functions and function expressions are similar to JavaScript functions, but you can explicitly declare parameter types and return values. Let's write a JavaScript function 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. The function is shown in the following listing.

Listing B.2 Calculating tax in JavaScript

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

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()` doesn't declare any types for the function parameters, you can guess them based on the parameter names. Now let's invoke it the wrong way, passing a `string` value for a number of dependents:

```
var 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 didn't have a chance to explicitly specify the types of the parameters. The next listing rewrites this function in TypeScript, declaring types for parameters and the return value.

Listing B.3 Calculating tax in TypeScript

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

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

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 saying, "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.

B.5.1 Default parameters

While declaring a function, you can specify default parameter values. For example:

```
function calcTax(income: number, dependents: number, state: string = 'NY'): n
     umber{
    // the code goes here
}
```

There's no need to change even one line of code in the body of `calcTax()`. You now have the freedom to invoke it with either two or three parameters:

```
let tax: number = calcTax(50000, 2);
// or
let tax: number = calcTax(50000, 2, 'NY');
```

The results of both invocations will be the same.

B.5.2 Optional parameters

In TypeScript, you can easily mark function parameters as optional by appending a question mark to the parameter name. The only restriction is that optional parameters must come last in the function declaration. When you write code for functions with optional parameters, you need to provide application logic that handles the cases when the optional parameters aren't provided.

Let's modify the tax-calculation function in the following listing: if no dependents are specified, it won't apply any deduction to the calculated tax.

Listing B.4 Calculating tax in TypeScript, modified

```
function calcTax(income: number, state: string = 'NY', dependents?: number):
     number{

    let deduction: number;

    if (dependents) {
        deduction = dependents * 500;
    }else {
      deduction = 0;
    }

    if (state === 'NY') {
        return income * 0.06 - deduction;
    } else if (state === 'NJ') {
        return income * 0.05  - deduction;
    }
}

let tax: number = calcTax(50000, 'NJ', 3);
console.log(`Your tax is ${tax}`);

tax = calcTax(50000);
console.log(`Your tax is ${tax}`);
```

Note the question mark in `dependents?: number`. Now the function checks whether the value for dependents was provided. If it wasn't, you assign 0 to the `deduction` variable ; otherwise, you deduct 500 for each dependent.

Running the preceding script will produce the following output:

```
Your tax is 1000
Your tax is 3000
```

NOTE TypeScript supports the syntax of fat-arrow expressions described in section A.5 in appendix A.

Function overloading

JavaScript doesn't support function overloading, so having several functions with the same name but different lists of arguments isn't possible. TypeScript supports function overloading, but because the code has to be transpiled into a single JavaScript function, the syntax for overloading isn't elegant.

You can declare several signatures of a function with one and only one body, where you need to check the number and types of the arguments and execute the appropriate portion of the code:

```
function attr(name: string): string;
function attr(name: string, value: string): void;
```

```
function attr(map: any): void;
function attr(nameOrMap: any, value?: string): any {
  if (nameOrMap && typeof nameOrMap === "string") {
      // handle string case
  } else {
      // handle map case
  }

  // handle value here
}
```

B.6 Classes

If you have Java or C# experience, you'll be familiar with the concepts of classes and inheritance in their classical form. In those languages, the definition of a class is loaded in memory as a separate entity (like a blueprint) and is shared by all instances of this class. If a class is inherited from another one, the object is instantiated using the combined blueprint of both classes.

TypeScript is a superset of JavaScript, which only supports *prototypal inheritance*, where you can create an inheritance hierarchy by attaching one object to the *prototype* property of another. In this case, an inheritance (or rather, a linkage) of *objects* is created dynamically.

In TypeScript, the `class` keyword is syntactic sugar to simplify coding. In the end, your classes will be transpiled into JavaScript objects with prototypal inheritance. In JavaScript, you can declare a constructor function and instantiate it with the `new` keyword. In TypeScript, you can also declare a class and instantiate it with the `new` operator.

A class can include a constructor, fields (properties), and methods. Declared properties and methods are often referred to as *class members* . We'll illustrate the syntax of TypeScript classes by showing you a series of code samples and comparing them with the equivalent ES5 syntax.

Let's create a simple `Person` class that contains four properties to store the first and last name, age, and Social Security number (a unique identifier assigned to citizens and residents of the United States). At left in figure B.2, you can see the TypeScript code that declares and instantiates the `Person` class; on the right is a JavaScript closure generated by the `tsc` compiler. By creating a closure for the `Person` function, the TypeScript compiler enables the mechanism for exposing and hiding the elements of the `Person` object.

TypeScript also supports class constructors that allow you to initialize object variables while instantiating the object. A class constructor is invoked only once during object creation. The left side of figure B.2 shows the `Person` class, which uses the `constructor` keyword that initializes the fields of the class with the values given to the constructor.

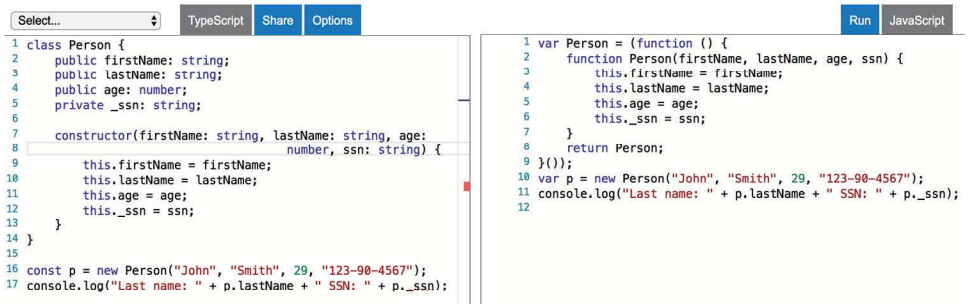


Figure B.2 Transpiling a TypeScript class into a JavaScript closure

B.6.1 Access modifiers

JavaScript doesn't have a way to declare a variable or a method as *private* (hidden from external code). To hide a property (or a method) in an object, you need to create a closure that neither attaches this property to the `this` variable nor `returns` it in the closure's return statement.

TypeScript provides `public`, `protected`, and `private` keywords to help you control access to object members during the development phase. By default, all class members have public access, and they're visible from outside the class. If a member is declared with the `protected` modifier, it's visible in the class and its subclasses. Class members declared as `private` are visible only in the class.

Let's use the `private` keyword to hide the value of the `_ssn` property so it can't be directly accessed from outside of the `Person` object. We'll show you two versions of declaring a class with properties that use access modifiers. The longer version of the class looks like the following listing.

Listing B.5 Using a private property

```
class Person {
    public firstName: string;
    public lastName: string;
    public age: number;
    private _ssn: string;

    constructor(firstName: string, lastName: string, age: number, ssn: string
     ) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this._ssn = ssn;
    }
}

const p = new Person("John", "Smith", 29, "123-90-4567");
console.log("Last name: " + p.lastName + " SSN: " + p._ssn);
```

Note that the name of the private variable starts with an underscore: `_ssn`. This is a naming convention for private properties.

The last line of listing B.5 attempts to access the `_ssn` private property from outside, so the TypeScript code analyzer will give you a compilation error: "Property '_ssn' is private and is only accessible in class 'Person'." But unless you use the `--noEmitOnError` compiler option, the erroneous code will still be transpiled into JavaScript:

```
const Person = (function () {
    function Person(firstName, lastName, age, _ssn) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this._ssn = _ssn;
    }
    return Person;
})();

const p = new Person("John", "Smith", 29, "123-90-4567");
console.log("Last name: " + p.lastName + " SSN: " + p._ssn);
```

The `private` keyword only makes it private in the TypeScript code, but the generated JavaScript code will treat all properties and methods of the class as public anyway.

TypeScript also allows you to provide access modifiers with constructor arguments, as shown in the short version of the `Person` class in the following listing.

Listing B.6 Using access modifiers

```
class Person {

    constructor(public firstName: string,
        public lastName: string, public age: number,  private _ssn: string) {
    }
}

const p = new Person("John", "Smith", 29, "123-90-4567");
```

When you use a constructor with access modifiers, the TypeScript compiler takes it as an instruction to create and retain class properties matching the constructor arguments. You don't need to explicitly declare and initialize them. Both the short and long versions of the `Person` class generate the same JavaScript, but we recommend using the shorter syntax as shown in figure B.3.

B.6.2 Methods

When a function is declared in a class, it's called a *method*. In JavaScript, you need to declare methods on the prototype of an object, but with a class, you declare a method by specifying a name followed by parentheses and curly braces, as you would in other object-oriented languages.

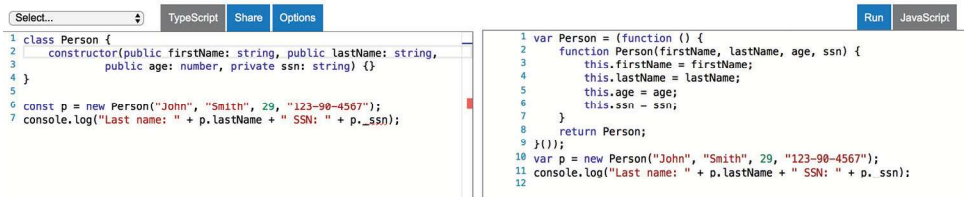


Figure B.3 Transpiling a TypeScript class with `constructor`

The next code listing shows how you can declare and use a `MyClass` class with a `doSomething()` method that has one argument and no return value.

Listing B.7 Creating a method

```
class MyClass {

   doSomething(howManyTimes: number): void {
      // do something here
   }
}

const mc = new MyClass();
mc.doSomething(5);
```

Static and instance members

The code in listing B 7, as well as the class shown in figure B.2, creates an instance of the class first and then accesses its members using a reference variable that points at this instance:

```
mc.doSomething(5);
```

If a class property or method were declared with the `static` keyword, its values would be shared between all instances of the class, and you wouldn't need to create an instance to access static members. Instead of using a reference variable (such as `mc`), you'd use the name of the class:

```
class MyClass{

   static doSomething(howManyTimes: number): void {
      // do something here
   }
}

MyClass.doSomething(5);
```

If you instantiate a class and need to invoke a class method from within another method declared in the same class, don't use the `this` keyword (as in, `this.doSomething(5)`), but still use the class name, as in `MyClass.doSomething(10);`.

B.6.3 Inheritance

JavaScript supports prototypal *object-based* inheritance, where one object can assign another object as its prototype, and this happens during runtime. TypeScript has the `extends` keyword for inheritance of classes, like ES6 and other object-oriented languages. But during transpiling to JavaScript, the generated code uses the syntax of prototypal inheritance.

Figure B.4 shows how to create an `Employee` class (line 9) that extends the `Person` class. On the right, you can see the transpiled JavaScript version, which uses prototypal inheritance. The TypeScript version of the code is more concise and easier to read.

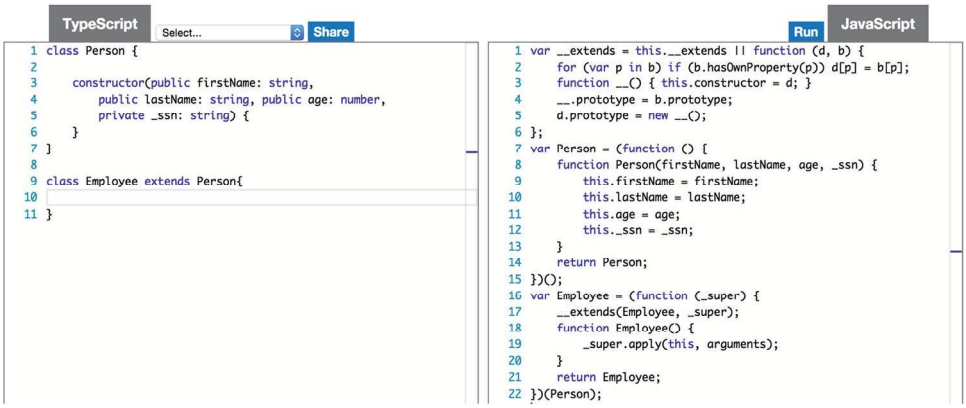


Figure B.4 Class inheritance in TypeScript

Let's add a constructor and a `department` property to the `Employee` class in the next listing.

Listing B.8 Using inheritance

```
class Employee extends Person {
    department: string;

    constructor(firstName: string, lastName: string,
            age: number, _ssn: string, department: string) {

        super(firstName, lastName, age, _ssn);

        this.department = department;
    }
}
```

Declares a property department

Creates a constructor that has an additional department argument

A subclass that declares a constructor must invoke the constructor of the superclass using super().

If you invoke a method declared in a superclass on the object of the subclass type, you can use the name of this method as if it were declared in the subclass. But sometimes you want to specifically call the method of the superclass, and that's when you should use the `super` keyword.

The `super` keyword can be used two ways. In the constructor of a derived class, you invoke it as a method. You can also use the `super` keyword to specifically call a method of the superclass. It's typically used with method overriding. For example, if both a superclass and its descendant have a `doSomething()` method, the descendant can reuse the functionality programmed in the superclass and add other functionality as well:

```
doSomething() {
    super.doSomething();

    // Add more functionality here
}
```

B.7 Interfaces

JavaScript doesn't support interfaces, which, in other object-oriented languages, are used to introduce a *code contract* that an API has to abide by. An example of a contract can be class X declaring that it implements interface Y. If class X won't include an implementation of the methods declared in interface Y, it's considered a violation of the contract and won't compile.

TypeScript includes the keywords `interface` and `implements` to support interfaces, but interfaces aren't transpiled into JavaScript code. They just help you avoid using the wrong types during development.

In TypeScript, we use interfaces for two reasons:

- Declare an interface that defines a custom type containing a number of properties. Then declare a method that has an argument of such a type. The compiler will check that the object given as an argument includes all the properties declared in the interface.
- Declare an interface that includes abstract (non-implemented) methods. When a class declares that it implements this interface, the class must provide an implementation for all the abstract methods.

Let's apply these two patterns by example.

B.7.1 Declaring custom types with interfaces

When you use JavaScript frameworks, you may run into an API that requires some sort of configuration object as a function parameter. To figure out which properties must be provided in this object, either open the documentation for the API or read the source code of the framework. In TypeScript, you can declare an interface that includes all the properties, and their types, that must be present in a configuration object.

Let's see how to do this in the `Person` class, which contains a constructor with four arguments: `firstName`, `lastName`, `age`, and `ssn`. This time, in the following listing, you'll declare an `IPerson` interface that contains the four members, and you'll modify the constructor of the `Person` class to use an object of this custom type as an argument.

Listing B.9 Declaring an interface

```
interface IPerson {
    firstName: string;
    lastName: string;
    age: number;
    ssn?: string;
 }

class Person  {
    constructor(public config: IPerson) {}
 }

let aPerson: IPerson = {
     firstName: "John",
    lastName: "Smith",
    age: 29
}

let p = new Person(aPerson);
 console.log("Last name: " + p.config.lastName );
```

Declares an IPerson interface with ssn as an optional member (note the question mark)

The Person class has a constructor with one argument of type IPerson.

Creates an aPerson object literal with members compatible with IPerson

Instantiates the Person object, providing an object of type IPerson as an argument

TypeScript has a structural type system, which means that if two different types include the same members, the types are considered compatible. In listing B.9, even if you didn't specify the type of the `aPerson` variable, it still would be considered compatible with `IPerson` and could be used as a constructor argument while instantiating the `Person` object. If you change the name or type of one of the members of `IPerson`, the TypeScript compiler will report an error.

The `IPerson` interface didn't define any methods, but TypeScript interfaces can include method signatures without implementations.

B.7.2 Using the implements keyword

The `implements` keyword can be used with a class declaration to announce that the class will implement a particular interface. Say you have an `IPayable` interface declared as follows:

```
interface IPayable {
  increase_cap: number;

  increasePay(percent: number): boolean
}
```

Now the `Employee` class can declare that it implements `IPayable`:

```
class Employee implements IPayable {
    // The implementation goes here
}
```

Before going into details, let's answer this question: why not just write all required code in the class rather than separating a portion of the code into an interface? Let's say you need to write an application that allows increasing salaries for the employees of your organization.

You can create an `Employee` class (that extends `Person`) and include the `increaseSalary()` method there. Then the business analysts may ask you to add the ability to increase pay to contractors who work for your firm. But contractors are represented by their company names and IDs; they have no notion of salary and are paid on an hourly basis.

You can create another class, `Contractor` (not inherited from `Person`), that includes some properties and an `increaseHourlyRate()` method. Now you have two different APIs: one for increasing the salary of employees, and another for increasing the pay for contractors. A better solution is to create a common `IPayable` interface and have `Employee` and `Contractor` classes provide *different implementations* of `IPayable` for these classes, as illustrated in the following listing.

Listing B.10 Using multiple interface implementations

The Person class serves as a base class for Employee.

The IPayable interface includes the signature of the increasePay() method that will be implemented by the Employee and Contractor classes.

The Employee class inherits from Person and implements the IPayable interface. A class can implement multiple interfaces.

The Employee class implements the increasePay() method. The salary of an employee can be increased by any amount, so the method prints the message on the console and returns true (allowing the increase).

The Contractor class includes a property that places a cap of 20% on pay increases.

The implementation of increasePay() in the Contractor class is different, invoking increasePay() with an argument that's more than 20 results in the "Sorry" message and a return value of false.

```
interface IPayable {
  increasePay(percent: number): boolean
}

class Person {
     // properties are omitted for brevity
}

class Employee extends Person implements IPayable {
    increasePay(percent: number): boolean {
        console.log(`Increasing salary by ${percent}`);
        return true;
    }
}

class Contractor implements IPayable {
    increaseCap:number = 20;

    increasePay(percent: number): boolean {
         if (percent < this.increaseCap) {
            console.log(`Increasing hourly rate by ${percent}`);
            return true;
        } else {
            console.log(`Sorry, the increase cap for contractors is
                     ${this.increaseCap}`);
            return false;
        }
    }
}
```

```
const workers: IPayable[] = [];
 workers[0] = new Employee();
workers[1] = new Contractor();

workers.forEach(worker => worker.increasePay(30));
```

Declaring an array of type IPayable lets you place any objects that implement the IPayable type there.

Now you can invoke the increasePay() method on any object in the workers array. Note that you don't use parentheses with the fat-arrow expression that has a single worker argument.

Running the preceding script produces the following output on the browser console:

```
Increasing salary by 30
Sorry, the increase cap for contractors is 20
```

Why declare classes with the implements keyword?

If you remove `implements Payable` from the declaration of either `Employee` or `Contractor`, the code will still work, and the compiler won't complain about lines that add these objects to the `workers` array. The compiler is smart enough to see that even if the class doesn't explicitly declare `implements IPayable`, it implements `increasePay()` properly.

But if you remove `implements IPayable` and try to change the signature of the `increasePay()` method from any of the classes, you won't be able to place such an object into the `workers` array, because that object would no longer be of the `IPayable` type. Also, without the `implements` keyword, IDE support (such as for refactoring) will be broken.

B.8 Generics

TypeScript supports parameterized types, also known as *generics*, which can be used in a variety of scenarios. For example, you can create a function that can take values of any type; but during its invocation, in a particular context, you can explicitly specify a concrete type.

Take another example: an array can hold objects of any type, but you can specify which particular object types (for example, instances of `Person`) are allowed in an array. If you were to try to add an object of a different type, the TypeScript compiler would generate an error.

The following code listing declares a `Person` class and its descendant, `Employee`, and an `Animal` class. Then it instantiates each class and tries to store them in the `workers` array declared with the generic type. Generic types are denoted by placing them in angle brackets (as in `<Person>`).

Listing B.11 Using a generic type

```
class Person {
    name: string;
}

class Employee extends Person {
    department: number;
}

class Animal {
    breed: string;
}

let workers: Array<Person> = [];

workers[0] = new Person();
workers[1] = new Employee();
workers[2] = new Animal();  // compile-time error
```

By declaring the `workers` array with the generic type `<Person>`, you announce your plan to store only instances of the `Person` class or its descendants. An attempt to store an instance of `Animal` in the same array will result in a compile-time error.

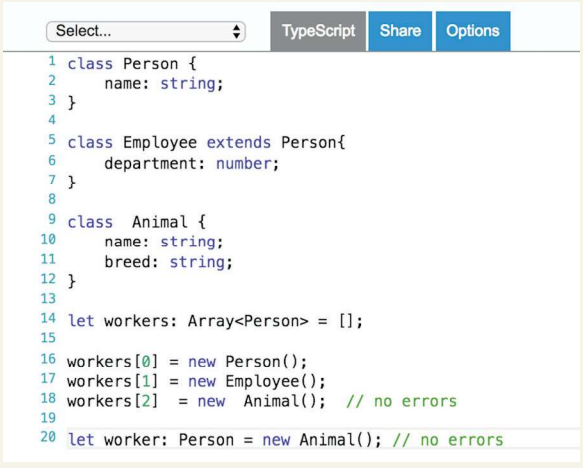
Nominal and structural type systems

If you're familiar with generics in Java or C#, you may get a feeling that you already understand this syntax. There's a caveat, though. Though Java and C# use a *nominal* type system, TypeScript uses a *structural* one. In a nominal system, types are checked against their names, but in a structural system, by their structure.

With a nominal type system, the following line would 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 illustrate it by adding the `name` property to the `Animal` class.



Structural type system in action

Now the TypeScript compiler doesn't complain about assigning an `Animal` object to a variable of type `Person`. The variable of type `Person` expects an object that has a `name` property, and the `Animal` object has it. This is not to say that `Person` and `Animal` represent the same types, but these types are compatible. On the other hand, trying to assign the `Person` object to a variable of type `Animal` will result in the compilation error "Property breed is missing in type Person":

```
let worker: Animal = new Person(); // compilation error
```

Can you use generic types with any object or function? No. The creator of the object or function has to allow this feature. If you open TypeScript's type definition file (lib.d.ts) on GitHub at http://mng.bz/I3V7 and search for "interface Array," you'll see the declaration of the `Array`, as shown in figure B.5. Type definition files are explained later in this appendix.

The `<T>` in line 1008 serves as a placeholder for the actual type. It means TypeScript allows you to declare a type parameter with `Array`, and the compiler will check for the specific type provided in your program. Earlier in this section, we specified a generic `<T>` parameter as `<Person>` in `let workers: Array<Person>`. But because

```
1004 ////////////////////////////
1005 /// ECMAScript Array API (specially handled by compiler)
1006 ////////////////////////////
1007
1008 interface Array<T> {
1009     /**
1010       * Gets or sets the length of the array. This is a number one higher than the h
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;
1027     /**
1028       * Combines two or more arrays.
1029       * @param items Additional items to add to the end of array1.
1030       */
```

Figure B.5 The fragment of lib.d.ts describing the `Array` API

generics aren't supported in JavaScript, you won't see them in the code generated by the transpiler. It's just an additional safety net for developers at compile time.

You can see another `T` in line 1022 in figure B.5. When generic types are specified with function arguments, no angle brackets are needed. But there's no `T` type in TypeScript. The `T` here means the `push` method lets you push objects of a specific type into an array, as in the following example:

```
workers.push(new Person());
```

You can create your own classes and functions that support generics, as well. The next listing defines a `Comparator<T>` interface that declares a `compareTo()` method, expecting the concrete type to be provided during method invocation.

Listing B.12 Creating an interface that uses generics

```
interface Comparator<T> {
     compareTo(value: T): number;
}

class Rectangle implements Comparator<Rectangle> {

    constructor(private width: number, private height: number) {};

    compareTo(value: Rectangle): number {
         if (this.width * this.height >= value.width * value.height) {
            return 1;
         } else  {
            return -1;
         }
    }
}

let rect1: Rectangle = new Rectangle(2,5);
let rect2: Rectangle = new Rectangle(2,3);

rect1.compareTo(rect2) === 1? console.log("rect1 is bigger"):
                             console.log("rect1 is smaller");

class Programmer implements Comparator<Programmer> {

    constructor(public name: string, private salary: number) {};

    compareTo(value: Programmer): number {
         if (this.salary >= value.salary) {
            return 1;
         } else  {
            return -1;
         }
    }
}

let prog1: Programmer = new Programmer("John",20000);
```

Declares a Comparator interface with a generic type

Creates a class that implements Comparator, specifying the concrete type Rectangle

Implements the method for comparing rectangles

Compares rectangles (the type T is erased and replaced with Rectangle)

Creates a class that implements Comparator, specifying the concrete type Programmer

Implements the method for comparing programmers

```
let prog2: Programmer = new Programmer("Alex",30000);

prog1.compareTo(prog2)===1? console.log(`${prog1.name} is richer`):
                            console.log(`${prog1.name} is poorer`) ;
```

Compares programmers (the type T is erased and replaced with Programmer)

B.9 The readonly modifier

ES6 introduced the `const` keyword that you can apply to variables, but not to properties of a class or interface. You can't write this:

```
class Person {
    const name: = "Mary";  // compiler error
}

const p = new Person();  // no errors
```

TypeScript adds a `readonly` keyword that can be applied to class properties:

```
class Person {
    readonly name = "Mary";  // no errors
}
```

You can initialize a `readonly` property only during its declaration or in the class constructor. Now if you'll try to write the code that modifies the value of the `name` property, the TypeScript compiler (or static analyzer) will report an error:

```
class Person {
    readonly name = "Mary";

    changeName() {
        this.name = "John";  // compiler error
    }
}
```

But creating an immutable object is a more interesting use case for applying the `readonly` modifier. In some cases, especially in Angular apps, you may want to ensure that an object is immutable, and you can't mutate the object by mistake. Let's try to apply `readonly` to an object property:

```
class Person {

    readonly bestFriend: { name: string } = {name: "Mary"};

    changeFriend() {
        this.bestFriend = { name: "John" }; // compiler error
    }

    changeFriendName() {
        this.bestFriend.name = "John";      // no errors
    }
}
```

An attempt to assign another object to the `bestFriend` variable results in a compilation error, because `bestFriend` is marked as `readonly`. But changing the internal property of the object represented by `bestFriend` is still allowed. To prohibit this, use the `readonly` modifier with each property of the object:

```
class Person {

    readonly bestFriend: { readonly name: string } = {name: "Mary"};

    changeFriend() {
        this.bestFriend = { name: "John" }; // compiler error
    }

    changeFriendName(newName: string) {
        this.bestFriend.name = "John";      // compiler error
    }
}
```

In Angular apps, you may want to store application state in an immutable object bound to the input property of a component. To enforce the creation of a new object instance whenever its properties change, write a function that creates a copy of the object with modification of the properties (see code samples in section A.7 in appendix A).

If an object has multiple properties, adding the `readonly` modifier to each of them is a tedious job, and you can use a read-only mapped type instead. The following example uses the `type` keyword to define a new type and generics to provide a concrete object to the `Readonly` class:

```
type Friend = Readonly<{ name: string, lastName: string }>;

class Person {

     bestFriend: Friend = {name: "Mary", lastName: "Smith"};

    changeFriend() {
        this.bestFriend = { name: "John" }; // compiler error
    }

    changeFriendName() {
        this.bestFriend.name = "John";     // compiler error
        this.bestFriend.lastName = "Lou";  // compiler error
    }
}
```

B.10 Decorators

There are different definitions of the term *metadata*. The popular definition is that metadata is data about data. We think of metadata as data that describes and enhances code. Internally, TypeScript decorators are special functions that add metadata enhancing the functionality of a class, property, method, or parameter. TypeScript decorators start with an `@` sign.

Decorators exist in Typescript, and they are proposed in ECMAScript. To properly transpile them, turn on experimental features in the Typescript transpiler by adding the following line in the tsconfig.json file:

```
"experimentalDecorators": true
```

In this section, we'll show you how to create a simple decorator that will print the information about the class it's attached to.

Imagine that you want to create a decorator, `UIComponent()`, that can accept an HTML fragment as a parameter. The decorator should be able to print the received HTML and understand the properties of the attached artifact—for example, a class. The following listing does this.

Listing B.13 A custom `UIComponent` decorator

```
function UIComponent (html: string) {
    console.log(`The decorator received ${html} \n`);

    return function(target) {
        console.log(`Creating a UI component from \n ${target} ` );
    }
}

@UIComponent('<h1>Hello Shopper!</h1>')
class Shopper {

    constructor(private name: string) {}
}
```

The `UIComponent()` function has one `string` parameter and returns another function that prints the content of the implicit variable `target`, which knows the artifact the decorator is attached to. If you compile this code into ES5 syntax and run it, the output on your console will look as follows:

```
The decorator received <h1>Hello Shopper!</h1>

Creating a UI component from
 function Shopper(name) {
        this.name = name;
    }
```

If you compile the same code into ES6, the output will be different, because ES6 supports classes:

```
The decorator received <h1>Hello Shopper!</h1>

Creating a UI component from
 class Shopper {
    constructor(name) {
        this.name = name;
    }
}
```

Under the hood, TypeScript uses the `reflect-metadata` library to query the structure of the artifact the decorator is attached to. This simple decorator knows what HTML you want to render and that your class has a member variable `name`. If you're a developer of a framework that needs to render UI, the code of this decorator can come in handy. The process of creating custom decorators is described in the TypeScript documentation at http://mng.bz/gz6R.

To turn a TypeScript class into an Angular component, you need to decorate it with the `@Component()` decorator. Angular will internally parse your annotations and generate code that adds the requested behavior to the TypeScript class. To turn a class variable into a component property that can receive values, you use the `@Input()` decorator:

```
@Component({
  selector: 'order-processor',
  template: `
    Buying {{quantity}} shares}
  `
})
export class OrderComponent {

  @Input() quantity: number;

}
```

In this example, the `@Component()` decorator defines the selector and a template (UI) for the `OrderComponent` class. The `@Input()` decorator enables the `quantity` property to receive values from the parent component via binding. When you use decorators, there should be a decorator processor that can parse the decorator content and turn it into code that the runtime (the browser's JavaScript engine) understands. Angular includes the `ngc` compiler that performs the duties of a decorator processor.

To use Angular decorators, import their implementation in your application code. For example, import the `@Component()` decorator as follows:

```
import { Component } from '@angular/core';
```

Angular comes with a set of decorators, but TypeScript allows you to create your own decorators regardless of whether you use Angular or not.

B.11 The union type

In TypeScript, 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:

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

Although TypeScript supports the `any` type, the preceding declaration provides some benefits compared to the declaration `let padding: any`. In the following listing, we'll review the code of one of the code samples from the TypeScript documentation at http://mng.bz/5742. This function can add left padding to the provided string. The

padding can be specified either as a string that has to prepend the provided argument or the number of spaces that should prepend the string.

Listing B.14 union.ts with any type

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

Provides the string and the padding of type any

For a string, uses concatenation

For a numeric argument, generates spaces

If the second argument is neither a string nor a number, throws an error

The following are examples of invoking `padLeft()`:

```
console.log( padLeft("Hello world", 4));         // returns "    Hello world"
console.log( padLeft("Hello world", "John says "));// returns "John says Hell
     o world"

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

But if you change the type of the `padding` to the union of a string or a number, the compiler will report an error if you try to invoke `padLeft()` providing anything other than a string or a number. This will also eliminate the need to throw an exception. The new version of the `padLeft()` function is more bulletproof, as you can see in the following listing.

Listing B.15 union.ts with a union type

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

Allows 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
```

Another benefit of using a union type is that IDEs have an autocomplete feature that will prompt you with allowed argument types, so you won't even have a chance to

make such a mistake. In section 15.2.3 in chapter 15, there's another practical example of using a union type.

B.12 Using type definition files

The purpose of type definition files is to describe an API of a JavaScript library (or a script) providing types offered by this API. Say you want to use the popular JavaScript library Lodash in your Typescript code. If you have a Lodash type definition file in your project, the TypeScript static analyzer will know what types are expected by Lodash functions, and if you provide wrong types, you'll get a compile-time error. Also, an IDE will offer autocomplete for the Lodash API.

Initially, a TypeScript community created a repository of TypeScript definition files called *DefinitelyTyped* at http://definitelytyped.org. In 2016, Microsoft created an organization, `@types`, at npmjs.org, and this is what we use now. This organization has more than 5,000 type definition files for various JavaScript libraries.

The suffix of any type definition filename is *d.ts*, and you install type definition files using npm. For example, to install type definition files for Lodash, run the following command:

```
npm i @types/lodash --save-dev
```

This will download the Lodash definitions in the node_modules/@types directory of your project and will also update the package.json file, so you won't need to run this command again.

When you install Angular, you're getting the definition files in Angular modules in the subfolders of the node_modules/@angular folder after running `npm install`, as explained in chapter 1. All required d.ts files are bundled with Angular npm packages, and there's no need to install them separately. The presence of definition files in your project will allow the TypeScript compiler to ensure that your code uses the correct types while invoking the Angular API.

For example, Angular applications are launched by invoking the `bootstrapModule()` method, giving it the root module for your application as an argument. The application_ref.d.ts file includes the following definition for this function:

```
abstract bootstrapModule<M>(moduleType: Type<M>,
compilerOptions?: CompilerOptions | CompilerOptions[]): Promise<NgModuleRef<M>>;
```

By reading this definition, you (and the `tsc` compiler) know that this function can be invoked with one mandatory module parameter of type `Type<M>` and an optional array of compiler options. If application_ref.d.ts wasn't a part of your project, TypeScript's compiler would let you invoke the `bootstrapModule` function with a wrong parameter type, or without any parameters at all, which would result in a runtime error. But application_ref.d.ts is present, so TypeScript would generate a compile-time error reading "Supplied parameters do not match any signature of call target." Type

definition files also allow IDEs to show context-sensitive help when you're writing code that invokes Angular functions or assigns values to object properties.

Specifying type definition files explicitly

To explicitly specify the type definition files located in the node_modules/@types directory, add the required files to the `types` section of tsconfig.json. Here's an example:

```
"compilerOptions": {
   ...
  "types": ["es6-shim", "jasmine"],
}
```

In the past, we used special type definition managers `tsd` and `Typings` to install type definition files, but these managers are no longer needed. If your application uses other third-party JavaScript libraries, install their type definition files with npm to get compiler help and autocomplete in your IDE.

B.13 Controlling code style with TSLint

Linters help to ensure that code complies with the accepted coding style. With TSLint, you can enforce specified rules and coding styles for TypeScript. For example, you can configure TSLint to check that the TypeScript code in your project is properly aligned and indented, that the names of all interfaces start with a capital I, that class names use CamelCase notation, and so on.

You can install TSLint globally using the following command:

```
npm install tslint -g
```

To install the TSLint node module in your project directory, run the following command:

```
npm install tslint
```

The rules you want to apply to your code are specified in a tslint.json configuration file, which is generated by running `tslint init`:

```
{
 "defaultSeverity": "error",
 "extends": [
     "tslint: recommended"
 ],
 "jsRules": {},
 "rules": {},
 "rulesDirectory": []
}
```

A file with recommended rules comes with TSLint, but you can use custom rules of your preference. You can check the recommended rules in the node_modules/tslint/lib/configs/recommended.js file. The core TSLint rules are documented at http://mng.bz/xx6B. Your IDE may support linting with TSLint out of the box. If you generated your project with Angular CLI, it already includes TSLint.

appendix C Using the npm package manager

This appendix is an overview of the tools we use to install Angular and its dependencies using the npm package manager.

For most of this book, we use Node.js for installing software. Node.js (or simply *Node*) isn't just a framework or a library: it's a JavaScript runtime environment as well. We use the Node runtime for running various utilities like npm or launching JavaScript code without a browser. We also use npm scripts to automate building, testing, and deploying Angular apps.

To get started, download and install the current version of Node.js from https://nodejs.org. After installation is complete, open your terminal or command window and enter the following command:

```
node --version
```

This command should print the version of Node installed, for example, 10.3.0. Node comes with the package manager npm, which we use to install Angular and other packages from the npm registry located at www.npmjs.com. This repository hosts Angular as well as more than 400,000 other JavaScript packages.

The Node.js framework

Node.js is also a framework that can be used to develop JavaScript programs that run outside the browser. You can develop the server-side layer of a web application in JavaScript or Typescript. We write a web server using Node in chapter 12 by using Node.js and Express frameworks. Google developed a high-performance V8 JavaScript engine for the Chrome browser, and it can also be used to run code written using the Node.js API. The Node.js framework includes an API to work with the filesystem, access databases, listen to HTTP requests, and more.

To install a JavaScript library, run the command `npm install`, or `npm i` for short. Say you want to install the TypeScript compiler locally. Open the Terminal in any directory and run the following command:

```
npm i typescript
```

After this command completes, you'll see a new subdirectory named node_modules, where the TypeScript compiler has been installed. npm always installs packages in the node_modules directory. If such a directory doesn't exist, npm will create it.

If you want to install a package globally, add the `-g` option:

```
npm i typescript -g
```

This time the TypeScript compiler will be installed not in the current directory, but globally in the lib/node_modules subdirectory of your Node.js installation.

If you want to install a specific version of the package, add the version number to the package name after the `@` sign. For example, to install Typescript 2.9.0 globally, use the following command:

```
npm i typescript@2.9.0 -g
```

All available options of the `npm install` command are described at https://docs.npmjs.com/cli/install.

In some cases, you may want to have the same package installed both locally and globally. As a example, you may have the TypeScript compiler 2.7 installed locally, and TypeScript 2.9 installed globally. To run the global version of this compiler, you enter the `tsc` command in your terminal or command window, and to run the locally installed compiler, you could use the following command from your project directory:

```
node_modules/.bin/tsc
```

A typical Node-based project may have multiple dependencies, and we don't want to keep running separate `npm i` commands to install each package. Creating a package.json file is a better way to specify all project dependencies.

C.1 Specifying project dependencies in package.json

To start a new Node-based project, create a new directory (for example, my-node-project), open your terminal or command window, and change the current working directory to the newly created one. Then run the `npm init -y` command, which will create the initial version of the package.json configuration file. Normally, `npm init` asks several questions while creating the file, but the `-y` flag makes it accept the default values for all options. The following example shows this command running in the empty my-node-project directory:

```
$ npm init -y
Wrote to /Users/username/my-node-project/package.json:
```

```
{
  "name": "my-node-project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Most of the generated configuration is needed either for publishing the project into the npm registry or while installing the package as a dependency for another project. We'll use npm only for managing project dependencies and automating development and build processes.

Because we're not going to publish it into the npm registry, remove all the properties except `name`, `description`, and `scripts`. Also, add a `"private": true` property, because it's not created by default. That will prevent the package from being accidentally published to the npm registry. The package.json file should look like this:

```
{
  "name": "my-node-project",
  "description": "",
  "private": true,
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  }
}
```

The `scripts` configuration allows you to specify command aliases that you can run in your command window. By default, `npm init` creates the `test` alias, which can be run like this: `npm test`. The generated script command include double ampersands, `&&`, which are used as a separator between commands. When you run `npm test`, it runs two commands: `echo \"Error: no test specified\"` and `exit -1`. npm scripts support about a dozen command names like `test`, `start`, and others. A list of these commands is at https://docs.npmjs.com/misc/scripts.

You can create your own command aliases with any names:

```
"scripts": {
    "deploy": "copyfiles -f dist/** ../server/build/public",
  }
```

Because `deploy` is a custom alias name, you need to run this command by adding the keyword `run`:

```
npm run deploy
```

In section 12.3.6 in chapter 12, we discuss how to write npm scripts for building and deploying an app on the web server.

If you generate package.json using the `npm init` command, it'll be missing two important sections: `dependencies` and `devDependencies`. Let's see how dependencies of an Angular project are specified. Figure C.1 shows a fragment from a package.json file of a typical Angular project.

```
"dependencies": {
  "@angular/animations": "^6.0.0",
  "@angular/common": "^6.0.0",
  "@angular/compiler": "^6.0.0",
  "@angular/core": "^6.0.0",
  "@angular/forms": "^6.0.0",
  "@angular/http": "^6.0.0",
  "@angular/platform-browser": "^6.0.0",
  "@angular/platform-browser-dynamic": "^6.0.0",
  "@angular/router": "^6.0.0",
  "core-js": "^2.5.4",
  "rxjs": "^6.0.0",
  "ws": "^5.2.0",
  "zone.js": "^0.8.26"
},
"devDependencies": {
  "@angular/compiler-cli": "^6.0.0",
  "@angular-devkit/build-angular": "~0.6.1",
  "typescript": "~2.7.2",
  "@angular/cli": "~6.0.1",
  "@angular/language-service": "^6.0.0",
  "@types/jasmine": "~2.8.6",
  "@types/jasminewd2": "~2.0.3",
  "@types/node": "~8.9.4",
  "codelyzer": "~4.2.1",
  "jasmine-core": "~2.99.1",
  "jasmine-spec-reporter": "~4.2.1",
  "karma": "~1.7.1",
  "karma-chrome-launcher": "~2.2.0",
  "karma-coverage-istanbul-reporter": "~1.4.2",
  "karma-jasmine": "~1.1.1",
  "karma-jasmine-html-reporter": "^0.2.2",
  "protractor": "~5.3.0",
  "ts-node": "~5.0.1",
  "tslint": "~5.9.1"
}
```

Figure C.1 Angular dependencies in package.json

It looks intimidating, but the good news is that you don't need to remember all these packages and their versions, because you'll be generating projects with Angular CLI, which will create the file package.json with the proper content.

The `dependencies` section lists all the packages required for your application to run. As you can see, the Angular framework comes in several packages with names starting with `@angular`. You may not need to install all of them. If your app doesn't use forms, for example, there's no need to include `@angular/forms` in your package.json.

The `devDependencies` section lists all the packages required to be installed on a developer's computer. It includes several packages, and none of them is needed on the production server. There's no need to have testing frameworks or a TypeScript compiler on the production machine, right?

To install a single package with npm, list the name of this package. For example, to add the Lodash library to your node_modules directory, run the following command:

```
npm i lodash
```

To add the package to node_modules and add the corresponding dependency to the `dependencies` section of your package.json file, you can explicitly specify the `--save-prod` option :

```
npm i lodash --save-prod
```

You can also use the abbreviation `-P` for `--save-prod`. If no options are specified, the `npm i lodash` command updates the `dependencies` section of the package.json file.

To add the package to node_modules and add the corresponding dependency to the `devDependencies` section of your package.json file, use the `--save-dev` option:

```
npm i protractor --save-dev
```

You can also use the abbreviation `-D` for `--save-dev`.

Sometimes the GitHub version of a package has an important bug fix that hasn't been released yet on npmjs.org. If you want install such a package from GitHub, you need to replace the version number in package.json dependencies with its location on GitHub. Changing the dependency to include the name of the GitHub organization and repository should allow you to install the latest builds from GitHub versions of this library. For example:

```
"@angular/flex-layout": "angular/flex-layout-builds"
```

The preceding configuration will work, assuming that the master branch of the Flex Layout library has no code issues preventing npm from installing it.

C.2 Semantic versioning

Numbering of Angular releases uses a set of rules known as *semantic versioning*. The version of a package consists of three digits—for example, 6.1.2. The first digit denotes a major release that includes new features and potentially breaking changes in the API. The second digit represents a minor release, which introduces new backward-compatible APIs but has no breaking changes. The third digit represents backward-compatible patches with bug fixes.

Take another look at figure C.1 in the previous section. Every package has a three-digit version, and many of them have additional symbols: ^ or ~. If the specified version has just the three digits, that means you instruct npm to install exactly that version. For example, the following line in package.json tells npm to install Angular CLI version 6.0.5, ignoring the newer versions even if they're available:

```
"@angular/cli": "6.0.5"
```

Many packages in that package.json file have the hat sign ^ in front of the version. For example:

```
"@angular/core": "^6.0.0"
```

This means you'll allow npm to install the latest minor release of version 6 if available. If the latest version of the Angular Core package is 6.2.1, it will be installed.

The tilde ~ means you want to install the latest patch for the given major and minor version:

```
"jasmine-core": "~2.99.1"
```

There are many other symbols you can use with version numbers with npm—see http://mng.bz/YnyW for details.

C.3 Yarn as an alternative to npm

Yarn (see https://yarnpkg.com) is another package manager that can be used as an alternative to npm. Prior to version 5, npm was slow, which was one reason we started using the faster Yarn.

Now npm is fast too, but Yarn has an additional benefit: it creates the file yarn.lock that keeps track of exact versions of packages that were installed in your project. Let's say your package.json file has a `"@angular/core": "^6.0.0"` dependency, and your project has no yarn.lock file. If version 6.1.0 is available, it'll be installed, and yarn.lock is created with a record about the version 6.1.0. If you run `yarn install` in a month, and if yarn.lock exists in your project, Yarn will use it and install version 6.1.0, even if version 6.2.0 is available.

The following fragment from yarn.lock shows that although the dependency in package.json for the `@angular/core` package was set as `^6.0.0`, version `6.0.2` was installed:

```
"@angular/core@^6.0.0":
  version "6.0.2"
  resolved "https://registry.yarnpkg.com/@angular/core/-/core-
     6.0.2.tgz#d183..."
  dependencies:
    tslib "^1.9.0"
```

In a team setup, you should check the yarn.lock file into the version control repository so every member of your team has the same versions of packages.

npm also creates the package-lock.json file, but npm isn't designed to install exact package version(s) listed in this file if you run `npm install` (see https://github.com/npm/npm/issues/17979). The good news is that starting from version 5.7, npm supports the `npm ci` command, which ignores the versions listed in package.json, but installs the versions listed in the package-lock.json file.

If at some point you decide to upgrade packages, overriding the versions stored in yarn.lock, run the `yarn upgrade-interactive` command, as shown in figure C.2.

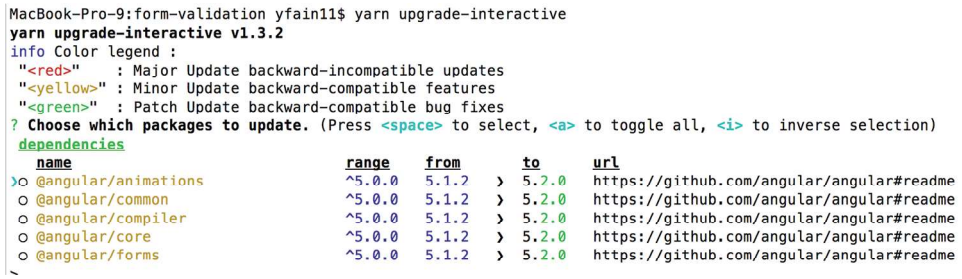


Figure C.2 Upgrading package versions with Yarn

Yarn works with your project's package.json file, so there's no need for any additional configuration. You can read more about using Yarn at https://yarnpkg.com/ en/docs.

TIP You can ask Angular CLI to use Yarn instead of npm while installing dependencies of the newly generated project. Starting with Angular CLI 6, you can do this by using the following command:

```
ng config --global cli.packageManager yarn
```

If you use older versions of Angular CLI, use the following command:

```
ng set --global packageManager=yarn
```

appendix D
RxJS essentials

Synchronous programming is relatively straightforward in that each line of your code is executed after the previous one. If you invoke a function in line 25 that returns a value, you can use the returned value as an argument for the function invoked in line 26.

Asynchronous programming dramatically increases code complexity. In line 37, you can invoke an asynchronous function that will return the value sometime later. Can you invoke a function in line 38 that uses the value returned by the previous function? The short answer is, "It depends."

This appendix is an introduction to the RxJS 6 library, which can be used with any JavaScript-based app. It shines when it comes to writing and composing asynchronous code. Because Angular uses the RxJS library internally, we decided to add a primer to this book.

The first library of reactive extensions (Rx) was created by Erik Meijer in 2009. Rx.NET was meant to be used for apps written with Microsoft .Net technology. Then the Rx extensions were ported to multiple languages, and in the JavaScript world, RxJS 6 is the current version of this library.

NOTE Though Angular depends on RxJS and can't function without it, RxJS itself is an independent library that can be used in any JavaScript app.

Let's see what being reactive means in programming by considering a simple example:

```
let a1 = 2;
let b1 = 4;
let c1 = a1 + b1;  // c1 = 6
```

This code adds the values of the variables `a1` and `b1`, and `c1` is equal to `6`. Now let's add a couple of lines to this code, modifying the values of `a1` and `b1`:

```
let a1 = 2;
let b1 = 4;
let c1 = a1 + b1;  // c1 = 6

a1 = 55;        // c1 = 6 but should be 59
b1 = 20;        // c1 = 6 but should be 75
```

While the values of `a1` and `b1` change, `c1` doesn't react to these changes, and its value is still `6`. You can write a function that adds `a1` and `b1` and invokes it to get the latest value of `c1`, but this would be an *imperative* style of coding, where you dictate when to invoke a function to calculate the sum.

Wouldn't it be nice if `c2` were automatically recalculated upon any `a1` or `b1` changes? Think of a spreadsheet program like Microsoft Excel, where you could put a formula like `=sum(a1, b1)` into the C1 cell, and C1 would react immediately upon changes in A1 and B1. In other words, you don't need to click any button to refresh the value of C1—the data is pushed to this cell.

In the *reactive* style of coding (as opposed to the imperative one), the changes in data drive the invocation of your code. Reactive programming is about creating responsive, event-driven applications, where an observable event stream is pushed to subscribers, who observe and handle the events.

In software engineering, Observer/Observable is a well-known pattern and is a good fit in any asynchronous-processing scenario. But reactive programming is a lot more than just an implementation of the Observer/Observable pattern. Observable streams can be canceled, they can notify about the end of a stream, and the data pushed to the subscriber can be transformed on the way from the data producer to the subscriber by applying one or more composable operators.

D.1 Getting familiar with RxJS terminology

We want to observe data, which means there's a data producer—a server sending data using HTTP or WebSockets, a UI input field where a user enters some data, an accelerometer in a smartphone, and so on. An *observable* is a function (or object) that gets the producer data and pushes it to the subscriber(s). An *observer* is an object (or function) that knows how to handle data elements pushed by the observable, as shown in figure D.1.

The main players of RxJS are as follows:

- *Observable*—Data stream that pushes data over time
- *Observer*—Consumer of an observable stream
- *Subscriber*—Connects observer with observable
- *Operator*—Function for en route data transformation

We'll introduce each of these players by showing multiple examples of their use. For complete coverage, refer to the RxJS documentation available at http://reactivex.io/rxjs.

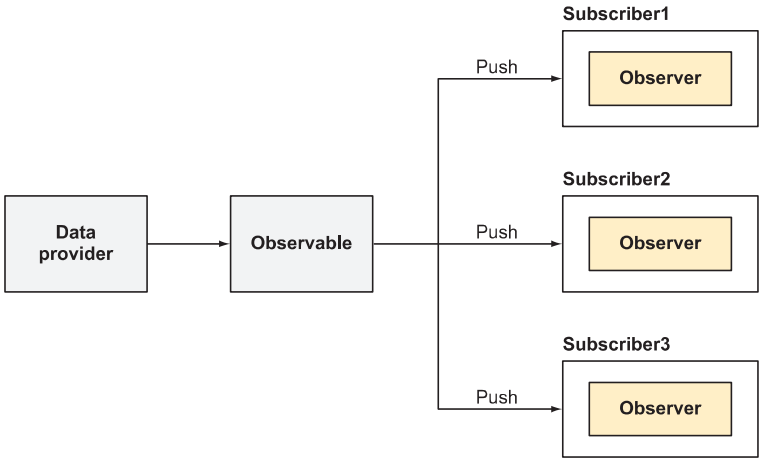


Figure D.1 The data flow from observable to observers

Hot and cold observables

There are two types of observables: hot and cold. The main difference is that a cold observable creates a data producer for each subscriber, whereas a hot observable creates a data producer first, and each subscriber gets the data from one producer, starting from the moment of subscription.

Let's compare watching a movie on Netflix to going into a movie theater. Think of yourself as an observer. Anyone who decides to watch *Mission: Impossible* on Netflix will get the entire movie, regardless of when they hit the play button. Netflix creates a new producer to stream a movie just for you. This is a cold observable.

If you go to a movie theater and the showtime is 4 p.m., the producer is created at 4 p.m., and the streaming begins. If some people (subscribers) are late to the show, they miss the beginning of the movie and can only watch it starting from the moment of arrival. This is a hot observable.

A cold observable starts producing data when some code invokes a `subscribe()` function on it. For example, your app may declare an observable providing a URL on the server to get certain products. The request will be made only when you subscribe to it. If another script makes the same request to the server, it'll get the same set of data.

A hot observable produces data even if no subscribers are interested in the data. For example, an accelerometer in your smartphone produces data about the position of your device, even if no app subscribes to this data. A server can produce the latest stock prices even if no user is interested in this stock.

Most of the examples in this appendix are about cold observables.

D.2 Observable, observer, and subscriber

As stated earlier, an observable gets data from a data source (a socket, an array, UI events) one element at a time. To be precise, an observable knows how to do three things:

- Emit the next element to the observer
- Throw an error on the observer
- Inform the observer that the stream is over

Accordingly, an observer object provides up to three callbacks:

- The function to handle the next element emitted by the observable
- The function to handle errors thrown by the observable
- The function to handle the end of a stream

The subscriber connects an observable and observer by invoking the `subscribe()` method and disconnects them by invoking `unsubscribe()`. A script that subscribes to an observable has to provide the observer object that knows what to do with the produced elements. Let's say you create an observable represented by the variable `someObservable` and an observer represented by the variable `myObserver`. You can subscribe to such an observable as follows:

```
let mySubscription: Subscription = someObservable.subscribe(myObserver);
```

To cancel the subscription, invoke the `unsubscribe()` method:

```
mySubscription.unsubscribe();
```

How can an observable communicate with the provided observer? By invoking the following functions on the observer object:

- `next()`, to push the next data element to the observer
- `error()`, to push the error message to the observer
- `complete()`, to send a signal to the observer about the end of a stream

You'll see an example of using these functions in section D.5.

D.3 Creating observables

RxJS offers several ways of creating an observable, depending on the type of the data producer—for example, a data producer for a DOM event, a data collection, a custom function, a WebSocket, and more.

Here are some examples of the API to create an observable:

- `of(1,2,3)`—Turns the sequence of numbers into an `Observable`
- `Observable.create(myObserver)`—Returns an `Observable` that can invoke methods on `myObserver` that you'll create and supply as an argument
- `from(myArray)`—Converts an array represented by the `myArray` variable into an `Observable`. You can also use any iterable data collection or a generator function as an argument of `from()`.

- `fromEvent(myInput, 'keyup')`—Converts the `keyup` event from an HTML element represented by `myInput` into an `Observable`. Chapter 6 has an example of using the `fromEvent()` API.
- `interval(1000)`—Emits a sequential integer (0,1,2,3...) every second

TIP There's a proposal for introducing `Observable` into future versions of ECMAScript. See https://github.com/tc39/proposal-observable.

Let's create an observable that will emit 1, 2, and 3 and subscribe to this observable.

Listing D.1 Emitting 1,2,3

```
of(1,2,3)
    .subscribe(
        value => console.log(value),
        err => console.error(err),
        () => console.log("Streaming is over")
);
```

Handles the error

Handles the value emitted by the observable

Handles the stream completion message

Note that you pass three fat-arrow functions to `subscribe()`. These three functions combined are the implementation of your observer. The first function will be invoked for each element emitted by the observable. The second function will be invoked in case of an error, providing the object representing the error. The third function takes no arguments and will be invoked when the observable stream is over. Running this code sample will produce the following output on the console:[1]

```
1
2
3
Streaming is over
```

NOTE In appendix A, we discuss using the `Promise` object, which can invoke an event handler specified in the `then()` function only once. Think of a `subscribe()` method as a replacement of the `then()` invocation on a `Promise` object, but the callback for `subscribe()` is invoked not just once, but for each emitted value.

D.4 Getting familiar with RxJS operators

As data elements flow from an observable to an observer, you can apply one or more *operators*, which are functions that can process each element prior to supplying it to the observer. Each operator takes an observable as an input, performs its action, and returns a new observable as an output, as seen in figure D.2.

Because each operator takes in an observable and creates an observable as its output, operators can be chained so that each observable element can go through several transformations prior to being handed to the observer.

[1] See it in CodePen: http://mng.bz/MwTz. Open the console view at the bottom to see the output.

Figure D.2 An operator: observable in, observable out

RxJS offers about 100 various operators, and their documentation may not always be easy to understand. On the positive side, the documentation often illustrates operators with marble diagrams. You can get familiar with the syntax of marble diagrams at http://mng.bz/2534. Figure D.3 shows how the RxJS manual illustrates the `map` operator with a marble diagram (see http://mng.bz/65G7).

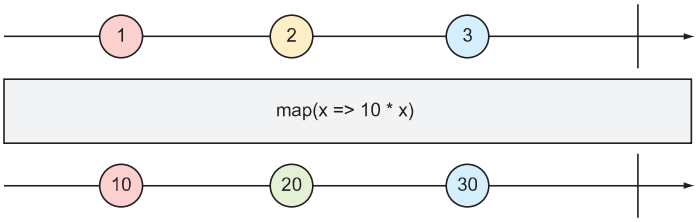


Figure D.3 The map operator

At the top, a marble diagram shows a horizontal line with shapes representing a stream of incoming observable elements. Next, there's the illustration of what a particular operator does. At the bottom, you see another horizontal line depicting the outgoing observable stream after the operator has been applied. The vertical bar represents the end of the stream. When you look at the diagram, think of time as moving from left to right. First, the value 1 was emitted, then time went by, 2 was emitted, then time went by, 3 was emitted, and then the stream ended.

The `map` operator takes a transforming function as an argument and applies it to each incoming element. Figure D.3 shows the `map` operator that takes a value of each incoming element and multiplies it by 10.

Now let's get familiar with the marble diagram of the `filter` operator, shown in figure D.4. The `filter` operator takes a function predicate as an argument, which returns `true` if the emitted value meets the criteria, and `false` otherwise. Only the values that meet the criteria will make it to the subscriber. This particular diagram uses

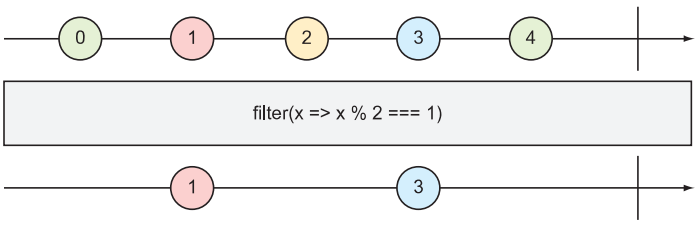


Figure D.4 The filter operator

the fat-arrow function that checks whether the current element is an odd number. Even numbers won't make it further down the chain to the observer.

Operators are composable, and you can chain them so the item emitted by the observable can be processed by a sequence of operators before it reaches the observer.

> **Deprecated operator chaining**
> Prior to RxJS 6, you could chain operators using the dot between operators.

Listing D.2 Dot-chainable operators

```
const beers = [
    {name: "Stella", country: "Belgium", price: 9.50},
    {name: "Sam Adams", country: "USA", price: 8.50},
    {name: "Bud Light", country: "USA", price: 6.50}
];

from(beers)
    .filter(beer => beer.price < 8)
     .map(beer => `${beer.name}: $${beer.price}`)
     .subscribe(
        beer => console.log(beer),
        err => console.error(err)
);

console.log("This is the last line of the script");
```

Applies the filter operator

Dot-chains the map operator

Starting with RxJS 6, the only way to chain operators is by using the `pipe()` method, passing to it comma-separated operators as arguments. The next section introduces *pipeable* operators.

D.4.1 Pipeable operators

Pipeable operators are those that can be chained using the `pipe()` function. We'll talk about dot-chaining operators first to explain why pipeable operators were introduced in RxJS.

If you have RxJS prior to version 6 installed, you can import dot-chaining operators from the rxjs/add/operator directory. For example:

```
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/filter';
```

These operators patch the code of the `Observable.prototype` and become a part of this object. If you decide later on to remove, say, the `filter` operator from the code that handles the observable stream, but you forget to remove the corresponding import statement, the code that implements `filter` would remain a part of `Observable.prototype`. When bundlers tried to eliminate the unused code (*tree shaking*), they may

decide to keep the code of the `filter` operator in the `Observable` even though it's not being used in the app.

RxJS 5.5 introduced pipeable operators, pure functions that don't patch the `Observable`. You can import operators using ES6 import syntax (for example, `import {map} from 'rxjs/operators'`) and then wrap them into a `pipe()` function that takes a variable number of parameters, or chainable operators.

The subscriber in listing D.2 will receive the same data as the one in the sidebar "Deprecated operator chaining," but it's a better version from the tree-shaking perspective, because it uses pipeable operators. This listing includes import statements, assuming that RxJS is locally installed.

Listing D.3 Using pipeable operators

```
import {map, filter} from 'rxjs/operators';
 import {from} from 'rxjs';
 ...
from(beers)
    .pipe(
          filter(beer => beer.price < 8),
          map(beer => `${beer.name}: $${beer.price}`)
      )
    .subscribe(
        beer => console.log(beer),
        err => console.error(err)
);
```

Imports pipeable operators from rxjs/operators instead of rxjs/add/operator

Imports the from() function

Wraps pipeable operators into the pipe() function

Now if you remove the line `filter` from listing D.2, the tree-shaking module of the bundlers (such as Webpack 4) can recognize that the imported function isn't used, and the code of the `filter` operator won't be included in the bundles.[2]

By default, the `from()` function returns a synchronous observable, but if you want an asynchronous one, use a second argument specifying an async scheduler:

```
from(beers, Scheduler.async)
```

Making this change in the preceding code sample will print "This is the last line of the script" first and then will emit the beers info. You can read more about the scheduler at http://mng.bz/744Y.

Now we'd like to introduce the `reduce` operator, which allows you to aggregate values emitted by an observable. A marble diagram of the `reduce` operator is shown in figure D.5. This diagram shows an observable that emits 1, 3, and 5, and the `reduce` operator adds them up, producing the accumulated value of 9.

The `reduce` operator has two arguments: an accumulator function where we specify how to aggregate the values, and the initial (seed) value to be used by the accumulator function. Figure D.5 shows that 0 was used as an initial value, but if we changed it to 10, the accumulated result would be 19.

[2] See it in CodePen: http://mng.bz/RqO5.

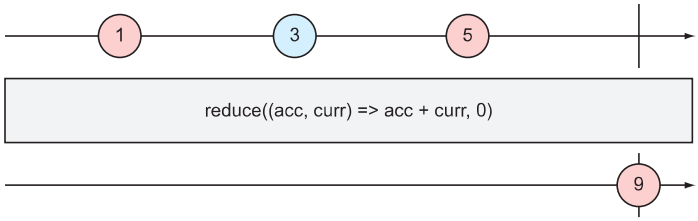


Figure D.5 The reduce operator

As you see in figure D.5, the accumulator function also has two arguments:

- `acc` stores the currently accumulated value, which is available for each emitted element.
- `curr` stores the currently emitted value.

The following listing creates an observable from the `beers` array and applies two operators to each emitted element: `map` and `reduce`. The `map` operator takes a `beer` object and extracts its price, and the `reduce` operator adds the prices.

Listing D.4 Using the `map` and `reduce` operators

```
const beers = [
    {name: "Stella", country: "Belgium", price: 9.50},
    {name: "Sam Adams", country: "USA", price: 8.50},
    {name: "Bud Light", country: "USA", price: 6.50},
    {name: "Brooklyn Lager", country: "USA", price: 8.00},
    {name: "Sapporo", country: "Japan", price: 7.50}
];

from(beers)
    .pipe(
      map(beer =>  beer.price),
       reduce( (total, price) => total + price, 0)
     )
    .subscribe(
        totalPrice => console.log(`Total price: ${totalPrice}`)
 );
```

Transforms the beer object into its price

Sums the prices

Prints the total price of all beers

Running this script will produce the following output:

```
Total price: 40
```

In this script, we were adding all prices, but we could apply any other calculations to the aggregate value, such as to calculate an average or maximum price.

The `reduce` operator emits the aggregated result when the observable completes. In this example, it happened naturally, because we created an observable from an array with a finite number of elements. In other scenarios, we'd need to invoke the

complete() method on the observer explicitly; you'll see how to do that in the next section.[3]

Code samples from this section have been turning the array into an observable, and magically pushing the array elements to the observer. In the next section, we'll show you how to push elements by invoking the next() function on the observer.

Debugging observables

The tap operator can perform a side effect (for example, log some data) for every value emitted by the source observable, but return an observable that's identical to the source. In particular, these operators can be used for debugging purposes.

Say you have a chain of operators and want to see the observable values before and after a certain operator is applied. The tap operator will allow you to log the values:

```
import { map, tap } from 'rxjs/operators';

myObservable$
  .pipe(
    tap(beer => console.log(`Before: ${beer}`)),
    map(beer => `${beer.name}, ${beer.country}`),
    tap(beer => console.log(`After: ${beer}`))
  )
  .subscribe(...);
```

In this example, you print the emitted value before and after the map operator is applied. The tap operator doesn't change the observable data—it passes it through to the next operator or the subscribe() method.

D.5 Using an observer API

An *observer* is an object that implements one or more of these functions: next(), error(), and complete(). Let's use an object literal to illustrate an observer, but later in this section, we'll use a simplified syntax with arrow functions:

```
const beerObserver = {
  next: function(beer) { console.log(`Subscriber got ${beer.name}`)},
  error: function(err) { console.err(error)},
  complete: function() {console.log("The stream is over")}
}
```

We can create an observable with the create method, passing an argument that represents an observer. When an observable gets created, it doesn't know yet which concrete object will be provided. That'll be known later, at the time of subscription:

```
const beerObservable$ = Observable.create( observer => observer.next(beer));
```

[3] See it in CodePen: http://mng.bz/68fR.

This particular observable thinks "When someone subscribes to my beers, they'll provide me with a concrete beer consumer, and I'll push one beer object to this guy." At the time of subscription, we'll provide a concrete observer to our observable:

```
beerObservable$.subscribe(beerObserver);
```

The observer will get the beer and will print on the console something like this:

```
Subscriber got Stella
```

The next listing has a complete script that illustrates creation of the observer, the observable, and the subscription. The `getObservableBeer()` function creates and returns an observable that will loop through the array of beers and push each beer to the observer by invoking `next()`. After that, our observable will invoke `complete()` on the observer, indicating that there won't be any more beers.

Listing D.5 Using `Observable.create()`

```
function getObservableBeer(){

    return Observable.create( observer => {

      const beers = [
        {name: "Stella", country: "Belgium", price: 9.50},
        {name: "Sam Adams", country: "USA", price: 8.50},
        {name: "Bud Light", country: "USA", price: 6.50},
        {name: "Brooklyn Lager", country: "USA", price: 8.00},
        {name: "Sapporo", country: "Japan", price: 7.50}
      ];

         beers.forEach( beer => observer.next(beer));

         observer.complete();
         }
    );
}

getObservableBeer()
   .subscribe(
      beer => console.log(`Subscriber got ${beer.name}`),
     error => console.err(error),
        () => console.log("The stream is over")
);
```

Creates and returns the observable object

Pushes each beer to the observer

Pushes the end of the stream message to the observer

Subscribes to the observable, providing the observer object in the form of three fat-arrow functions

The output of this script is shown next:[4]

```
Subscriber got Stella
Subscriber got Sam Adams
```

[4] See it in CodePen: http://mng.bz/Q7sb.

```
Subscriber got Bud Light
Subscriber got Brooklyn Lager
Subscriber got Sapporo
The stream is over
```

In our code sample, we were invoking `next()` and `complete()` on the observer. But keep in mind that an observable is just a data pusher, and there's always a data producer (the array of beers, in our case) that may generate an error. In that case, we'd invoke `observer.error()`, and the stream would complete. There's a way to intercept an error on the subscriber's side to keep the streaming alive, discussed in section D.9.

It's important to note that our data producer (the array of beers) is created inside the `getObservableBeer()` observable, which makes it a cold observable. A WebSocket could be another example of the producer. Imagine we have a database of beers on the server, and we can request them over a WebSocket connection (we could use HTTP or any other protocol here):

```
Observable.create((observer) => {
  const socket = new WebSocket('ws://beers');
  socket.addEventListener('message', (beer) => observer.next(beer));
  return () => socket.close(); // is invoked on unsubscribe()
});
```

With cold observables, each subscriber will get the same beers, regardless of the time of subscription, if the query criteria (in our case, show all beers) are the same.

D.6 Using RxJS Subject

An RxJS `Subject` is an object that contains an observable and the observer(s). This means you can push the data to its observer(s) using `next()`, as well as subscribe to it. A `Subject` can have multiple observers, which makes it useful when you need to implement for *multicasting*—emitting a value to multiple subscribers, as shown in figure D.6.

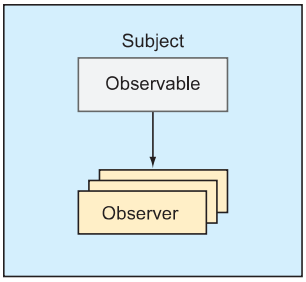


Figure D.6 RxJS `Subject`

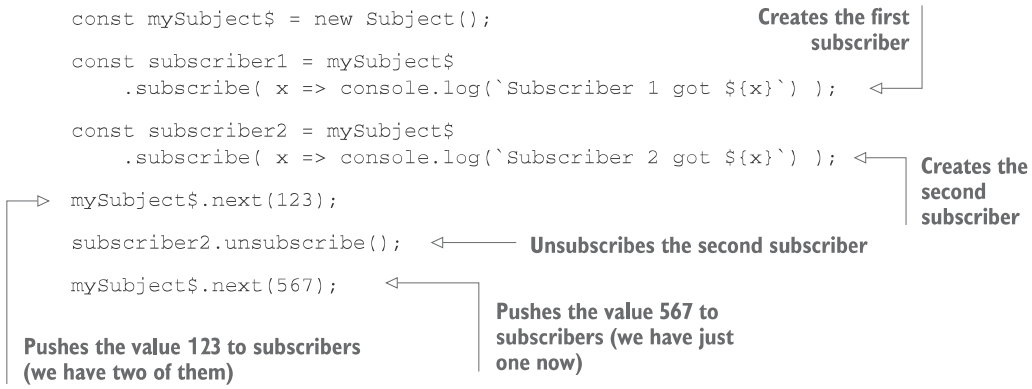
Say you have an instance of a `Subject` and two subscribers. If you push a value to the subject, each subscriber will receive it:

```
const mySubject$ = new Subject();

const subscription1 = mySubject$.subscribe(...);
const subscription2 = mySubject$.subscribe(...);
...
 mySubject$.next(123); // each subscriber gets 123
```

The following example has one `Subject` with two subscribers. The first value is emitted to both subscribers, and then one of them unsubscribes. The second value is emitted to one active subscriber.

Listing D.6 One subject and two subscribers

```
const mySubject$ = new Subject();

const subscriber1 = mySubject$
    .subscribe( x => console.log(`Subscriber 1 got ${x}`) );

const subscriber2 = mySubject$
    .subscribe( x => console.log(`Subscriber 2 got ${x}`) );

mySubject$.next(123);

subscriber2.unsubscribe();

mySubject$.next(567);
```



Running this script produces the following output on the console:[5]

```
Subscriber 1 got 123
Subscriber 2 got 123
Subscriber 1 got 567
```

TIP There's a naming convention to end the names of variables of type `Observable` or `Subject` with a dollar sign.

Now let's consider a more practical example. A financial firm has traders who can place orders to buy or sell stocks. Whenever the trader places an order, it has to be given to two scripts (subscribers):

- The script that knows how to place orders with a stock exchange
- The script that knows how to report each order to a trade commission that keeps track of all trading activities

The following listing, written in TypeScript, shows how to ensure that both subscribers can receive orders as soon as a trader places them. We create an instance of `Subject` called `orders`, and whenever we invoke `next()` on it, both subscribers will receive the order.

[5] See it in CodePen: http://mng.bz/jx16.

Listing D.7 Broadcasting trade orders

```
enum Action{
     Buy = 'BUY',
    Sell = 'SELL'
}

class Order{
     constructor(public orderId: number, public traderId: number,
    ➥public stock: string, public shares: number, public action:Action){}
}

const orders$ = new Subject<Order>();

class Trader {

    constructor(private traderId:number, private traderName:string){}

    placeOrder(order: Order){
        orders$.next(order);
     }
}

const stockExchange = orders$.subscribe(
     ord => console.log(`Sending to stock exchange the order to
    ➥${ord.action} ${ord.shares} shares of ${ord.stock}`));
const tradeCommission = orders$.subscribe(
     ord => console.log(`Reporting to trade commission the order to
    ➥${ord.action} ${ord.shares} shares of ${ord.stock}`));

const trader = new Trader(1, 'Joe');
const order1 = new Order(1, 1,'IBM',100,Action.Buy);
const order2 = new Order(2, 1,'AAPL',100,Action.Sell);

trader.placeOrder( order1);
trader.placeOrder( order2);
```

Uses enums to declare which actions are allowed for orders

A class representing an order

A subject instance that works only with Order objects

A class representing a trader

When an order is placed, pushes it to subscribers

A stock exchange subscriber

A trade commission subscriber

Places the first order

Places the second order

Running listing D.6 produces the following output:[6]

```
Sending to stock exchange the order to BUY 100 shares of IBM
Reporting to trade commission the order to BUY 100 shares of IBM
Sending to stock exchange the order to SELL 100 shares of AAPL
Reporting to trade commission the order to SELL 100 shares of AAPL
```

NOTE In listing D.6, we use TypeScript enums that allow us to define a limited number of constants. Placing the actions to buy or sell inside an `enum` provides additional type checking to ensure that our script uses only the allowed actions. If we used string constants like `"SELL"` or `"BUY"`, the developer could misspell a word (`"BYE"`) while creating an order. By declaring `enum Action`, we restrict possible actions to `Action.Buy` or `Action.Sell`. Trying to use `Action.Bye` results in a compilation error.

[6] See it in CodePen: http://mng.bz/4PIH.

TIP We wrote listing D.6 in TypeScript, but if you want to see its JavaScript version, run `npm install` and the `tsc` commands in the project that comes with this appendix. The original code is located in the subject-trader.ts file, and the compiled version is in subject-trader.js.

Chapter 6 contains an example of using a `BehaviorSubject`—a special flavor of `Subject` that always emits its last or initial value to new subscribers.

D.7 The flatMap operator

In some cases, you need to treat each item emitted by an observable as another observable. The outer observable emits the inner observables. Does that mean you need to write nested `subscribe()` calls (one for the outer observable and another for the inner one)? No, you don't. The `flatMap` operator autosubscribes to each item from the outer observable.

Some operators are not explained well in RxJS documentation, and we recommend you refer to the general ReactiveX (reactive extensions) documentation for clarification. The `flatMap` operator is better explained at http://mng.bz/7RQB, which states that `flatMap` is used to "transform the items emitted by an observable into observables, then flatten the emissions from those into a single observable." This documentation includes the marble diagram shown in figure D.7.

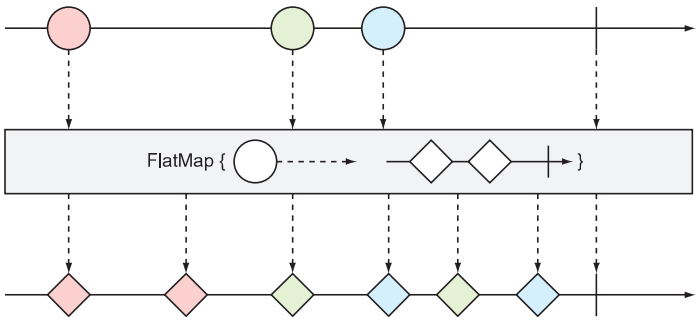


Figure D.7 The `flatMap` operator

As you see, the `flatMap` operator takes an emitted item from the outer observable (the circle) and unwraps its content (the inner observable of diamonds) into the flattened output observable stream. The `flatMap` operator merges the emissions of the inner observables, so their items may interleave.

Listing D.8 has an observable that emits drinks, but this time it emits not individual drinks, but palettes. The first palette has beers, and the second, soft drinks. Each palette is an observable. We want to turn these two palettes into an output stream with individual beverages.

Listing D.8 Unwrapping nested observables with `flatMap`

```
function getDrinks() {

    const beers$ = from([
         {name: "Stella", country: "Belgium", price: 9.50},
        {name: "Sam Adams", country: "USA", price: 8.50},
        {name: "Bud Light", country: "USA", price: 6.50}
    ], Scheduler.async);

    const softDrinks$ = from([
         {name: "Coca Cola", country: "USA", price: 1.50},
        {name: "Fanta", country: "USA", price: 1.50},
        {name: "Lemonade", country: "France", price: 2.50}
    ], Scheduler.async);

    return Observable.create( observer => {
            observer.next(beers$);
             observer.next(softDrinks$);
             observer.complete();
        }
    );
}

// We want to "unload" each palette and print each drink info

getDrinks()
  .pipe(flatMap(drinks => drinks))
   .subscribe(
       drink => console.log(`Subscriber got ${drink.name}: ${drink.price}`),
      error => console.err(error),
      () => console.log("The stream of drinks is over")
  );
```

Creates an async observable from beers

Creates an async observable from soft drinks

Emits the beers observable with next()

Emits the soft drinks observable with next()

Unloads drinks from pallets into a merged observable

Subscribes to the merged observable

This script will produce output that may look as follows:[7]

```
Subscriber got Stella: 9.5
Subscriber got Coca Cola: 1.5
Subscriber got Sam Adams: 8.5
Subscriber got Fanta: 1.5
Subscriber got Bud Light: 6.5
Subscriber got Lemonade: 2.5
The stream of observables is over
```

Are there any other uses of the `flatMap` operator besides unloading palettes of drinks? Another scenario where you'd want to use `flatMap` is when you need to execute more than one HTTP request, in which the result of the first request should be given to the second one, as shown in the following listing. In Angular, HTTP requests return observables, and without `flatMap()`, this could be done (it a bad style) with nested `subscribe` calls.

[7] See it in CodePen: http://mng.bz/F38l.

Listing D.9 Subscribing to an HTTP request in Angular

```
this.httpClient.get('/customers/123')
  .subscribe(customer => {
              this.httpClient.get(customer.orderUrl)
              .subscribe(response => this.order = response)
  })
```

The `HttpClient.get()` method returns an `Observable`, and the better way to write the preceding code is by using the `flatMap` operator, which autosubscribes, unwraps the content of the first observable, and makes another HTTP request:

```
import {flatMap} from 'rxjs/operators';
...
httpClient.get('/customers/123')
    .pipe(
        flatMap(customer => this.httpClient.get(customer.orderUrl))
    )
    .subscribe(response => this.order = response);
```

Because a `flatMap` is a special case of `map`, you can specify a transforming function while flattening observables into a common stream. In the preceding example, we transform the value `customer` into a function call `HttpClient.get()`.

Let's consider one more example of using `flatMap`. This one is a modified version of the subject-trader example used earlier. This example is written in TypeScript, and it uses two `Subject` instances:

- `traders$`—This `Subject` keeps track of traders.
- `orders$`—This `Subject` is declared inside the `Trader` class and keeps track of each order placed by a particular trader.

You're the manager who wants to monitor all orders placed by all traders. Without `flatMap`, you'd need to subscribe to `traders$` (the outer observable) and create a nested subscription for `orders$` (the inner observable) that each subject has. Using `flatMap` allows you to write just one `subscribe()` call, which will be receiving the inner observables from each trader in one stream, as shown in the following listing.

Listing D.10 Two subjects and `flatMap`

```
enum Action{
     Buy = 'BUY',
    Sell = 'SELL'
}
```

Uses TypeScript enums to define action types

```
class Order{
    constructor(public orderId: number, public traderId: number,
         public stock: string, public shares: number, public action: Action)
     {}
}
```

```
let traders$ = new Subject<Trader>();

class Trader {

    orders$ = new Subject<Order>();

    constructor(private traderId: number,
    ➥public traderName: string) {}
}

let tradersSubscriber = traders$.subscribe
➥(trader => console.log(`Trader ${trader.traderName} arrived`));

let ordersSubscriber = traders$
   .pipe(flatMap(trader => trader.orders$))
   .subscribe(ord =>
        console.log(`Got order from trader ${ord.traderId}
       ➥to ${ord.action} ${ord.shares} shares of ${ord.stock}`));

let firstTrader = new Trader(1, 'Joe');
let secondTrader = new Trader(2, 'Mary');

traders$.next(firstTrader);
traders$.next(secondTrader);

let order1 = new Order(1,1,'IBM',100,Action.Buy);
let order2 = new Order(2,1,'AAPL',200,Action.Sell);
let order3 = new Order(3,2,'MSFT',500,Action.Buy);

// Traders place orders
firstTrader.orders$.next(order1);
firstTrader.orders$.next(order2);
secondTrader.orders$.next(order3);
```

Declares the Subject for traders

Each trader has its own Subject for orders.

Starts with the outer observable traders$

Extracts the inner observable from each Trader instance

The function subscribe() receives a stream of orders.

NOTE The enum containing string constants defines the action types. You can read about TypeScript enums at http://mng.bz/sTmp.

In this version of the program, the `Trader` class doesn't have a `placeOrder()` method. We just have the trader's `orders$` observable push the order to its observer by using the `next()` method . Remember, a `Subject` has both an observable and an observer.

The output of this program is shown next:

```
Trader Joe arrived
Trader Mary arrived
Got order from trader 1 to BUY 100 shares of IBM
Got order from trader 1 to SELL 200 shares of AAPL
Got order from trader 2 to BUY 500 shares of MSFT
```

In our example, the subscriber prints the orders on the console, but in a real-world app, it could invoke another function that would place an order with the stock exchange for execution.[8]

[8] See it in CodePen: http://mng.bz/4qC3.

D.8 The switchMap operator

Whereas `flatMap` unwraps and merges *all the data* from the outer observable values, the `switchMap` operator handles the data from the outer observable but cancels the inner subscription being processed if the outer observable emits a new value. The `switchMap` operator is easier to explain with the help of its marble diagram, shown in figure D.8.

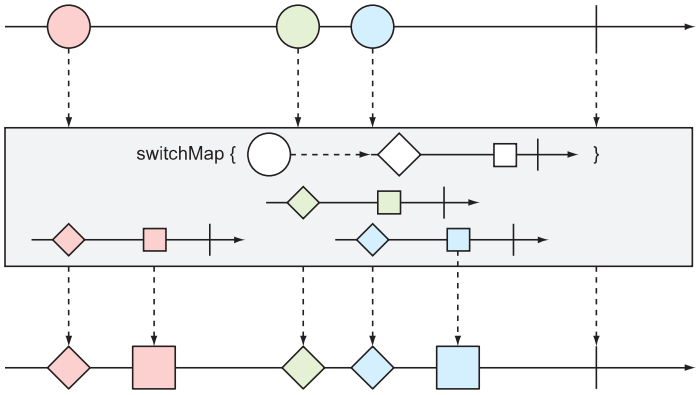


Figure D.8 The `switchMap` operator

For those reading the printed edition of this book, we need to say that the circles in the outer observable are red, green, and blue (from left to right). The outer observable emits the red circle, and `switchMap` emits the items from the inner observable (red diamond and square) into the output stream. The red circle was processed without any interruptions because the green circle was emitted after the inner observable finished processing.

The situation is different with the green circle. `switchMap` managed to unwrap and emit the green diamond, but the blue circle arrived *before* the green square was processed. The subscription to the green inner observable was cancelled, and the green square was never emitted into the output stream. The `switchMap` operator *switched* from processing the green inner observable to the blue one.

Listing D.11 has two observables. The outer observable uses the `interval()` function and emits a sequential number every second. With the help of the `take` operator, we limit the emission to two values: `0` and `1`. Each of these values is given to the `switchMap` operator, and the inner observable emits three numbers with an interval of 400 ms.

Listing D.11 Two observables and `switchMap`

```
let outer$ = interval(1000)
                .pipe(take(2));
```

Outer observable

This take operator will take only the first two items from the stream.

```
let combined$ = outer$
    .pipe(switchMap((x) => {
            return interval(400)      <-- Inner observable
               .pipe(
                 take(3),
                 map(y => `outer ${x}: inner ${y}`)
               )
          })
      );

combined$.subscribe(result => console.log(`${result}`));
```

The output of listing D.10 is shown next:

```
outer 0: inner 0
outer 0: inner 1
outer 1: inner 0
outer 1: inner 1
outer 1: inner 2
```

Note that the first inner observable didn't emit its third value, `2`. Here's the timeline:

1. The outer observable emits `0` and the inner emits `0` 400 ms later.
2. 800 ms later, the inner observable emits `1`.
3. In 1000 ms, the outer observable emits `1`, and the inner observable is unsubscribed.
4. The three inner emissions for the second outer value went uninterrupted because it didn't emit any new values.

If you replace `flatMap` with `switchMap`, the inner observable will emit three values for each outer value, as shown here:[9]

```
outer 0: inner 0
outer 0: inner 1
outer 0: inner 2
outer 1: inner 0
outer 1: inner 1
outer 1: inner 2
```

The chances are slim that you'll be writing outer and inner observables emitting integers. Chapter 6 explains a very practical use of the `switchMap` operator.

Just think of a user who types in an input field (the outer observable), and HTTP requests are being made (inner observable) on each `keyup` event. The circles in figure D.8 are the three characters that the user is typing. The inner observables are HTTP requests issued for each character. If the user entered the third character while the HTTP request for the second one is still pending, the inner observable gets cancelled and discarded.

[9] See it in CodePen: http://mng.bz/Y9IA.

TIP The `interval()` function is handy if you want to invoke another function periodically based on a specified time interval. For example, `interval(1000).subscribe(n => doSomething())` will result in calling the `doSomething()` function every second.

D.9 Error handling with catchError

The Reactive Manifesto (see www.reactivemanifesto.org) declares that a reactive app should be resilient, which means the app should implement a procedure to keep it alive in case of a failure. An observable can emit an error by invoking the `error()` function on the observer, but when the `error()` method is invoked, the stream completes.

RxJS offers several operators to intercept and handle an error before it reaches the code in the `error()` method on the observer:

- `catchError(error)`—Intercepts an error, and you can implement some business logic to handle it
- `retry(n)`—Retries an erroneous operation up to *n* times
- `retryWhen(fn)`—Retries an erroneous operation as per the provided function

Next, we'll show you an example of using the pipeable `catchError` operator. Inside the `catchError` operator, you can check the error status and react accordingly. Listing D.12 shows how to intercept an error and, if the error status is 500, switch to a different data producer to get the cached data. If the received error status isn't 500, this code will return an empty observable, and the stream of data will complete. In any case, the `error()` method on the observer won't be invoked.

Listing D.12 Intercepting errors with `catchError`

```
.pipe(
   catchError(err => {
    console.error("Got " + err.status + ": " + err.description);

    if (err.status === 500){
        console.error(">>> Retrieving cached data");

        return getCachedData();  // failover
    } else{
      return EMPTY;  // don't handle the error
    }
}))
```

Listing D.13 shows the complete example, where we subscribe to the stream of beers from a primary source—`getData()`—which randomly generates an error with the status 500. The `catchError` operator intercepts this error and switches to an alternate source: `getCachedData()`.

Listing D.13 Implementing failover with `catchError`

```
function getData(){
  const beers = [
      {name: "Sam Adams", country: "USA", price: 8.50},
      {name: "Bud Light", country: "USA", price: 6.50},
      {name: "Brooklyn Lager", country: "USA", price: 8.00},
      {name: "Sapporo", country: "Japan", price: 7.50}
  ];

  return Observable.create( observer => {
      let counter = 0;
      beers.forEach( beer => {
            observer.next(beer);
             counter++;

            if (counter > Math.random() * 5) {
                 observer.error({
                    status: 500,
                    description: "Beer stream error"
                  });
            }
          }
      );

      observer.complete();}
  );
}

// Subscribing to data from the primary source
getData()
  .pipe(
     catchError(err => {
       console.error(`Got ${err.status}: ${err.description}`);
      if (err.status === 500){
          console.error(">>> Retrieving cached data");
          return getCachedData();
       } else{
        return EMPTY;
       }
    }),
    map(beer => `${beer.name}, ${beer.country}`)
  )
  .subscribe(
      beer => console.log(`Subscriber got ${beer}`),
      err => console.error(err),
      () => console.log("The stream is over")
  );

function getCachedData(){
   const beers = [
      {name: "Leffe Blonde", country: "Belgium", price: 9.50},
      {name: "Miller Lite", country: "USA", price: 8.50},
      {name: "Corona", country: "Mexico", price: 8.00},
      {name: "Asahi", country: "Japan", price: 7.50}
  ];
```

Emits the next beer from the primary data source

Randomly generates the error with the status 500

Intercepts the error before it reaches the observer

Fails over to the alternative data source

Doesn't handle the non-500 errors; returns an empty observable to complete the stream

The alternate data source for failover

```
  return Observable.create( observer => {
      beers.forEach( beer => {
              observer.next(beer);
          }
      );

      observer.complete();}
  );
}
```

The output of this program can look as follows:[10]

```
Subscriber got Sam Adams, USA
Subscriber got Bud Light, USA
Got 500: Beer stream error
>>> Retrieving cached data
Subscriber got Leffe Blonde, Belgium
Subscriber got Miller Lite, USA
Subscriber got Corona, Mexico
Subscriber got Asahi, Japan
The stream is over
```

[10] See it in CodePen: http://mng.bz/QBye.

index

Numerics

A