**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:



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:



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.

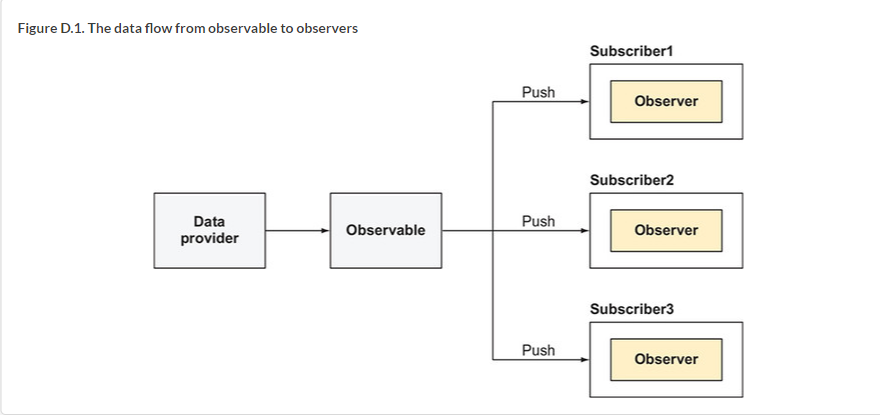
Wouldn’t it be nice if c2 were automatically recalculated upon any a1 or b1 changes? Think of a spreadsheet program like Microsoft Excel, where you could put a formula like =sum(a1, b1) into the C1 cell, and C1 would react immediately upon changes in A1 and B1. In other words, you don’t need to click 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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/15#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fig01).



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>.

##### 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:



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



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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/42#!/book/angular-development-with-typescript-second-edition/appendix-d/app04lev1sec5).

### 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.

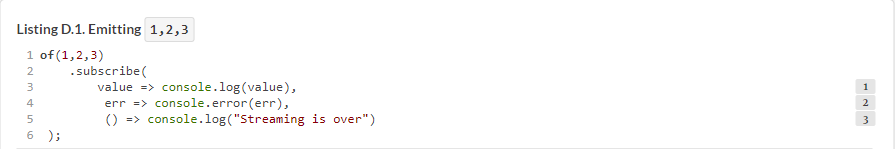
Here are some examples of the API to create an observable:

* **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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/49#!/book/angular-development-with-typescript-second-edition/chapter-6/ch06) has an example of using the fromEvent() API.
* **interval(1000)—**Emits a sequential integer (0,1,2,3...) every second

##### TIP

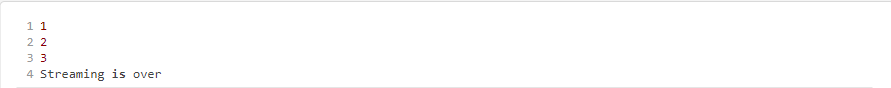
There’s a proposal for introducing Observable into future versions of ECMAScript. See <https://github.com/tc39/proposal-observable>.

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



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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/59#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fn1)]

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



##### NOTE

In [appendix A](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/62#!/book/angular-development-with-typescript-second-edition/appendix-a/app01), 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.

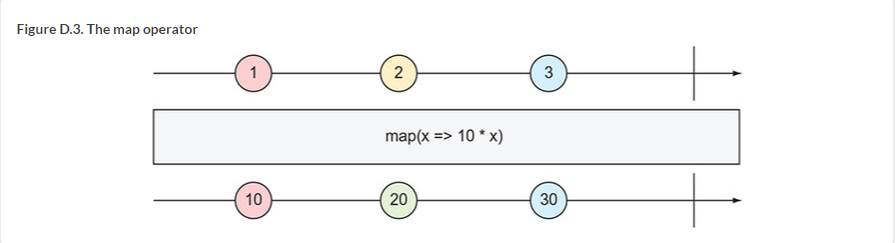
### D.4. Getting familiar with RxJS operators

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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/65#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fig02).



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.

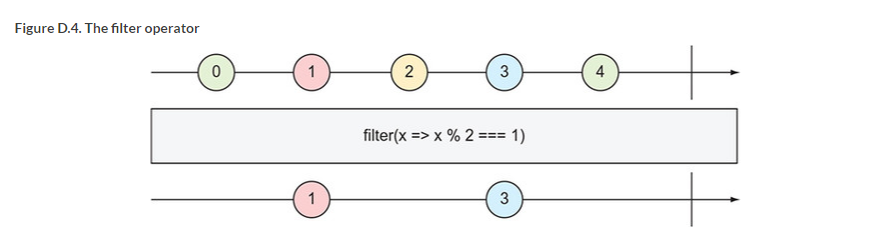
RxJS offers about 100 various operators, and their documentation may not always be easy to understand. On the positive side, the documentation often illustrates operators with marble diagrams. You can get familiar with the syntax of marble diagrams at <http://mng.bz/2534>. [Figure D.3](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/65#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fig03) shows how the RxJS manual illustrates the map operator with a marble diagram (see <http://mng.bz/65G7>).



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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/72#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fig03) shows the map operator that takes a value of each incoming element and multiplies it by 10.

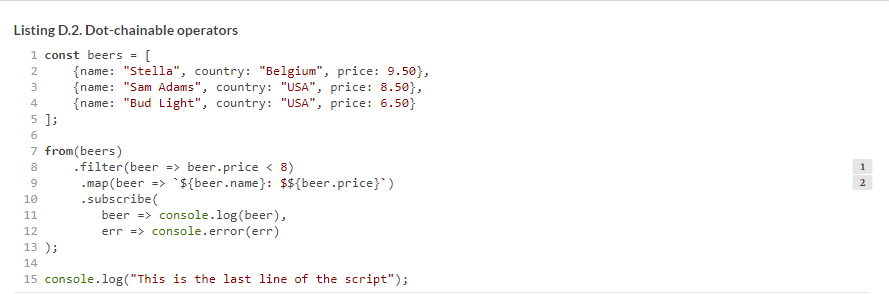
Now let’s get familiar with the marble diagram of the filter operator, shown in [figure D.4](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/72#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fig04). The filter operator takes a 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 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.

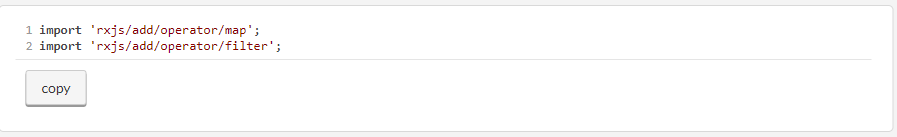


Starting with RxJS 6, the only way to chain operators is by using the pipe() method, passing to it comma-separated operators as arguments. The next section introduces *pipeable* operators.

#### **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:



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.

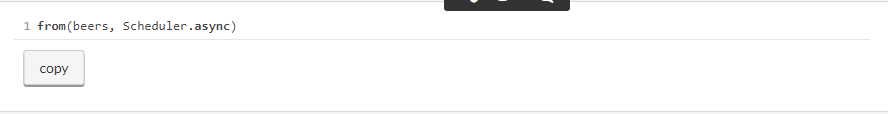
The subscriber in [listing D.2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/84#!/book/angular-development-with-typescript-second-edition/appendix-d/app04ex02) will receive the same data as the one in the sidebar “[Deprecated operator chaining](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/84#!/book/angular-development-with-typescript-second-edition/appendix-d/app04sb02),” 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.



Now if you remove the line filter from [listing D.2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/89#!/book/angular-development-with-typescript-second-edition/appendix-d/app04ex02), 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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/89#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fn2)]

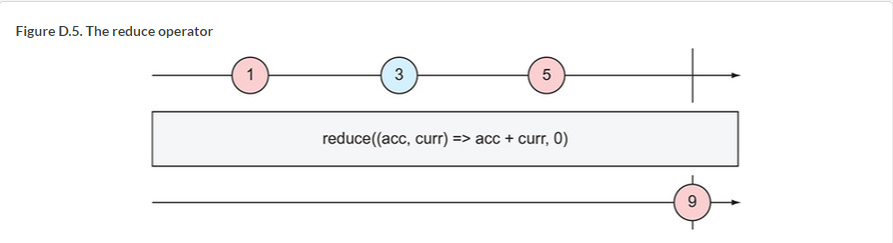
2See it in CodePen: <http://mng.bz/RqO5>.

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



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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/90#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fig05). This diagram shows an observable that emits 1, 3, and 5, and the reduce operator adds them up, producing the accumulated value of 9.

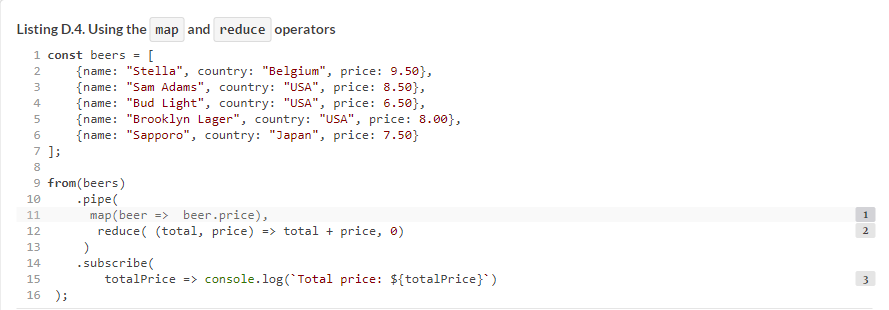


The reduce operator has two arguments: an accumulator function where we specify how to aggregate the values, and the initial (seed) value to be used by the accumulator function. [Figure D.5](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/96#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fig05) shows that 0 was used as an initial value, but if we changed it to 10, the accumulated result would be 19.

As you see in [figure D.5](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/96#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fig05), 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.



Running this script will produce the following output:



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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/104#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fn3)]

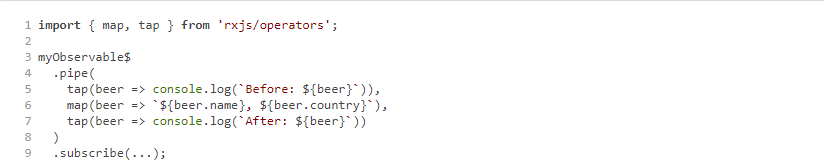
3See it in CodePen: <http://mng.bz/68fR>.

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.

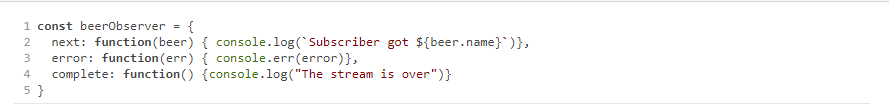
Say you have a chain of operators and want to see the observable values before and after a certain operator is applied. The tap operator will allow you to log the values:



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:



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:



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:



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

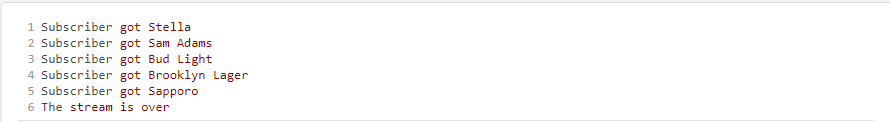


The next listing has a complete script that illustrates creation of the observer, the observable, and the subscription. The getObservableBeer() function creates and returns an observable that will loop through the array of beers and push each beer to the observer by invoking next(). After that, our observable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



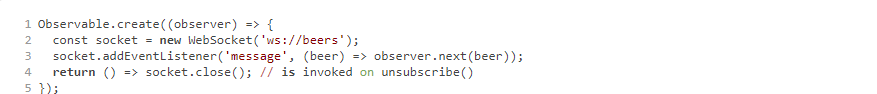
The output of this script is shown next:[[4](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/app04fn4)]

4See it in CodePen: <http://mng.bz/Q7sb>.



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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/127#!/book/angular-development-with-typescript-second-edition/appendix-d/app04lev1sec9).

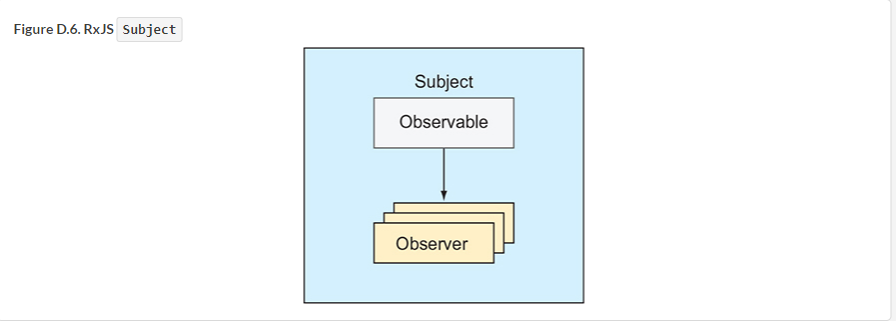
It’s important to note that our data producer (the array of beers) is created inside the getObservableBeer() observable, which makes it a cold observable. A WebSocket could be another example of the producer. Imagine we have a database of beers on the server, and we can request them over a WebSocket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



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**

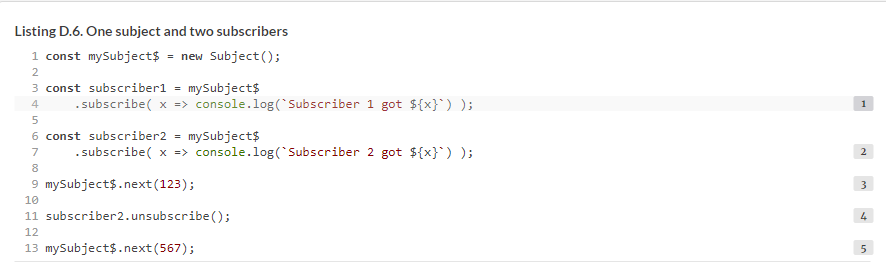
An RxJS Subject is an object that contains an observable and the observer(s). This means you can push the data to its observer(s) using next(), as well as subscribe to it. A Subject can have multiple observers, which makes it useful when you need to implement for *multicasting*—emitting a value to multiple subscribers, as shown in [figure D.6](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/129#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fig06).



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

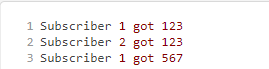


The following example has one Subject with two subscribers. The first value is emitted to both subscribers, and then one of them unsubscribes. The second value is emitted to one active subscriber.



Running this script produces the following output on the console:[[5](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/app04fn5)]

5See it in CodePen: <http://mng.bz/jx16>.



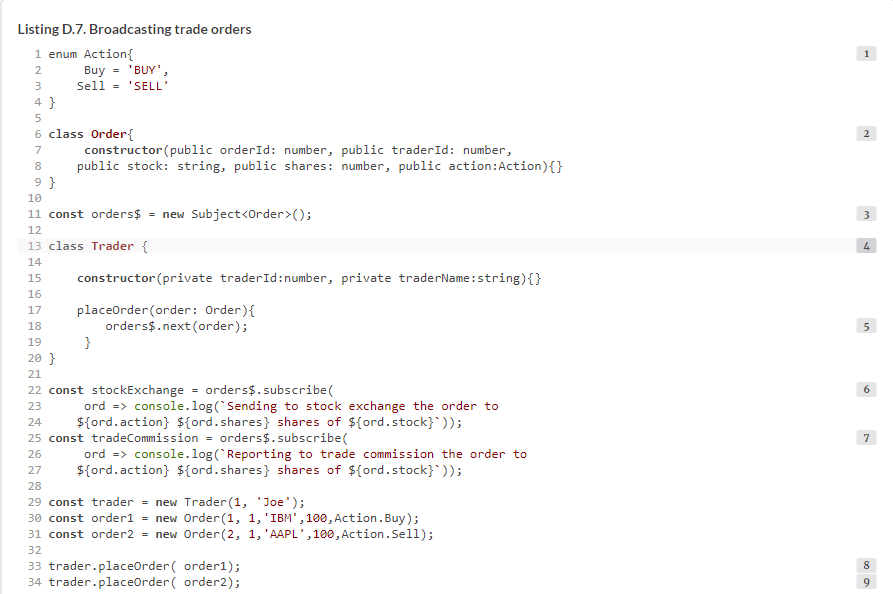
##### **TIP**

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):

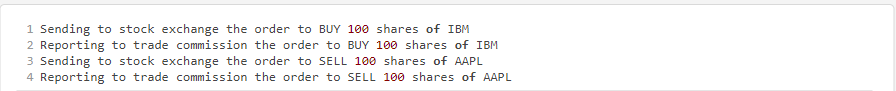
* The script that knows how to place orders with a stock exchange
* The script that knows how to report each order to a trade commission that keeps track of all trading activities

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.



Running [listing D.6](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/app04ex06) produces the following output:[[6](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/app04fn6)]

6See it in CodePen: <http://mng.bz/4PIH>.



##### **NOTE**

In [listing D.6](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/151#!/book/angular-development-with-typescript-second-edition/appendix-d/app04ex06), 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.

##### **TIP**

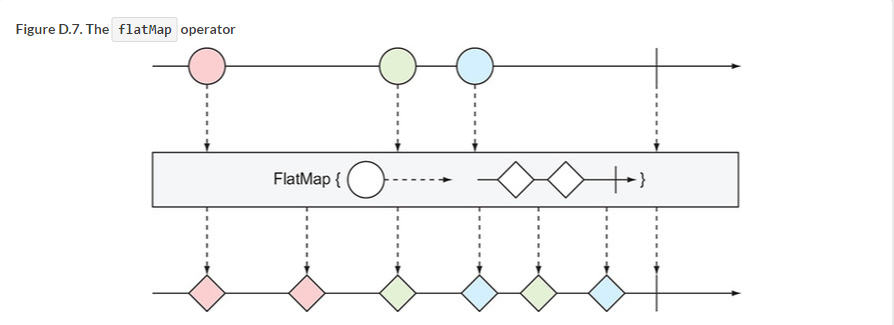
We wrote [listing D.6](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/151#!/book/angular-development-with-typescript-second-edition/appendix-d/app04ex06) in TypeScript, but if you want to see its JavaScript version, run npm install and the tsc commands 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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/151#!/book/angular-development-with-typescript-second-edition/chapter-6/ch06) 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**

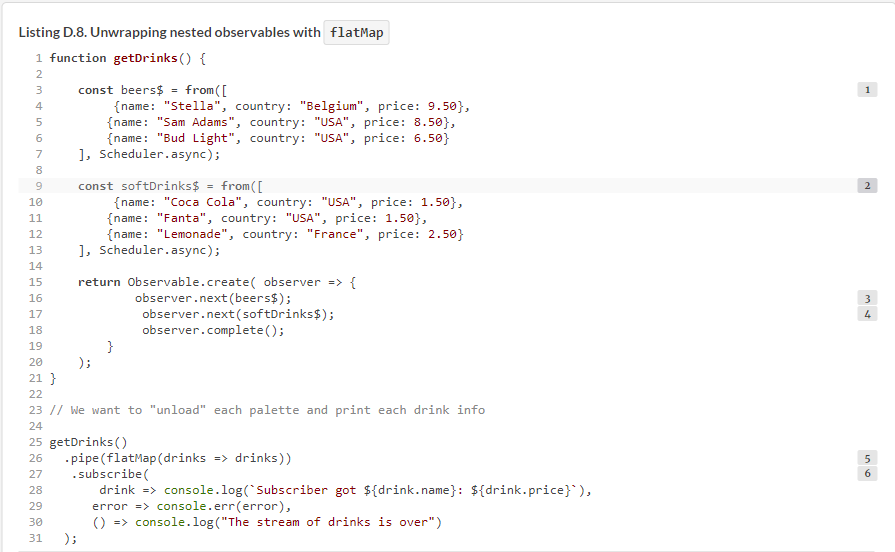
In some cases, you need to treat each item emitted by an observable as another observable. The outer observable emits the inner observables. Does that mean you need to write nested subscribe() calls (one for the outer observable and another for the inner one)? No, you don’t. The flatMap operator autosubscribes 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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/155#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fig07).



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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/160#!/book/angular-development-with-typescript-second-edition/appendix-d/app04ex08) 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.

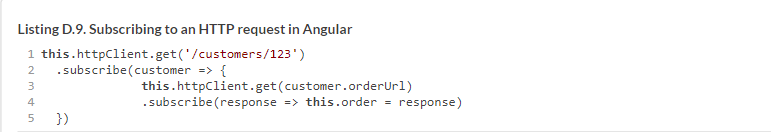


This script will produce output that may look as follows:[[7](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/app04fn7)]

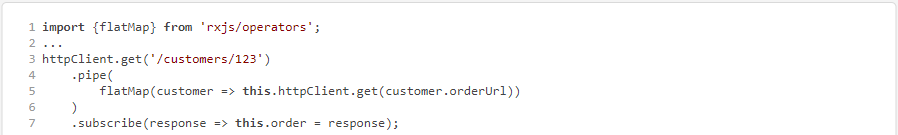
7See it in CodePen: <http://mng.bz/F38l>.



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.



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:

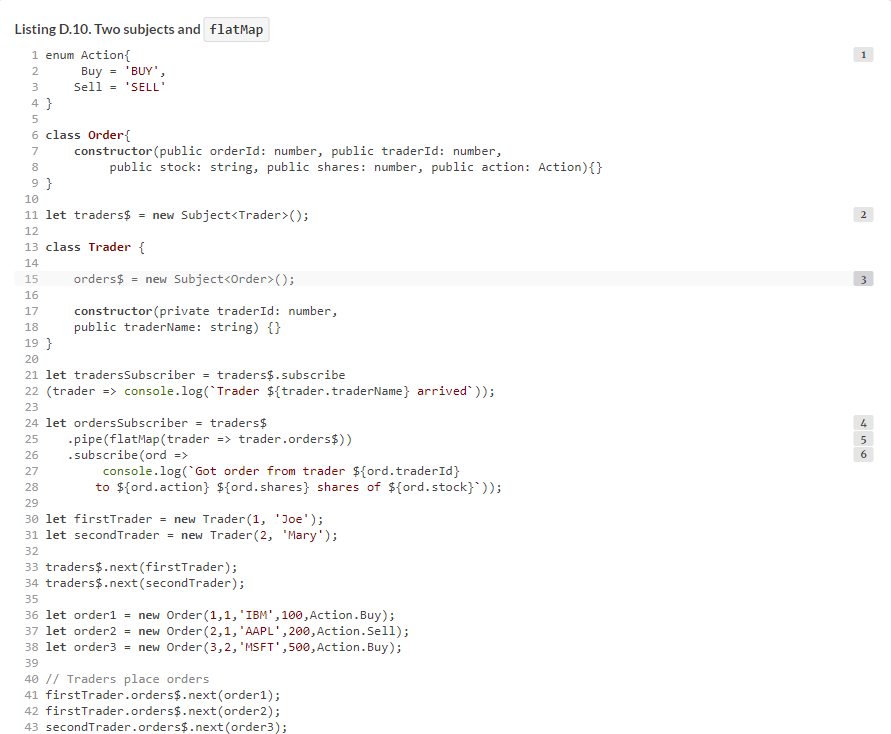


Because a flatMap is a special case of map, you can specify a transforming function while flattening observables into a common stream. In the preceding example, we transform the value customer into a function call HttpClient.get().

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.

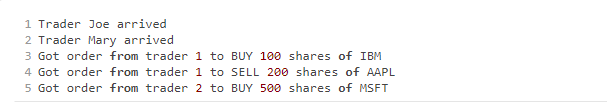


##### NOTE

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:

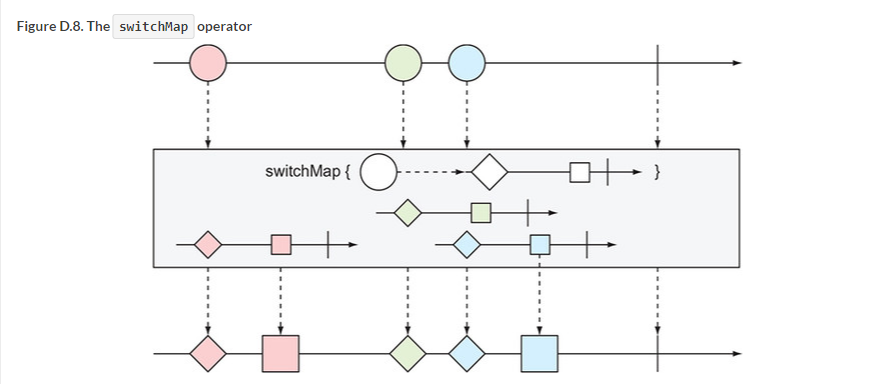


In our example, the subscriber prints the orders on the console, but in a real-world app, it could invoke another function that would place an order with the stock exchange for execution.[[8](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/179#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fn8)]

8See it in CodePen: <http://mng.bz/4qC3>.

### **D.8. The switchMap operator**

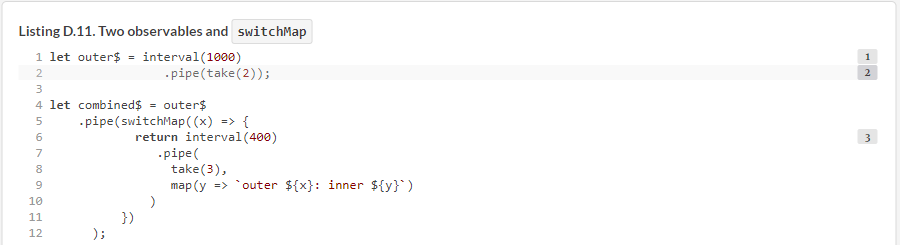
Whereas flatMap unwraps and merges *all the data* from the outer observable values, the switchMap operator handles the data from the outer observable but cancels the inner subscription being processed if the outer observable emits a new value. The switchMap operator is easier to explain with the help of its marble diagram, shown in [figure D.8](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/179#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fig08).



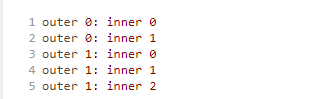
For those reading the printed edition of this book, we need to say that the circles in the outer observable are red, green, and blue (from left to right). The outer observable emits the red circle, and switchMap emits the items from the inner observable (red diamond and square) into the output stream. 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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/186#!/book/angular-development-with-typescript-second-edition/appendix-d/app04ex11) 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.



The output of [listing D.10](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/app04ex10) is shown next:



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.

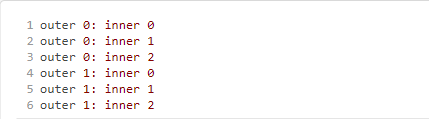
**2**.  800 ms later, the inner observable emits 1.

**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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/193#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fn9)]

9See it in CodePen: <http://mng.bz/Y9IA>.



The chances are slim that you’ll be writing outer and inner observables emitting integers. [Chapter 6](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/199#!/book/angular-development-with-typescript-second-edition/chapter-6/ch06) 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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/199#!/book/angular-development-with-typescript-second-edition/appendix-d/app04fig08) 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.

##### 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.

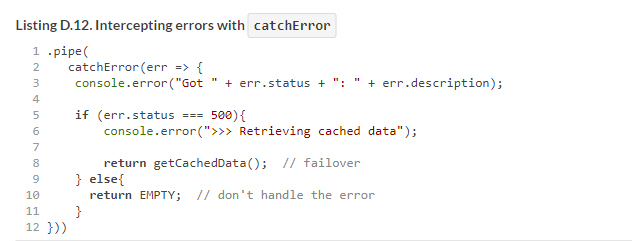
### **D.9. Error handling with catchError**

The Reactive Manifesto (see [www.reactivemanifesto.org](http://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.

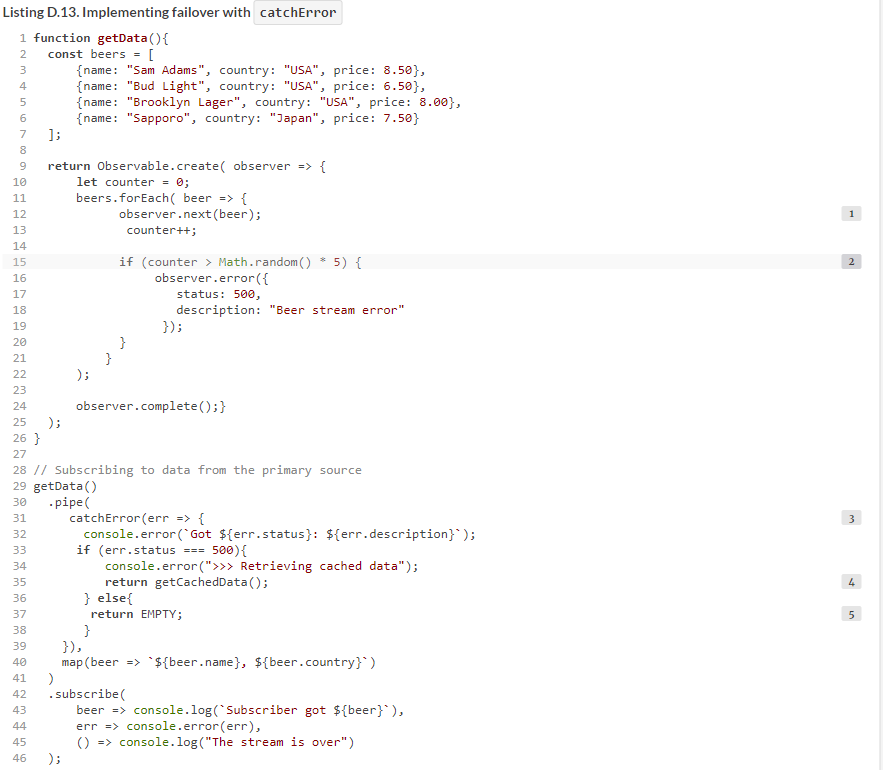
RxJS offers several operators to intercept and handle an error before it reaches the code in the error() method on the observer:

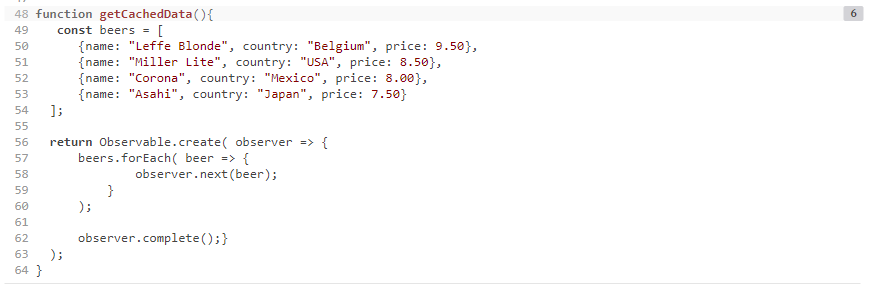
* **catchError(error)—**Intercepts an error, and you can implement some business logic to handle it
* **retry(n)—**Retries an erroneous operation up to *n* times
* **retryWhen(fn)—**Retries an erroneous operation as per the provided function

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](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/205#!/book/angular-development-with-typescript-second-edition/appendix-d/app04ex12) 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.13](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/1#!/book/angular-development-with-typescript-second-edition/appendix-d/app04ex13) 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().





The output of this program can look as follows:[[10](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/appendix-d/app04fn10)]

10See it in CodePen: <http://mng.bz/QBye>.

