Covers RxJS 5

RxJS IN ACTION

Paul P. Daniels
Luis Atencio

MEAP

MANNING

MEAP Edition
Manning Early Access Program
RxJS in Action
Covers RxJS 5
Version 7

For more information on this and other Manning titles go to
www.manning.com

welcome

You've been hearing the term Reactive bounced around a lot lately. Websites and platforms are now proudly sporting banners and proclaiming loudly, "We are Reactive!". It's easy to dismiss much of the buzz as another fad in the ever-shifting cauldron of software development. And, while some skepticism is certainly in order, our goal with this book is to break through the noise and give you the real story on RxJS. We want to show you how to use this tool to improve not just the performance of your next application, but to encourage a new way of thinking about software in a world where data no longer stands still.

Realize it or not, JavaScript is perhaps the best language out there for a paradigm shift. As the *de facto* language for the web, it harnesses a simplicity not easily matched, as well as strong functional roots, and an "async by nature" design that make it an ideal candidate for reactive programming. JavaScript is also a language that is constantly evolving, so rather than resist change, it actively embraces it and uses it to reach new places never before thought possible (read: Node.js).

RxJS takes these attributes along with principles of functional programming and builds on them by adding a coherent library to manage and extract data in the modern world of asynchronous programming. In what some call the "Golden Age of APIs", data is cheap and it can come from many different sources. What makes our lives difficult as developers is organizing and coordinating that data into meaningful and actionable results (How do we turn mouse clicks into web searches into online purchases?). Orchestrating this kind of data will take a new paradigm.

To really benefit from this book, you will need to be comfortable writing and building applications in JavaScript. Some familiarity with the role of scope, and how functions and closures work, will help you get the optimal experience from this book. As will at least some exposure to DOM manipulation, network communication (i.e. ajax requests) and event handling via callbacks. These are not dealbreakers, however, and the even less-experienced programmers will learn useful lessons from this book.

Reactive extensions for Java Script approaches problems differently from what you're used to, and at times you may find it tough going. Rest assured this feeling is natural; learning a new paradigm is always full of bumps and pitfalls. To soften the learning curve, we have divided this book into three sections. Part I deals with the fundamentals of RxJS and the basics of what it means to be reactive. Part II introduces topics you are likely to see should you build an application with RxJS. Finally, Part III will peek under the hood and tackle the advanced topics that make RxJS tick.

Both the topic, and this book, are sitting on the cutting edge, as we reference the latest version, RxJS 5, for our examples and on the website.

We are excited about the possibilities of RxJS, and excited also to share them with you. Our intention is to create the best learning resource available, and to make this journey as

smooth as possible for you. If you have questions or comments please post them in the Author Online Forum. Your input will help us create the most relevant and helpful book for people learning this exciting technology. Welcome aboard; we're glad to have you along for the ride!

—Paul Daniels and Luis Atencio

brief contents

1 Thinking Reactively

This chapter covers:

- Understanding the challenges of asynchronous JavaScript code and overcoming the limitations of callback and promise-based solutions
- Using streams to model both static, dynamic, and time-bound data
- Learning about observable streams to handle unbounded data pushed into your application in a functional manner
- Exploring the advantages of thinking reactively to simplify dealing with the composition of asynchronous data flows

Right now, somewhere in the world someone just created a tweet, a stock price just dropped and, most certainly, a mouse just moved. Tiny pinpricks of data that light up the internet and pass ubiquitously through semi-conductors scattered across the planet. A deluge of data propagates from any connected device. What's this got to do with you? As you push your code to production, this fire hose of events is pointed squarely at your JavaScript application, which needs to be prepared to handle it effectively. This creates two important challenges: scalability and latency.

As more and more data is received, the amount of memory that our application consumes or requires will grow linearly or, in worst cases, exponentially; this is the classic problem of *scalability* and trying to process it all at once will certainly cause the user interface to become unresponsive. You will see that buttons may no longer appear to work, fancy animations will lag, and the browser may even flag the page to terminate, which is an unacceptable notion for the modern web.

This problem is not new, though in recent years there has been exponential growth in the sheer scale of the number of events and data that JavaScript applications are required to process. This quantity of data is too big to be held readily available and stored in memory for

us to use. Instead, we must create ways to fetch it from remote locations asynchronously, resulting in another big challenge of interconnected software systems: *latency*, which can be difficult to express in code.

While modern system architectures have improved dramatically to include faster network devices and highly concurrent processing, the libraries and methods for dealing with the added complexity of remote data have not made the same strides. For example, when it comes to fetching data from a server or running any deferred computation, most of us still rely on the use of callbacks, a pattern that quickly breaks down when business rules evolve and change, or the problem you're trying to solve involves data that lives not in one, but potentially different remote locations.

The solution is not only in which library to use, but which paradigm suits these types of problems best. In this book, first you'll learn about the fundamental principles of two emerging paradigms functional programming (FP) and reactive programming (RP). This exhilarating composition is what gives rise to Functional Reactive Programming (FRP), encoded in a library called RxJS (or rx.js), which is the best prescription to deal with asynchronous and event-based data sources effectively. Our prescriptive roadmap has multiple parts. First we'll learn about the principles that lead to thinking reactively as well as look at the current solutions, their drawbacks, and how RxJS improves upon them. With this new found mindset, we'll dive into RxJS specifics and learn about the core operators that will allow you to express complex data flows of bounded or unbounded data in a succinct and elegant manner. You'll learn why RxJS is ideal for applications of any size that are event-driven in nature. So, along the way you'll find real-world examples that demonstrate using this library to combine multiple pieces of remote data, auto-completing input fields, drag and drop, processing user input, creating responsive UIs, parallel processing, and many others. These examples are intended to be narrow in scope as we work through the most important features of RxJS; finally, all of these new techniques will come together to end our journey with a full-blown web application using a hybrid React/Rx architecture.

The goal of this chapter is to give a broad view of the topics you will be learning about in this book. We will focus on looking at the limitations of the current solutions and point you to the chapters that show how RxJS addresses them. Furthermore, we'll learn how to shift our mindset to think in terms of *streams*, also known as *functional sequences of events*, which RxJS implements under the hood through the use familiar patterns such as Iterator and Observer. Finally, we'll explore the advantages of RxJS to write asynchronous code, minus the entanglement caused by using callbacks, that also scales to any amount of data. Now, understanding the differences between these two worlds is crucial, so let's begin there.

1.1 Synchronous versus asynchronous computing

In simple terms, the main factor that separates the runtime of synchronous and asynchronous code is latency, also known as "wait time." Coding explicitly for time is difficult to wrap our head around; it's much easier to reason about solutions when you're able to see the execution

occur synchronously in the same order as you're writing it: "do this, then immediately do that."

But the world of computing does not grant such luxuries. In this world of highly networked computing, the time it takes to send a message and receive a response represents critical time in which an application can be doing other things, such as responding to user inputs, crunching numbers or updating the UI. It's more like: "do this (wait for an indeterminate period of time), then do that." The traditional approach of having applications sit idle waiting for a database query to return, a network to respond, or a user action to complete is simply not acceptable; so you need to take advantage of asynchronous execution so that the application is always responsive. The main question here is whether it's acceptable to block the user on long running processes.

1.1.1 Issues with blocking code

Synchronous execution occurs when each block of code must wait for the previous block to complete before running. Without a doubt, this is by far the easiest way to implement code because you simply put the burden on your users to wait for their processes to complete. Many systems still work this way today such as ATM machines, point of sales systems, and other dumb terminals. Writing code this way is much easier to grasp, maintain, and debug; unfortunately, due to JavaScript's single-threaded nature, any long running tasks such as waiting for a AJAX call to return or a database operation to complete, shouldn't be done synchronously. It creates awful experience for your users because it causes the entire application to sit idle waiting for the data to be loaded and wasting precious computing cycles that could easily be executing any other code. This will block further progress on any other tasks that you might want to execute, which in turn leads to artificially long load times, as shown in figure 1.1:

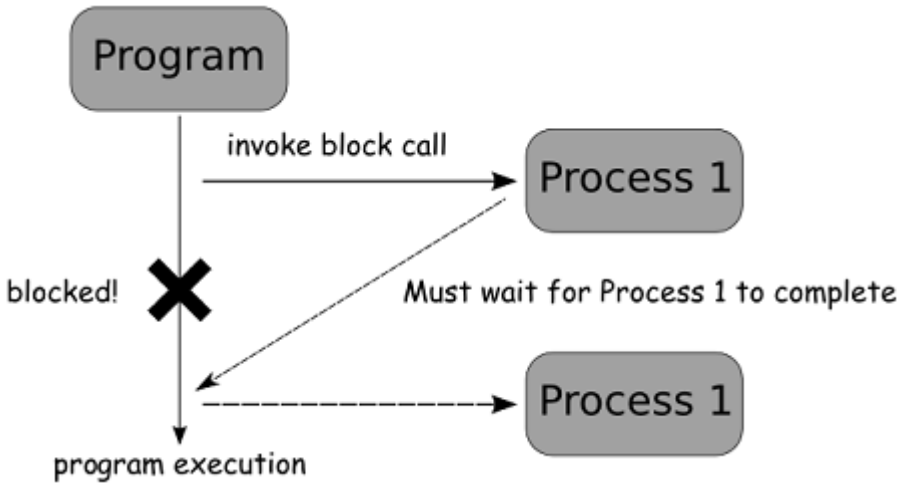

Figure 1.1 Shows a program that invokes two processes synchronously. A process in this case can be as simple a function call, an IO process, or a network transaction. When Process1 runs, it blocks anything else from running.

In this case, the program makes a blocking call to Process 1, which means it must wait for it to return control back to the caller, so that it can proceed with Process 2. This might work well for kiosks and dumb terminals, but browser user interfaces could never be implemented this way. Not only does it create a terrible user experience, but also browsers may deem your scripts unresponsive after a certain period of inactivity and terminate them. Here's an example of making an HTTP call that will cause your application to block waiting the server to respond:

```
let items = ajax('/data'); ❶
items.forEach(item => {
  // process each item
});
```

❶ Loading server side data synchronously halts program execution. The nature of the data is not important right now, it's some generic sample data pertaining to our application

A better approach would be to invoke the HTTP call and perform other actions while you're waiting on the response. Long running tasks are not the only problem; as we said earlier, mouse movement generates a rapid succession of very quick, fine-grained events. Waiting to process each of these synchronously will cause the entire application to become unresponsive whether it's long wait times or handling hundreds of smaller waits quickly. So, what can you do to handle these types of events in a non-blocking manner? Luckily, JavaScript gave us callback functions.

1.1.2 Non-blocking code with callback functions

Using functions as callbacks have been a staple of JavaScript development for years. They are used in everything from mouse clicks and key presses, to handling remote HTTP requests or file IO. JavaScript, being a single threaded language requires such a construct in order to maintain any level of usability. Callback functions were created to tackle the problem of blocking for long running operations to complete by allowing you to provide a handler function that the JavaScript runtime will invoke once the data it's ready for use. In the meantime, your application can continue carrying out any other task, as shown in figure 1.2:

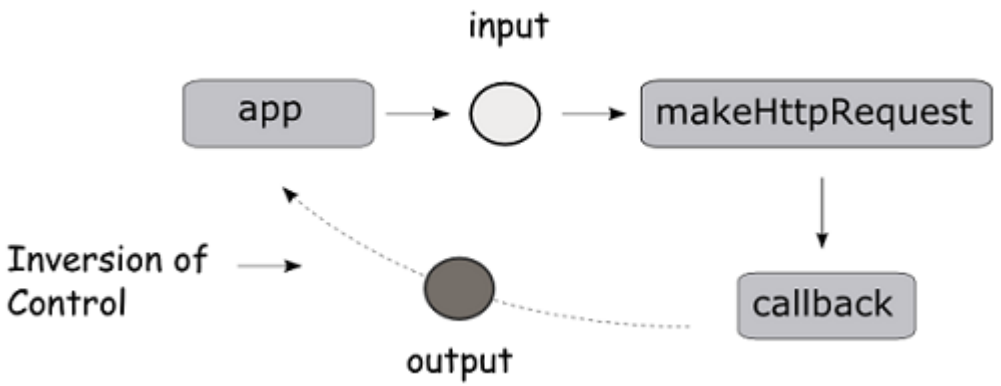

Figure 1.2 Callback functions in JavaScript create an inversion of control where functions call the application back, instead of the other way around

Unlike the previous code that makes a blocking HTTP call that you must wait for, using callbacks creates an *inversion of control* that permits your application to continue executing the next lines of code. Inversion of control in this sense refers to the way in which certain parts of your code receive the flow of control back from the runtime system. In the case, the runtime "calls you" (or returns control back to you) via the function handler when the data is ready to be processed; hence the term "callback." Let's take a look at this alternative:

```
ajax('/data',  ❶
  items  => {  ❷
     items.forEach(item => {
        // process each item ❸
     });
});
beginUiRendering(); ❹
```

❶ No explicit return value
❷ Declaration of callback function
❸ All processing carried out within callback body after the data has been fetched from the server
❹ This function begins immediately after ajax is called

Callback functions allow you to invoke code asynchronously, so that the application can return control back to you at a later time. This allows the program to continue with any other task in the meantime. In this code sample above, the HTTP functions runs in the background and immediately returns control back to the caller to begin rendering the UI; it only handles the contents of the items *after* it has completely loaded. This behavior is ideal as it frees up the application to make progress on other tasks such as loading the rest of a web-page, as in this case. As you'll see throughout this book, asynchronous code is a good design for IO-bound work like fetching data from the web or a database. The reason this works is that IO processes are typically much slower than any other type of instruction. So we allow them to run in the background as they're not dependent on processor cycles to complete.

Syntax Check

In the previous code sample, the second parameter of ajax() is the callback function. In that code, as in many parts of the book, we'll be using the ECMAScript 6 Lambda Expression syntax[1], which offers a terser and succinct way of invoking functions. Also called Arrow Functions, lambda expressions behave somewhat similar to an anonymous function call, which you're probably very familiar with. The subtle difference has to do with what the keyword this refers to. On some rare occasions, when the value of this is important, we'll call it out in the text and switch to using an anonymous function expression.

1.1.3 Understanding time and space

Certainly, asynchronous functions allow us to stay responsive, but they come at a price. Where synchronous programs allow us to reason directly about the state of the application, asynchronous code forces us to reason about its *future* state. What on earth does this mean? State can be understood simply as a snapshot of all the information stored into variables at any point in time. This information is created and manipulated via sequences of statements. Synchronous code can be thought of as an ordered, step-by-step execution of statements shown in figure 1.3:

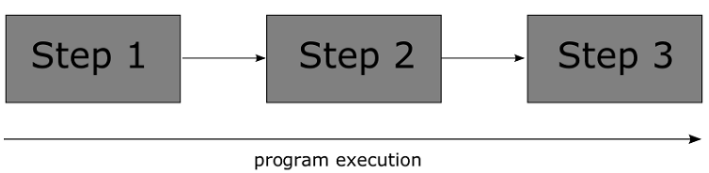

Figure 1.3 Synchronous is a step-by-step sequential execution of statements where each step depends on the previous one to run.

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

In this model, it's easy to determine at any point in time what the states of the variables are and what will occur next, which is why it's very easy to write and debug. But when tasks have different wait times or complete at different times, it's very difficult to guarantee how they'll behave together. Functions that terminate at unpredictable times are typically harder to deal with without the proper methods and practices. When this happens, the mental model of our application needs to shift to compensate for this additional dimension. Compare figure 1.3 to the model in figure 1.4, which not only grows vertically, but also horizontally.

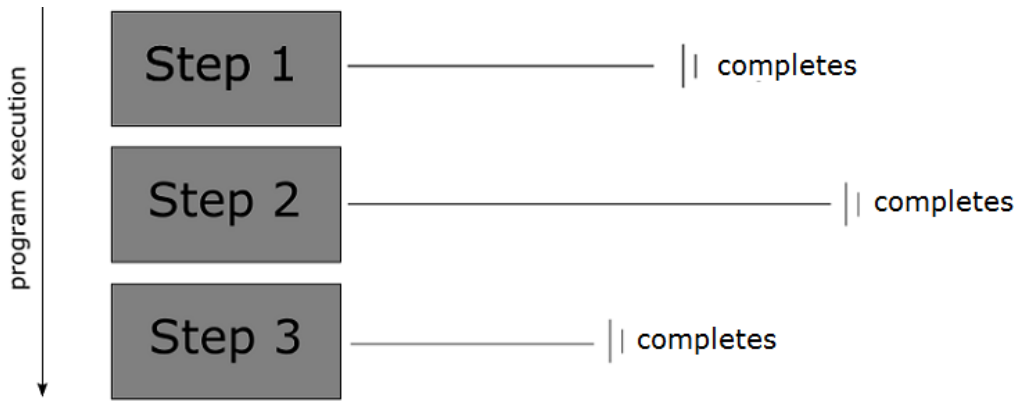

Figure 1.4 In asynchronous execution steps that are invoked in sequence need not terminate all at the same time. So, there's absolutely no guarantee that you can rely on the data in step 1 from step 2, for example.

As of now, if steps 1, 2, and 3 were independent tasks then executing them in any order wouldn't be a problem. However, if these were functions that shared any global state, then their behavior would be determined by the order in which they are called or by global state of the system. These conditions we refer to as *side effects*, which you'll learn more about in chapter 2, and involve situations where you need to read or modify an external resource like a database, the DOM, console, and others. Functions with side effects can perform very unreliably when run in any arbitrary order. In functional and reactive programming, we learn to minimize them by using *pure functions*, and you'll learn in this book this is extremely advantageous when dealing with asynchronous code.

So, assuming that our functions were side effect free, we still have another important issue—*time*. Steps 1, 2, and 3 might complete instantly, or might not depending on the nature of the work. The main question is: how can we guarantee these steps run in the correct order? As you've probably done many times before, the proper way to achieve is by *composing* these functions together, so that the output of one becomes the input to the next and, therefore, a chain of steps is created. The traditional approach that ensures the proper sequence of steps takes place is to nest a sequence of callbacks, and the model of your application's runtime resembles figure 1.5:

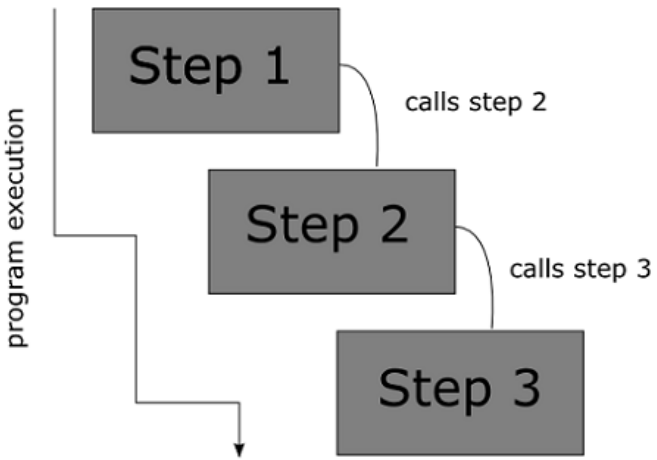

Figure 1.5 In order to guarantee the proper order of steps and asynchronous invocation takes place, we use callback functions to transfer control of the application once a long-running operation terminates.

Undoubtedly, this nested control flow is much harder to reason about than the synchronous, straight-line model of figure 1.4 done earlier. In figure 1.5, Step 1 runs first, which then calls Step 2 as soon as it completes, then Step 3 executes, and so on for the rest of the steps. This suggests the presence of a *temporal dependency* or time-coupling between these steps, which means that one can begin as soon as the previous finishes—it's a chain of commands. In this scenario, the callback functions are used to respond to the asynchronous request that happened before them, and begin processing its data. This happens typically when making sequential AJAX requests, but can also happen when mixing in any other event-based system whether it be key presses, mouse movements, database reads and writes, and others; all of these systems rely on callbacks.

1.1.4 Are callbacks out of the picture?

The short answer is: No. The need for a paradigm to tackle event-based or asynchronous code is not necessary when you're dealing with simple interactions with users or external services. If you're just writing a simple script that issues a single remote HTTP request, RxJS is a bit of overkill, and callbacks still remain the perfect solution. On the other hand, a library that mixes functional and reactive paradigms together really begins to shine when implementing state machines of moderate to advanced complexity such as dynamic UIs or service orchestration. Some examples of this can be, you need to orchestrate the execution of several business processes that consume several microservices, data mashups, or perhaps you're implementing features of a rich UI made up of several widgets on the page that interact with one another.

Consider the task of loading data from the client originating from different remote server-side endpoints. To coordinate amongst them, we would need several nested AJAX requests where each step wraps the processing of the data residing within each callback body and the logic of invoking the next step, as you saw previously in figure 1.6. Below is a possible solution for this, which requires the use of three composed callback functions to load data sets that potentially live in the same host or different hosts, together with its related meta information and files:

```
ajax('<host1>/items',    ❶
   items => {
    for (let item of items) {
    ajax(`<host2>/items/${item.getId()}/info`, ❷
         dataInfo => {
         ajax(`<host3>/files/${dataInfo.files}`, ❸
             processFiles);
       });
    }
});
beginUiRendering();
```

❶ **Load all items we want to display**
❷ **For reach item, load additional meta information**
❸ **For each meta record, load associated files**

Now while you might think this code looks trivial, if continuing this pattern, you'll begin to sink into horizontally nested calls—our model starts to grow horizontally. This trend is informally known in the JavaScript world as "callback hell," a design that you'll want to avoid at all costs if you want to create maintainable and easy to reason about programs. There's actually another hidden problem with this code. Can you guess what it is? It occurs when you mix a synchronous artifact like a `for..of` imperative block invoking asynchronous functions. Loops are not aware that there's latency in those calls, so they will always march ahead no matter what, which can cause some really unpredictable and hard to diagnose bugs. In these situations, you can improve things by creating closures around your asynchronous functions, which you can do by simply using `forEach()` instead of the loop.

```
ajax('<host1>/items',
   items => {
      items.forEach(item => {    ❶
     ajax(`<host2>/items/${item.getId()}/info`,
      dataInfo => {
         ajax(`<host3>/files/${dataInfo.files}`,
             processFiles);
       });
     });
});
```

❶ **The forEach() method of arrays will properly scope each item object into the nested HTTP call.**

This is why in RxJS, and functional programming in general for that matter, all loops are virtually eliminated! Instead, in chapters 4 and 5 you will learn about operators that allow you

to spawn sequences of asynchronous requests taking advantage of pure functions to keep all of the information properly scoped. Another good use of callbacks is to implement APIs based on Node.js event emitters. Let's jump into this next.

1.1.5 Event Emitters

Event emitters are very popular mechanism for asynchronous event-based architectures. The DOM, for instance, is probably one of the most widely known event emitter. On the server, like Node.js, there are certain kinds of objects that periodically produce events that cause functions to be called. In Node.js, the `EventEmitter` class is used to implement APIs for things like WebSocket IO or file reading/writing so that if you're iterating through directories and a file of interest is found, the object can emit an event referencing this file for you to execute any additional code.

Let's implement a simple object to show this API a bit. Consider a simple calculator object that can emit events like "add" and "subtract," which you can hook any custom logic into:

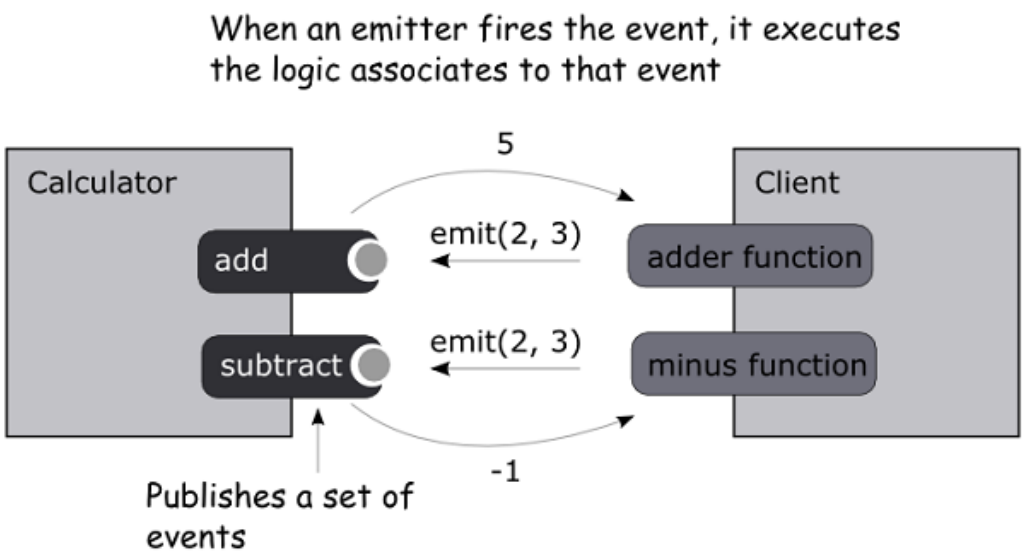

Figure 1.6 Node emitter object representing a simple calculator, which exposes two events: an add and a subtract

Here's some code for a calculator adder event:

```
const EventEmitter = require('events');      ❶

class Calculator extends EventEmitter {}     ❷

const calc = new Calculator();
```

```
calc.addListener('add', (a, b) => {        ❸
  console.log(a + b);
});
calc.addListener('subtract', (a, b) => {   ❸
  console.log(a - b);
});


calc.emit('add', 2, 3);      //-> Prints 5
calc.emit('subtract', 2, 3); //-> Prints -1
```

❶ **Load the events module**
❷ **Create a custom emitter**
❸ **Handle the 'add' event**

Subscribing to an event emitter is done through the `addListener()` method, which allows you to provide the callback that will be called when an event of interest is fired. Unfortunately, event emitters have all of the same the same problems associated with using callbacks to handle to emitted data coming from multiple composed resources. Overall, composing nested asynchronous flow is difficult.

The JavaScript community as a whole has made strides in the right direction to solve the types of issues described above. With the help of patterns emerging from functional programming, an alternative available to you with ES6 is to use Promises.

1.2 Better callbacks with promises

All hope is not lost; we promise you that. Promises are not part of the RxJS solution, but they work together perfectly well. JavaScript ES6 introduces promises to represent any asynchronous computation that is expected to complete in the future. With promises you can chain together a set of actions with future values to form a *continuation*. Let's understand what this means first and how writing your functions in a pure manner facilitates this technique.

One solution to the problem posed with callbacks that will allow you to have some degree function decoupling, readability, and perhaps targeted try/catch blocks, is to use a *continuation-passing style* (CPS). In its simplest form, this just means breaking apart your sequences of inlined callbacks into individual functions themselves. If these functions are side effect free, then it becomes easier to use them as continuations.

Going back to the example of making invoking several HTTP requests, continuation functions are passed in as arguments of each running `ajax()` in order to "continue" carrying or transforming the data forward to where it needs to be. We won't cover continuation-passing

in this book but you can read more about it here in this great post by Dr. Axel Rauschmayer[2]. Here's how you can rewrite the data loading example using CPS:

```
var fetchFiles = dataInfo => {
   ajax(`<host3>/data/files/${dataInfo.files}`,  processFiles);
}

var processItems = items => {
   items.forEach(item => {
        ajax(`<host2>/data/${item.getId()}/info`, fetchFiles);
   });
}

ajax('<host1>/items', processItems});
```

Arguably this code is now more modular than before, yet still not intuitive to follow. In essence, we want code to read something like:

```
Fetch all items, then                    (1)
   For-each item fetch all files, then   (1)
      Process each file
```

(1) The key term in this phrase is "then," which suggests that presence of time in these queries and that each action is chained to the next in the future

This is where promises come in. A `Promise` is a data type that wraps an asynchronous or long-running operation, a future value, with the ability for you to *subscribe* to its result, or its error, respectively. Promises are considered to be fulfilled when its underlying operation completes, at which point subscribers will receive the computed result. Because you can't alter the value of a promise once it's been executed, it's actually an immutable type, which is a functional quality we seek for in our programs. Different promise implementations exist based on the Promise/A+ protocol, and it's designed to provide some level of error handling and continuations via the `then()` methods. Here's how I can tackle the same example:

```
ajax('<host1>/items')
    .then(items => items.forEach(
         item => ajax(`<host2>/data/${item.getId()}/info`)
                 .then(dataInfo =>
                      ajax(`<host3>/data/files/${dataInfo.files}`))
            .then(processFiles);
     )
);
```

Now this looks very similar the statement we wrote before! Being a more recent addition to the language, and inspired in functional programming design, promises are more versatile and idiomatic than callbacks. And of course, to tie it all together, promises are more reliable when

[2] http://www.2ality.com/2012/06/continuation-passing-style.html

the function arguments passed to `then()` are pure. Now you're beginning to see why functional programming is so important for understanding RxJS.

The use of `then()` explicitly implies that there's time involved amongst these calls, which is a really good thing. If any step fails, you also have matching `catch()` blocks to handle errors and potentially continue the chain of command if necessary, as shown in figure 1.7:

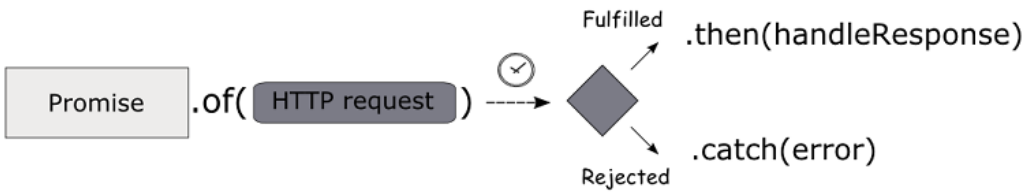

Figure 1.7 Promises create a flow of calls chained by then methods. If the promise is fulfilled, the chain functions continues; otherwise, the error is delegated to the promise catch block

Of course, promises also have shortcomings, or else we wouldn't be talking about Rx. The drawback of using promises, is that they're unable to handle data sources that produce more than one value, like mouse movements or sequences of bytes in a file stream. Promises are also missing important features like debouncing, throttling, and the ability to retry from failure—all present in RxJS. The most important downside, moreover, is that because promises are immutable, they can't be cancelled. So, for instance, if you use a promise to wrap the value of a remote HTTP call, there's no hook or mechanism for you to cancel that work. This is unfortunate because HTTP calls, based on the `XmlHttpRequest` object, can be aborted[3], but this feature is not honored through the promise interface. These limitations reduce their usefulness and force developers to write some of the cancellation logic themselves or seek other of libraries.

Collectively, promises and event-emitters are solving what are essentially the same problems in slightly different ways. Both approaches are used for different use cases (promises for single value returns like HTTP requests and event emitters for multiple value returns like mouse click handlers), but they do so mostly because of their own implementation constraints, not because the use cases are actually different. The result of this is that in many scenarios a developer must use both in order to accomplish their goal, which can often lead to disjointed and confusing code.

The problems of readability, hard to reason about code, and the downsides of current technology that we've discussed so far are not the only ones that we, as developers, need to worry about with asynchronous code. In this next section, we'll outline more concretely why we need to switch to a different paradigm altogether to tackle these issues head on.

[3] https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/abort

1.3 The need for a different paradigm

For many years now, we've learned to use many JavaScript async-libraries; everyone has their own preference whether it be JQuery, Async.js, Q.js, and others, yet they all fall short one way or another. We believe that it's not a matter of just using a library, but the right paradigm for the job. As a result, by combining functional and reactive programming paradigms, RxJS will help you address the following issues:

1. Familiar control flow structures (like `for` and `while` loops) with asynchronous functions don't work well together because they are not "async-aware;" that is, there're oblivious of wait-time or latency between iterations.
2. Error handling strategies become easily convoluted when you being nesting try/catch blocks within each callback. In chapter 7, we're going to approach error handling from a functional perspective. Also, if you wanted to implement some level of retry logic, at every step, this will incredibly difficult even with the help of other libraries.
3. Business logic is tightly coupled together within the nested callback structure you need to support. It's plain to see that the more nested your code is, the harder it is to reason about. Functions that are deeply nested become entangled with other variables and functions, which is problematic in terms of readability and complexity. It would be ideal to be able to create more reusable and modular components in order to have loosely coupled business logic that can be maintained and unit tested independently. We will cover unit testing with RxJS in Chapter 9.
4. Excessive use of closures. Functions in JavaScript create a closure around the scope in which they're declared. So nesting them means that you need to be concerned about not just the state of the variables passed in as arguments, but also the state of all external variables surrounding each function declaration, causing side effects to occur. In the next chapter, you'll learn how detrimental side effects can be and how functional programming addresses this. Side effects increase the cognitive load of the state of your application, making it virtually impossible to keep track of what's going on in your programs. Throw in a few loops and conditional if-else statements to the mix and you'll regret the day a bug occurs impacting this functionality.
5. It's difficult to detect when events or long running operations go rogue and need to be cancelled. Consider the case of a remote HTTP request that is taking too long to process. Is the script unresponsive or is the server just slow? It would be ideal to have an easy mechanism to cancel events cleanly after some predetermined amount of time. Implementing your own cancellation mechanism can be very challenging and error prone even with the help of other third-party libraries.
6. One good quality of responsive design is to always throttle a user's interaction with any UI components, so that the system is not unnecessarily overloaded. In chapter 4, you'll learn how to use *throttling* and *debouncing* to your advantage. Manual solutions for achieving this are typically very hard to get right and involve functions that access data outside of their local scope, which breaks the stability of your entire program.

7. It's very rare to think of memory management in JavaScript applications, especially client-side code. After all, the browser takes care of most of these low-level details. However, as UIs become larger and richer we can begin to see that lingering event listeners may cause memory leaks, and the size of the browser process to grow. It's true that this was a lot more prevalent in older browsers; nevertheless, the complexity of today's JavaScript applications is no match for the applications of years past.

This long list of problems can certainly overwhelm even the brightest developers. The truth of the matter is that the very paradigms that help us tackle these problems are hard to express in code, which is why a tool like RxJS is necessary to redefine our approach.

You learned that promises certainly move the needle the right direction (and RxJS integrates with promises seamlessly if you feel the need to do so), but what we really need is a solution that abstracts out the notion of latency away from our code while allowing us to model our solutions using a linear sequence of steps through which data can flow over time, as shown in figure 1.8:

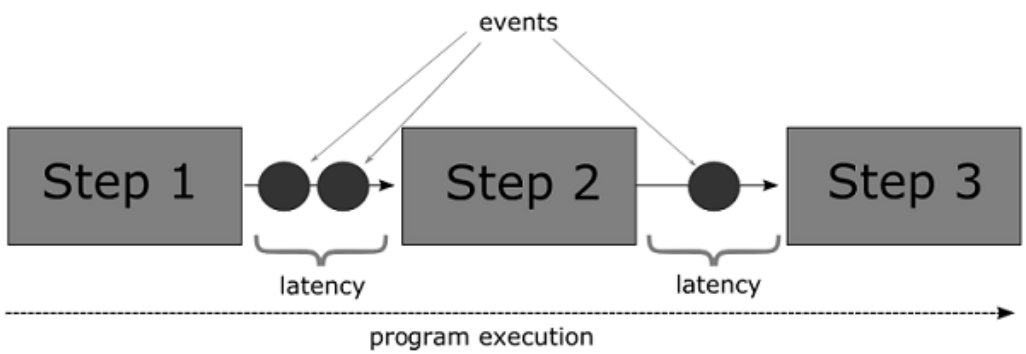

Figure 1.8 RxJS can treat asynchronous data flows with a programming model that resembles a simple chain of sequential steps.

In essence, we need to combine the ability to decouple functionality like event emitters, with the fluent design pattern of promises, all into a single abstraction. Moreover, we also need to work with both synchronous and asynchronous code, handle errors, discourage side effects, and scale out from one to a deluge of events. This is certainly a long laundry list of things to take care of.

As you think about this, ask yourself: how can we write code as a linear sequence of steps that only acts after some event has occurred in the future? How do we combine it with other code that might have its own set of constraints? Our desire for synchronicity is not just about convenience, it's what we're used to. Unfortunately, most of the common language constructs that we use in synchronous code are simply not well-suited for asynchronous execution. This lack of language support for things like "async-try/catch," "async-loops," and "async-conditionals" means that developers must often roll their own. It's not surprising that in the

past few years other people have asked the same questions and come together with the community at large to address these challenges, imploding into what's known as the *Reactive Extensions*—we have arrived.

1.4 The Reactive Extensions for JavaScript (RxJS)

RxJS is an elegant replacement for callback or promised-based libraries, using a single programming model that treats any ubiquitous source of events, whether it be reading a file, making an HTTP call, clicking a button, or moving the mouse, in the exact same manner. For example, instead of handling each mouse event independently with a callback, with RxJS you handle all of them combined.

As you'll learn in chapter 9, it's also inherently robust and easy to test with a vibrant and well-supported community to support it. The power of RxJS derives from being built on top of the pillars of functional and reactive programming, as well a few popular design patterns such as *Observer* and *Iterator* that have been used successfully for years. Certainly, RxJS did not invent these patterns, but it found ways to use them within the context of functional programming. We'll discuss FP and its role in RxJS further in the next chapter; at this time, in order to take full advantage of this framework, the key takeaway from this section is that you must learn to think in terms of *streams*.

1.4.1 Thinking in streams: data flows & propagation

Whether you deal with thousands of key presses, move movement events, touch gestures, remote HTTP calls, or single integers, RxJS treats all of these data sources in exactly the same way, which we'll refer to as *data streams* from now on.

> **STREAMS** Traditionally, the term "stream" was used in programming languages as an abstract object related to IO operations such as reading a file, socket, or requesting data from an HTTP server. For instance, Node.js implements readable, writable, and duplex streams for doing just this. In the reactive programming world, we'll expand the definition of a stream to mean absolutely *any* data source that can be consumed.

Reactive programming entails a mental shift in the way you reason about your program's behavior, especially if you come from an imperative background. We'll illustrate this shift in mindset with a simple exercise:

```
var a = 20;
var b = 22;
var c = a + b; //-> 42

a = 100;
c = ?
```

You can easily predict the value of `c` in this case: 42. The fact that I changed `a`, did not have any influence on the value of `c`. In other words, there's no *propagation of change*. This is the

most important concept to understand in reactive programming. Now, we'll show you a pseudo-JavaScript implementation of this:

```
A$ = [20];      ❶
B$ = [22];      ❷
C$ = A$.concat(B$).reduce(adder); //-> [42] ❸

A$.push(100);   ❹
C$ = ?
```

❶ Create a stream initialized with the value 20
❷ Create a stream initialized with the value 22
❸ Concatenate both streams, and apply an adder function, to get a new container with 42
❹ Push a new value into A$

First, we'll explain some of the notation we're using here. Streams are containers or wrappers of data very similar to an Array, so we used the array literal notation `[]` to symbolize this. Also, it's common to use the `$` suffix to qualify variables that point to streams. In the RxJS community, this is known as "Finnish notation," attributed to Andre Staltz, who is one the main contributors of RxJS and Finnish.

We created two streams `A$` and `B$` with one numerical value inside each. Since they are not primitive objects in JavaScript or have a plus "`+`" overloaded operator, we need symbolize addition by simply concatenating both streams and applying an *operator method* like `reduce` with an adder function (this should be somewhat familiar to you if you've worked with these Array methods). This represents `C$`.

Array extras

JavaScript ES5 introduced new Array methods, known as the "Array extras," which enable some level of native support for functional programming. These include map, reduce, filter, some, every, and others.

So what happens to `C$` if the value 100 is pushed onto `A$`? In an imperative program, nothing will actually happen except that `A$` would have an extra value. But in the world of streams, where there's change propagation, if `A$` receives a new value (a new event), this state is pushed through any streams that it is a part of. In this case `C$` gets the value 122. Confused yet? *Reactive programming is oriented around data flows and propagation*. In this case, you can think of `C$` as an "always-on" variable that *reacts* to any change and causes actions to ripple through it when any constituent parts changes. Now let's see how RxJS implements this concept.

1.4.2 Introducing the RxJS project

RxJS, which stands for *Reactive Extensions for JavaScript*, is the result of many efforts to manage the myriad of problems that manifest in asynchronous programming outlined earlier. It's an open-source framework ported by Matthew Podwysocki from Rx.Net (Reactive Extensions for .Net), itself open-source and created by Microsoft. RxJS has now evolved as a

community-driven project owned by Ben Lesh from Netflix, sanctioned by Microsoft as RxJS 5. This latest version is a complete overhaul of the previous version with a brand new architecture, a laser-focus on performance, and drastic simplification of the API surface. It offers several distinct advantages over other JavaScript solutions, as it provides idiomatic abstractions to treat asynchronous data similar to how you would treat any source of synchronous data, like a simple array. You can obtain installation details in Appendix A.

If you were to visit the main web site for the Reactive Extensions project (http://reactivex.io/), you'll find it defined as:

> "An API for asynchronous programming with observable streams"

By the end of this chapter, you'll be able to parse out exactly what this means. We will demystify these concepts and put you on the right path to tackle the problems presented in this book.

Let's see what thinking in streams looks like more concretely in RxJS. In figure 1.9, we show a simple break down of a stream (or pipeline) approach to handling data. A pipeline is a series of logic blocks that will be executed, in order, when data becomes available[4]. On the left side of figure 1.9 we have data sources, which produce various forms of data to be consumed by an application. And on the right are our data consumers, the entities that subscribe to (or listen for) these events and will do something with data we they receive, such as present it on a chart or save it to a file. In the middle is our data pipeline. During this middle step, data that is coming from any of the data sources that are being "observed" is filtered and processed in different ways so that it can be more easily consumed by the consumers.

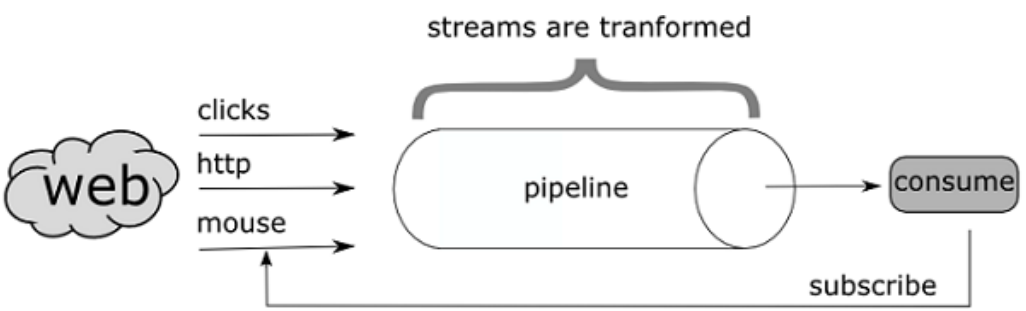

Figure 1.9 A generic data processing pipeline deals with a constant stream of asynchronous data, moving it from a producer (ex. a user clicking on the mouse) to a consumer (code that reacts the click). The pipeline will process data before it is passed the consumer for consumption.

[4] You can relate this to the popular *Pipes and Filter* design pattern.

You can subscribe to streams and implement functions within the pipeline that will be called (therefore react) when an event occurs (it's this pipeline component where the principles of functional programming will come into play, as you'll learn about in chapter 2).

A stream is nothing more than *a sequence of events over time*.

A popular example that you can relate to would be an Excel spreadsheet. You can easily bind functions onto cells that subscribe to the values of other cells and respond in real time as soon as any of the bounded cells changes.

A stream is a very abstract concept that works exactly like this, so we'll slowly wind up to it and break it down starting with some popular constructs you're familiar with.

1.4.3 Everything is a stream

The concept of a stream can be applied to any data point that holds a value; this ranges from just a single integer to bytes of data received from a remote HTTP call. RxJS provides lightweight data types to subscribe and manage streams as a whole that can be passed around as first-class objects and combined with other streams. Learning how to manipulate and use streams will be one of the central topics of this book. At this point, we haven't talked about any RxJS specific objects, for now we'll assume that an abstract data type, a container called `Stream` exists. We can create one from a single value as such:

```
Stream(42);
```

At this point, this stream remains dormant and nothing has actually happened, until there's a subscriber (or observer) that listens for it. This is very different from promises shown earlier that execute their operations as soon as its created. Instead, streams are *lazy* data types, which means that they execute only after a subscriber is attached. In this case, the value 42 which was lifted into the stream context navigates or propagates out to at least one subscriber. After it receives the value, the stream is completed.

```
Stream(42).subscribe(
   val => {  (1)
      console.log(val); //-> prints 42
   }
);
```

(1) **Using a simple function that will be called with each event in the stream**

Subject-Observer pattern

Behind RxJS is a fine-tuned Subject-Observer design pattern. It involves an object (the Subject), which maintains a list of subscribers (each an Observer) that are notified of any state changes. This pattern has had many applications, especially as an integral part of the Model View Controller (MVC) architecture where the View layer is constantly listening for Model changes. However, the rudimentary observer pattern has its drawbacks due to memory leaks related

to improper disposal of observers. You can learn more about in the famous book Design Patterns: Elements of Reusable Object-Oriented Software, known casually as the "Gang of Four" book.

RxJS draws inspiration from this pattern for its Publish-Subscribe methodology targeted at asynchronous programs, but adds a few extra features out-of-the-box like: signals that indicate when a stream has completed, lazy initialization, cancellation, resource management, and disposal. Later on we'll talk about the components of an RxJS stream.

Furthermore, we can extend this example to a sequence of numbers:

```
Stream(1, 2, 3, 4, 5).subscribe (
   val => {
      console.log(val); //-> prints 1, 2, 3, 4, 5
   }
);
```

Or even arrays:

```
Stream([1, 2, 3, 4, 5])
  .filter(num => (num % 2) === 0) ❶
  .map(num => num * num)  ❶
  .subscribe(
      val => {
        console.log(val); //-> prints 4, 16
      }
);
```

❶ Streams also support the Array.map() and Array.filter() functions introduced in ES5 to process the contents within the array

In this sample above, the set of operations that occurs between the creation of the *Producer* of the stream (in this case the array), and the *Consumer* (the function that logs to the console) is what we'll refer to as the *Pipeline* (we'll expand on these concepts shortly). The pipeline is what we'll study thoroughly in this book and is what allows you to transform a given input into the desired output. In essence, it is where your business logic will be executed, as outlined in figure 1.10:

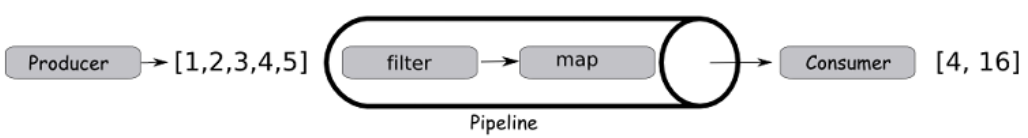

Figure 1.10 Shows a simple producer (an array of numbers) which emits events linearly. These events are submitted through the pipeline and transformed. The final data is then sent to all subscribers to be consumed.

Up until now, we've created streams from *static* data sources: numbers (or strings), sequences, and arrays. But the power of RxJS extends beyond that with the ability to treat *dynamic* data sources in exactly the same way, as if *time* didn't factor into the equation.

1.4.4 Abstracting the notion of time from your programs

Indeed, "time is of the essence." The hardest part of asynchronous code is having to deal with latency and wait time. You saw earlier how callbacks and promises can be used to cope with these concerns, each with their own limitations. RxJS brings this notion of "continuous sequences of events over time" as a first-class citizen of the language—finally a true event subsystem for JavaScript. In essence, this just means that RxJS *abstracts over time under the same programming model regardless of source*, so that you can transform your data as if your code was completely linear and synchronous. This is brilliant because you now can process a sequence of mouse events just as easily as processing an array of numbers.

Looking at figure 1.11, streams are analogous to a real world monthly magazine subscription. Our subscription to the magazine is actually a collection of magazines that are separated by time, i.e. there are twelve magazines annually, however, we only receive one every month. Upon receiving a magazine, we usually perform an action on it (read it or throw it away). There are additional cases that we can also consider such as the time between magazine deliveries being zero, whereby we would receive all the magazines at once, or there might be no magazines (and someone would be getting an angry email). In all the above cases, because we only perform the action upon receiving the magazine we can think of this process as reactive (because we are *reacting* to receiving a magazine). A non-reactive version of this would be going to a newspaper stall at the airport. Here you can also find magazines, but now you won't receive additional magazines, only the ones that you buy at the stall. In practice this would mean that you are only receiving updates when you happen to be near a magazine stand rather than every time a new magazine becomes available.

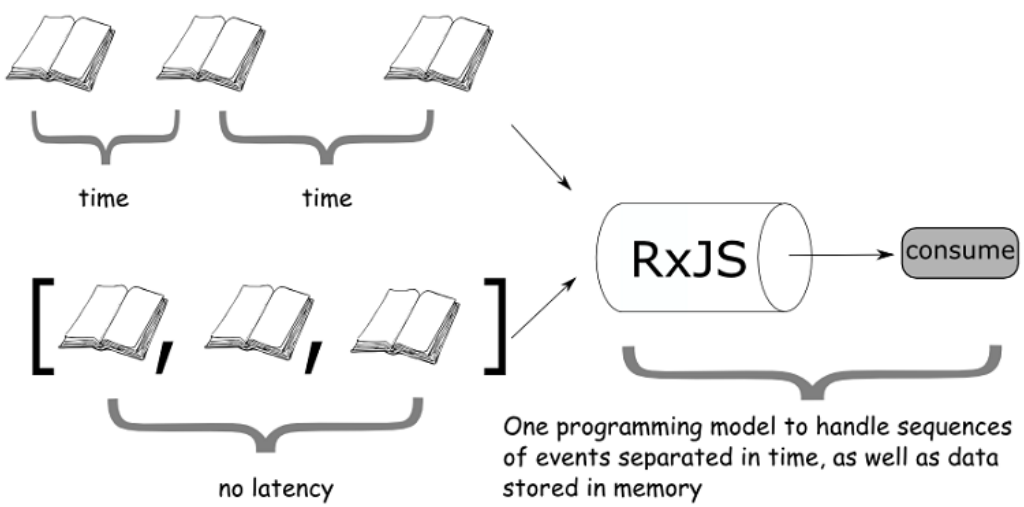

Figure 1.11 Not only does RxJS handle sequential, but using the same programming model, it can just as easily work with asynchronous events (bound by time). This mean that the same level of reasoning applied to linear programs can also be applied to non-linear programs with latency and wait times.

Rx allows you to take this magazine subscription metaphor and apply it to a wide range of use cases: loading files from disk or over a network, processing user input, handling real-time services like an RSS and Twitter feeds. Following the same examples before, with RxJS you can consume a stream of time-based asynchronous sequences of events, just like you did with normal synchronous data

```
Stream(loadMagazines('/subscriptions/magazines'))
  .filter(magazine => magazine.month === 'July')  ❶
  .subscribe(
      magazine => {
        console.log(magazine.title);
        //-> prints Dr. Dobbs "Composing Reactive Animations"
      }
);
```

❶ Using the well-known Array.filter() operator, this time with magazine subscriptions, to retrieve only the July edition

These type of services produce data in real-time at irregular intervals and the data produced forms the foundation of an event stream. In the case of a service like Twitter, we can think of the Twitter API as a producer of tweets, of which some will be interesting and some not so much. In general, in most cases we are interested in creating logic that processes the content of the tweet rather than diving into the intricacies of network communication. As we mentioned earlier, this logic is made up of several components, which we'll look at in more detail.

1.4.5 Components of an Rx Stream

The RxJS stream is made up of several basic components, each with specific tasks and lifetimes with respect to the overall stream. You've seen some examples of these earlier, and now we'll introduce them more formally. They are:

- Producers
- Consumers
- Data pipeline
- Time

PRODUCERS

Producers are the source of our data. A stream must always have a producer of data, which will be the starting point for any logic that we will perform in RxJS. In practice, a producer is created from something that generates events independently (anything from a single value, an array, mouse clicks, to a stream of bytes read from a file). The Observer pattern defines producers as the *Subject*; in RxJS, we call them *Observables*, as in something that is *able to be observed*.

Observables are in charge of pushing notification, so we refer to this behavior as a fire-and-forget, which means that we will never expect the producer to be involved in the *processing* of events, only the emission of them.

TC-39 OBSERVABLE SPEC The use observables have proven to be so successful from the previous version the library (RxJS 4), that it's proposed to be included in the next major release of JavaScript[5]. Fortunately, RxJS 5 follows this proposal closely to remain completely compatible.

CONSUMERS

To balance the producer half of the equation, we must also have a consumer to accept events from the producer and process them in some specific way. When the consumer begins listening to the producer for events to consume we now have a stream and it is at this point that the stream begins to push events; we'll refer to a consumer as an *Observer*.

Streams only travel from the producer to the consumer, not the other way around. In other words, a user typing on the keyboard produces events that flow down to be consumed by some other process. This means that part of understanding how to think in streams will mean understanding how to think about parts of an application as "upstream" or "downstream" to determine the direction that the data will flow. With respect to RxJS a stream will always flow from an upstream observable to a downstream observer, and both components are loosely coupled, which increases the modularity of your application as shown in figure 1.12.

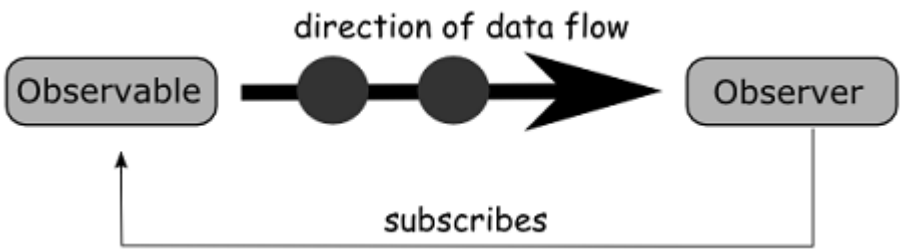

Figure 1.12 Direction of the events always move from observables into observers, and never the other way around.

For instance, a keyboard event handler would be "upstream" because it would only produce events, but not consume them, while code that should perform logic based on key presses would be "downstream". At a fundamental level, a stream will only ever require the producer and the consumer. Once the latter is able to begin receiving events from the former, we have effectively created a stream. Now what can we do with this data? All of that happens within the data pipeline.

[5] https://github.com/tc39/proposal-observable

DATA PIPELINE

One advantage of RxJS is that we can manipulate or edit the data as it passes from the producer to the consumer. This is where the list of methods (known as observable operators) will come into play. Manipulating data en route means that we can adapt the output of the producer to match the expectations of the consumer. Doing so promotes a *separation of concerns*[6] between the two entities and it's a big win for the modularity of your code. This design principle is typically extremely hard to accomplish in large-scale JavaScript applications, but RxJS facilitates this model of design.

TIME

And now the implicit factor behind all of this, time. For everything RxJS there is always an underlying concept of time, which we can use to manipulate streams. The time factor permeates through all the components that we have discussed so far. It is an important and abstract concept to grasp, so we'll look at it in detail in later chapters. For now, we need only understand that time need not always run at normal speed, and we can build streams that run slower or faster depending on our requirements. Luckily, this won't be an issue if you decide to use RxJS. Figure 1.13 provides a visualization of each of the parts of the RxJS:

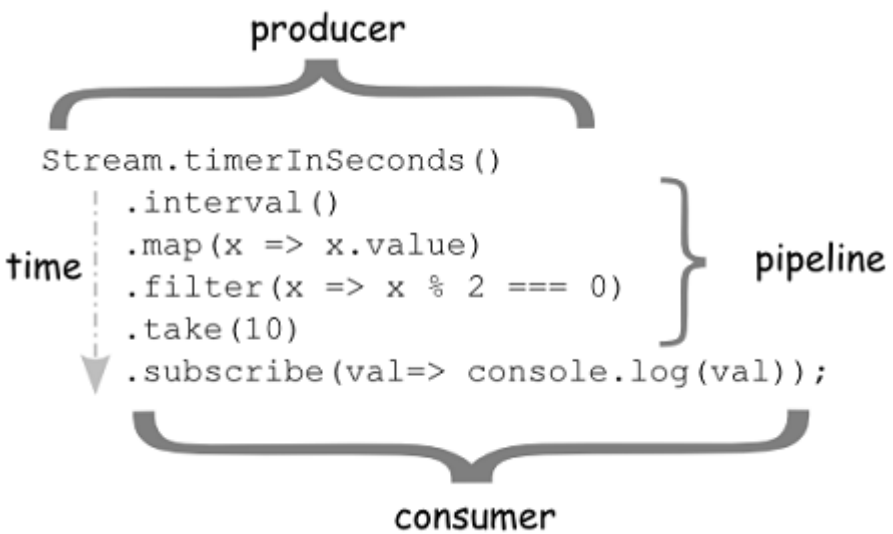

Figure 1.13 Sample code highlighting the different components of a stream.

[6] Separations of concerns in this case refers to the use of functions with single responsibility

If you pay close attention to the structure of a stream, you'll notice that this closely resembles the pattern used in promises. Because what started out as a nested callback "pyramid of doom:"

```
ajax('<host1>/items',
      items => {
         items.forEach(item => {     ❶
        ajax(`<host2>/items/${item.getId()}/info`,
         dataInfo => {
             ajax(`<host3>/files/${dataInfo.files}`,
                processFiles);
          });
        });
});
```

Was drastically improved using promises:

```
ajax('<host1>/items')
    .then(items => items.forEach(
         item => ajax(`<host2>/data/${item.getId()}/info`)
                 .then(dataInfo =>
                      ajax(`<host3>/data/files/${dataInfo.files}`))
            .then(processFiles);
     )
);
```

And now streams extend this behavior with powerful operators that break this down even further:

```
Stream(ajax('<host1>/items')
  .streamMap(item =>
        Stream(ajax(`<host2>/data/${item.getId()}/info`)))    ❶
  .streamMap(dataInfo =>
        Stream(ajax(`<host3>/data/files/${dataInfo.files}`))) ❶
  .subscribe(processFiles);
```

❶ **Streams can also compose other streams**

Remember that our `Stream` object here is merely abstract artifact designed to show you how the paradigm works. In this book, you'll learn to use the actual objects that implement these abstract concepts to design your applications using a functional and reactive model. However, RxJS doesn't oblige you to you use only a single paradigm, as it's often the combination of them that creates the most flexible and maintainable designs.

1.5 Reactive and other programming paradigms

Every new paradigm that you'll encounter during your programming career will require you to modify your thinking to accommodate the primitives of the language. For example, object-oriented programming (OOP) puts *state* within objects which are the central of unit of abstraction, and the intricacy of the paradigm comes from the interactions that arise when they interact with one another. In a similar fashion, functional programming (FP) places

behavior at the center of all things with functions as the main unit of work. Reactive programming RP, on the other hand, requires us to see data as a constantly *flowing stream of change* as opposed to monolithic data types, or collections holding all of an application's state.

Now you're probably wondering: Am I allowed to choose only one? Or can I combine them together into the same code base? The beauty behind all of this is that you can actually use all of them together. Many prominent figures in our industry have attested this. In other words, RxJS doesn't force upon you a certain style of development or design pattern to use—it is *un-opinionated*. Thankfully it also works orthogonally to most libraries. As you'll see later on, it's a simple matter in most cases to adapt an existing event stream such as a DOM event handler into an observable. In fact, the library provides many operators for such operations baked directly into it. It will even support more unusual design patterns such as those we will see when we use a library like React or Redux (which we will see in the last chapter).

In practice, you can use OOP to model your domain, and a powerful combination of reactive and functional programming (a combination known as functional reactive programming FRP) to drive your behavior and events. When it comes to managing events, there's an important theme that you'll soon begin to see in code involving Rx. Unlike in OOP where state or data is *held* in variables or collections, state in RP is *transient*, which means that data never remains stored but actually flows through the streams that are being subscribed to, which makes event handling easy to reason about and test.

Another noticeable difference is the style used in both paradigms. On one hand, OOP is typically written imperatively. In other words, you instantiate objects that keep track of state while running through a sequence of statements revealing how those objects interact and transform to arrive at your desired solution.

On the other hand, RxJS code encourages you to write declaratively, which means your code expresses the *what* and not the *how* of what you're trying to accomplish. RxJS follows a simple and declarative design inspired in functional programming. No longer will you be required to create variables to track the progress of your callbacks or worry about inadvertently corrupting some closed over outer state causing side effects to occur. Besides, with RxJS it becomes easy to manage multiple streams of data, filtering and transforming them at will. By creating operations that can be chained together we are also able to fluently create pipelines of logic that sound very much like spoken sentences like: "when I receive a magazine for the month of July, notify me."

In this chapter, you learned how RxJS elegantly combines both functional and reactive paradigms into a simple computing model that places observables (streams) at the forefront. Observables are pure and side effect free, with a powerful arsenal of operators and transformations that will allow you compose your business logic with asynchronous operations elegantly. We chose to keep the code abstract for now as we work through some of the new concepts. But quickly we'll ramp up to a comprehensive theoretical and practical understanding of the library, so that that you can begin to apply it immediately at work or on your personal projects. Now it's time to start really thinking in streams, and that's the topic of the next chapter.

1.6 Summary

- Asynchronous can be very difficult to implement because existing programming patterns don't scale to complex behavior
- Callbacks and promises can be used to deal with asynchronous code, but have many limitations when targeted against large streams generated from repeated button clicks or mouse movements.
- RxJS is a reactive solution that can more concisely and declaratively deal with large amounts of data separated over time.
- RxJS is a paradigm shift that requires seeing and understanding data in streams with propagation of change.
- Streams are made up from a producer (observable), where data flows through a pipeline, arriving at a consumer (observer). This same programming model is used whether or not data is separated by time.

2

Reacting with RxJS

This chapter covers:

- Looking at streams as the main unit of work
- Understanding functional programming's influence on RxJS
- Identifying different types of data sources and how to handle them
- Modeling data sources as RxJS observables
- Consuming observables with observers

When writing code in an object-oriented way, we are taught to decompose problems into components, interactions, and states. This breakdown occurs iteratively and on many levels, with each part getting further sub-divided into more components, until at last we arrive at a set of cohesive classes that implement a well-defined set of interactions. Hence, in the object-oriented approach, classes are the main unit of work. Every time a component is created, it will have state associated with it, and the manipulation of that state in a structured fashion is what advances application logic forward. For example, consider a typical online banking web site. Banking systems contain modules that encapsulate not only the business logic associated with withdrawing, depositing, and transferring money, but also domain models that store and manage other properties, such as account and user profiles. It's the manipulation of this state (its behavior) that causes the data to transform into your desired output. In other words, behavior is driven by the continuous mutation of a system's state. If such a system is designed using object-oriented programming, the units of work are the classes responsible for modeling accounts, users, money, and others.

RxJS programming works a bit differently. In reactive programming, in general, the fundamental unit of work is the stream.

In this chapter we will ask you to think in terms of streams (think reactively) and design code that, instead of holding on to data, allows it to flow through and applies transformations

along the way until it reaches your desired state. You will learn how to handle different types of data sources, whether static or dynamic, as RxJS streams using a consistent computational model based on the *Observable* data type. Unlike other JavaScript libraries, however, using RxJS in your application means much more than implementing new APIs; it means that you must approach your problems not as the sum of the set of states manipulated by methods in classes, but as a sequence of data that continuously travels from the producers to the consumers through a set of operators that implement your desired behavior.

This way of thinking places the notion of time at the forefront, which runs as the undercurrent through the components of an RxJS stream and causes data to be never stored but rather transiently flowing. Relating this to a real-world physical water stream, we often think of the data source as the top of the stream and the data consumer as the bottom of the stream. Hence, data is *always traveling downstream*, in a single direction, like water in a river, and along the way you can have control dams in charge of transforming the nature of this stream. Thinking this way will help you understand how data should move through an application.

This is not to say that this understanding will come easily—like any new skill, it must be built up over time and through iterative application of the concepts. As you saw in the pseudo-streams example in chapter 1, the notion of data in motion versus [data] kept in variables is a difficult one for most people to wrap their head around. In this book, we will provide you with the necessary tools to ease this learning curve. To begin building [your toolkit], this chapter starts by laying down the groundwork to help you better understand streams. Many of the basic principles behind reactive programming derive from functional programming, so let's start there.

2.1 Functional programming as the pillar of reactive programming

The abstractions that support reactive programming are built on top of Functional programming (FP), so FP is the foundation for reactive programming. In fact, a lot of the hype around reactive programming derives from the development communities and the industry realizing that functional programming offers a compelling way to design your code. This is why it's important for you to have at least a basic understanding of the FP principles. If you have a solid background in functional programming, you are free to skip this section; however, we recommend you read along because it will help you better understand some of the design decisions behind RxJS.

Just like in chapter 1, we ask you to take another quick glance at the main website for the Reactive Extensions project (http://reactivex.io). In it, you will find the following definition:

> **[RxJS] is a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming.**

You learned about the main components of the Observer pattern in chapter 1 (Producer and Consumer), now you'll learn the about the other parts that gave rise to the Rx project, which are functional programming and iterators. Here's a diagram (Figure 2.1) that better illustrates the relationship between these paradigms:

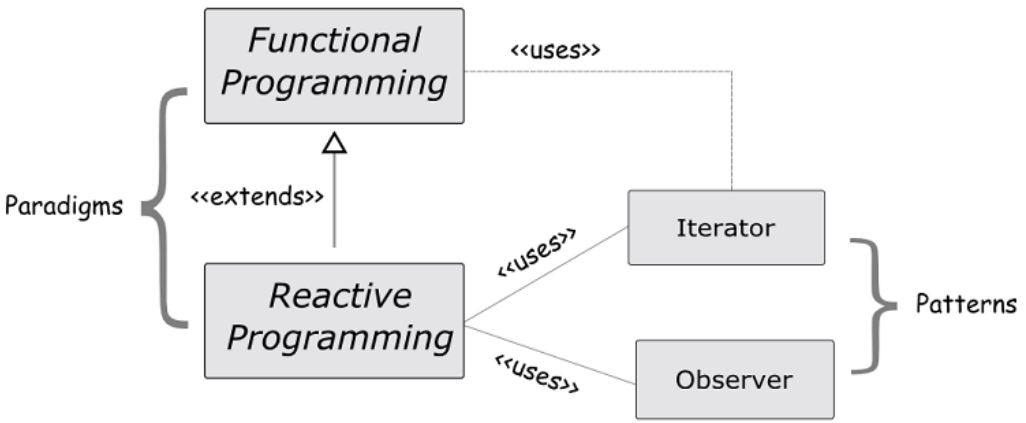

Figure 2.1 Reactive programming paradigm builds and extends from functional programming. Also, it leverages commonly-known design patterns such as Iterator and Observer

Let's begin with understanding the basics of functional programming.

2.1.1 Functional programming

Functional programming is a software paradigm that emphasizes the use of functions to create programs that are declarative, immutable, and side-effect free. Did you trip over the word 'immutable'? We agree with you; the notion of a program that doesn't ever change state is a bit mind bending. After all, that's why we put data in variables and modify them to our hearts content. All of the object-oriented or procedural application code you've written so far relies on changing and passing variables back and forth to solve problems. So, how can we accomplish the same goals without doing this? Take the example of a clock. When a clock goes from 1:00 pm to 2:00 pm, it's undoubtedly changing, isn't it? But to frame this from a functional point of view, we argue that instead of a single clock instance mutating every second, it's best to return new clock instances every second. Theoretically, both would arrive at the same time, and both would give you a single state at the end.

RxJS borrows lots of principles from FP, particularly *function chaining*, *lazy evaluation*, and the notion of using an abstract data type to orchestrate data flows. These are some of design decisions that drive the development of RxJS' stream programming via the `Observable` data type. Before we dive in, we'll explain the main parts of the FP definition we just gave, and then show you a quick example involving arrays.

To reiterate, functional programs are:

1. **Declarative**: functional code has a very peculiar style, which takes advantage of JavaScript's higher-order functions to apply specialized business logic. As you'll see later on, function chains (also known as pipelines) describe data transformation steps in a very idiomatic manner. Most people see SQL syntax as a perfect example of declarative code.
2. **Immutable**: an immutable program (and by this we mean any immutable function, module, or whole program) is one that never changes or modifies data after it's been created, or after its variables have been declared. This can be a very radical concept to grasp, especially coming from an object-oriented background. Functional programs treat data as immutable, constant values. A good example of a familiar module is the String type, as none of the operations change the string they operate; rather, they all return new strings. A good practice that you'll see us use throughout the book is to qualify all of our variables with `const` to create nicely block-scoped immutable variables that can't be reassigned. This doesn't solve all of the problems of immutability, but gives you a little extra support when your data and functions are shared globally.
3. **Side effect free**: functions with side effects depend on data residing outside of its own local scope. A function's scope is made up of its arguments and any local variables declared within. Interacting with anything outside of this (like reading a file, writing to the console, rendering elements on an HTML page, and more), are considered side effects and should be avoided or, at the very least, isolated. In this book, we'll learn how RxJS deals with these issues by pushing the effectful computations into the subscribers.

In general, mutations and side effects make functions unreliable and unpredictable. That is to say, if a function alters the contents of an object inadvertently, it will compromise other functions that expect this object to keep its original state. The OO solution to this is to encapsulate state, and protect it from direct access from other components of the system. In contrast, FP deals with state by eliminating it, so that your functions can confidently rely on it to run.

Figure 2.2 revisits the dependency between two functions `doWork()` and `doMoreWork()` through a shared state variable called `data`.

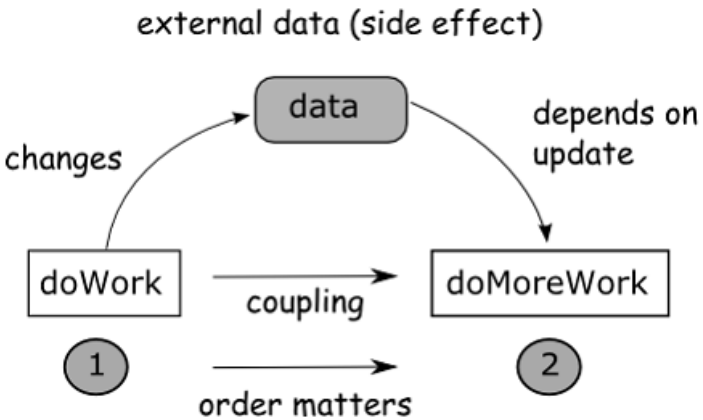

Figure 2.2 Function doWork() is temporally coupled to doMoreWork() due to the dependency on shared state (side effect). Hence, doWork() must called before doMoreWork() or the program ceases to work

This coupling presents an issue because `doMoreWork` now relies on `doWork` to run first. Two issues may occur:

- The result of `doMoreWork()` depends entirely on the successful outcome of `doWork()`, and on no other parts of the system changing this variable.
- Unit tests against this function can't be done in isolation, as they should be, so your test results are susceptible to the order in which the test cases are run (in chapter 9 we explore testing in much more detail).

Shared variables, especially in the global scope, add to the cognitive load of reasoning about your code because these variables demand that you keep track of them as you trace through it. Another way you can think of global data is as a hidden parameter into all of your functions. So, the more global the state you have to maintain, the harder it is for you to maintain your code. The example in figure 2.2 is a very obvious side effect, but they are not always this clear. Consider this trivial function that returns the lowest value in a numerical array:

```
const lowest = arr => arr.sort().shift();
```

While this code may seem harmless to you, it packs a terrible side effect. Can you spot it? This function actually changes the contents of the input array, as shown in the snippet below. So if you used the first element of the array somewhere else, that's completely gone now.

```
let source = [3,1,9,8,3,7,4,6,5];
let result = lowest(source); //-> 1
   console.log(source); //-> [3, 3, 4, 5, 6, 7, 8, 9] (1)
```

(1) **The original array changed!**

Later on, we'll talk about a functional library that provides a rich set of functions for working with arrays immutably, so that things like this don't inadvertently creep up on you.

Matters get worse if you had concurrent asynchronous processes where data structures are shared and used in different components. Because latency is unpredictable, you would need to either nest your function calls, or use some other robust synchronization mechanism to ensure they execute and mutate this state in the right order; otherwise, you'll experience very random and hard-to-troubleshoot bugs.

Fortunately, JavaScript is single-threaded, so we don't need to worry about shared state running through different threads. However, as JavaScript developers, we deal quite often with concurrent code when either working with web workers or simply making simultaneous HTTP calls. Consider this trivial, yet frequent, use case illustrated in figure 2.3, which involves asynchronous code mixed with synchronous code. This presents a tremendous challenge because the latter assumes that the functions executing before it have completed successfully, which might not necessarily be the case if there's some latency:

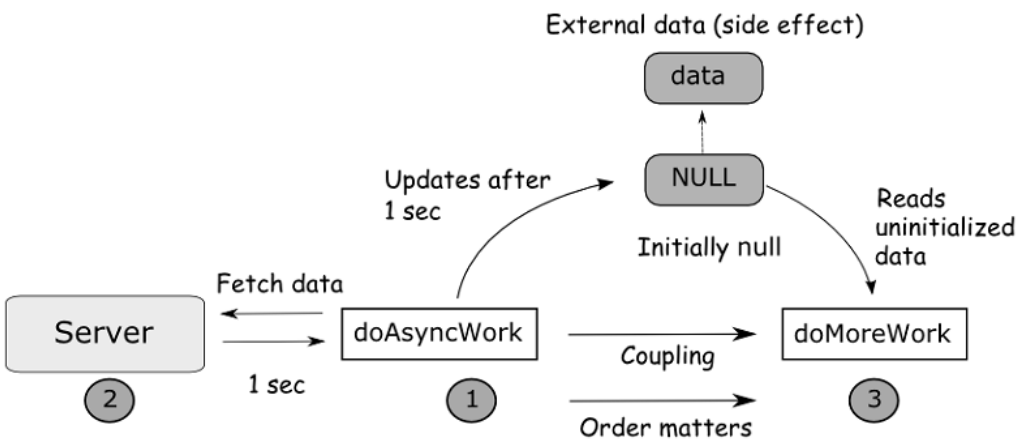

Figure 2.3 Function doAsyncWork() is an example of a remote call that fetches data from the server. Suppose this call has a latency around 1 second, depending on network conditions. Immediately after, the next function runs doMoreWork() expecting that a piece of shared data had already been initialized. Due to this latency, the shared data had not been initialized and the execution of doMoreWork() is compromised.

In this scenario, `doAsyncWork()` fetches some data from the server, which is never a constant amount of time. So, `doMoreWork()` fails to run properly as it reads data that hadn't yet been initialized. This is what callbacks and promises help you solve, so that you don't have to hard code your own timeouts in order to anticipate latency; dealing directly with time is a recipe for disaster as your code will be extremely brittle, hard to maintain, and will cause to you to come in to work during a weekend when your application is experiencing slightly more traffic than usual. Working with data immutably, using functional programming, with the help of an asynchronous library like RxJS, can make these timing issues disappear—immutable variables

are protected against time. In chapters 4 and 6, we'll cover timing and synchronization with observables, which offers a much superior solution to this problem.

Even though JavaScript is not a pure functional language, with a bit of discipline and the help of the proper libraries, you can use it completely functionally. As you learn to use RxJS, we ask that you also begin to embrace a functional coding style; it's something we believe strongly about and promote in all code samples in this book.

Aside from using `const` to safeguard the variable's reference, JavaScript also has support for a versatile array data structure with methods such as: `map`, `reduce`, `filter`, and others; these are known as higher-order or first-class functions, and are one of the most important functional qualities in the language, allowing you to express JavaScript programs in an idiomatic way. A higher-order function is defined as one that can accept as argument as well as return other functions; they are used extensively with RxJS, as with any functional data type.

Listing 2.1 shows a simple program that takes an array of numbers, extracts the even numbers, computes their squares, and sums their total.

Listing 2.1 Processing collections with map, reduce, and filter

```
const isEven = num => num % 2 === 0;
const square = num => num * num;
const add = (a, b) => a + b;

const arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

arr.filter(isEven).map(square).reduce(add);  //-> 220
```

In this example above, because these operations are side-effect free, this program will always produce the same value (220) given the same input array.

Where can I find this code?

All of the code for this book can be found in the RxJS in Action GitHub repository https://github.com/RxJSInAction. In there you will two sub-repositories. Under **rxjs-in-action** you will find a simple application that contains the code for all individual chapter listings 1 through 9. All samples are presented as runnable snippets of RxJS code that you can interact with. Also, under the **banking-in-action** repository you will find our web application that showcases RxJS embedded into a React/Redux architecture.

If you imagine for a second having to write this program using a non-functional or imperative approach, you'll probably need to write a loop, a conditional statement, and a few variables keeping track of things. Functional programming, on the other hand, raises the level of abstraction and encourages a style of declarative coding that clearly states the purpose of a program, describing *what* it does and not *how* it does it. Nowhere in this short program is the presence of a loop, if/else, or any imperative control flow mechanism.

In fact, one of the main themes in FP which you'll use as well in reactive, is learning to *program without loops*. In listing 2.1, we took advantage of `map`, `reduce`, and `filter` to hide

manual looping constructs—and allow you to implement looping logic through functions arguments. Moreover, these functions are also immutable, which means that new arrays are created at each step of the way, keeping the original intact.

Aside from using a mutable loop counter, another reason for eliminating loops, as we'll explain later on, is that they are designed to work with synchronous flows. Once you start fetching data asynchronously, loops begin to break down and you're going to have to resort to creating artificial closures in your code to ensure the order of operations is correct. To drive this point home, let's say you needed to fetch stock prices for different symbols. One approach could be something like:

```
let symbols = ['FB', 'GOOGL', 'CTXS']; (1)
for(let symbol of symbols) {
  ajax(`<host>/stocks?symbol=${symbol}`, (2)
     response => {  (3)
        console.log(`Symbol: ${symbol} | Stock price:  ${resp.price}`);
  });
}
```

(1) Stock symbols used are Facebook (FB), Google (GOOGL), and Citrix (CTXS
(2) Use an external service to fetch stock data
(3) Print the stock symbol with its price

It seems intuitive to loop through the set of symbols to fetch price information for, and display, each symbol with its price as it arrives. But the result is quite surprising. While the price information is correct (at the time of writing), the stock symbol is not. Running this program prints:

```
"Symbol: CTXS | Stock price:  119.379997"
"Symbol: CTXS | Stock price:  747.599976"
"Symbol: CTXS | Stock price:  85.500000"
```

Why is CTXS repeated? Because the synchronous for-loop is not "async-aware." Chapter 4 deals with timing, and we'll show you the RxJS way of getting this right.

Going back to our discussion, side-effect-free functions are also known as *pure*, because they are predictable when working on collections of objects or streams. You should always strive for purity whenever possible as it makes your programs easy to test and reason about.

Want to learn more about functional programming?

JavaScript's Array object has a special place in functional programming because it behaves as an extremely powerful data type called a functor. In a simple sense, functors are containers that can wrap data and expose a mapping method that allows you to immutably apply transformations on this data, as seen by the Array.map() method. As you'll see learn later on, RxJS streams follow this same functor-like design.

Functional programming is a huge subject to cover. In this book, we will only cover enough of FP to help you to understand and be proficient with RxJS and reactive programming. If you'd like more information about FP and FP topics, you can read about them in detail in Functional Programming in JavaScript (Manning 2016) by Luis Atencio.

The code shown in listing 2.1, which works very well with arrays, also translates to streams. Continuing with our pseudo Stream data type that we began in chapter 1, look at how similarly arrays and streams work when processing some number sequence:

```
Stream([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
  .filter(isEven)
  .map(square)
  .reduce(add)
  .subscribe(console.log);  //-> 220
```

You can clearly see how Rx was inspired by functional programming. All we had to do was to wrap the array into a stream, and then subscribe to it to listen for the computed values that derive from the sequence of steps declared in the stream's pipeline. This is the same as saying that *streams are containers that you can use to lift data (events)* into their context, so that you can apply sequences of operations on this data until reaching your desired outcome. Fortunately, you're very familiar with this concept already by working with arrays for many years. I can lift a value into an array and map any functions to it. Suppose I declare some simple functions on strings like `toUpper`, `slice`, and `repeat`:

```
['rxjs'].map(toUpper).map(slice(0, 2)).map(repeat(2)); //-> 'RXRX'
```

The ancient Greek philosopher Heraclitus once said: "You can never step into the same river twice." He formulated this statement as part of his doctrine on *change* and *motion* being central components of the universe—everything is constantly in motion. This epic realization is what RxJS streams are all about: as data continuously flows and moves through it, orchestrated through this data type we're learning about called a Stream. Despite being dynamic, streams are actually immutable data types. Once a stream is declared to wrap an array, listen for mouse clicks, or respond to an HTTP call, you can't mutate it or add a new value to it afterwards—you must do it at the time of declaration. Hence, we're *specifying the dynamic behavior of an object or value declaratively and immutably*. We'll revisit this topic a bit more in the next chapter.

Moreover, the business logic of this program is pure and takes advantage of side effect free functions that are mapped onto the stream to transform the produced data into the desired outcome. The advantage of this is that all side effects are isolated and pushed onto the consumers (logging to the console in this case). This separation of concerns is ideal and keeps our business logic clean and pure. Figure 2.4 shows the role that the producer and consumer play:

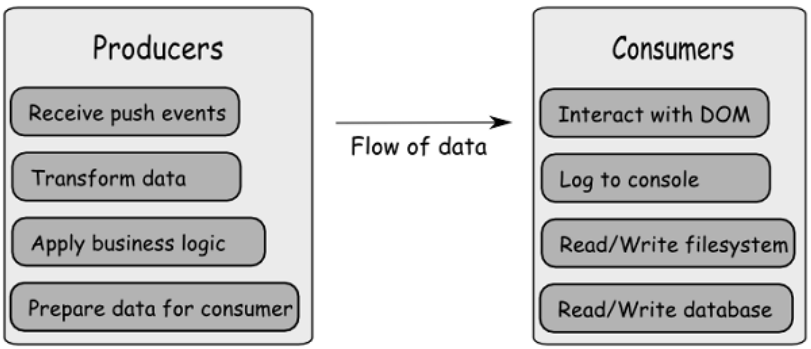

Figure 2.4 Events emitted by producers are pushed through a pipeline of side-effect free functions, which implement the business logic of you program. This data flows to all observers in charge of consuming and displaying it.

Another design principle of streams that's borrowed from functional programing is lazy evaluation. Lazy evaluation simply means that code is never called until actually needed. In other words, functions won't evaluate until their results are used as part of some other expression. In this example below, the idea is that streams sit idle until a subscriber (a consumer) is attached to it, only then will it emit the values 1 – 10:

```
Stream([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
  .filter(isEven)
  .map(square)
  .reduce(add); ❶
```

❶ Nothing actually runs here because there's no subscriber added

When a subscriber begins listening, the stream will emit events downstream through the pipeline in a single unidirectional flow from the producer to the consumer. This is beneficial if your functions have side effects because the pipeline runs in a single direction, helping you to ensure an orderly execution of your function calls. This is another reason to avoid side effects at all costs, especially when you begin combining multiple streams, as things can revert into the tangled mess that we're trying to get rid of in the first place. Lazy evaluation is actually a mandatory requirement for streams because they emit data infinitely to handle mouse movement, key presses, and other asynchronous messages. Otherwise, storing the entire sequence of mouse movements in memory could make your programs crash.

REACTIVE MANIFESTO One of the key principles of a reactive system is the ability to stay afloat under varying workloads—known as *elasticity*. Now, obviously this has lots of architectural and infrastructural implications that extend beyond the scope of this book; however, corollary to this is that the paradigm you use should not change whether you're dealing with 1, 100, or thousands of events. RxJS offers a single computing model to handle finite as well as infinite streams. The reactive manifesto (http://www.reactivemanifesto.org) is a working group that aims at identifying patterns for building reactive systems. It really has no direct relation to the Rx libraries, but philosophically there are many points in common.

For instance, without lazy evaluation code that uses infinite streams like this will cause the application to run out of memory and halt:

```
//1
Stream.range(1, Number.POSITIVE_INFINITY) /(1)
  .take(100)
  .subscribe(console.log);
```

```
//2
Stream.fromEvent('mousemove') (2)
   .map(e => [e.clientX, e.clientY])
   .subscribe(console.log);
```

(1) **Read infinitely many numbers in memory**
(2) **Listen to all mouse moves the user is performing**

In example 1 above, lazy evaluation makes the stream smart enough to understand that it will never need to actually run through all the positive numbers infinitely before taking the first 100. And even if the amount of numbers to store is big, streams will not persistently hold on to data; instead any data emitted is immediately broadcast out to all subscribers at the moment it gets generated. In example 2, imagine if we needed to store in memory the coordinates of all mouse movements on the screen; this could potentially take up a huge amount. Instead of holding to this data, RxJS lets it flow freely and uses the Iterator pattern to traverse any type of data source irrespective of how it's created.

2.1.2 The Iterator pattern

A key design principle behind RxJS streams is to give you a familiar traversal mechanism, just as you do with arrays. Iterators are used to traverse containers of data in a structure-agnostic way, or independent of the underlying data structure used to harness these elements, whether it's an array, a tree, a map, or even a stream. In addition, this pattern is effective at *decoupling the business logic applied at each element from the iteration itself*. The goal is just to provide a single protocol for accessing each element and moving on to the next, as shown in figure 2.5:

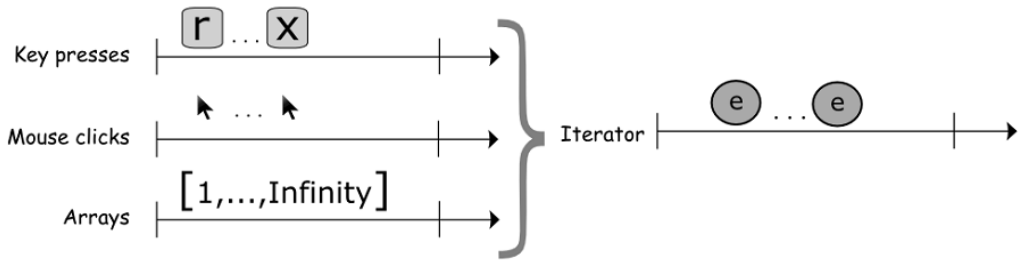

Figure 2.5 Iterators abstract the traversal mechanism, whether it be a for or a while loop, so that processing any type of data is done in the exact same way

We'll explain this pattern briefly now, and later on you'll see how this applies to streams. The JavaScript ES6 (or ES2015) standard defines the Iterator protocol which allows you to define or customize the iteration behavior of any `Iterable` object. The iterable objects you're most familiar with are arrays and strings. ES6 added `Map` and `Set` as well. With RxJS, we're going to treat streams as iterable data types as well.

We can make any object iterable by manipulating its underlying iterator. I'll be using some ES6 specific syntax to show this. Consider an iterator object that traverses an array of numbers and buffers a set amount of contiguous elements. Here, the business logic performed is the buffering itself, which can be very useful to group elements together to form numerical sets of any dimension, like the ones illustrated in figure 2.6:

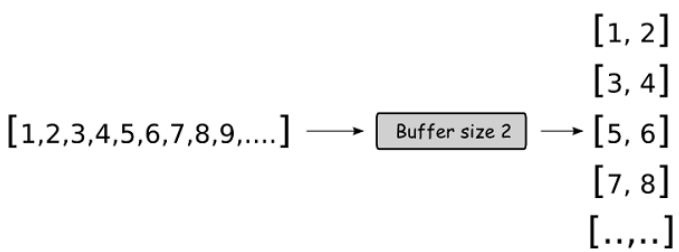

Figure 2.6 Using an iterator to display sets of numbers of size 3

Now let's see what the code would look like. Listing 2.2 shows the internal implementation of this custom iterator, which contains the buffer logic:

Listing 2.2 Custom BufferIterator function

```
function BufferIterator(arr, bufferSize = 2) {          (1)
    this[Symbol.iterator] = function () {               (2)
        let nextIndex = 0;

        return {
            next: () => {                               (3)
```

```
            if(nextIndex >= arr.length) {
              return {done: true};                    ❹
            }
            else {
              let buffer = new Array(bufferSize);
              for(let i = 0; i < bufferSize; i++) {   ❺
                buffer[i] = (arr[nextIndex++]);
              }
              return {value: buffer, done: false};   ❻
            }
          }
        }
    };
}
```

❶ Assigning a default buffer size of 2
❷ Overrides the provided array's iterator mechanism. Symbol.iterator represents the array's iterator function.
❸ The next() function is part of the Iterator interface and marks the next element in the iteration
❹ Returning an object with a done = true property, causes the iteration mechanism to stop
❺ Creating a temporary buffer array to group contiguous elements
❻ Return the buffered items, and a status of done = false, which will indicate to the iteration mechanism to continue

Any clients of this API need to only interact with the `next()` function, as outlined in the class diagram in figure 2.6. The business logic is hidden from the caller, the `for…of` block, which is the main goal of the iterator pattern:

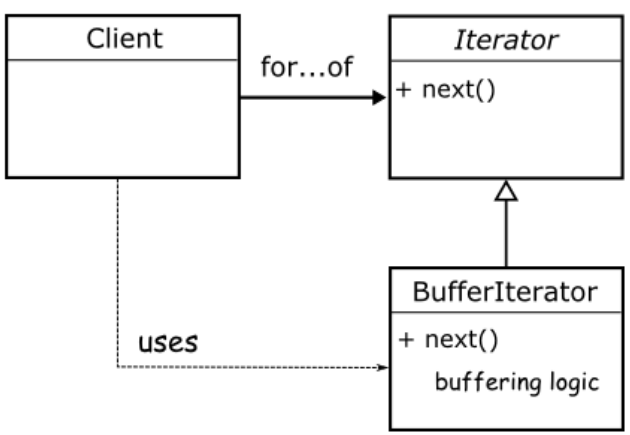

Figure 2.7 A class diagram (UML) highlighting the components of the Iterator pattern. The *Iterator* interface defines the next() function, which is implemented by any concrete iterator (BufferIterator). Users of this API need only interact with the interface, which is general and applies to any custom traversal mechanism

The `next()` function in listing 2.2 is used to customize the behavior of the iteration through `for...of` or any other looping mechanism. As you'll see later on, RxJS observers also implement a similar interface to signal to the stream to continue emitting elements.

Did iterators throw you for a loop?

The ES6 Iterator/Iterable protocols are really powerful features of the language. RxJS development predates this protocol, so it doesn't use it at its core, but in many ways the pattern is still applied. We won't be using Iterators in this book; nevertheless, we recommend you learn about them. You can read more about this protocol here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#iterator

Iterators allows you to easily take advantage of the JavaScript runtime to take care of the iteration on your behalf. Below we show some examples of this using our simple numerical domain. In fact, buffering is built into RxJS and it's really useful to gather up a sequence of events and make decisions about the nature of these events or apply additional logic. An example of this is when you need to invoke an expensive operation in response a sequence of mouse events, like drag and drop. Instead of running expensive code at each mouse position, you buffer a specific amount of them, and emit a single response taking all into account. Implementing this yourself would be really tricky, because it would involve time management and keeping external state that tracks the frequency and speed with which the user moves the mouse; certainly you'll want to delegate this to libraries that understand how to manage all of this for you. We'll study buffers in more detail in chapter 4. In RxJS, buffers are not really implemented like listing 2.2, but it serves to show you an example of how you can buffer data using iterators, which is how you think about these sorts of operations. Let's show our `BufferIterator` in action:

```
const arr = [1, 2, 3, 4, 5, 6];

for(let i of new BufferIterator(arr, 2)) {  ❶
    console.log(i);
}
//-> [1, 2] [3, 4] [5, 6]

for(let i of new BufferIterator(arr, 3)) {  ❷
    console.log(i);
}
//-> [1, 2, 3] [4, 5, 6]
```

❶ Buffer two elements at once
❷ Buffer three elements at once. Notice how the iteration mechanism is completed separated from the buffering logic

When you subscribe to a stream, you will be traversing through many other data sources such as mouse clicks and key presses in the exact same way. Theoretically speaking, because our pseudo `Stream` type is an iterable object, I could traverse a set of key press events as well with a conventional loop:

```
const stream = Stream(R, x, J, S)[Symbol.iterator](); ❶

for(let keyEvent of stream) {                          ❷
    console.log(event.keyCode);
}
//-> 82, 120, 74, 83
```

1. Creating a stream that wraps key presses for those 4 letters
2. Traversing a stream is semantically equivalent as subscribing to it (more on this later)

Streams in RxJS also respect the `Iterator` interface, and subscribers of this stream will listen for all of the events contained inside of it. As you saw previously, iterators are great at decoupling the iteration mechanism, and data being iterated over, from the business logic. When data itself defines the control flow of the program, this is known as *data-driven* code.

2.2 Stream's data-driven approach

RxJS encourages a style of development known as data-driven programming. The data-driven approach is a way of writing code such that we can separate the behavior of an application from the data that is passing through it. This is a core design decision of RxJS and the main reason why you can use the same paradigm to process arrays, mouse clicks, or data from AJAX calls.

In the object-oriented approach, we place more emphasis on the supporting structures rather than the data itself. This explains why pure OO languages like Java have many different implementations to store a sequential collection of elements, each tackling different use cases: `Array`, `ArrayList`, `LinkedList`, `DoublyLinkedList`, `ConcurrentLinkedList`, and others. To put it another way, imagine that you ran a local florist that performed deliveries. Your business in this case is importing flowers, cutting them, packaging them, handling orders and sending those orders out for delivery. These tasks are all part of your business logic, that is, they are the important bits that your customers care about, and the parts that bring in revenue. Now imagine that in addition to those tasks you were also tasked with designing the type of delivery van to use. Creating these structures is itself a full-time job, and one that would likely distract from your primary business without meaningfully lending to it.

Data, as in the data that we care about, and that which gives rise to search engines, websites and video games, are the flowers of software design. Creating software should therefore be about how we manipulate data rather than the how we create approximations of real-world objects (as we might in object-oriented programming). Bringing data to the forefront and separating it from the behavior of the system is at the heart of data-driven/centric design. Similarly, loosely coupling functions from the objects that contain data is a design principle of functional programming and, by extension, reactive programming.

To be driven by data is to be compelled to act by the presence of it, and let it fuel our logic. Without data to act upon, behavior should do nothing. This idea of data giving life to the behavior ties back to our earlier definition of what it means to be reactive—react to data instead of waiting for it. Streams are nothing more than a passive description of a process that sit idle when nothing is pushed through them and not consumer is attached:

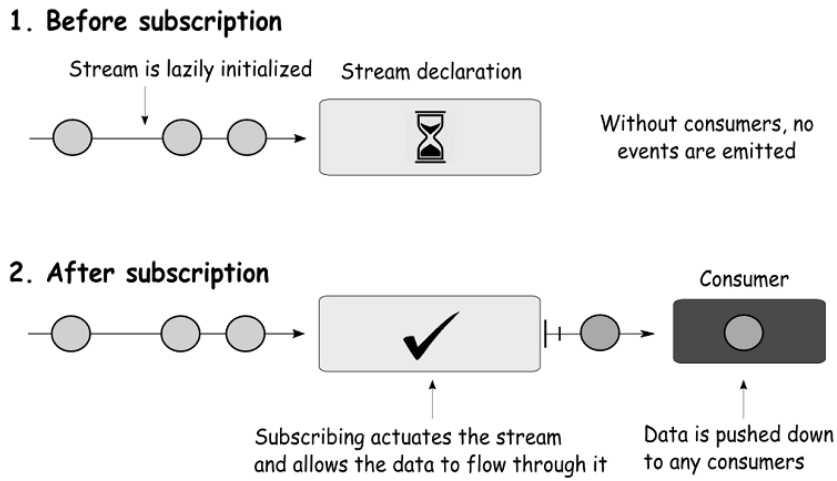

Figure 2.8 Initially, streams are lazy programs that wait for a subscriber to become available. Any events received at this point are discarded. Subscribing to the stream puts the wheels in motion and event data flows through the pipeline and out for consumers to use

This design pattern seems intuitive to most people simply because we think of data as requiring some sort of behavior in order to be meaningful. In a physics simulation the mass of a ball is just a decimal number without context until the behavior of gravity is applied to it. Thus, if we are to imagine that both are intertwined by nature, it seems only natural that they should cohabitate logically within an object. In theory this would seem to be a fairly obvious approach, and indeed the prevalence and popularity of object-oriented programming stands testament to its power as a programming paradigm.

However, it turns out that the greatest strength of object-oriented design is also perhaps its greatest weakness. The intuition of representing components as objects with intrinsic behavior makes sense to a certain extent, but much like the real world, it can become difficult to reason about as the complexity of the application grows. For instance, if we hadn't used the `BufferIterator` type before, we would've had to implement the buffering logic with the application logic that uses this data. To keep things simple, we just logged the numbers to the screen, but in real life you will use iterators for something more meaningful.

The data-centric approach seeks to remedy this issue by separating the concerns of data and behavior, through its Producer/Consumer model. Data would be lifted out of the behavior logic and instead it would pass through it. Behavior could be loosely linked such that the data moved from one part of the application to another, independent from the underlying implementation. Earlier you saw how iterators help with this:

```
Stream([1, 2, 3, 4, 5, 6])
   .buffer(2)
   .subscribe(console.log)); //-> [1, 2] [3, 4] [5, 6]
```

Each step in the pipeline resides within its own scope that is externalized from the rest of the logic. In this case, you can see that just like iterators, the buffering step is done separately from the code acting on the data. By doing so, we have both declared the intent of each step and effectively decoupled the data from the underlying implementation, because each component only reacts to the step that preceded it.

Furthermore, producers come in all shapes and sizes. Event emitters are one of the most common ones, they are used to respond to events like mouse clicks or web-requests. Also, there are timer-based sources like `setTimeout` and `setInterval` that will execute a task at a specified point in the future. Then there are subtler ones such as arrays and strings, which we might recognize as collections of data but not necessarily producers of data.

Traditionally, when dealing with each of these data sources, you've been conditioned to think of them as requiring a different approach. For instance, event emitters require named event handlers, promises require the continuation-passing "thenable" function, `setTimeout` needs a callback, and arrays need a loop to iterate through them. What if we told you that all of these data sources can be consolidated and processed in the exact same way?

2.3 Wrapping data sources with Rx.Observable

All along, we've been using a pseudo data type called Stream as a substitute for the real `Rx.Observable` type available in RxJS 5. We did this to help you understand the paradigm, and what it means to think in streams, rather than focus on the specifics of this library. In this section, we're going to begin diving into the RxJS 5 APIs (for information about installing RxJS 5 on the client or on the server, please visit Appendix A). Through the `Rx.Observable` type, we can subscribe to events produced from different types of data sources.

> **ES7 SPECIFICATION** One of the key design decisions behind the development of RxJS 5 was to create an Observable type that follows the proposed Observable specification slated for the next version of JavaScript ES7. You can find all of the details of this API here: https://github.com/zenparsing/es-observable

You can lift a heterogeneous set of inputs into the context of an observable object. Doing so allows you to unlock the power of RxJS to transform/manipulate them to reach your desired outcome. First, let's identify these different types of data:

2.3.1 Identifying different sources of data

We've mentioned earlier that the advantage of separating data and behavior, is that you can reason about a holistic model to account for any type of data. Hence, the first step to breaking the data free is to understand that all of these data sources are in fact the same when viewed through a data-driven (or stream-driven?) lens. First, let's re-categorize the types of data we'll encounter. Rather than dealing with them as strict JavaScript types let's look at some broader categories of data.

EMITTED DATA

Emitted data is data that will be created as a result of some sort of interaction with the system, this can be either from a user interaction such as a mouse click, or a system event like a file read. As we alluded to in chapter 1, some of these will have at most one event, that is, we request data and then at some point in the future we receive a response. For this, promises can be a really good solution. Others, like a user's clicks and key presses, are part of a continuous process, and this requires us to treat them as event emitters that produce multiple discrete events at future times.

STATIC DATA

Static data is data that is already in existence and present in the system (in memory). For example, an array or a string. Artificial unit test data also falls into this category. Interacting with it is usually a matter of iterating through it. If we were wrapping a stream around an array, for instance, the stream would never actually store the array, it would just extend it with a mechanism that flushes the elements within the array (based on iterators). Arrays are a common and heavily used static data source, but we could also think of associative arrays or maps as unordered static data. Most of the examples so far have dealt with static data such as strings, numbers, and arrays, as we use them to illustrate some of the basic concepts. In later parts of the book, we'll focus on emitted data and generated data.

GENERATED DATA

Generated data is data that we create periodically or eventually, like a clock sounding a chime every quarter hour; it can also be something more procedural like generating the Fibonacci sequence using ES6 generators. In the latter case, since the sequence is infinite it is not feasible to store it all in memory. Instead each value should be generated on-the-fly and yielded to the client as needed. In this category, we can also place the traditional `setTimeout` and `setInterval` functions which use a timer to trigger events in the future.

Just like the saying: "When you're a hammer, every problem looks like a nail," the `Rx.Observable` data type can be used to normalize and process each of these data sources using a single programming model—it's the hammer. With this approach, you gain the most code reuse and avoid creating specific ad-hoc functions to deal with the idiosyncrasies of each event type.

2.3.2 Creating RxJS Observables

In Rx, an observer subscribes to an observable. As you learned in chapter 1, this is analogous to the Subject-Observer pattern with the subject acting as the observable; `Rx.Observable` represents the object that pushes notifications for observers to receive. The observers asynchronously react to any events emitted from the observable, which allows your application to remain responsive instead of blocking in the face of a deluge of events. This is ideal to implement asynchronous, responsive code both on the client and the server.

`Rx.Observable` has a different meaning to different people. To functional programming purists, it falls under a special category called a functor, an *endofunctor* to be exact. (We don't cover functors in this book because they are not essential to understanding Rx, but if you want learn more about them, they are explained in the functional programming book mentioned earlier.) To most others, it's simply a data type that wraps a given data source, present in memory or eventually in the future, and allows you to chain operations on to them by invoking observable instance methods sequentially.

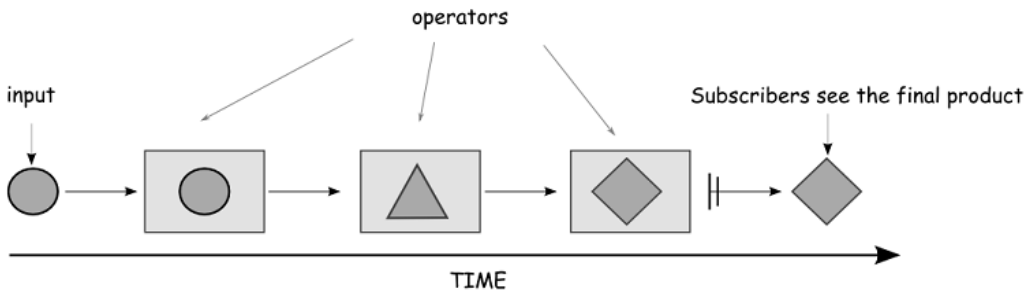

Figure 2.9 Shows the sequential application of methods or operators that transform an input into the desired outcome, which is what subscribers see.

Here's a quick look at how observables implement chaining extremely well:

```
Rx.Observable.from(<data-source>) ❶
   .operator1(...)                ❷
   .operator2(...)
   .operator3(...)
   .subscribe(<process-output>);  ❸
```

❶ **Wrap a data source with a stream**
❷ **Invoke a sequence of operations chained by the "dot" operator. In chapter 3, we will spend a lot more time with observable instance methods**
❸ **Process results**

Whether you choose to accept one definition over the other, it's important to understand that an observable doesn't just represent a value *now*, but also the *idea of a value occurring in the future*. In functional programming, this is the same definition given to pure functions, which are nothing more than to-be-computed values, and part of the reason why we refer to the "methods" invoked on an observable instance as *operators*.

Because observables in RxJS are immutable data types, this pattern works quite well and should not look that foreign to you. Consider a familiar data type, strings. Look at this trivial example and notice its similarity to the pattern above:

```
String('RxJS')
   .toUpperCase()
   .substring(0, 2)
```

```
  .concat(' ')
  .repeat(3)
  .trim()
  .concat('!') //->  "RX RX RX!"
```

Now, learning about a shiny new tool is always exciting, and there is a tendency among developers to try and use that tool in every conceivable situation where it might potentially apply. However, as is often the case, no tool is meant for every situation and it is just as important to understand where RxJS will not be used.

We can divide our computing tasks into four groups within two different dimensions. The first dimension is the number of pieces of data to process. The second is the manner in which the data must be processed, that is, synchronously or asynchronously. In enumerating these possibilities, we want to highlight where RxJS would be most beneficial to your applications.

2.3.3 When and where to use RxJS

Learning to use a new tool is as important as learning when not to use it. The types of data sources we'll be dealing with in this book can be classified into the four different categories listed in figure 2.10, which we'll explain next:

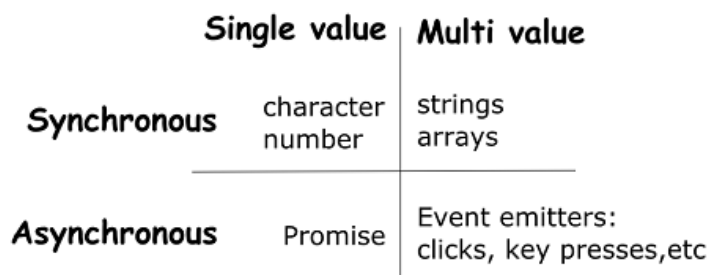

Figure 2.10 Different types of data sources with examples in each cuadrant

SINGLE-VALUE, SYNCHRONOUS

The simplest case is that we would have only a single piece of data. In programming we know that there are operations that return a single value for each invocation. This is the category of any function that returns a single object. We can use the `Rx.Observable.of()` to wrap a single, synchronous value. As soon as the subscriber is attached, the value is emitted (we haven't yet explained the details behind `subscribe`, but we'll cover that in a bit)

```
Rx.Observable.of(42).subscribe(console.log); //-> 42
```

While there are cases where we'll need to wrap single values, in most cases if your goal is just to perform simple operations on them (concatenating another string, adding another number, and others), an observable wrapper may be overkill. The only times we'll wrap simple values with observables is when they combine with other streams.

MULTI-VALUE, SYNCHRONOUS

We can also group single items together to form collections of data, mainly for arrays. In order to apply the same operation that we used on the single item on all of the items, we would traditionally iterate over the collection and repeatedly apply the same operation to each item in the collection. With RxJS, it works in exactly the same way

```
Rx.Observable.from([1, 2, 3, 4, 5]).subscribe(console.log);
// -> 1, 2, 3, 4, 5

Rx.Observable.from('RxJS').subscribe(console.log);
// -> "R", "x", "J", "S"
```

The RxJS `from()` operator is probably one of the most common ones to use. And to make it a bit more idiomatic, RxJS has overloaded the `forEach` observable method as well, with the exact same semantics as `subscribe`:

```
const map = new Map();

map.set('key1', 'value1');
map.set('key2', 'value2');

Rx.Observable.from(map).forEach(console.log);
//-> ["key1", "value1"] ["key2", "value2"]
```

Both of these groups operate synchronously, which means each subsequent block of code must wait for the previous block to complete before executing. In the multi-value example, each item will be processed serially (one-by-one) until the collection is exhausted. This behavior is useful when dealing with items that have been pre-allocated, like arrays, sets, or maps, or if they can be generated, in place, on demand. Essentially, you can consider synchronous behavior to be actions on demand with results returning immediately (or at the very least before any further processing is done). When this is not the case, data is known as asynchronous.

SINGLE-VALUE, ASYNCHRONOUS

This brings us to the second dimension of computing tasks, where RxJS gives you the most benefits. This dimension addresses whether a task will execute synchronously or asynchronously. In the latter case, code is only guaranteed to run at some time in the future, thus subsequent code blocks cannot rely on any execution of a previous block having already taken place. As with the first dimension we also have a single value case, where the result of a task will result in a single return value. This kind of operation is usually used to load some remote resource via an AJAX call or wait on the result of some non-local calculation wrapped in a promise, without blocking the application. In either case, after the operation is initiated, it will expect a single return value or an error.

As we mentioned previously, in JavaScript this case is often handled using promises. A promise is similar to the single value data case in that it only resolves or errors once. RxJS has

methods to seamlessly integrate with them. Consider this simple example of a promise resolving into a single, asynchronous value:

```
const fortyTwo = new Promise((resolve, reject) => {
   setTimeout(() => {
      resolve(42);
   }, 5000);
});

Rx.Observable.fromPromise(fortyTwo)
   .map(increment)
   .subscribe(console.log); //-> 43

console.log('Program terminated');
```

> **NOTE** The promised value is being computed asynchronously; however, promises differ from observables in that they are executed *eagerly*, as soon as they're declared.

Running this program as-is produces the following output:

```
'Program terminated'

43 //-> after 5 seconds elapse
```

And because promises are single-value, immutable, they are never run again. So, if we subscribe to it 10 seconds later, it will just return the same value ten times—this is a desirable trait of a promise by design. In chapter 7, you'll learn that we can retry a promise observable and force it to be executed many times by nesting within another observable, which has support for retries. Using the version of `ajax(url)` that returns a promise, I can write the following:

```
Rx.Observable.fromPromise(ajax('/data'))
   .subscribe(data => console.log(data.id));
```

Another frequently used alternative is to use jQuery's deferred objects, which also implement the `Promise` interface. Particularly, you can use functions like `$.get(url)` or `$.getJSON(url)`:

```
Rx.Observable.fromPromise($.get('/data'))
   .subscribe(data => console.log(data.id));
```

MULTI-VALUE, ASYNCHRONOUS

For those keeping score, this brings us to our fourth and final group of computing tasks. The tasks in the fourth group are those that will produce multiple values yet do so asynchronously. We create this category especially for the DOM events, which are all asynchronous and can occur infinitely many times. This means that we will need a mix of semantics from both the Iterator and the Promise pattern. More specifically we need a way to process infinitely many items in sequence and capture any errors that occur. These items could be data fetched from

remote AJAX calls or data generated from dragging the mouse across the screen. For this we need to invert our control structures to operate asynchronously.

The typical solution to problems of this nature would be to use an `EventEmitter`. It provides hooks or callbacks to which closures can be passed, in this way it is very much like the Promise. However, an event emitter does not stop after a single event, instead, it can continue to invoke the registered callbacks for each event that arrives, creating a practically infinite stream of events. The emitter will fulfill both of our criteria for handling multi-value, asynchronous events. However, it's not without its share of problems. Though simple to use, event emitters do not scale well for larger systems, as their simplicity leads to a lack of versatility. The semantics for unsubscribing and disposing of them are cumbersome and there is no native support for error handling.

Rather, we can use RxJS to wrap event emitters and bring in all of the benefits and versatility of observables with it. The following code attaches a callback to a click event on a link HTML element:

```
const link = document.querySelector('#google');                ❶

const clickStream = Rx.Observable.fromEvent(link, 'click')     ❷
  .map(event => event.currentTarget.getAttribute('href'))      ❸
  .subscribe(console.log); //-> http://www.google.com
```

❶ Query the DOM for the link HTML element
❷ Create an observable around click events on this link
❸ Extract the links HREF attribute

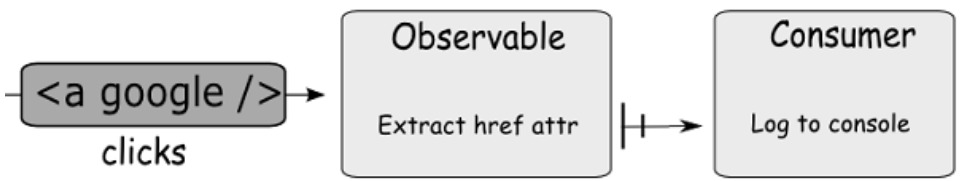

Figure 2.11 Observable that wraps click events and passes them down to the observer for processing.

NOTE In this example, the `subscribe()` method was used to process click events and perform the required business logic, in this case extracting the `href` attribute. Later on, when we cover the Observable instance methods that form the pipeline, you'll see concrete examples of how to decouple the business logic from the printing of the result.

You can also use Observables to wrap any custom event emitters. Going back to our calculator emitter in Node.js, instead of listening for the `add` event:

```
addEmitter.on('add', (a, b) => { ❸
  console.log(a + b); //-> Prints 5
});
```

We can subscribe to it:

```
const Rx = require('rx');

Rx.Observable.fromEvent(addEmitter, 'add',
   (a, b) => ({a: a, b: b}))
   .map(input -> input.a + input.b)
   .subscribe(console.log); //-> 5

addEmitter.emit('add', 2, 3);
```

In this section, we only covered a few of the ways for creating observables with RxJS. Later on, we tackle more complex problems as well as new Observable methods.

2.3.4 To push or not to push

Event emitters have been around as long as the JavaScript language. In that time, they have not had any significant improvements to their interface in the latest releases of the language. This contrasts with promises, iterators, and generators, which were part of the JavaScript ES6 specification and are already supported in many browsers at the time of writing. This is one of the reasons why RxJS is so important; it brings many improvements to JavaScript's event system.

Event emitters parse through a sequence of events asynchronously, so they come really close to being an iterator and, hence, a stream. The difference, however, lies in the way data is consumed by its clients—whether it is pulled or pushed. This is extremely important to understand, because most of the literature for RxJS defines observables as objects that represent *push-based* collections. Figure 2.12 highlights the main difference between both mechanism, which we'll explain immediately:

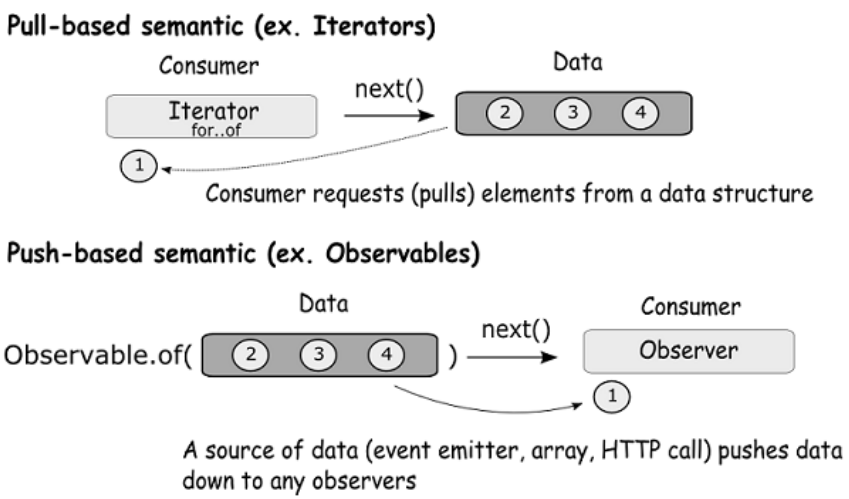

Figure 2.12 Notice the positions of the consumer and the direction of the data. In pull-based semantics, the consumer requests data (iterators work this way); while, in pushed-based semantics, data is sent from the source to the consumer without it requesting it. Observables work this way

Iterators use a pull-based semantic. This means that the consumer of the iterator is responsible for requesting the next item from the iterator. This "data-on-demand" model has two major benefits. First, it creates an abstraction over the data structure that is being used. Essentially, any data source that exposes some common method of iteration can be used interchangeably with one another. The second benefit of data-on-demand is for sequences of data that result from some calculation. Such is the case with JavaScript generators.

For instance, for a Fibonacci number sequence, which is itself infinite, we only need to calculate numbers as they are requested rather than wasting computing time generating parts of a sequence that the caller doesn't care about. This is immensely helpful if the data source is expensive or difficult to calculate. In listing 2.3 we use a Generator to create a lazy Fibonacci calculator. Generators are nothing more than iterators behind the scenes, so each value will only be produced when the consumer calls (or pulls) the `next()` method.

Listing 2.3 Fibonacci function using generators

```
function* fibonacci() {          ❶
  var first = 1, second = 1;     ❷
  for(;;) {
    var sum = second + first;
    yield sum;                   ❸
    first = second;
    second = sum;
  }
}

const iter = fibonacci();        ❹
iter.next(); //-> {value: 2, done: false}
iter.next(); //-> {value: 3, done: false}
iter.next(); //-> {value: 5, done: false}
```

❶ A generator function is denoted by the * "star" notation
❷ Fibonacci sequence must be initialized with at least two values
❸ yield will return the result of each intermediate step in the loop.
❹ Create the generator

Want to learn more about generators?

Generators are a language feature added into JavaScript as part of the ES6 specification. From a syntax point of view, it introduces the function* and yield keywords. A function with an asterisk declares that a function behaves as a generator, which means it can exit with a return value via yield, and later re-entered. Under the hood, generators don't actually execute immediately, but return an Iterator object, which is accessed via its next() method. Through this Iterator object, a generator can pause and resume remembering exactly where it left off and any context (closure) is kept across re-entrances. They're a rare, but powerful construct to use to produce infinite data using a given formula or template. If you want to learn more about them, we recommend you read the documentation: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*

A pull-based paradigm is useful in cases where we know that a value can be returned immediately from a computation. However, in scenarios like listening for a mouse click, where the consumer has no way of knowing when the next piece of data will become available, this paradigm breaks down. It is for this reason we require a corresponding type on the asynchronous side which is *push-based*—opposite of the pull-based approach. In a push paradigm, the producer is responsible for creating the next item, while the consumer only listens for new events. As an example of this, consider your phone's email client. A pull-based mechanism that checks for new email every second can drain the resources of your mobile device quickly; whereas, with push email, or any push notifications for that matter, your email client only needs to react to any incoming messages once.

RxJS observables use push-based notifications, which means they don't request data; rather, data is pushed onto them so that they can react to it. Push notifications bring the reactive paradigm to life. RxJS proposes observables as an improvement over event emitters because they are more versatile and extensible. The observable also serves as a better contemporary to the Iterator type, given that it possesses similar semantics, but with push-based mechanism.

So, you can see from all of our discussion so far how iterators and promises can be potential data sources that can be wrapped as observables, even though we earlier classified them as distinct groups. This ability to adapt not just the types it is replacing, but also types from other groups is immensely powerful—observables work equally well across synchronous and asynchronous boundaries. It not only makes interfacing with legacy code incredibly easy, but also it allows consumer code to be written independently of how the producer is implemented.

> **WATCH OUT!** **This power comes with responsibility as well, for while we are able to convert anything our hearts desire into Observables, it doesn't always mean that we should. In particular, processes that are strictly synchronous and iterative or will only ever deal with a single value do not need to be "reactivized" just for the sake of being cool. Even though Observables are cheap to create, there is a bit of overhead associated with just applying simple operations on data. For instance, just transforming a string from lower to uppercase does not need to be wrapped with an observable, you should directly use the string methods. So, don't be reactive just because you can.**

In RxJS, we will always have a pipeline that takes data from source to the corresponding consumer. Data will always be created or materialized from a data source. Again, the type of data source is not relevant to how our abstraction of none operates. Likewise, when data reaches the end of its journey and must be consumed, it is immaterial where the data came from. We'll reiterate that the separation and abstraction of these two concepts, data production and data consumption, is important for three reasons:

- It enables us to hide differences of implementation behind a common interface, which lets us focus more on the business logic of our task. This has the benefit of not only optimizing development time but also reducing code complexity by removing extra

noise from code

- The separation of production and consumption builds a clear separation of concerns and makes the direction of data flow clear.
- It makes streams testable by allowing you to attach mock versions of the producer and wire the corresponding matching expectations in the observer.

Now that you understand how streams can be constructed, we're missing the last piece and where observers come into play—stream consumption.

2.4 Consuming data with observers

Every piece of data that gets emitted and processed through an observable needs a destination. In other words, what was the purpose of capturing and processing a certain event? Observers are created within the context of a subscription, which means that the result of calling `subscribe()` on an observable source is a `Subscription` object. As observables operate synchronously or asynchronously, the consumer of an observable must in some way support the inversion of control that also happens with callbacks. This is consistent with its push-based mechanism. That is, because we don't know when a DOM element, for instance, will fire an event or the result of an AJAX call will return, observables must be able to call into or signal the observer structure that more data is available by using the observer's `next()` method as illustrated in figure 2.13. This mechanism is directly inspired in the Iterator and Observer patterns. An iterator doesn't know (or really care) about the size of the data structure it's looping over, or if it will ever end, it only knows whether there's more data to process.

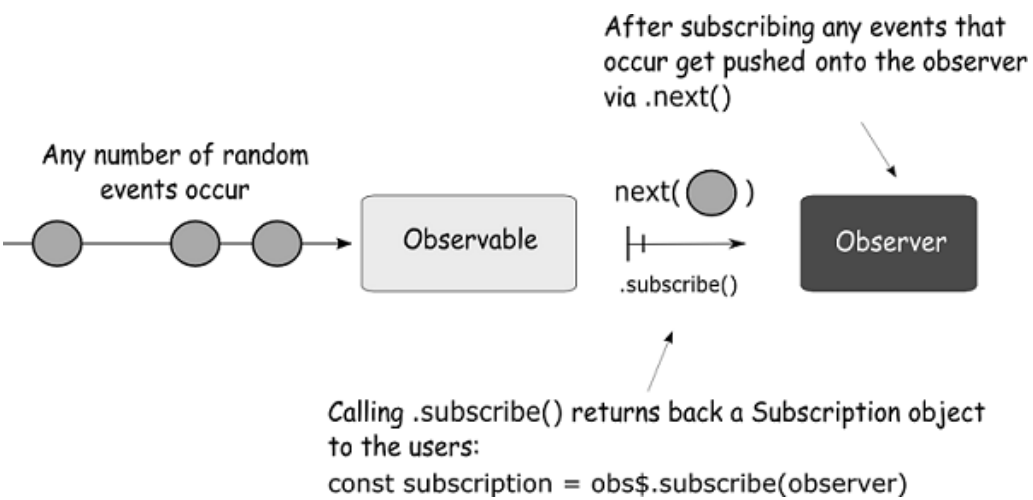

Figure 2.13 Shows Observables calling into an observer's methods. Observers expose a simple iterator-like API with a next() method. Upon subscription, an object of type Subscription is returned back to the user, which they can use for cancellation and disposal as we'll discuss in a bit.

Through a concise iterator-like API, observables are able to signal to its subscribers whether or not more events have occurred. This gives us the flexibility to control how what data observers receive.

2.4.1 The Observer API

An observer is registered with an observable in much the same way that we registered callbacks on an event emitter. An observable becomes aware of an observer during what is known as the subscription process, which you've seen a lot of so far. The subscription process is a way for you to pass an observer reference into an observable, creating a managed, one-way relationship.

Figure 2.14 shows how observables call an observer's methods to signal more data, completion, and even errors. As you can see, aside from `next()`, there 2 other methods called on observers: `error()`, and `complete()`.

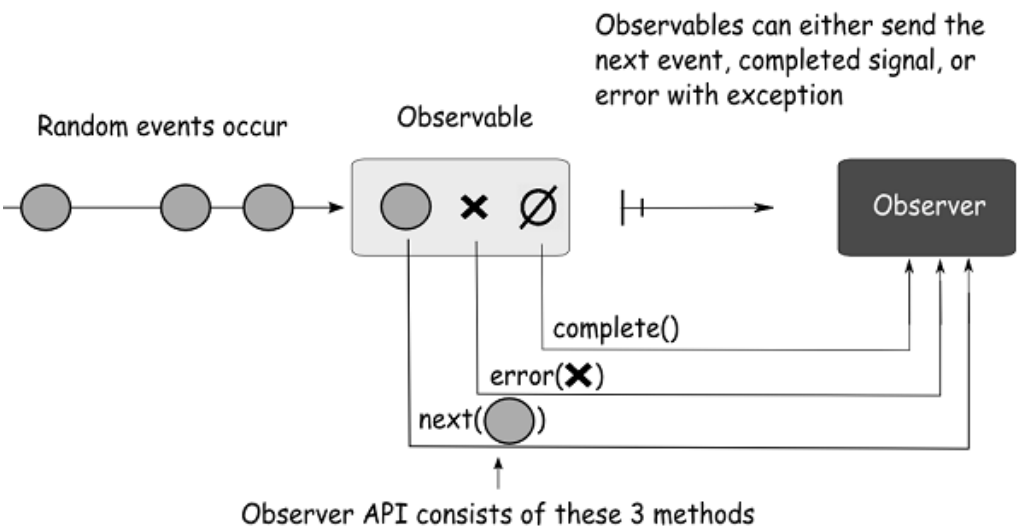

Figure 2.14 Observables call into the observer API to either send the next event in the stream, the completed flag a stream is flushed out, or any errors that occur during the pipeline operators. We will discuss more about error handling in later chapters.

Figure 2.14 shows that once the `subscribe` method is called, an observer is implicitly created with an API that exposes three (optional) methods: `next`, `complete`, and `error` (in RxJS 4 these were called `onNext`, `onCompleted`, and `onError`, respectively). In code the resulting object has the following structure:

```
var observer = {
    next: function () {
        // process next value
```

```
    },
    error: function () {
        // alert user
    },
    complete: function () {

    }
}
```

Up until now, we've only really used a single function call to process the results somehow, which maps to `next()`. Each method serves a specific purpose in the lifetime of the Observer as shown in table 2.1.

Table 2.1 Defining the Observer API

Name	Description
`next(val):void`	Receives next value from an upstream Observable. This is the equivalent of `update` in the Observer pattern. When a single function is passed into `subscribe()` instead of an observer object, it maps to the observer's `next()`.
`complete():void`	Receives a completion notification from the upstream Observable. Subsequent calls to `next` (if any) are ignored.
`error(exception):void`	Receives an error notification from the upstream Observable. This would indicate that it encountered an exception and will not be emitting any more messages to the Observer (subsequent calls to `next` are ignored). Generally, error objects are passed in, but you could customize this to pass other types as well.

Alternatively, you can use this API directly to by creating your own observable.

2.4.2 Creating bare observables

Most of the time, you will use the RxJS factory operators like `from()`, `of()`, as you learned at the beginning of this chapter, to instantiate observables. In practice, these should cover all of your needs. However, it's important to understand how observables work under all of the nice RxJS abstraction, and how they interact with the observer to emit events. So, we'll show you a very barebones model of an observable that emits events asynchronously and exposes the mechanism to unsubscribe. At the core, an observable is just a function that processes a set of inputs and returns a subscription back to the caller to manage the disposal of the stream:

```
const observable = events => {
  const INTERVAL = 1 * 1000;
  let schedulerId;

  return {
     subscribe: observer => {
       schedulerId = setInterval(() => {
         if(events.length === 0) {
            observer.complete();
```

```
            clearInterval(schedulerId);
            schedulerId = undefined;
         }
         else {
            observer.next(events.shift());
         }
       }, INTERVAL);

       return {
          unsubscribe: () => {
             if(schedulerId) {
                 clearInterval(schedulerId);
             }
          }
       };
     }
  }
};
```

You can call this function by passing the observer object:

```
let sub = observable([1, 2, 3, 4, 5, 6, 7, 8]).subscribe({
  next: console.log,
  complete: () => console.log('Done!')
});
```

Of course, this is very simplistic model of RxJS, and there's so much more that goes into it. But the main takeaway here is that an observable behaves like a function that begins chipping away at the data pushed into it as soon as a subscriber is available; the subscriber has the keys to turn the stream off via `sub.unsubscribe()`; Now, let's move on to using RxJS.

Using RxJS, I can register an observer object through `Rx.Observable.create()`. Like the code above, this function expects an observer object that you can use to signal the next emitted event by invoking its `next()` method. Most of the time, you'll provide the observer object literal directly into the subscription and use the static `create()` method when you want full control of how and when the data is emitted from the observable through the observer API. For instance, you create observables artificially by just calling into the observer's methods directly:

```
const source$ = Rx.Observable.create(observer => {
   observer.next('4111111111111111');
   observer.next('5105105105105100');
   observer.next('4342561111111118');
   observer.next('6500000000000002');
   observer.complete(); ❶
}); ❷

const subscription = source$.subscribe(console.log); ❷
```

❶ If an Observable is finite, you can signal its completion by calling the Observer's complete() method.
❶ At this point, the observable stands idle and none of the data is actually emitted or passed into the observer
❷ With subscribe(), the observer logic is executed; in this case, it's just printing to the console

A marble diagram of this stream would look like:

Figure 2.15 A marble diagram showing a synchronous set of events ended by a call to complete()

This sample code is very simple as it just emits a series of account numbers, but we can do much more. In fact, we could create our own observables with custom behavior that can be reused anywhere in our application.

2.4.3 Observable modules

Directly calling the observer object allows you to define the data that's pushed to the subscriber. How this data is generated and where it comes is all encapsulated into the observable's context—kind of like a module. For instance, suppose we wanted to create a simple progress indicator widget that can be used when a user is performing a long running operation. This module will emit percentage values 0 to 100% at a certain speed:

Listing 2.4 Custom progress indicator module using RxJS

```
const progressBar$ = Rx.Observable.create(observer => {
   const OFFSET = 3000;  (1)
   const SPEED =  50;    (2)

   let val = 0;
   function progress() {
     if(++val <= 100) {
       observer.next(val);          (2)
       setTimeout(progress, SPEED); (3)
     }
     else {
       observer.complete();         (4)
     }
   };
   setTimeout(progress, OFFSET);    (1)
});

const label = document.querySelector('#progress-indicator');

progressBar$
  .subscribe(
    val   => label.textContent = (Number.isInteger(val) ? val + "%" : val),
    error => console.log(error.message),
    ()    => label.textContent = 'Complete!'
);
```

(1) Start the progress indicator counter after 3 seconds
(2) Emit a new progress value every 50 milliseconds

3. Calling the progress function recursively
4. Send the complete signal after reaching 100%

The business logic of how the values are generated and emitted belongs in the observable, while all of the details of rendering, whether you want a simple number indicator or use some third-party progress bar widget, are for the caller to implement within the observer.

NOTE You could also achieve this by using RxJS' time operators. More about this in the next chapter.

Using these methods gives us more opportunities to react to the different states of the program. Stepping back into our discussion about iterators and generators in chapter 2, observers operate very similarly to these artifacts. The key difference is that the iterator uses a pull-based mechanism as opposed to an observable's push-based nature—an observable pushes values into an observer. For iterators and generators, the consuming code is controlling the pace of consumption. For instance, a `for` loop controls (or requests) what to pull from an iterator or a generator, not the other way around. This means that each time a new piece of data is needed (by a call to `next()` or `yield`), the consumer of the iterator will call the appropriate method to advance the state of the iterator. Here's an example again using the Fibonacci sequence:

```
for (let nums of new BufferIterator(arr, 2)) {
  console.log(nums);
}

for(let num of fibonacci()) {
  console.log(num);
}
```

The loop pulls the next element from the iterator by calling .next()

The loop pulls data from the generator function, requesting it to *yield* the next element.

Figure 2.16 Figure that shows the pull mechanism of iterators

As a result of this, iterators must have a way to inform the consumer that there are no longer any items for consumption. Bank tellers are real world iterators. Each time a customer comes up they must be handled first before the next customer can be helped. When the teller becomes available they yell "NEXT!" to "pull" the next customer in. If they were to call "NEXT!" with no one present they would know that the line was complete and it might be safe to take their lunch break.

Something to keep in mind, though, is that infinite event emitters, like the DOM, will never fire the `complete()` function (or `error()` for that matter) on any of its events. But for finite event sequences, when an observer is called with either of these methods, it knows that contractually it will not receive any more messages from its owning observable. This again is a tight parallel to an iterator, which by definition should stop returning values when the iteration generates an exception or completes.

Consider a simple promise object that resolves to the value 42 after 5 seconds (shown in figure 2.17):

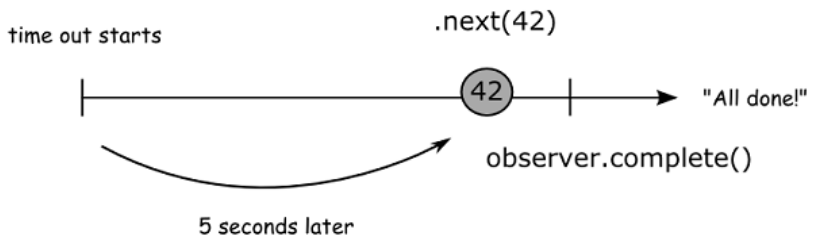

Figure 2.17 An observable (wrapped promise) that emits a value after 5 seconds

We mentioned in chapter 1, that promises can be used to model an immutable, single (future) value. We'll use the `setTimeout()` function to simulate this, now instead of creating our own observable, we'll use the generic creational methods in RxJS, such as

```
Rx.Observable.fromPromise():
const computeFutureValue = new Promise((resolve, reject) => {
   setTimeout(function () {
      resolve(42); ❶
   }, 5000);
});

Rx.Observable.fromPromise(computeFutureValue)
   .subscribe(
     function next(val) {
        console.log(val);
     },
     function error(err) {
        console.log(`Error occurred:  ${err}`);
     },
     function complete() {
        console.log('All done!');
     }
);

//-> 42
//-> "All done!"
```

❶ **Resolve the promise after 5 seconds have elapsed**

Because promises emit a single value, this stream will eventually send the completed status after 5 seconds have passed, printing "All done!" at the end. Now, suppose that instead of a resolved promise, something goes wrong in computing this value and the promise is rejected:

```
const computeFutureValue = new Promise((resolve, reject) => {
   setInterval(function () {
      reject(new Error('Unexpected Exception!'));
   }, 5000);
});
```

This will cause the observable to invoke the `error()` method on the observer and print the following message after 5 seconds:

```
"Error occurred: Unexpected Exception!"
```

This is quite remarkable because RxJS not only takes care of error handling for us (without messy, imperative try/catch statements), but also provides logic that ties in with promise semantics of resolve/reject. We will cover all there's to know about error handling in chapter 7.

An important take away from this discussion about observers is that the callbacks passed to it are, for all practical purposes, future code. That is, we don't know when the callbacks will actually be called, so other code should not make assumptions about their execution. This relates to the larger point made earlier about the nature of the code within a stream. Because one of our goals is to move away from the messy business of keeping track of state changes, avoiding the introduction of side effects is one of the ways that we can keep our streams "pure" and prevent unwanted changes from adversely seeping into the application logic. This works well with RxJS because pure functions can run in any order, and at any time (now or in the future), and will always yield the correct results.

With observers, we finish introducing the tree main parts of RxJS: producers (observables), the pipeline (business logic), and the consumers (observers). This chapter is just the start of learning how to think reactively (and functionally). It will take much more time and many more examples to truly understand how we can think reactively, but you were able to get your feet wet on some advanced APIs. Much of what you've seen so far has been abstract in nature with very little coding, but this step is crucial for understanding how this approach differs from ones we have been taught in the past. In the next chapter, we'll look more closely at the operations that we can perform on streams as well as how we can cancel them if needed. Doing so, we're officially taking the training wheels off and introducing you to the core operations for building applications in RxJS.

2.5 Summary

- Thinking in streams and the role functional programming plays in combination with RxJS.
- The declarative style of RxJS allows us to translate almost exactly from our problem statement into working code.
- Data sources can often operate quite differently even within the observable contract
- How mouse clicks, HTTP requests, or simple arrays are really all the same under eyes of observables.
- Understand the difference between push-based and pull-based.
- Wrapping data sources is the first step to creating a pipeline/Observable.
- Observables abstract the notion of production and consumption of events such that we can separate production, consumption and processing into completely self-contained constructs.
- Observers expose an API with three methods: `next()`, `complete()`, and `error()`.

3

Core operators

This chapter covers

- Introducing disposal of streams
- Exploring common RxJS operators
- Building fluent method chains with map, reduce, and filter
- Additional aggregate operators

In the first two chapters, you learned that RxJS draws lots of inspiration from functional programming and, of course, reactive programming. Both paradigms are oriented around data flows and the propagation of change through a chain of functions known as operators. Operators are pure functions that create a new observable based on the current one—the original is unchanged. In this chapter, you'll learn about some of the most widely used RxJS observable operators that you can use to create a pipeline that transforms a sequence of events into the output you desire.

A common theme in this chapter will be creating observables using a declarative style of coding, which originates from functional programming. You can lift sequences of data of any size into an observable context, as well as data generated or emitted over time, with the goal of creating a unified programming model for any type of data source, static or dynamic. Before we dive into the operators used to apply transformations onto the data that flows through an observable sequence, it's important to understand that unlike many AJAX libraries, observables can be cancelled.

3.1 Evaluating and cancelling streams

Imagine making a long running AJAX call requesting lots of data from the server. However, shortly after spawning this call, the user navigates from away the page by clicking on some other button. What happens to the original AJAX request? Consider another example. You

begin a client-side interval to poll for certain data to become available, but an exception occurs and the data never becomes available? Should these processes be allowed to run wild and take up system resources? We're guessing: No.

A stream, as it exists in RxJS, is an object with a deterministic lifespan defined almost entirely by you, the programmer. JavaScript, unlike some other languages, has very few real distinct types, most of those types mirroring the simplicity of JSON. Additionally, there is little support within JavaScript for memory management as this has historically been left to browser manufacturers to worry about. While both of these features make JavaScript a marvelously simple language to learn and use, for the most part, they also obscure a bit what's really happening under your application's plumbing.

In languages like C and C++ there exists an extremely fine-grained approach to not only control the specific data structure you use, but also its exact lifetime in memory—you have complete control of allocating and deallocating objects in memory. On the other hand, in JavaScript the lifetime of objects is controlled by the garbage collector, and rightfully so. The garbage collector is a process that is operated by the runtime engine that's running your application. It will periodically run and free up memory that is associated with any unused references. The garbage collector does so by keeping track of the references that are kept between various objects in the application—this is known as *ref counting*. When it detects that an object is no longer referenced, it becomes a candidate for disposal. A failure to find references that are no longer in use results in a memory leak. Memory leaks are generally an indicator of either sloppy design or reference tracking and can result in a run-away system footprint that results in either the user or the system killing your application, since it becomes unresponsive at that point.

> **CAUTION** This notion of "automatic" garbage collection gives us JavaScript developers a false impression that we should not care about memory management. This is a mistake when attempting to write our own event handling code. RxJS frees us from this by implementing a mechanism to unsubscribe or effectively clean up attached listeners from any event emitters such as the DOM.

In older browser implementations this used to be a big problem, particularly with Internet Explorer's event handling system. Modern browsers are now much more efficient at this and libraries such as RxJS are tuned to avoid many of these problems.

3.1.1 Downside of eager allocation

An important point to remember when dealing with RxJS is that the lifetime of a stream does not start with the creation of an observable. It begins when it is subscribed to. Hence, there's barely no overhead in creating and initializing one, as it begins in a dormant state and does not generate or emit events without an observer subscribed to it. It's analogous to the old adage: if an observable is created your application, and no one subscribes to it, does it emit an event?

In computing terms, an object that creates data only when it is needed is known as a *lazy* data source. This is in sharp contrast to JavaScript, which has strict *eager* evaluation. The terms lazy and eager refer to when an application requests memory from the system and how much it requests upfront. Lazy allocation is always done when the space is actually needed (or on-demand), while eager allocation is performed upfront as soon as the object is scoped. In the eager scheme, there's an upfront cost to allocation and there exist the possibility that you'll over allocate because you don't know how much space will actually be used. Lazy allocation, on the other hand, will wait until the space is needed and pay the penalty for allocation at runtime. This allows frameworks to be really smart and avoid over-allocating space in certain situations. To illustrate the difference, we'll show you how a popular JavaScript array method, `slice()`, would work under eager and lazy evaluation. Consider a function called `range(start, end)`, which generates an array of numbers from start to end. Generating an infinite amount of elements, and taking the first 5 would look like:

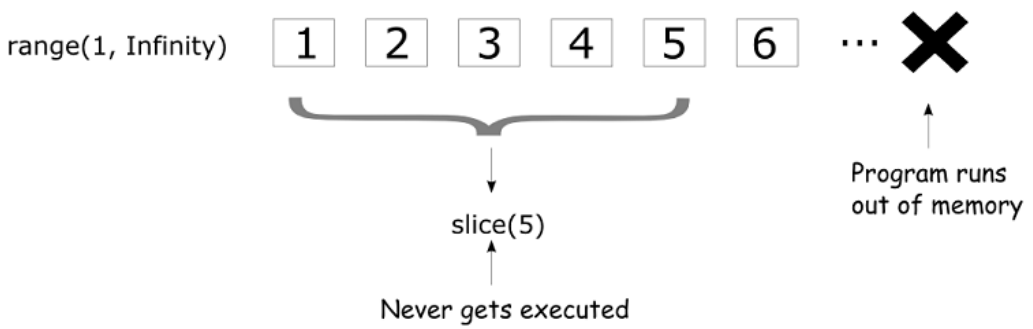

Figure 3.1 In an eager allocation scheme, the program halts before it can execute the slice(5) function because it runs out of memory

In code:

```
range(1, Number.POSITIVE_INFINITY).slice(0, 5); //-> Browser halts
```

With JavaScript's eager evaluation this code will never get past the `range()` function, since it needs to generate numbers infinitely (or until you run out of memory and crash). In other words, eager evaluation means execute each portion of an expression fully before moving on to the next. On the other hand, if JavaScript functions were lazy, then this code would only need to generate the first 5 elements:

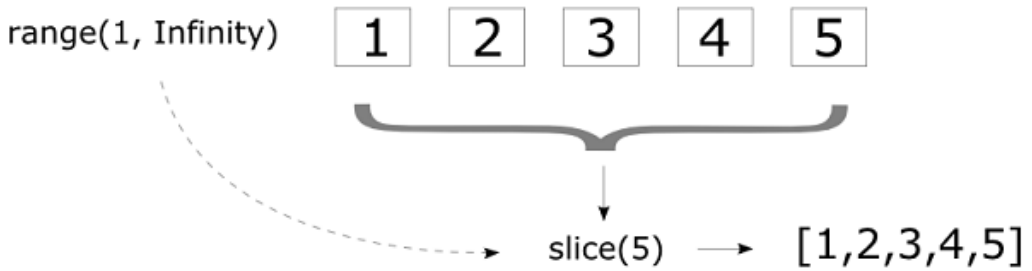

Figure 3.2 In a lazy allocation scheme, the runtime waits until the result of this expression is needed, I only then run through the program allocating only the resources it needs.

In this case, the entire evaluation of the expression waits until the result of the expression is needed, and only then evaluate. In RxJS, the strategy is precisely this to wait until a subscriber subscribes to the observable expression and then begin initialize any required data structures. You'll see later on that using lazy evaluation will allow it to perform internal data structure optimizations and reuse.

3.1.2 Lazy allocation and subscribing to observables

RxJS avoids premature allocation of data in two ways. The first, as we mentioned, is the use of a lazy subscription mechanism. The second is that an observable pushes data as soon as the event is emitted instead of holding it statically in memory. In chapter 4, we'll discuss the buffering operators, which can be used to transiently store data for either a period of time or as reacting to a certain condition, if you wished to do so. But by default, the data is emitted down as soon as it's received.

A lazy subscription means that an observable will remain in a dormant state until it is activated by an event that it finds interesting. Consider this example that again generates an infinite number of events separated by half a second:

```
 const source$ = Rx.Observable.create(observer => {
let i = 0;
setInterval(() => {
  observer.next(i++); ❶
}, 500);
 });
```

❶ This interval will keep on emitting events every 500 milliseconds until something stops it

The cost of allocating an observable instance is fixed, unlike an array which is a dynamic object with potential unbounded growth. It must be this way, otherwise, it would be impossible to store each and every user's clicks key presses, or mouse moves. To activate `source$`, an observer must first subscribe to it via `subscribe()` (alternatively `forEach()`). A call to either of these will take the observable out of its dormant state and inform it that it can begin producing values—in this case, starting the allocation for events 1, 2, 3, 4, 5, and so on

every half a second. Since an observable is really an abstraction over many different data sources, the effect of this will vary from source to source.

The second advantage to a lazy subscription is that the observable does not hold onto data by default. In the previous example, each event generated by the interval will be processed and then dropped. This is what we mean when we say that the observable is streaming in nature rather than pooled. This *discard-by-default* semantic mean that you never have to worry about unbounded memory growth sneaking up on you causing memory leaks. When writing native event-driven JavaScript code, especially in older browsers, memory leaks can occur if we neglect event management and disposal.

3.1.3 Disposing of subscriptions: explicit cancellation

Just as important as when memory is allocated, is when it is deallocated or released back to the application. For example, rich JavaScript UIs bind event handlers potentially to thousands of elements. After the user has finished interacting with a certain part of the UI, there's no reason for those objects needing to exist and take up memory any longer. As discussed earlier, the garbage collector is fairly smart about how it cleans up memory. Unfortunately, it is only able to do so if those references are actually found to be unused, or if there are no reference cycles formed, which tends to occur frequently in native event handling code.

It's very easy to initialize objects and then forget about them without ever removing references to them, which prevents the application from ever recovering that memory back (this might not be a concern with small scripts, but can easily become an issue with modern client-heavy applications). For example, we can see the problem better in this simple code where we listen for right-click events on a menu item, perhaps to show a custom context menu:

```
document.addEventListener('mouseup', e => {
  if (e.button === 2)
    showCustomContextMenu();
  e.stopPropagation();
});
```

Many developers may not even recognize the problem with the code above. The problem arises from the fact that in order to *unsubscribe* from this event we need the reference to the function that was passed into the event handler (the inlined lambda expression). Because we're trying to use this idiom as much as possible, we ended up creating a handler that we cannot unsubscribe from, that is if we even remembered to unsubscribe from it at all. To make matters worse, if we had nested event handlers or subscribed to other events from within this one, we create yet another level of complexity and potential for more memory to leak.

For older web applications (the web 1.0 years), memory de-allocation was not so much of a problem because navigation between pages forced a page reload which in turn cleared out JavaScript's runtime footprint. Today, as single page applications grow in popularity (the web 2.0+ era) and clients become more modern and richer, memory pressure becomes a real

threat; objects can now conceivably exist for the duration of the entire application's lifespan loaded into the browser.

This is why we need sophisticated libraries like RxJS. In RxJS, the producer is the one responsible for unsubscribing. Managing a subscription is handled through an object of type `Subscription` (also known as a `Disposable` in RxJS 4) returned from a call to `subscribe()` or `forEach()`, which implements the mechanism to dispose of the source stream. If we have finished with the observable and no longer wish to receive events from it, we can call `unsubscribe()` to tear it down; this is known as *explicit cancellation*. Here's a short example

```
   const mouseClicks = Rx.Observable.fromEvent(document, 'mouseup');
   const subscription = mouseClicks.subscribe(someMouseClickObserver);

  ... moments later

  subscription.unsubscribe(); (1)
```

(1) **Tear down the stream and free up any allocated objects**

This tearing down process will stop further events from going to any registered observers and will immediately release all resources allocated by the observable. The subscription instance handles the entire unsubscription process and it's able to do this because every observable also defines how it will be disposed.

Recall that in chapter 2 we introduced the `Rx.Observable.create()` method, which could be used to create arbitrary observables. The final step in creating it was that the `subscribe` would need to know how to dispose of it, and that's our responsibility to implement. Going back to our progress indicator code, we'll add the unsubscription mechanism at the end:

Listing 3.1 Disposing of an observable

```
const progressBar$ = Rx.Observable.create(observer => {
   const OFFSET = 3000;
   const SPEED =  50;

   let val = 0;
   let timeoutId = 0;
   function progress() {
     if(++val <= 100) {
       observer.next(val);
       timeoutId = setTimeout(progress, SPEED);
     }
     else {
       observer.complete();
     }
   };
   timeoutId  = setTimeout(progress, OFFSET);

   return () => { (1)
      clearTimeout(timeoutId);
    };
});
```

❶ Function that executes when the unsubscribe method is called. Describes how to cancel that timeout upon disposal

The function added at the of our observable body becomes the body of the `unsubscribe()` method of the returned subscription object. In essence, each observable provides the keys to its own destruction during its creation. Every time a subscription occurs from an observer, it passes back a way to clean itself up (analogous to the `finally` clause after a try/catch). As every observable provides this self-contained, self-destruct button, we can also compose their subscriptions such that we can always tear them down correctly no matter how complex the underlying observable is.

> **CUSTOM OBSERVABLES** If you're creating a custom observable with `create()`, and it happens to emulate an infinite interval stream, you're responsible for supplying the proper unsubscribe behavior, or it will run indefinitely and cause memory to leak.

The examples we've shown so far for the most part involve setting timed intervals to generate events, which support cancelation through `clearInterval()`; but what happens to data sources that don't support cancellation? Let's jump into that next.

3.1.4 Cancellation mismatch between RxJS and other APIs

RxJS pbservables provide a straightforward mechanism for cancelling and disposing of the event streams. However, this simplicity can be deceiving when used in conjunction with other JavaScript APIs. For example, you might encouter problems when trying to cancel observables that wrap promises; let's take a look at listing 3.2:

Listing 3.2 Disposing of a promise

```
const promise = new Promise((resolve, reject) => {    ❶
    setTimeout(() => {
       resolve(42);
    }, 10000);
});
promise.then(val => {
   console.log(`In then(): ${val}`);       ❷
});
const subscription$ = Rx.Observable.fromPromise(promise).subscribe(val => {
   console.log(`In subscribe(): ${val}`); ❸
});
subscription$.unsubscribe(); ❹
```

❶ Create a promise that resolves to 42 after 10 seconds
❷ Handle the resolved promise value
❸ Wrap an Observable around the Promise API
❹ Attempt to dispose of the Observable

As you can see from listing 3.2, we disposed of the observable thinking it would also take care of the underlying promise. The observable object itself was properly disposed of; surprisingly,

although we attempted to explicitly cancel the event as well, after 10 seconds this program emits the following (apparently JavaScript promises can't be broken after all):

```
"In then(): 42"
```

So what happened? This process is explained in figure 3.8:

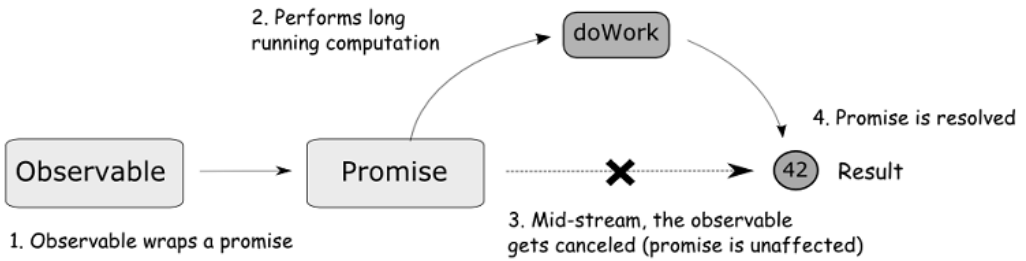

Figure 3.3 The cancellation of an observable does not affect the underlying promise.

What happens is that *promises were not design to be cancelled*. Once a promise object begins executing (gets into a pending status), it tries to become fulfilled by either resolving or rejecting the underlying result, as the case may be.

RxJS makes it easy to integrate with external APIs; however, you must be mindful that there's a mismatch of design philosophies between an API designed to emit a single value (Promise) and one that supports infinite values (Observable). This is one use case, but it could also happen if you integrate with other APIs that aren't RxJS-aware. Most of the time, though, we don't have to worry about cancelling subscriptions ourselves because lots of RxJS operators do this for us.

Now that we've covered creating and cancelling streams, in this next section, we'll begin with the more popular operators that are essential to any RxJS program.

3.2 Popular RxJS observable operators

Though the subscription and disposal semantics of RxJS are useful in managing resources to avoid leaky event handlers, they are only part of the story. However, keep in mind what thinking reactively is all about; instead of you controlling what goes on a stream by creating a custom observable and pushing events through `observer.next()`, it's preferable to relinquish that control and react when the time comes—you want to always be reactive! This means allowing RxJS factory operators (`of()`, `from()`, and others) to wrap an event source of interest and create the observable sequence with which to apply the business logic you desire. Hence, being reactive involves defining what a program *will do* when a value is pushed some time in the future.

This is where RxJS shines, and it's due to its fully loaded arsenal of out-of-the-box operators, which you can use to create expressive streams of logical data flows. You can

create flows to solve virtually any problem from creating responsive web foms, drag and drop, and even games.

An operator is a small piece of declarative functionality that allows us to inject logic into an observable's pipeline. An operator *is a pure*, *higher-order function* as well, which means it never changes the observable object is operating under (called *the source*), but rather returns a new observable that continues the chain. It is at this point where functional programming best practices come into play since the functions comprising your business logic, the building blocks of your solution, should be done using pure functions as much as practically possible. These operators can be used to inspect, alter, create, or delay events after they leave the data source but before they reach the consumer; in other words, anything that comprises your business logic pipeline is handled by the combination of one or more operators, which drive the execution of the pure functions of your program. And if that's not enough, to put the icing on the cake, RxJS operators are also lazily evaluated!

In chapter 1 we highlighted four fundamental types of computing tasks that we can perform. We split them into two dimensions depending on whether they performed work synchronously or asynchronously, and whether they acted on single values or collections. Manipulating a single value is a relatively trivial task (known as a singleton stream), given that we can inspect its properties and manipulate it directly. In most cases, though, we want streams to act across a range of values rather than just one and done. The computing model behind RxJS encourages you to work with function chains that process data, similar to a conveyor belt in an assembly line as shown in figure 3.4:

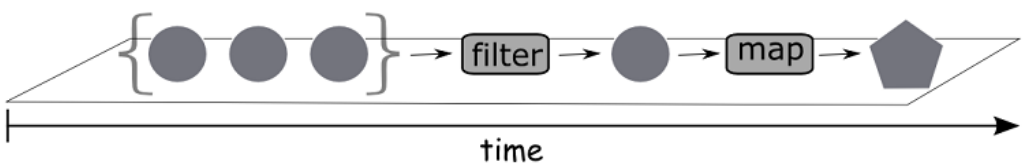

Figure 3.4 An assembly line where operators represent individual stations and each has their own task to perform on each data that passes by them.

Another important design principle of RxJS is to provide a familiar computing model to what you're accustomed to. Inspired in the Array#extras APIs introduced in ES5, RxJS features its own version of core operators such as: `map`, `filter`, and `reduce`. Because these are some of the more frequently used, let's start with these.

3.2.1 Introducing the core operators

Operators come in two variaties: as instance or as static methods of the observable type. Part of the RxJS 5 rework was the drastic simplication of the API surface, which consisted of a sheer reduction of operators as well as a simplification of their usage. Hence, most of the operators in RxJS 5 can be invoked as static or as instance methods (when we say instance, we refer to invoking them using the dot "." notation on an observable instance).

RxJS comes built in with many operators that handle many common tasks such as working with collections, extracting elements from the stream, manipulating and transforming the data, handling errors, and others. In this section we'll focus on the three that you'll use about 80% of the time: `map`, `filter`, `reduce`, as well as a variation of `reduce` called `scan`.

MAPPING OPERATIONS ON OBSERVABLES

By far the most common operator that you will likely come across when dealing with RxJS is `map()`. RxJS is not the only library to implement it, and they all follow the same functional prorgramming principles. In FP, `map()` belongs to a category of operations called *trasnformational* because it changes the nature of data running through the observable by applying a function; therefore, it is a single output value or a one-to-one transformation. In symbolic notation we write it as `map :: x -> f(x)`, where for a given value `x` we can associate an input of `x` with an output of `f(x)`. Consider a quick example that applies a given percetange value onto a set of prices:

```
const sixPercent = x => x + (x * .06);

Rx.Observable.of(10.0, 20.0, 30.0, 40.0)
  .map(sixPercent)  ❶
  .subscribe(console.log); //-> 10.6, 21.2, 31.8, 42.4
```

❶ **Applies this function onto each value of the source Observable**

Mapping functions is a fundamental process when transforming data from one type to another. For example, say you had a list of user IDs that you wanted to fetch GitHub information for. Mapping a function like `ajax()` over the set of IDs yields an array of JSON account objects.

In RxJS, we want to map functions across all the elements emitted from an observable. To help better visualize operators, we'll make use of the marble diagrams. Recall that arrows and symbolic characters representing the various operations that convert the input stream into the output stream, as shown in figure 3.5:

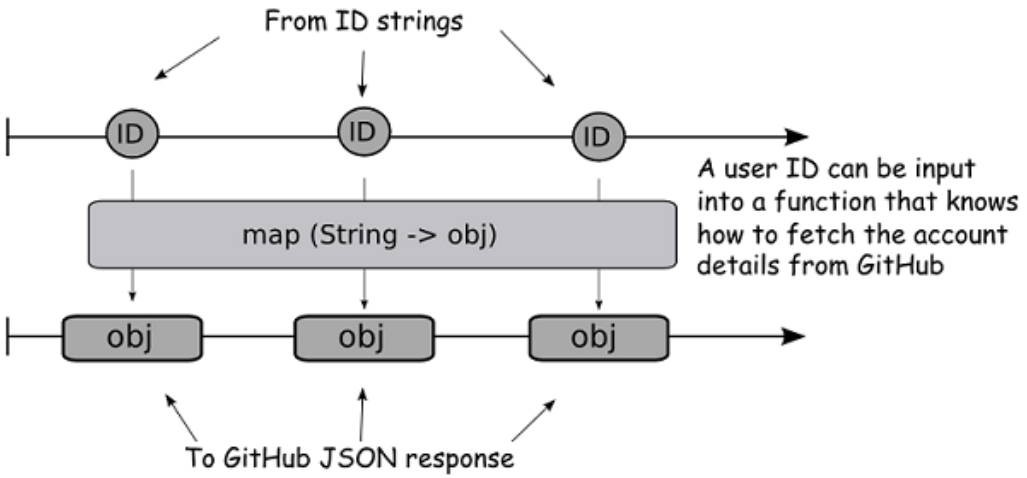

Figure 3.5 The map operator will produce a 1-to-1 transformation which will convert an input value into an output value by a given process. In this case, map takes a URL string and converts it into an array of users by means of the mapping function. Operators are encoded inside a box that illustrates the function that is passed in.

Using these vertical transformations like in figure 3.5 is how we'll depict operations that take one form of data and convert it to another. By design, this function in RxJS has the exact same signature as that of Array[7]:

```
Array.prototype.map         :: a => b; for all a in Array<a>
Rx.Observable.prototype.map :: a => b; for all a in Observable<a>
```

As with arrays, observable's `map()` is also immutable, which means it won't actually change the original, but instead transform the value passed through it. Also, as you can see in figure 3.5 the size of the output will always be the same as the input because mapping is a one-to-one relationship that preserves structure. What exactly the transform function is, is left up to you to decide depending on your business logic, `map()` simply guarantees that it will be called on every value passing through the stream as it's propagated downstream to the next operator in the chain.

We mentioned briefly before that all of the RxJS operators are pure. To show you what this means, we'll show the case of `map()`. Transforming a String into an ID, looks like this:

[7] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

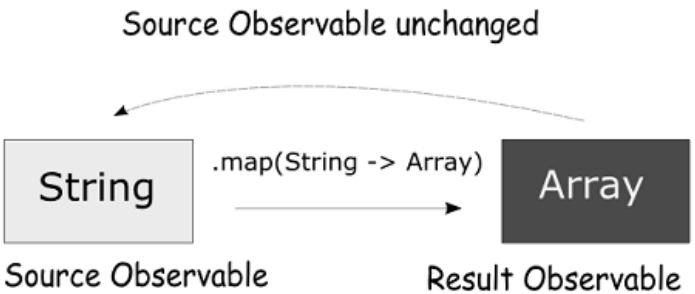

Figure 3.6 Mapping a function from String to Array to a source observable creates a new Observable with the result of the function.

Now let's show the code for this. Suppose we need to convert a collection of strings into a corresponding comma-separated value (CSV) array. Listing 3.3 shows a simple stream that will accomplish this:

Listing 3.3 Mapping functions over streams

```
Rx.Observable.from([
   'The quick brown fox',
   'jumps over the lazy dog'
   ])
 .map(str => str.split(' ')) ❶
 .do(arr => console.log(arr.length)) ❷
 .subscribe(console.log);
```

❶ Map a set of function to extract the value from the event
❷ RxJS .do() is a utility operator. Very useful for effectful actions such as logging to the screen. This can be very handy for debugging or tracing the values flowing through a stream

For those familiar with design patterns, mapping functions is analogous to the *Adapter* pattern (as shown in the famous "Gang of Four" book titled *Design Patterns: Elements of Reusable Object-Oriented Software*, 1994). In the adapter pattern, we have an object that interacts with two otherwise incompatible interfaces and allows information to flow between them, that is, adapting them to each other. In a similar fashion, we also use `map()` to create a type compatibility between the producer and the consumer of data. In other words, the purpose of using it is to convert the raw input data into something the consumers can understand. So, in this way, the adaptation is done from the producer to the consumer, in that direction.

But sometimes, there can be too much data to process and we may not be interested in all of it. For this, there's an operator to discard unwanted events.

FILTERING OUT UNWANTED EVENTS

Filtering is the process of removing unwanted items from a stream. The criteria to remove these elements is passed in as selector function, also called the *predicate*. Here's a simple

example of how this operator works. Say you need you need to place restrictions on input boxes for numerical quantities. It's probably a good idea to place a business rule over textboxes rejecting any non-numerical input. Whenever we're thinking about rejecting, removing, narrowing, or selecting, we can do that easily using a *filtering* operator called `filter()` and inspecting the `keyCode` property of the keystroke:

Listing 3.4 Filtering events from a stream

```
const isNumericalKeyCode = code => code >= 48 && code <= 57;
const input = document.querySelector('#input');
Rx.Observable.fromEvent(input, 'keyup')
  .pluck('keyCode')           (1)
  .filter(isNumericalKeyCode) (2)
  .subscribe(code => console.log(`User typed:
        ${String.fromCharCode(code)}`));
```

(1) Extract this property from the object passing through the observable
(2) Accept only keys in the numerical range

Also, you can use it to ignore unwanted mouse clicks, touch events, and others. It could be that you are only interested in data that meets certain criteria, or you only need a certain subset of the data. In some cases, allowing too much data through can have an adverse effect on the performance of your application. Think about building an API for users to access all their account history for the month; if on every request you simply dumped their entire account history you would quickly find both your API and your clients overwhelmed. To make matters worse, your application won't scale to the size of data being processed. Filtering could be used to generate different views if the user only wanted to debits, credits, or transactions after a certain month.

An easy way to think about filtering is to consider the job interview process (every developer's favorite activity). When recruiting people for a specific job one of the first things one would look for is the candidate's previous experience in order to determine if they have the right skillset for the position. If the job requires programming, then we would expect that the candidates should have some sort of programming background listed on their resume. Otherwise, we could exclude them from our final interview list.

Suppose, we modeled our applicant screening process as an array, then we could write our filtering operation using the Array's `filter()`[8] method. Because Observables implement the exact same filtering semantics, you're already familiar with using a predicate function (also called a *discriminant*) in `filter()` that returns `true` for candidates that will be considered for the interview process to move on to the next round. Here's the dataset we'll use:

```
let candidates = [
      {name: 'Brendan Eich', experience : 'JavaScript Guru'},
```

[8] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

```
      {name: 'Emmet Brown', experience: 'Historian'},
      {name: 'George Lucas', experience: 'Sci-fi writer'},
      {name: 'Alberto Perez', experience: 'Zumba Instructor'},
      {name: 'Bjarne Stroustrup', experience: 'C++ Developer'}
];
```

Whether this data arrives as a result of an AJAX call or a DOM event, the observable treats it all the same way. So, for now we'll stick with a simple array. In this case, we can wrap the data with an Observable and keep only the candidates that will be considered for this JavaScript job:

```
const hasJsExperience = bg => bg.toLowerCase().includes('javascript');

const candidates$ = Rx.Observable.from(candidates);
candidates$
   .filter(candidate => hasJsExperience(candidate.experience)) ❶
   .forEach(console.log); //-> prints "Brendan Eich"
```

Here's what's happening behind the scenes shown in figure 3.7. Like `map()`, `filter()` works vertically removing values from the resulting stream:

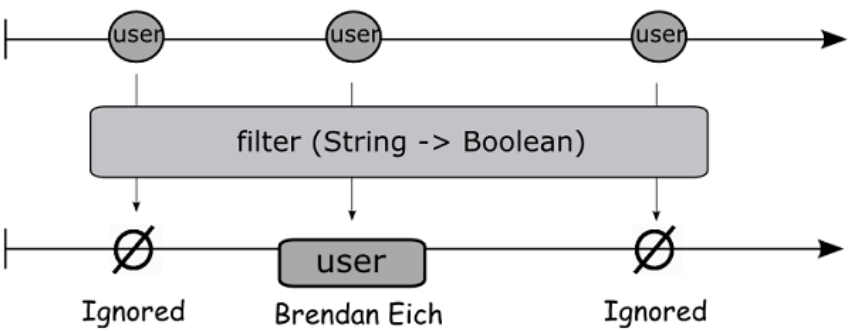

Figure 3.7 The filter operator is used to discard candidates that do not have any JavaScript experience

Functions `map()` and `filter()` are very similar in that they take a single function as its parameter. However, where the function passed to `map` converted the input value into an output value, the `filter()` function is used merely as a criterion that decides whether to keep the event in the stream or not. As you know, JavaScript being loosely typed will accept any "truthy" value as a pass, while any "falsey" values will cause it to reject the event.

Truthy vs. Falsey

In Javascript truthy is any value which can be coerced to a true boolean value. This includes objects, arrays, non-zero numbers, non-empty strings and of course the true boolean value. Meanwhile, falsey would be represented by 0, null, undefined, or false. In practice, though Javascript will accept all these types without question, it is often best for clarity's sake to simply return a boolean value.

Map and filter work very well together in scenarios where you don't want to apply a mapping function to each element, but to only the subset you care about. But filter is not the only function married to map, let's not forget about the powerful map/reduce combinations.

AGGREGATING RESULTS WITH REDUCE

Sometimes we aren't interested in acting on each item in a collection in isolation, sometimes we want to look at the collection in aggregate rather than piecemeal. For instance, suppose we want to take the average value of a collection of numbers or we want to turn a sequence into a mathematical series. We refer to this type of operation as a reduction or an aggregation with the result as a single value output instead of another collection. Once again, Arrays come with a built in `reduce` operator for this purpose[9] and observables follow suit. Reduce is a little bit more involved than the other two, here's the function signature:

```
Rx.Observable.reduce(accumulatorFunction, [initialValue]);
```

The accumulator function is called on every element and it's given the current running total and the new value as parameters. The initial value (optional) is used to begin the accumulation process; we're using 0 to begin the addition. Here's a simple example to illustrate how `reduce()` works. Suppose we wanted to compute the user's spending for the month by totaling all of his transactions. For this example, these transaction objects have a property called `amount`:

Listing 3.5 Using reduce to compute spending

```
const add = (x, y) => x + y;

Rx.Observable.from([
    {
       date: '2016-07-01',
       amount: -320.00,
    },
    {
       date: '2016-07-13',
       amount: 1000.00,
    },
    {
       date: '2016-07-22',
       amount: 45.0,
    },
  ])
  .pluck('amount') ❶
  .reduce(add, 0)  ❷
  .subscribe(console.log);
```

❶ Extract the 'amount' property

[9] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce

❷ Reduce the set of amount values with an add function

It's important to notice that `reduce()` with Observables works a bit different than `map()` and `filter()`. With arrays, `reduce()` does not return another array; instead it produces a single raw value which is the result of the reduction. The observable's `reduce()`, on the other hand, continues the previous pattern of returning a new singleton observable. This distinction will become important in section 3.3 when we talk more about operator chaining. Here's a visual representation of the previous code. Reduction is an operation that moves horizontally through the stream:

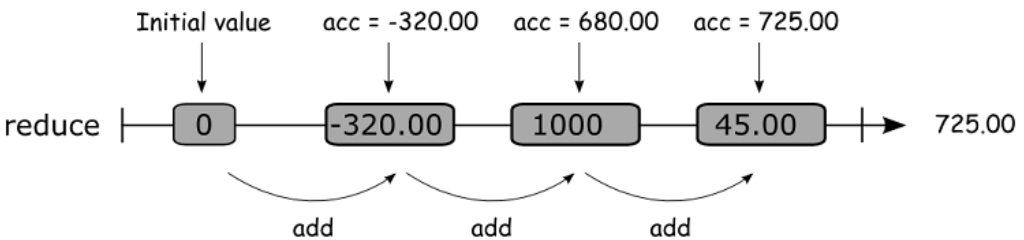

Figure 3.8 Shows the reduce operator moving horizontally accumulating every value through the stream using the add function

Suppose we needed to traverse through the candidate stream and group all the candidates with technical background (i.e. with knowledge of C++ or JavaScript)

```
Rx.Observable.from(candidates)
  .filter(candidate => {  ❶
    const bg = candidate.experience.toLowerCase();
    return bg.includes('javascript') || bg.includes('c++');
  })
  .reduce((acc, obj) => {
      acc.push(obj.name); ❷
      return acc;
    }, [])  ❸
  .forEach(console.log); //-> ["Brendan Eich", "Bjarne Stroustrup"]
```

❶ Filter all candidates that have no knowledge of a programming language
❷ Add candidate name to array
❸ Begin with an empty array (called the *seed*)

As you can see, reduce applies an accumulator function over the observable sequence initialized with a first seed value, which will be used to begin the aggregation process. Because `reduce()` returns a single value, there's a need for partial accumulation as well. We'll look at a variation of `reduce()`, called `scan()`.

SCANNING AGGREGATE DATA

RxJS uses `scan()` to apply an accumulator function over an observable sequence (just like `reduce()`), but returns each intermediate result as the accumulation process is happening and not all at once. This is very useful to obtain progress information about how data is being aggregated with each event.

Changing the previous code to use `scan()` as a direct swap-in replacement of `reduce()` reveals the intermediate steps of the accumulation:

```
Rx.Observable.from(candidates)
  .filter(candidate => {
    const bg = candidate.experience.toLowerCase();
    return bg.includes('javascript') || bg.includes('c++');
  })
  .scan((acc, obj) => { ❶
      acc.push(obj.name);
      return acc;
    }, [])
  .subscribe(console.log);
//-> ["Brendan Eich"]   ❷
     ["Brendan Eich", "Bjarne Stroustrup"]  ❷
```

❶ **Scan can be used as a direct replacement of reduce. In RxJS 4, you would have had to change the seed parameter to be first one. This was fixed in RxJS 5, now having the same signature as reduce**

❷ **As soon as it finds the first event, it emits it and accumulates it. A second emission happens when the second event is found returning the current state of the accumulation**

Aside from `scan()`, the symmetries between arrays and observables are no coincidence. This signature was chosen specifically because it is so simple and because it is one that many JavaScript developers are already familiar with. It is here though that the similarities end. Remember that these methods by themselves don't actually cause any work to run on the stream (only an subscriber can); instead, when an operator is called on an observable it is simply configuring the Observable for future values. Recall our definition of a stream as a specification of a dynamic value. This is a key distinction between the operators that we will see with arrays and those with observables, and you'll learn in later chapters that arrays represent work happening now while observables represent work in the future.

3.3 Sequencing operator pipelines with aggregates

One principle of functional programming is the ability to construct lazy function chains. In this section, we'll show you how to mix and match the main observable operators you just learned about together with a few other functions known as *aggregates*. Aggregate functions let you do really useful things like keeping track of a running total, taking only a subset of the total set of data, returning default values, and others. Some functional libraries you might have heard of or used before such as Lodash.js and Underscore.js have ample support for this. First, it's important to understand that observable sequences must be self-contained.

3.3.1 Self-contained pipelines and referential transparency

Function chains utilize JavaScript's power of higher-order functions to act as the single providers of the business logic. You saw examples of this before such as the filter function taking a predicate parameter. Also, observable pipelines should be self-contained, which essentially means they are side effect free (keep in mind that if your business logic functions are pure, your entire program is pure and stable as well). A pure pipeline does not allow any references to leak out of the observable's context. Once an event is lifted into the context, it's contained and transformed through a sequence of operators. Earlier, we showed that it is possible to group operations together to create more expressive logic. In RxJS we call this process operator chaining or *Fluent Programming*. The analogy of a self-contained pipeline works great as a visualization aid, shown in figure 3.9:

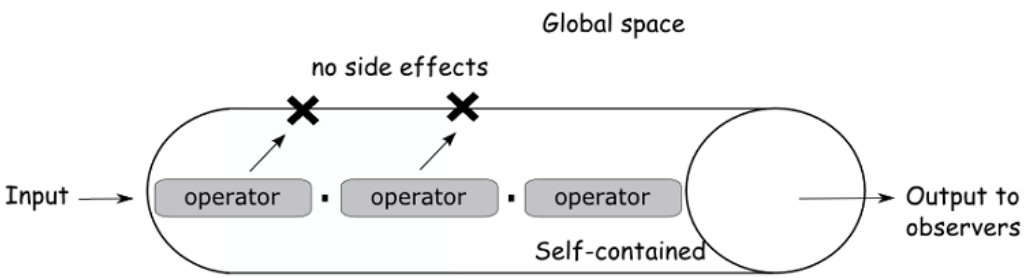

Figure 3.9 A self-contained pipeline is one where all of its operations are side effect free and work strictly on the data coming from previous operators. Operators might be any of: map, filter, reduce, and others you'll learn about in this book.

Consider this example:

```
let sinceLast = new Date();

Rx.Observable.fromEvent(document, 'mouseup')
     .filter(e => {
         let timeElapsed = new Date() – sinceLast; ❶
         sinceLast = new Date(); ❶
         return timeElapsed < 200;
}).subscribe();
```

❶ **Careless side effects of reading and writing to an external variable**

The code above is an example of poorly designed scope management in which a state variable `sinceLast` is allowed to live outside of the observable's context. The result of this is that the observable is no longer stateless and the lifecycle of the state and the Observable are now dependent.

It's important to understand that when you create an observable you're, in fact, creating an ecosystem or a *bounded context*. That ecosystem is a closed loop that begins with a subscription and ends with a disposal. If we were to look at the observable through a

functional programming lens, the internals of that observable should remain completely stateless and walled-off some the rest of the application. The scope of the callbacks that are passed into the operators should remain small and local. Mixing code that has external side effects not only introduces difficult to track complexity, but also removes one of the key advantages to using observables which is its well-defined lifespan—creation and disposal should leave the system in the exact same state it found it in.

What is a bounded context?

A bounded context is a design principle originating from Domain-Driven Design, that states that entities pertaining to a single domain model should be highly cohesive, and only expose the necessary interface to interact with other contexts. We can extend this definition to the Observable type as a form of context that hides the nature of the data that's pushed through it, allowing you to trasnform it by a ubiquitous language made up from the limited set of operators being exposed, and independently of what happens in the ouside world.

At a glance, a single subscription to this observable will function correctly, assuming that no other code manipulates `sinceLast`. However, if this observable is subscribed to a second time, the result is no longer the same. Observables must always produce the same results given the same events passing through it (i.e. pressing the same key combination should always yield the same data to the observers), a quality known in functional programming as *referentially transparent*.

Each invocation of `subscribe()` does more than simply start an event emitter. It spins off a brand new pipeline that will be independent of any other pipelines that were created by subsequent calls to `subscribe()`. This behavior is intentional in order to minimize side-effects and be referentially transparent; similarly, the result of an observable should be the result of the data passed through it, not the number of parallel observables that are also active. You will see in the next chapter on dealing with time in RxJS, the sort of operation used in the code sample above is completely unnecessary.

As mentioned earlier, the operator chain is actually core to the design of an RxJS operator: every operator must perform some work on the data passing through it, and then wrap it into another observable instance that gets returned[10]. In this manner, the subscription just gets internally passed around from one context to the next. To show how this works, we'll add our own operator into the observable prototype; this operator is the logical inverse of `filter()`, called `exclude()` and is shown in listing 3.6:

Listing 3.6 Customer exclude operator

```
function exclude(predicate) {
  return Rx.Observable.create(subscriber => { ❶
      let source = this; ❷
```

[10] https://github.com/ReactiveX/rxjs/blob/master/doc/operator-creation.md#advanced

```
      return source.subscribe(value => {
           try {          ❸
              if(!predicate(value)) {
                  subscriber.next(value);      ❹
              }
           }
           catch(err) {
              subscriber.error(err);
           }
         },
         err => subscriber.error(err),         ❺
         () => subscriber.complete());
   });
}

Rx.Observable.prototype.exclude = exclude;     ❻
```

❶ Create a new observable context to return with the new result
❷ Because we're in a lambda function, 'this' points to the outer scope.
❸ Catch errors from user-provided callbacks
❹ Pass the next value to the new operator in the chain
❺ Be sure to handle errors appropriately and pass them along
❻ Adding the operator by extending the Observable prototype

As you can see from the snippet above, every operator creates a brand new observable, transforming the data in its own way and delegating it to the next subscriber in the chain. We can use it to exclude all even numbers as such:

```
Rx.Observable.from([1, 2, 3, 4, 5])
  .exclude(x => x % 2 === 0)
  .subscribe(console.log);
```

Furthermore, operation chaining in combination with observable's lazy evaluation, gives RxJS an important performance advantage over arrays, which we'll discuss next.

3.3.2 Performance advantages of sequencing with RxJS

Aside from the declarative style of development that encourages you to write side effect free code, the primary advantage of using observable operators is that there is little to no performance penalty for chaining two methods like map and filter. Behind the scenes, RxJS produces very little overhead because observables themselves are very lightweight and inexpensive to create. On the other hand, operator calls on arrays create new instances along the way, which naturally incurs more memory allocations when the collection being processed is large. We'll show this with a simple example that uses the full set of parameters for `map()` and `filter()` array functions.

```
   const original = [1,2,3];
   const result = original
      .filter((x, idx, arr) => {     ❶
        console.log(`filtering ${x}, same as original?
             ${original === arr}`); ❷
```

```
      return x % 2 !== 0;
    })
    .map((x, idx, arr) => {          ❶
      console.log(`mapping, same as original?  ${original === arr}`); ❷
      return x * x;
  });   result; //-> [1, 9]
```

❶ Map and filter expose extra parameters such as the current index and the source array. Typical implementations of these methods don't use these parameters, but it's good to know they're there.
❷ Logging to the console within the pipeline is considered a side effect. We're bending the rule here a bit to illustrate this concept.

Running this code logs the following messages

```
"filtering, same as original? true"
"filtering, same as original? true"
"filtering, same as original? true"
"mapping, same as original? false"
"mapping, same as original? false"
```

You can visualize the difference between both approaches in figure 3.10:

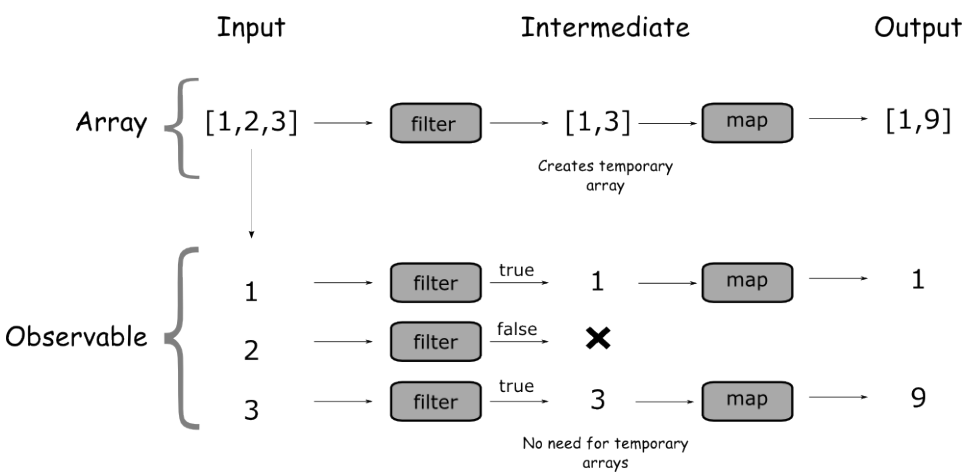

Figure 3.10 The arrays filter and map operators generate intermediate, wasteful data structures. RxJS observables are optimized and process events entirely through all functions at once, avoiding intermediate storage altogether

RxJS, by contrast, does not create intermediate data structures. As you can see in the example above, `filter()` works on the same data structure as the original because it's first on the chain. This operation returns a brand new array instance that becomes the new owning object on which we call `map`. This can be inefficient on very large collections because new data structures are created and used only once before being garbage collected. In RxJS, the underlying data structure is optimized to process each item through the pipeline from the

producer to the consumer at once, avoiding the creation of extra data structures along the way. Let's convert the same code to use Observables

```
Rx.Observable.from(original)
     .filter(x => {
        console.log(`filtering ${x}`);
        return x % 2 !== 0;
     })
     .map(x => {
        console.log(`mapping ${x}`);
        return x * x;
    })
    .subscribe();
```

Running this code shows you that each element (or mouse click, key press, asynchronous data, and others) passes through the pipeline one by one without creating intermediary storage. The first value "1" passes through filtering and then through mapping before "2" and "3" are looked at.

```
"filtering 1"
"mapping 1"
"filtering 2"
"filtering 3"
"mapping 3"
```

Now this is really optimal. This fluent chaining pattern hinges on the return type of all Observable methods to always return observables. As you know, in arrays the reduce operator breaks the chain of commands because it doesn't return an array, so further chaining becomes impossible. In RxJS every operator will return an Observable instance so that it can support further chaining. This property means that there is a virtually unlimited variety of combinations that can be assembled. While observables are abstractions over various data sources, its operators are actually just abstractions of those abstractions. That is, just like the adapter methods used to create observables from other library types, an operator is simply an adapter to convert an existing Observable into a new one with more specific functionality.

Before we continue having fun building more chains, we'll introduce another set of aggregate methods that will become handy for building nice expressive business logic. Table 3.1 briefly explains each of these aggregate functions.

Table 3.1 More aggregate operators

Name	Description
`take(count)`	**Filtering operator. Returns a specified amount (`count`) of contiguous elements from an Observable sequence. Later on we'll see this is really useful to extract a finite set of events from an otherwise infinite stream.**
`first, last`	**A refinement on the take function. Returns the first element in the Observable stream or the last, respectively.**

`min, max`	Filtering operators. Work on Observables that emit numbers returning the minimum or maximum value of a finite stream, respectively.
`do`	Utility operator. Invokes an action for each element in the observable sequence to perform some type of side effect. This operator is really for debugging and tracing purposes and can be plugged into any step in the pipeline.

Now, let's have some fun with some examples that put some of these to work in listing 3.7:

Listing 3.7 Using aggregate operators

```
Rx.Observable.from(candidates)
  .pluck('experience')
  .take(2) (1)
  .do(val => console.log(`Visiting ${val}`)) (2)
  .subscribe(); // prints "Visiting JavaScript Guru"
                        "Visiting Historian"
```

(1) Take only the first two elements (another filtering operator)
(2) Perform the logging routine and pass alone the observable sequence

Effectful computations

The `do` operator is known as an *effectful* computation, which means it will typically cause an effect like IO, a database insert, append to the DOM, write to a file, etc—all of these side effects, of course. The reason why `do()` still presevers the chain is rooted on a functional proramming artifact called the *K combinator.* In simple terms, this is is just a function that executes any effect but ignores its outcome, just passing the value along in the stream to the next operator. In a way, it's just a bridge that intercepts the stream that allows you to invoke any function. It's also known in other libraries as the `tap()` operator.

Being able to use this repertoire of operators is certainly beneficial as it frees us from having to write them ourselves, reducing the probably for bugs to occur (you can find a complete list of all of the operators used in this book in Appendix B—you're free to use it as a guide). Nevertheless, the functions passed into these operators do as your sole responsibility, so please exert thorough testing of them. We'll revisit testing further in chapter 9.

In this chapter we have talked at length about several of the core operators that come bundled with RxJS. We have purposefully avoided specifically enumerating all the operators that are available for mapping, filtering, and other tasks. That job is better left to the reference material on GitHub or on the internet[11]. Instead we wanted to understand how operators are used in conjunction with observables to build chains of logic that let us write streams declaratively, so that they are both easy to understand and easy to extend. We have chosen what we think of as the set of core operators. We explored how we can build complex

[11] http://xgrommx.github.io/rx-book/index.html

logic intuitively using fluent operators. These are operators that act primarily on a single observable and do not introduce any time-based operations. In the next chapter we will explore the time aspect of observables which allows us to handle with future data.

3.4 Summary

- Streams provide their own mechanisms for cancellation and disposal, which is an improvement over JavaScript's native event system
- The observable data type enables fluent function chaining that allows for the sequential application of operators, using a familiar model to that of arrays.
- Unlike JavaScript's native Promises, observables have built-in capabilities for disposal and cancellation
- Functions injected into the operators of an observable sequence contain the business logic of your application and should be side effect free
- Observables are self-contained with indefinitely chainable operators
- Operators act independently of each other and only work on the output of the operator that preceded it
- The order and type of operators used determines the behavior and the performance characteristics of an observable

4

It's About Time You Used RxJS

This chapter covers:

- Understanding time in RxJS
- Using time as a new dimension of your programs
- Building Observable streams with time
- Learning about RxJS operators like debounce and throttle
- Analyzing event data with buffering

Time is a tricky business. We spoke earlier about the challenges that exist when the code you write is not synchronous, in that it may have unpredictable wait times from one instruction to the next. We've defined observables as *infinite sequences of events*, and now we add the last part of the puzzle to this definition—*over time*. Relating this back to the ancient Greek Heraclitus when he said: "time is also always in motion," and so are observables.

Observables are infinite sequences of events over time

You can accurately measure the time a synchronous program takes by simply adding the execution time of its constituent functions; this doesn't hold for asynchronous programs because instructions are not linearly executed, as shown in figure 4.1:

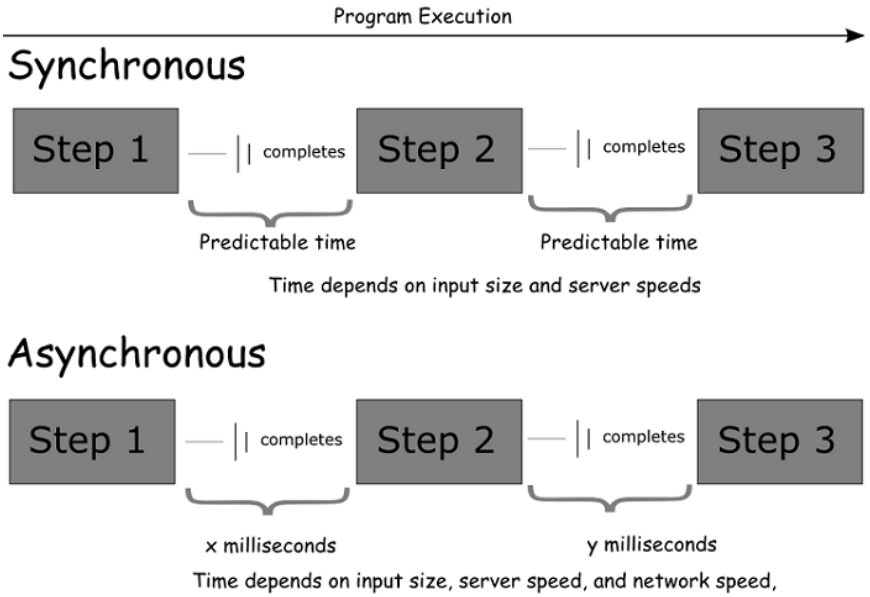

Figure 4.1 In synchronous code (top), operations are predictable and will typically depend on the input size and speed of the environment. Asynchronous programs (bottom) depend on many other factors including the speed of the network.

Generally speaking, you should never try to inject wait times into your code in an attempt to time your operations; asynchronous code is unpredictable and many factors can alter the period of time an AJAX call takes to respond or a long running computation to finish. So instead of dealing with time directly yourself and trying to guess when certain operations complete, *you should react to them*.

In previous chapters we talked about how observables react to future events, but we skirted around talking about specifics of what the future is. We also mentioned the issue with latency in passing, but never addressed it head on. This chapter will focus on first giving you a brief introduction to time as viewed by RxJS), and then exploring how to use operators to affect not only the output of a sequence of events, but also "when" this output will occur and how that type of transformation can be useful. Having direct control over time such as being able to schedule certain actions to occur, or generate data in set timed intervals is essential to creating responsive user interfaces that interact with user actions. Keep in mind that modern users have high expectations that web user interfaces behave like native applications, so that any clicks, key presses, or any other type of action is immediately acknowledged.

4.1 Why worry about time?

Time is of the essence, and in computing it's extremely essential. Many years ago the world of User Experience (UX) and design adopted the rule of the "Powers of Ten" to create guidelines about what is an acceptable amount of time a user can wait for an application to respond. The study can be summarized as such:

1. At 0.1 seconds the user feels as though their actions are causing a direct impact on the application. The interactions are real and pleasant.
2. From 0.1 to 1 second, the user still feels in control of the application enough to stay focused on their activity. For web applications, pages or sections of a page should display within 1 second.
3. From 1 to 10 seconds, the user gets impatient and notice they're waiting for a slow computer to respond.
4. After 10 seconds, the flow is completely broken and the user is likely to leave the site.

Time is the undercurrent that causes your data to flow within a stream. And you can see from this study that it's a crucial aspect of any successful application. JavaScript applications are notorious for being exposed to time a lot, and we don't mean using any date/time libraries. We're referring to the conflicting tasks of having to balance fetching data from remote locations, slow networks, user animations, scheduled events, and others—all making it incredibly challenging.

Before we get into this topic, it's important to realize that time-based functions rely on external state directly or indirectly. Why do we mean by this? Well, from a pure functional programming perspective, functions that deal with time are *inherently impure*. Time is a dimension that's not necessarily local to a function–it's actually global to the entire application and forever changing.

> **IMPURE JAVASCRIPT FUNCTIONS** **Some frequently used JavaScript functions like `Date.now()` and `Math.random()` are impure because you can never guarantee a consistent return value**

Despite this incompatibility from a pure functional programming standpoint, RxJS is still the right tool for the job. By chaining operators, as you already know, most of these issues are addressed by virtue of their sequenced, synchronous execution that threads time through and minimizes the impact of this side effect. In previous chapters we have seen how we can build a pipeline out of array-like operators that use higher order functions, such as `map()`, `filter()` and `reduce()`. Time, being a dimension that does not exist with Arrays does not have a direct analogy to any array methods. However, this does not mean we can't introduce time into operators and end up with the same fluent design.

RxJS comes bundled with many of the tools to inspect and manipulate time right out of the box (in chapter 9, we'll learn how to work with virtual time in unit tests, essentially by mocking time). Before we dive into these new operators, let's review JavaScript's own timing mechanisms and how they can easily interact with RxJS.

4.2 Understanding asynchronous timing with JavaScript

The runtime of an asynchronous application depends on factors outside of its control such as network, file system, server speed, and others; all of these become bottlenecks to code that would otherwise execute instantly on a CPU. An asynchronous event has two main challenges. It is:

- Ambiguous - in that it may or may not happen at any time in the future.
- Conditional - it is dependent on the correct execution of a previous task, i.e. loading data from a file or database.

The reason RxJS is a game-changer is that it allows you to treat asynchronous tasks as if their execution order were synchronous. In simpler terms, it's designed to serialize operations so that one piece of code executes only after another piece of code has completed. This is possible through the orchestration layer of observables so that you can handle time implicitly or explicitly.

4.2.1 Implicit timing

Consider the example of a relay race. In a relay race the runners will run as fast as they can around the track. Every time a runner finishes their set distance, he or she passes the baton to the next runner. The winner of the race is always the team who collectively crosses the finish line first. JavaScript functions that use callbacks work under this same philosophy. This is why all of the client-side AJAX APIs, as well as all of the streaming IO APIs in Node.js, to name a few, declare callback parameters—batons. Figure 4.2 shows that time factors into many types of JavaScript problems, whether it's fetching data from the server, a database, or simply handling user input:

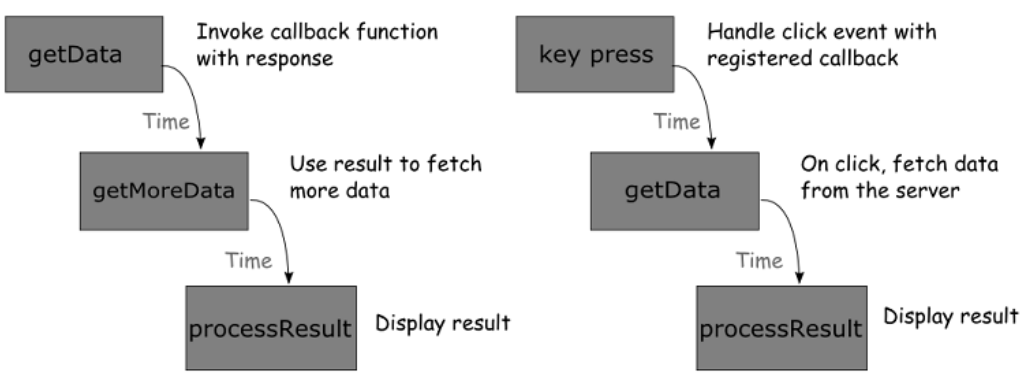

Figure 4.2 The dimension of time is implicit in IO tasks such as fetching data from the server and handling user input. On the left a sequence of AJAX calls is run that fetch necessary from the server before processing. On the right, we listen for a key press DOM event, and a result fetch from the server. As each step completes, data (the baton) is passed from one step to the next.

Both use cases above would involve the use of nested callbacks to pass the baton along the way and ensure their synchronicity. On the left, we have two nested HTTP calls that depend on each other to fetch the required data from the server. On the right, there's the use case of intercepting a DOM event which causes some data to be fetched from the server. By treating both scenarios as streams, observables internally take care of passing the baton for us through the operator's internal subscription mechanism that you learned about in chapter 3; our job is to wait and react accordingly. Now let's look at another form of timing in JavaScript, explicit timing.

4.2.2 Explicit Timing

Unlike implicit timing, explicit timing has the following desirable characteristics:

- Concrete - it will happen at a set time
- Explicit - at a time we clearly define and control
- Non-conditional - it will always happen, unless an error occurred or the stream is canceled.

Think of any time you have had to write an event that occurred a few seconds after the user performed some action, or maybe delayed an animation for a set amount of time. These are some examples of when you have explicitly declared that you wanted something to occur in the future as well as exactly when you wanted it to happen.

Use cases for these explicitly timed operations tend to revolve around two general categories: user-centric and resource-centric. In the former case we are concerned with creating something that is perceivable to the human eye. An animation, a dialog, or a validation message are examples of user-centric timings. While some animations are simply superfluous, they are also an important part of drawing the user's attention to where they should act next and creating a connection with the user interface so that it's always responsive. By carefully timing how elements move and react to the user's interaction, we can subtly guide them through what could otherwise be a difficult experience.

In the resource-centric case you can use timing to reduce demands on a given resource. Network/IO operations, rapid user input, and CPU intensive calculations are all instances of scenarios where reducing the number of method calls could have a significant boost to performance. In these cases, we could either constrain the number of calls or their impact by specifying a timeout. Another way of handling resources is through buffering or caching a certain subset of elements that they can be worked at once. One example of this that comes to mind is when we need to apply many database operations where it's preferable to just do a single bulk operation (we'll show how buffering helps us in this respect at the end of this chapter).

Explicit timing is very similar to a train or airline schedule. Tasks such as moving us from point A to point B do not happen as soon as there's availability. Trains and planes will depart at their scheduled time (for the most part, of course, but that's a separate issue). As such we

know that an airplane does not leave simply because we are on it (or we've subscribed to it). It will only leave on or after its departure time.

Explicitly timing events is a useful property in computing because it means that we can exercise some control over when a piece of code is run rather than relying on the implied timing of executing operations in sequence. The latter is unreliable for any sort of exact timing, because it is beholden to the speed and availability of processors, memory and network latency. In practical terms this simply means that the behavior of an application would be very different on a desktop as opposed to a smart phone, so explicitly defined behavior is sometimes necessary. Explicit time can be used to invoke timed tasks in sequence, such as hiding and showing messages to the user after a set number of seconds, displaying a notification dialog that guides the user to the next step, implementing a count-down clock indicating an action needs to be completed by a certain time, and others.

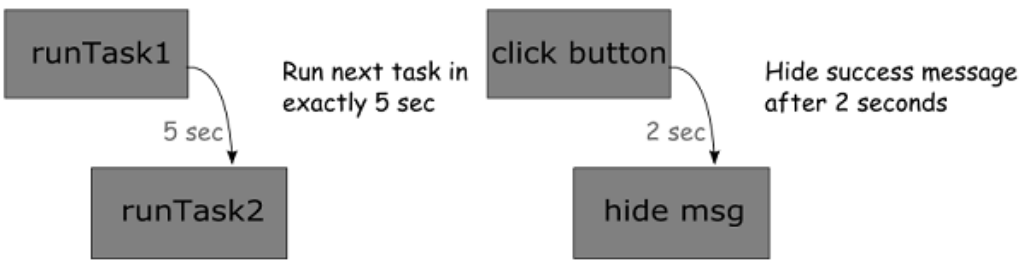

Figure 4.3 Explicitly timed tasks can execute with a delay or can overlap in time.

Now that you understand both modalities, let's see what JavaScript has to offer. If you've used functions such as `setTimeout()` or `setInterval()` before, then you've already been exposed to JavaScript's timing interfaces.

4.2.3 The JavaScript timing interfaces

There are two well-known interfaces for accomplishing explicit timing in JavaScript (there are actually several more but they are not universal so we will leave them out for now). Both methods use time relatively, that is, all declarations of future tasks will be done by a time offset relative to the current time ("now") within the application. For instance, if we show a count-down clock for an action that is to be completed within 3 minutes, this action would complete 3 minutes relative to the execution of the timed operator in use (or now + 3 min).

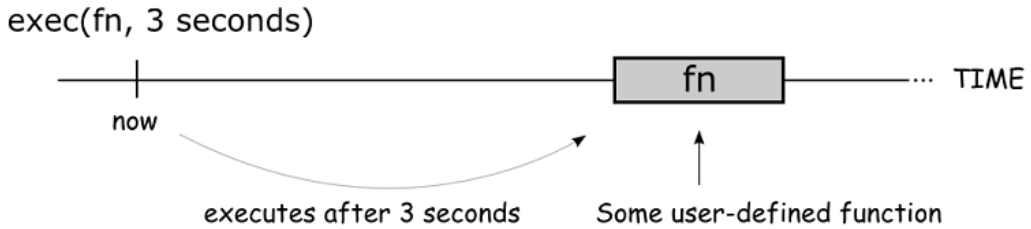

Figure 4.4 Shows explicitly timing some function after 3 seconds offset time

Generally, there are two types of explicit time in RxJS. Relative time is also called offset time and it represents only a delta measurement from now. Absolute time will always refer to a specific instance in time which can either be in the past or in the future.

When is "now?"

Now in JavaScript will always be the time provided by the system at the time a particular line of code executes. If declaring a time of new Date(), we would expect the Date to be the system time in milliseconds since 1970 at the moment that the line is evaluated. For some of the examples involving dates and time, we will be using a library called moment.js (installation instructions available in Appendix A). Moment.js provides a simple API for accessing and manipulating dates and time.

We'll go over JavaScript's most common timing operations that are part of the `WindowTimers` utilities and their equivalent operators in RxJS, starting with `setTimeout()`.

SETTIMEOUT

The `setTimeout()` function (a method of the global context object) sets up an explicit one-time task to execute at a specific point in the future (in milliseconds) relative to now. Invoking the function will tell the JavaScript runtime that some body of code should be executed milliseconds after `setTimeout()` is called, which is considered time zero milliseconds. This is the programmatic equivalent of setting an egg timer to notify you when your cake is done baking and is great if we don't want something to be executed synchronously with the rest of your program. For instance, we could schedule some CSS to slide the account details panel to the right after a short delay to make the interaction with the page richer.

```
setTimeout(() =>
  document.querySelector('#panel')
     .setAttribute('style', 'slide-right'), 1000);
```

In a way, `setTimeout()` could be considered the time based counterpart to the promise—a single emitted function that is invoked in the future. However, this offers none of the same versatility that a promise or RxJS does. For example, there's no mechanism for error handling, operator chaining, or cancellation. Clearly we can do better!

With our newly discovered understanding of streams we should also be able to recognize that this is again a simple observable, one that emits once to each subscriber. We can create an observable that wraps over `setTimeout()`, which applies some CSS action downstream into the observer to avoid conflating it with any business logic.

Listing 4.1 Working with Observables and setTimeout()

```
const source$ = Rx.Observable.create(observer => {  (1)
    const timeoutId = setTimeout(() => {
      observer.next();      (2)
      observer.complete(); (2)
    }, 1000);              (2)

    return () => clearTimeout(timeoutId);           (3)
  });

  source$.subscribe(() =>  (4)
      document.querySelector('#panel').style.backgroundColor = 'red'); (5)
```

(1) **Wrap everything inside of the Observable factory method.**
(2) **Send the single next and completed flags a second after the subscription occurs**
(3) **Define unsubscribe behavior**
(4) **Subscribe to the Observable to start the timer**
(5) **Perform CSS operations**

In listing 4.1 we have created an observable that will fire once after a second and then complete, without being tied to any business logic, and we provided the unsubscription mechanism of the timer by passing the `clearTimeout(timeoutId)` back as the disposal logic. It turns out, however, that this kind of boilerplate code is unnecessary since the RxJS library already possess implementations for these base cases. The `setTimeout()` method can be substituted with the `timer()` operator which creates an observable that will emit a single event after given period of time. Listing 4.2 performs the same action as before, using a one second timer to emit an action that changes the layout of an HTML element:

Listing 4.2 Creating a simple animation with an RxJS timer

```
Rx.Observable.timer(1000)  (1)
  .subscribe(()=>
     document.querySelector('#panel').style.backgroundColor = 'red'); (2)
```

(1) **Timer factory function in milliseconds**
(2) **Adds a custom CSS class to the selector element after a set period of time has elapsed.**

We can illustrate this using a marble diagram as follows:

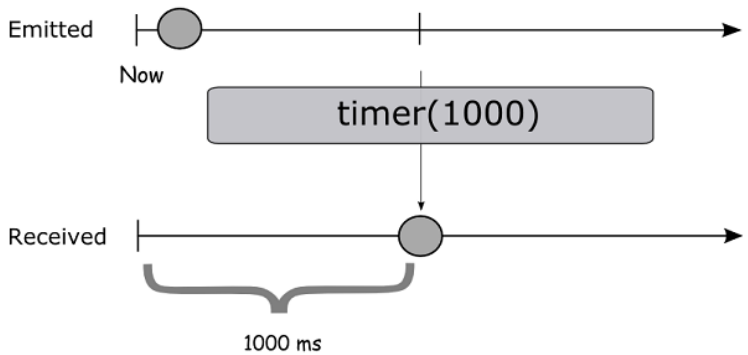

Figure 4.5 Emit an action after 1 second. This is similar to using a 1 second timeout.

Notice that we don't need to worry about callbacks any more. While the result of listing 4.2 is the same as listing 4.1, taking advantage of RxJS operators has drastically improved the readability of this simple program. As an additional plus, the timer is now emitting a generic event which can be used by several consumers if we wanted to, rather than being forced to cram several callbacks together.

SETINTERVAL

The other commonly used method is `setInterval()`. Where the `setTimeout()` bore similarities to a promise, `setInterval()` more closely resembles event emitters in that it can create multiple events over a span of time, specified in milliseconds. Now, instead of a single operation being performed, this function will invoke the operation repeatedly by specifying how far apart in time those calls should be. Using this type of operation, we could easily create a simple counter that would tell the user how long he's been on the same page:

```
let tick = 0; (1)
setInterval(() => {
  document.querySelector('#ticker').innerHTML = `${tick}`;  (2)
  tick++;      (2)
}, 1000);      (3)
```

(1) External state keeping track of the ticker
(2) Updates the counter on the screen (clear side effect)
(3) Sets the time period for the interval (in milliseconds) to 1 second

Unfortunately, the sample code above uses side effect to increment the counter. Instead, consider using a pure observable context to create a container around the effect and push out new counts as events into the observer. Listing 4.3 below creates a simple observable in charge of spawning a 2 second interval. At every interval it increments a running counter and pushes the event to any downstream observers. In light of what you learned in chapter 3 and

because JavaScript's time intervals can run infinitely, we also included the cancellation of the event. The cancellation process also occurs explicitly after 8 seconds have elapsed.

Listing 4.3 Explicit time using JavaScript timing functions and RxJS

```
const source$ = Rx.Observable.create(observer => {
    let num = 0;
    const id = setInterval(() => {
      observer.next(`Next ${num++}`);
      }, 2000);                          ❶

      return () => {                     ❷
         clearInterval(id);
      }
});

const subscription = source$.subscribe(
    next  => console.log(next),          ❸
    error => console.log(error.message),
       () => console.log('Done!')        ❹
  );

setTimeout(function () {                 ❺
      subscription.unsubscribe();
}, 8000);
```

❶ Every two seconds print the next number count
❷ The returned object contains the cancellation logic for this observable (the disposal handler).
❸ Handle next
❹ Flag when stream is done
❺ After 8 seconds, cancel the interval

This code wraps an observable over a JavaScript explicit `setInterval()` function, which emits a count after 2 seconds. After 8 seconds have elapsed, the stream is canceled and disposed of, generating a total of 8 / 2 = 4 events. Notice that because it was cancelled, the completed function of the observer is also omitted. Running this code yields:

```
"Next 0"
"Next 1"
"Next 2"
"Next 3"
"Done!"
```

As you saw in listing 4.3, `setInterval()` can be used within an observable. In its current form, however, it isn't particularly useful. For instance, there is no way to track the number of invocations, short of tracking it ourselves with an external variable. An actual observable operator would be more useful because it can forward state downstream, while remaining infinitely more extensible.

Fortunately, we don't have to implement this method either, because it already exists as the `interval()` operator, which gives us a very simple compact and generic form. Essentially,

you can subscribe to the interval and begin receiving periodic events and in true RxJS form we have also gained automatic disposal semantics for free!

Both the interval and the timer emulate the existing behavior that we see from the traditional JavaScript interfaces, however they do so without the associated entanglement. While interval emits the number of milliseconds in the interval, you don't actually have to use it. Most RxJS implementations use interval to simply monitor and react to an external resource. For instance, we can make periodic AJAX requests to an external API and detect when something has changed. This is really useful for applications like stock tickers, weather trackers, and other real-time applications. In the next section we will look at how we can introduce time-based operators into a stream to implement a stock ticker component.

4.3 Back to the future with RxJS

The time-based operators in RxJS come in several different flavors. The factory functions that resemble generic versions of the `setInterval()` and `setTimeout()` methods will have function signatures resembling these RxJS operators, respectively:

```
Rx.Observable.interval(periodInMillis)
Rx.Observable.timer(timeoutInMillis)
```

These static methods (`interval()` and `timer()`) work just like the other factory methods that we saw in previous chapters for creating observables, except that where most of those either wrapped other sources or emitted events immediately upon subscription, the time-based factory methods will only emit after a provided amount of time has elapsed.

> **SCHEDULERS** There's actually another parameter called a *scheduler* that's passed into either `interval()` or `timer()`, as well as other operators. You can imagine how unit testing code with long timers is virtually impossible to test. We'll learn about schedulers in unit testing in chapter 9.

We've illustrated these timing operators by themselves; however, in real-world problems they are usually combined with observable streams that generate meaningful data. This combination is extremely powerful, because you can use the timers to synchronize the frequency with which you consume data from an observable.

We're going to start implementing our stock ticker widget using only functional and reactive primitives. We'll be coming back to this example and adding more features as you explore and get more comfortable with RxJS. At the moment, we'll use a made up symbol ABC to keep things simple and instead of fetching its stock price using a real web service, we'll emulate it using random numbers. In the next chapter, we'll tie everything together and fetch stock data using actual AJAX calls to a real service endpoint. Here's our simple function that fabricates random numbers:

```
const newRandomNumber = () => Math.floor(Math.random() * 100); ❶
```

❶ Maps the number into the range [0,100].

NOTE Of course this function has side effects, but we're only using it as a random producer of events, not part of our business logic.

At a high level, the process of fetching stock data works in the following way, shown in figure 4.6. Our program consists on generating a random value every two seconds, which we'll use to emulate continuous stock quote prices of our made-up ABC company:

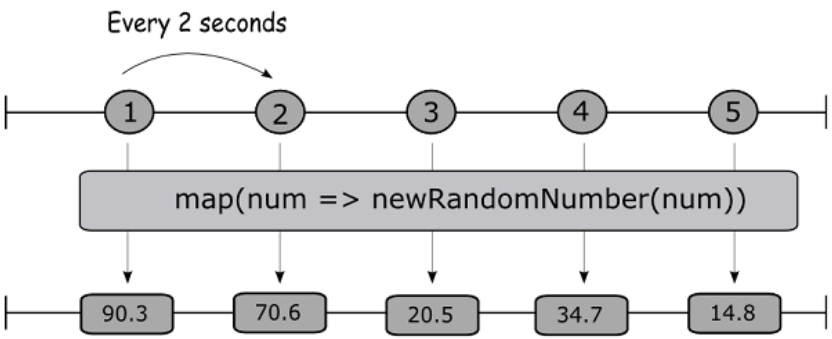

Figure 4.6 The process of fetching the stock data involves using a Promise to make a remote HTTP call against a stock service. In this case, we'll just use a random number generator function to emulate stock price changes, and revisit this in chapter 5.

Stocks prices are made up of a numerical portion and a currency string. We can model this with a simple Money *Value Object*:

```
const Money = function (currency, val) {
  return {
    value: function () {
      return val;
    },
    currency: function () {
      return currency;
    },
    toString: function () {
      return `${currency} ${val}`;
    }
  };
};
```

A Value Object is a design pattern used to represent simple immutable data structures whose equality is not based on the entity itself, but on their values. In other words, two value objects are equal only if they have the same content or their corresponding properties contain the same values. These objects are ideal to transfer immutable state from one component to another.

The stock ticker widget is kicked-off by a 2 second interval, which will set the pace for how often notifications are pushed onto our observable stream. Subscribers receive each value and display update the DOM.

Listing 4.4 Simulating a simple the stock ticker widget

```
const USDMoney = Money.bind(null, 'USD');❶

Rx.Observable.interval(2000) ❷
   .skip(1) ❸
   .take(5) ❹
   .map(num => new USDMoney(newRandomNumber()))
   .forEach(price => {
       document.querySelector('#price').textContent = price;
    });
```

❶ Creating a new function to manage USD
❷ Creating a 2 second interval
❸ Skip the first number emitted, zero
❹ Because this is an infinite operator, simulate only 5 values

As we've mentioned before, you can use RxJS' time operators to control the advancement of the stream they're part of. Another variation of `interval()` is an instance operator called `timeInterval()`, which gives us a bit more extra information such as the count as well as recording the time in between intervals, which we can use with `do()` to print the time elapsed between price refreshes:

Listing 4.5 Augmenting stock data with the time interval

```
Rx.Observable.interval(2000)
   .timeInterval() ❶
   .skip(1)
   .take(5)
   .do(int =>
      console.log(`Checking every ${int.interval} milliseconds`)) ❷
   .map(int => new USDMoney(newRandomNumber())) ❸
   .forEach(price => {
       document.querySelector('#price').textContent = price;
    });
```

❶ Augments the interval value as an object that also includes the precise number of milliseconds between intervals.
❷ The interval property contains the number of milliseconds elapsed between one interval and the next
❸ The value property returns the number of intervals emitted by the Observable

This function actually reveals the fraction of delay present in the call to compute the random number, checking the log shows:

```
"Checking every 2000 milliseconds"
"Checking every 2002 milliseconds"
"Checking every 1999 milliseconds"
"Checking every 2001 milliseconds"
"Checking every 2000 milliseconds"
```

The two types of timing options we've discussed so far (explicit and implicit) are not mutually exclusive. Using one does not preclude the use of the other. However, using them together does require understanding of how time propagates to downstream operators. This is

especially important to introduce a new operator called `delay()`. Unlike when we use only implicitly timed operations, where each operation will initiate directly after the previous one, explicit timing can introduce new issues with ordering.

To visualize this, consider the `delay()` operator. It accepts a time in milliseconds and can be used to *time shift the entire observable sequence*. This means that if an event arrives at 1 second time to `delay()`, it would actually be emitted to the subscribe method at 3 seconds if the delay was 2 seconds long. You can visualize this with the following marble diagram in figure 4.7:

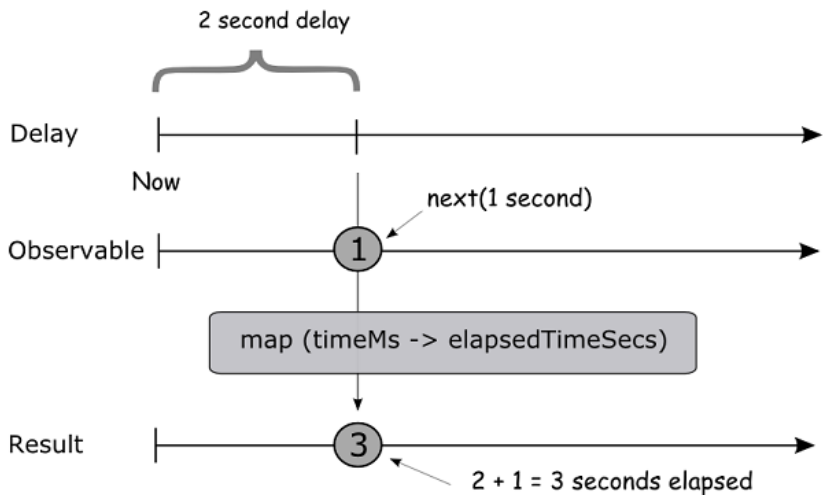

Figure 4.7 A marble diagram of how the delay operator affects the output Observable.

Here's a code sample to illustrate its use:

Listing 4.6 Showcase the delay operator

```
Rx.Observable.timer(1000) ❶
   .delay(2000)  ❷
   .timeInterval()
   .map(int => Math.floor(int.interval / 1000)) ❸
   .subscribe(seconds => console.log(`${seconds} seconds`)); ❹
```

❶ Emit a value after one second
❷ Delay the entire sequence by a 2 second offset
❸ Compute the time elapsed using the interval value from timeInterval()
❹ Convert and round the result to seconds

This code reflects what figure 4.7 shows, how the stream is affected by the delay operator to create a shifted Observable sequence. Running this code will print "3 seconds" rather than the initial time. The effect of this process raises two important points about the nature of a delay:

1. Each operator will only affect the propagation of an event, not its creation.
2. Time operators act sequentially.

Let's discuss each of these points individually:

4.3.1 Propagation

The first effect applies to all operators but is especially critical with explicitly timed operations because it can have drastic effects on the performance of your application. As operators have no knowledge about the specific observable to which they are attached to, they are unable to affect the production of events (remember that we can think of each operator as a work station on an assembly line). In order to maximize decoupling and throughput, each operator must work independently of one another. This decoupling can lead to problems, however, if one station drastically outpaces another in terms of production. Suppose an assembly line had one station that painted a part and another that required that part sit for at least an hour for the paint to dry. If the first station produced one part every minute, we would end up with 60 parts waiting to dry at any given moment. The deficit would be greater the larger the ratio of production to propagation.

This means that for an operator like `delay()`, it will only affect the propagation of the events downstream after they've been generated, not during. Consider a simple observable example that shows a delay firing all at once, for an array with multiple values. Instead of delaying each event in the stream, delay shifts the entire sequence by a specific period of time:

Listing 4.7 Delay shifts the entire observable sequence

```
Rx.Observable.of([1, 2, 3, 4, 5])
      .do(x => console.log(`Emitted: ${x}`)) ❶
      .delay(200)
      .subscribe(x => console.log(`Received: ${x}`));
```

❶ Use the .do() operator to introduce an effectul computation, in this case log to the console the emitted data
❷ Delay propagation of the event by 200 ms

You might expect that each "Emitted" event would be followed immediately by a "Received," followed by a delay of 200 milliseconds before the next "Emitted" - "Received" pair. This is a common mistake for newcomers to RxJS. In reality, the result is:

```
Output:

"Emitted: 1,2,3,4,5"

// 200 milliseconds later...

"Received: 1,2,3,4,5"
```

This result is something much different than you might have expected, because the generation of the events is independent from the delay operator. This is, in fact, exactly the same as the factory worker scenario where production and propagation are not matched. Figure 4.8 illustrates what's happening:

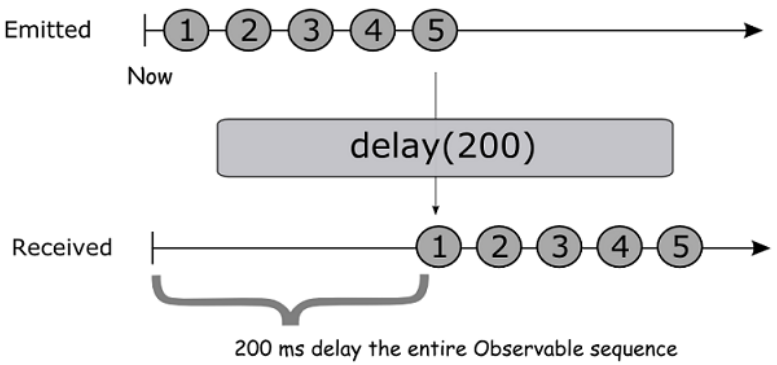

Figure 4.8 A 200 ms delay injected into the pipeline shifts the entire observable sequence instead of each event

A corollary to this idea is that in order for a delay to work, it must buffer the events it receives before emitting them at the right time. The `delay()` operator has a fixed constant value called a "bounded" upper limit that is proportional to the number of events received and their frequency. This can be calculated with the following relation:

```
# of events received / time * (x time units)
```

For the most part, as long as an operator will eventually propagate, a buffer will always remain bounded and it will not grow beyond a certain size (we will come back to buffering in a bit). It's worth mentioning this behavior as it is often confusing for newcomers to RxJS who see delay and think that the production of the sequence can be delayed or somehow controlled downstream. Another important aspect of these time-based operators is that they act sequentially.

4.3.2 Sequential time

As you've already seen by now, when operators are chained together they always operate in sequence, where operators earlier in the chain will execute before operators later in the chain, this downstream flow is a core design of RxJS observables. You would expect this to hold when dealing with time-based operators, as well. That is to say, if we were to chain multiple delays, we would expect that the actual delay downstream should be the sum of each of them. While `delay()` is sequential, its execution as it appears in the stream declaration with respect to other non-time operators isn't, which can be confusing. Listing 4.8 shows how `delay()`

stays true to its definition to shift the entire observable sequence regardless of where it's placed in the sequence:

Listing 4.8 Sequential delay operators

```
Rx.Observable.from([1, 2])
    .delay(2000)
    .concat(Rx.Observable.from([3, 4]))
    .delay(2000)
    .concat(Rx.Observable.from([5, 6]))
    .delay(2000)
    .subscribe(console.log);
```

❶ Chaining multiple delay operators together

Based on your intuition of how non-time operators work, you would expect the element pairs [1, 2], [3, 4], and [5, 6] to be emitted two seconds apart, but this is not the case. Each subsequent delay would receive an event after the preceding one expires, thus creating a delay of 2000 + 2000 + 2000= 6000 milliseconds with respect to the entire observable sequence, printing [1,2,3,4,5,6] after all 6 seconds have elapsed, as shown in figure 4.9:

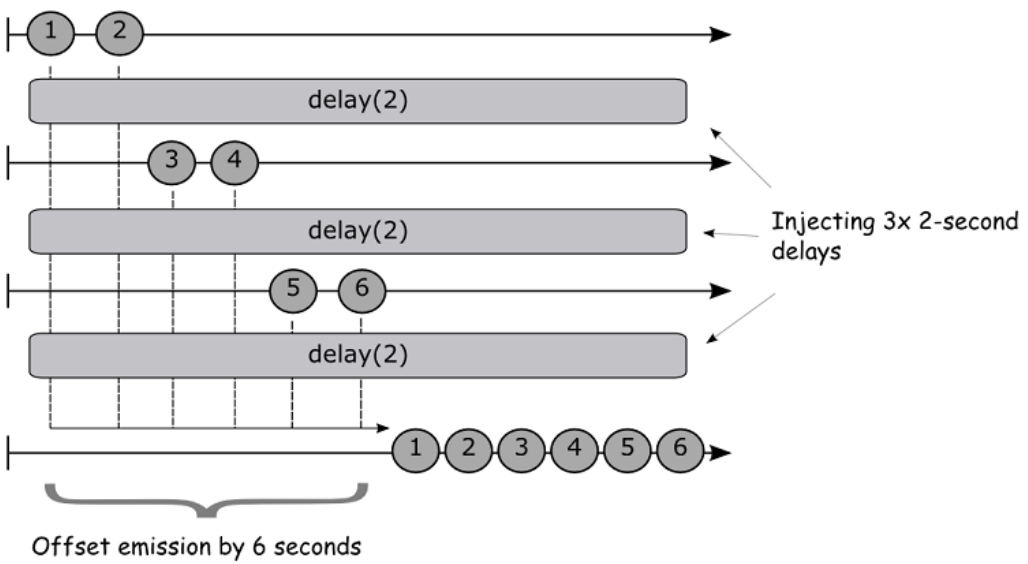

Figure 4.9 Injecting several delay operators has the effect of compounding one delay operator equal to the aggregate amount of wait time

We can relate this to a downstream river with control dams along the way that temporarily delay the flow of water. When the water reaches its destination, however, all of the water

would be there at once. So, don't make the mistake of thinking that embedding multiple delays into a stream will actually exert its effect at each stage in the pipeline.

Now that you've understood some of RxJS time operators, let's put them in action and mix them up with other familiar operators. One of the main areas of concern when building responsive user interfaces is dealing with user input. For instance, we would expect that the application we're building would react accordingly whether the user was pressing a button once, or rapidly many times. Consider the DOM events fired by a text box on each keypress. Should the application handle each and every single change, or could we just process the result when the entire word is entered? Let's examine this next.

4.4 Handling user input

Both the `interval()` and `timer()` static methods are used to create observables and initiating an action after a timed offset. These, together with `delay()`, are probably the most familiar combinations when scheduling a future action that executes once, at a set interval, or after a set time, respectively. These operators are ideal to be used with explicit events for which we know the action to perform, and schedule it to be run at some later time. But what happens when sequences of events are generated from a dynamic event emitter, like a user's mouse move or key presses, that can emit potentially lots of events in a short period of time. In this case, we're probably not interested in processing each and every one of them, but events in between.

In this section, we'll take a look at two of the most useful observable mechanisms: debounce and throttle. Both perform very similar functions, so we'll learn to apply debouncing to implement a smart search program first starting with an imperative version before moving into a fully reactive version.

4.4.1 Debouncing

In signal and circuit design, it's common to *debounce* a signal so that a manual input signal doesn't appear like multiple. This is common when building switches and other types of manual user interactions. The same happens when software interacts with humans for which RxJS offers an operator called `debounceTime()`, which emits an event from an observable sequence only after a particular timespan (in milliseconds) has passed without it emitting any other item—essentially sending one and not many within a certain time frame. You can think of this operator as belonging to the filtering category of operators, using time as the predicate to decide which events to keep. In simple software terms, debouncing means: "execute a function or some action only if a certain period of has passed without it being called." In our case, it means that an event is emitted from an observable sequence if a set timespan has passed without emitting another value, and may be represented with this marble diagram:

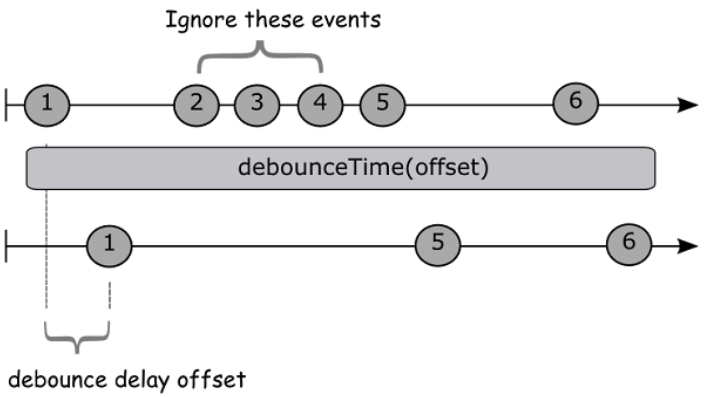

Figure 4.10 Generic debounce operation allows the emission of an item only after a certain time span has elapsed before another event is emitted.

Let's see a simple example showcasing the debouncing operator, that emits the most recent click after a rapid succession of clicks:

```
Rx.Observable.fromEvent(document, 'click')
    .debounceTime(1000)
    .subscribe(c => {
       console.log(`Clicked at position
            ${c.clientX} and ${c.clientY}`)
    });
```

With this code the user can generate a burst of click events, but only the last one will be emitted after a second of inactivity.

Observable factory vs. instance methods

Static methods and instance methods on some web sites are referred to as Observable methods and Observable instance methods, respectively. The static methods are defined directly on the Rx.Observable object and are not part of the object's prototype. These are typically used for initiating the declaration of an observable instance; for example; Rx.Observable.interval(500). The observable instance methods are included in the object's prototype and are used as members of the chained pipeline after you've initiated an observable declaration. We've referred to these simply as operators in previous chapters for brevity. For example: Rx.Observable.prototype.delay(), Rx.Observable.prototype.debounceTime(), and others.

Let's put this operator to the test. Consider the example of a smart search widget that allows you to easily look up articles from Wikipedia by giving you suggestions as you type:

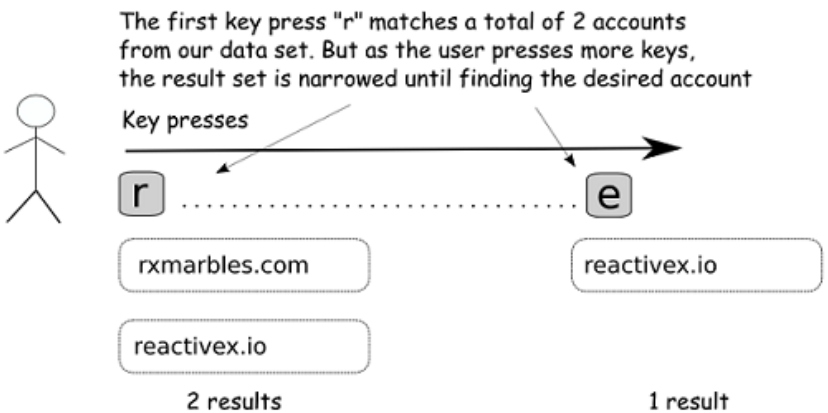

Figure 4.11 The user interacts with the search box. As the user types on the keyboard the list of search results filters down. This is typical of modern search engines.

Running this code generates the following output: if I type "r" into the text box it will suggest two possible results: "rxmarbles.com" and "reactivex.io." Additionally, typing "e" into the box will filter down the results further to just "reactivex.io." As the user types their keywords, we recognize that making web requests after each letter pressed is a bit wasteful, and we would be better off if we allowed the user to type first and wait for a specific amount of time before making the expensive roundtrip request. This has the benefit of restricting the number of web requests that are actually made to the server while the user is still typing, which is better resource utilization overall. When interacting with a third-party service, like the Wikipedia API, is good to do this so that you don't hit your rate limits. Preferably, the program should back-off until the user has stopped typing for a brief period (indicating that they are not sure what to type next) before looking for possible suggestions. This is both to prevent the additional network congestion of initiating requests that will never be seen, and also to avoid the annoying user experience of having the type-ahead flicker as the user types.

Prior to RxJS, we would need to implement this ourselves, probably using `setTimeout()`. The timeout is reset every time the user types a new key. If the user does not type for a short duration than the timeout will expire and the function will execute. We'll need to cover a little more ground in order to start streaming from remote services, so in the meantime, we'll use a small data set and return to this program in the next chapter:

```
let testData = [
       'github.com/Reactive-Extensions/RxJS',
       'github.com/ReactiveX/RxJS',
       'xgrommx.github.io/rx-book',
   'reactivex.io',
       'egghead.io/technologies/rx',
       'rxmarbles.com',
```

```
      'https://www.manning.com/books/rxjs-in-action'
    ];
```

To implement this, we're going to need HTML elements for the search box and a container to show results. As the user types into the search box, any results will get inserted into this container

```
const searchBox = document.querySelector('#search'); //-> <input>
const results = document.querySelector('#results');  //-> <ul>
```

Here's a possible implementation of a smart search box using a typical imperative or procedural solution without debouncing. The goal of this code sample is to illustrate how you would typically approach this problem without thinking in terms of functional programming and streams:

```
searchBox.addEventListener('keyup', function (event) { ❶
   let query = event.target.value;
   let searchResults = [];
   if(query && query.length > 0) {
      for(result of testData) {     ❷
        if(result.startsWith(query)) {
          searchResults.push(result);
        }
      }
   }
   if(searchResults.length === 0) { ❸
      clearResults(results);
   }
   else {
      for(let result of searchResults) {
        appendResults(result, results);
      }
   }
});

function clearResults(container) { ❹
  while(container.childElementCount > 0) {
     container.removeChild(container.firstChild);
  }
}

function appendResults(result, container) { ❺
    let li = document.createElement('li');
    let text = document.createTextNode(result);
    li.appendChild(text);
    container.appendChild(li);
}
```

❶ **Listen for all 'keyup' events on that search box**
❷ **Loop through all of our test data URLs and find matches**
❸ **If no matches are found, clear the list of search results; otherwise, append the items found**
❹ **Function used to clear search results container**
❺ **Function used to append a result onto the container**

This code is very simple. Building a smart search box involves binding the `keyup` event and using the value in the textbox to look up possible search results. If there's a match, the results are appended to the DOM; otherwise, the DOM is cleared. For this we created two functions `clearResults()` and `appendResults()`,respectively. This flow is shown in figure 4.12:

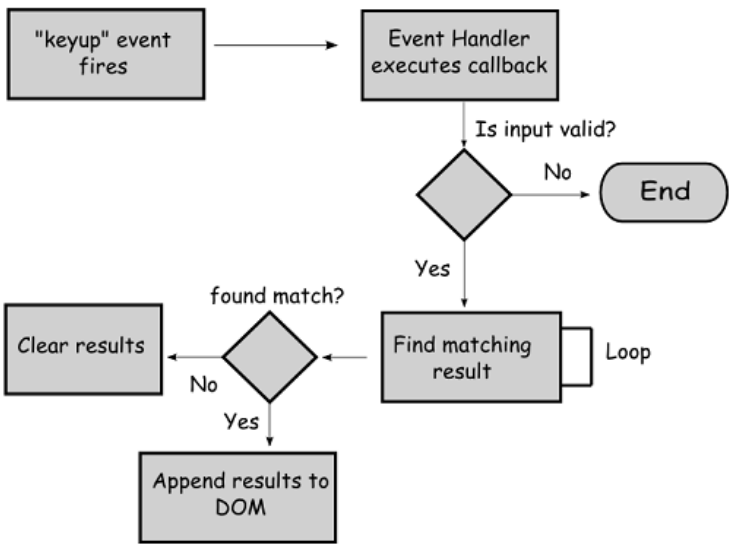

Figure 4.12 Shows the different stages of this simple program from the moment the event fires, finding a correct search result, and writing the data on the page. Notice the use of a couple of conditional statements and loops. This is the mark of true imperative design.

This code has no debouncing logic, so essentially it will issue queries against the test array every letter the user enters letters into the search box. By adding debouncing logic to this imperative code example, we end up with listing 4.9:

Listing 4.9 Manual debouncing logic for smart search widget

```
var timeoutId = null;  ❶

searchBox.addEventListener('keyup', function (event) {

   clearTimeout(timeoutId); ❷

   timeoutId = setTimeout(function (query) { ❸
     console.log('querying...');
     let searchResults = [];
     if(query && query.length > 0) {
       for(result of testData) {
        if(result.startsWith(query)) {
          searchResults.push(result);
        }
      }
```

```
      }
      if(searchResults.length === 0) {
        clearResults(results);
      }
      else {
        for(let result of searchResults) {
          appendResults(result, results);
        }
      }
   }, 1000, event.target.value);  ❹
});
```

❶ Register the current timeout
❷ As the user presses the key, clear the current timeout to initiate a new one
❸ Start a new timeout that will fire after 1 second of no interaction
❹ Debounce for one second

In Listing 4.9, whenever the user types a key, that key press event will trigger the event listener. Inside of the event listener we will first clear any existing pending timeouts using `clearTimeout()`. Then we will reset the timer. The result of all this is that the task won't execute until the input has "cooled off" or there is a delay between inputs. In this case the delay in inputs dovetails nicely with the perception of "helping the user along" if they hesitate in typing.

While the number of lines of code added here are minimal, there are several things that make this approach less than desirable. For one, we have been forced to create an external `timeoutId` variable that is accessible within the callback's closure and exists outside of the scope of the event handler. This introduces more of the dreaded global state pollution, which is a clear sign of side effects. Further, there's no real separation of concerns. The operation itself is not terribly indicative of what it is intended for and the timeout value together with all the business logic gets buried and entangled with the debouncing logic—adding additional logic was very invasive.

It would be nicer if we could clean up this operation. Let's wear our functional hats and start by creating a method similar to `setTimeout()` that will encapsulate the debouncing mechanism, separate it from the rest of the business logic, and lift the method out of the closure. From that we could expect a function signature as in listing 4.10.

Listing 4.10 Dedicated debounce method using vanilla JavaScript

```
const copyToArray = arrayLike => Array.prototype.slice.call(arrayLike);

function debounce(fn, time) {
  let timeoutId;        ❶
  return function() {   ❷
    const args = [fn, time]
        .concat(copyToArray(arguments)); ❸
    clearTimeout(timeoutId);            ❹
    timeoutId = window.setTimeout.apply(window, args); ❺
  }
}
```

1. Store timeoutId externally so it can be shared.
2. Return a function that wraps the original callback
3. Capture the arguments object into an actual array
4. Reset the timer
5. Proxy the arguments to the setTimeout() method.

This new `debounce()` can now wrap the request logic, and allows us to elegantly decouple event handling logic from debouncing logic. Using this method, we can extend the imperative version of the search code:

Listing 4.11 Using custom debounce method

```
function sendRequest(query) { (1)
   console.log('querying...');
   let searchResults = [];
   if(query && query.length > 0) {
      for(result of testData) {
         if(result.startsWith(query)) {
             searchResults.push(result);
         }
      }
   }
   if(searchResults.length === 0) {
       clearResults(results);
   }
   else {
      for(let result of searchResults) {
         appendResults(result, results);
      }
   }
}

let debouncedRequest = debounce(sendRequest, 1000); (2)

searchBox.addEventListener('keyup', function (event) {
  debouncedRequest(event.target.value);
}); (3)
```

1. Helper method to send HTTP requests
2. Wrap this helper method with debounce()
3. Invoke the debounced version of the function after handling user input

As you can see, just like in previous chapters we are able to remove much of the ugly and bug-prone extrinsic state by compartmentalizing behavior into small functions. However, by doing this we have introduced another problem which is that the result of the task completion is no longer readily available. We need to now push all of the event handling logic into `sendRequest()` because there is no way to forward the result outside of the closure. All of this serves to show that implementing your own debounce logic can be quite daunting and imposes lots limitations to your design.

Fortunately, with RxJS this becomes extremely simple, and the fact that we push all of the DOM interaction into the observer means our business logic is greatly simplified from the

complex flow chart shown above, to the following abstract model where data moves always forward from producer to consumer in its typical unidirectional manner:

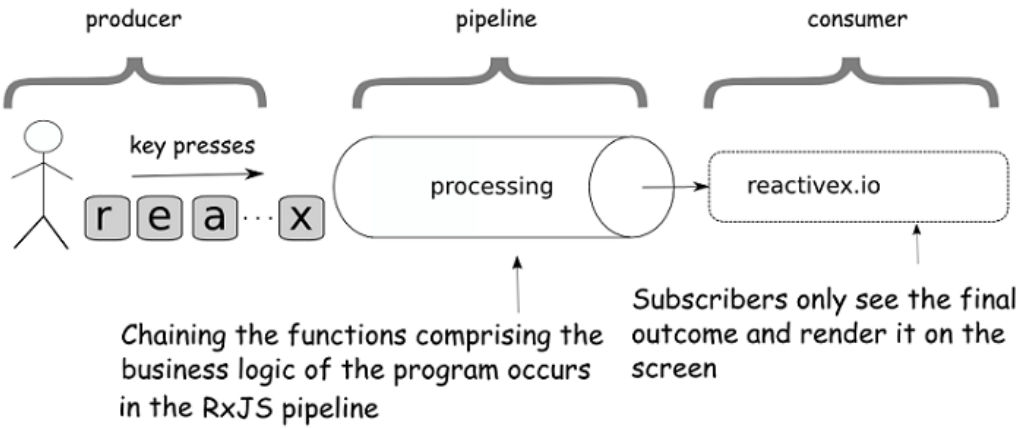

Figure 4.14 RxJS uses the pipeline to process data from the producers in a way that's consumable and acceptable to the consumers to display and do more work on. Reactive state machines are modeled using marble diagrams, more on this later in this section.

When thinking reactively, a debounce operation is simply a filter embedded into the processing pipeline that uses time to remove certain events from the observer. Hence, we can use observables to explicitly inject time into our stream as a first class entity. For this functional version of the program, we'll create a pure, more streamlined version of `sendRequest()`, and use `debounceTime()` to implement all of the debouncing logic for us. Listing 4.12 shows you the functional-reactive version:

Listing 4.12 A simple debounce-optimized search program

```
const notEmpty = input => !!input && input.trim().length > 0;

const sendRequest = function(arr, query) {
  return arr.filter(item => {
    return query.length > 0 && item.startsWith(query);
  });
}

const search$ = Rx.Observable.fromEvent(searchBox, 'keyup')
  .debounceTime(1000) ❷
  .pluck('target', 'value')
  .filter(notEmpty)   ❸
  .do(query => console.log(`Querying for ${query}...`))
  .map(query =>       ❹
        sendRequest(testData, query))
  .forEach(result => {
    if(result.length === 0) {
      clearResults(results);
```

```
    }
    else {
      appendResults(result, results);
    }
  });
```

1. Refactor sendRequest() to be more functional and return the list of matched search results
2. Injecting a debounce offset of 1 second, after capturing the user's input. This will allow the user to type any characters in the timespan of a second, before requests are sent over.
3. Helper function to check whether a string is empty
4. Maps the search results into the source observable. For now we're creating a test data set. In chapter 5 we'll learn add AJAX calls into our streams

The advantage of this approach should be readily apparent. Note that, just like other operators, `debounceTime()` simply slots into the stream. RxJS' time abstraction allows the introduction of time to be done transparently and in conjunction with all of our other operators, seamlessly. Figure 4.15 shows how debouncing affects the user input is passed through the stream:

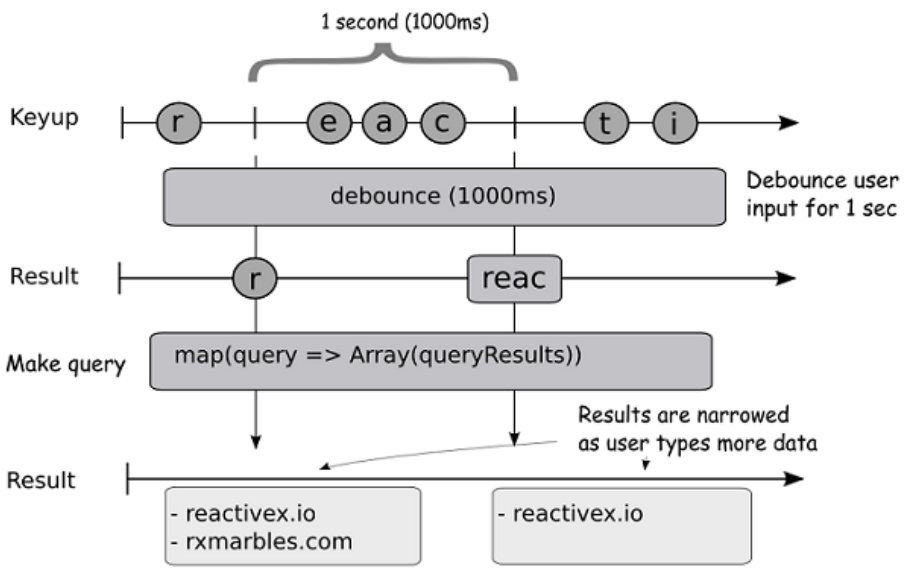

Figure 4.15 Debouncing the event stream allows the user to input a rapid set of characters so that they can be processed all at once, instead of querying for data at every key press.

As we mentioned before, this kind of behavior is quite common and testimony of RxJS' extensible design. In scenarios where the operation that needs to be performed is expensive or the resources available to the application are limited such as those on a mobile platform, limiting the amount of extraneous computation is an important task, and debouncing is a way

to control that. This is a more efficient way of handling user input, and the Wikipedia servers agree with us. Another way to achieve this is with a sister operator to debounce known as throttle.

4.4.2 Throttling

The `debounceTime()` operator has a close sister called `throttleTime()`. Throttling ignores values from an observable sequence that are followed by another value before a certain period of time. In simple terms this means: "execute a function at most once every period of time," as shown in this marble diagram:

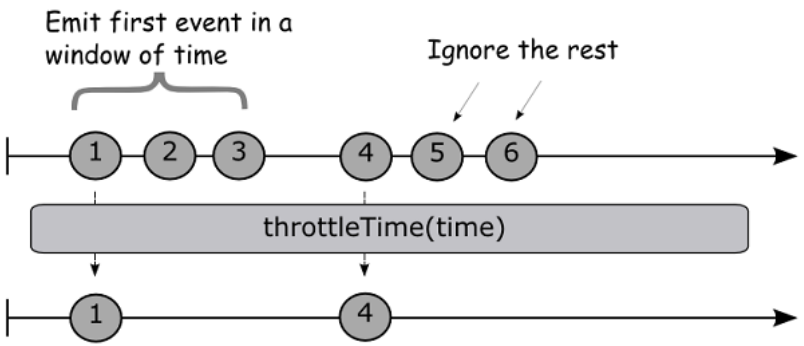

Figure 4.16 Throttling events at most one to be emitted in a specified period of time, in this case the first event in the timespan window

A good use case of this is, say you're executing an expensive computation in response to a user scrolling or moving the mouse. It's probably best to wait for the user to finish scrolling, instead of executing running this function thousands of times. This train of thought can also work well with banking sites for controlling important action buttons like withdrawing from an account, or popular shopping sites like Amazon to add extra logic around the "1-click buy" buttons. Let's see how to throttle mouse moves:

Listing 4.13 Controlling button action with throttle

```
Rx.Observable.fromEvent(document, 'mousemove')
  .throttleTime(2000)  (1)
  .subscribe(event => {
     console.log(`Mouse at: ${event.x} and ${event.y}`);
  });
```

(1) Plugging a 2 second throttle to prevent click bursts

With throttling in place, even if the user moves the mouse rapidly, it will only ever fire once in a 2 second period, instead of emitting hundreds of events in between. We really only care

where the mouse cursor lands. The effect of throttle in this program can be seen in figure 4.17:

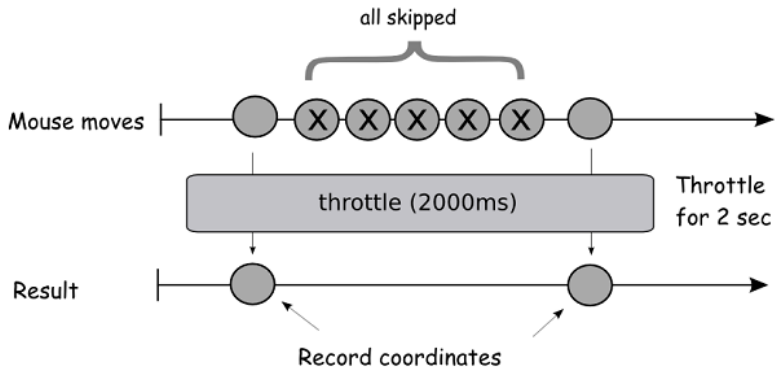

Figure 4.17 Throttling button clicks allows the application to ignore the accidental repeated clicking preventing the withdraw action from executing multiple times.

Up until now, this chapter has been all about understanding the basics of time in RxJS. We explored several of the operators to demonstrate the power of RxJS to simplify the concepts of time and make coding with it much easier. We demonstrated that RxJS operators allow us to intuitively use time as part of an Observable.

Time-based operators like `delay()` and others contain buffering or caching logic under the hood in order to temporarily defer emitting the events without any loss of data; this is how RxJS is able to control or manipulate the time within the events of an observable sequence. This can be a very powerful feature to use in your own applications as well, so RxJS exposes buffering operators for you to use directly in order to temporarily store data of a certain amount or for a certain period of time. This is analogous to building control dams along the way to harvest the streams not only for a specific period of time, but also of a certain size, so that you can make decisions and potentially transform the stream before it flows through. In this section we'll take up a notch. The plan is to use buffering as a means to temporarily cache data, together with timed operators to debounce or throttle user input.

4.5 Buffering in RxJS

We've mentioned many times before that streams are stateless and they do not store any data. In chapter 2, we showed how you can create a small repository of data within a custom iterator, we called it `BufferedIterator`, which we used to format and change the nature of the elements being iterated over.

RxJS recognizes it's useful to temporarily cache some events like mouse moves, clicks, and key presses, instead processing a deluge of events all at once, and apply some business logic onto them before broadcasting them out to subscribers; depending on the nature of this cached data, you might allow them to flow through as is, or perhaps create a whole new event

that subscribers see; the buffering operators provide underlying data structures that transiently stores data in the past so that you can work with it in batches instead of whole, as shown in figure 4.18:

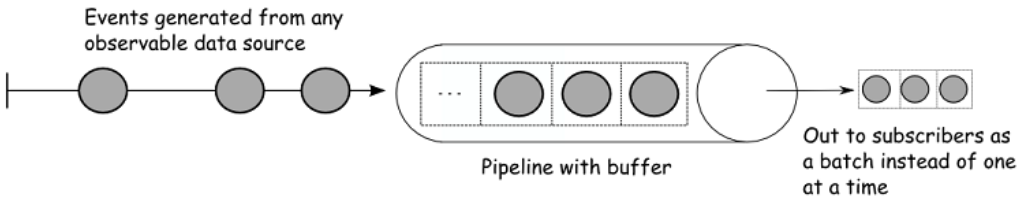

Figure 4.18 Shows buffers plugged into an RxJS pipeline. A buffer of size 3 as in this case can store up to three events at a time, and then emit them all at once as an array of observables

One important thing to understand is what happens to the data emitted as a result of buffering. Instead of a single observable output, as we would normally expect and as you've seen all along, subscribers receive an array of observables. Buffering is useful for tasks where the overhead of processing items is large and therefore it's better to deal with multiple items at once. A good example of this is when reacting to a user moving the mouse or scrolling a web page. Because mouse movement emits hundreds of events at once, you might want to buffer a certain amount, and then emit an Observable in response to where the mouse or the page is.

Table 4.1 provides a list of the observable instance methods that we'll explore in this chapter:

Table 4.1 API documentation for buffer operators

Name	Description
`buffer(observable)`	Buffers the incoming observable values until the passed `observable` observable emits a value, at which point it emits the buffer on the returned observable and starts a new buffer internally, awaiting the next time `observable` emits.
`bufferCount(number)`	Buffers a number of values from the source observable then emits the buffer whole and clears it. At this point a new buffer is internally initialized.
`bufferWhen(selector)`	Opens a buffer immediately, then closes the buffer when the observable returned by calling `selector` emits a value. At that time, it immediately opens a new buffer and repeats the process.
`bufferTime(time)`	Buffers events from the source for a specific period of time. After the time is passed, the data is emitted and a new buffer initialized internally.

Generally speaking, this ability to capture a temporary set of data gives you a window of time that you can use to examine and make decisions about the nature or frequency of the data

coming in. The buffer operators achieve this by grouping the data of an observable sequence into a collection (an array), and provides a second parameter called a *selector function* which can you use to transform or format the data beforehand.

We'll start with the `buffer()` operator. `buffer()` gathers events emitted by source observables into a buffer until a passed in observable, called the *closing observable*, emits an event. Then, at this point it flushes out the buffered data, and starts a new buffer internally. To show this, we'll use `timer(0, 50)`. A period argument of 50 causes the timer to emit subsequent values every 50 milliseconds. We'll buffer the events with a closing timer observable of 500 milliseconds; hence, we should expect 500 / 50 = 10 events to be emitted at once. Here's a marble diagram showing this process in figure 4.19:

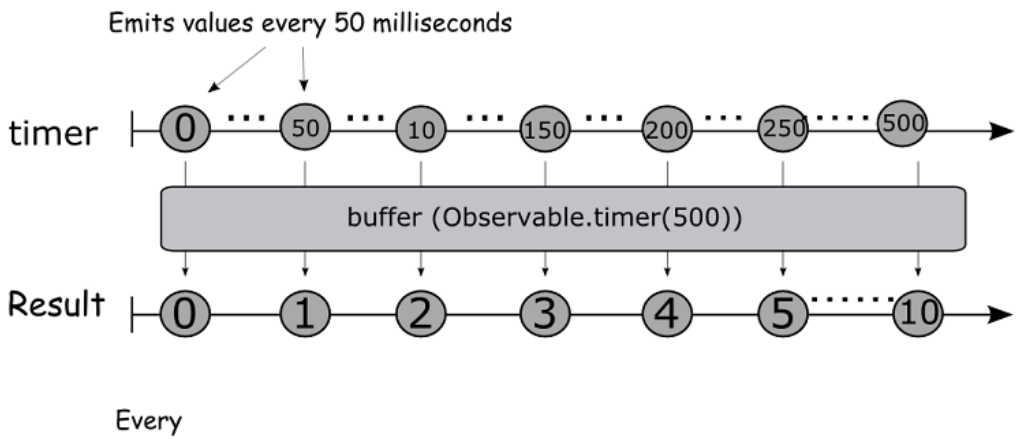

Figure 4.19 Shows a timer with a 50 millisecond period emitted values every millisecond.

This can be implemented with the following code:

```
Rx.Observable.timer(0, 50)
  .buffer(Rx.Observable.timer(500))            ❶
  .subscribe(
    function (val) {
      console.log(`Data in buffer: [${val}]`); ❷
  });
//-> "Data in buffer: [0,1,2,3,4,5,6,7,8,9]"
```

❶ **Buffer uses a "closing observable" which is simply the criteria when which to stop buffering data, in this case after 500 ms of caching events.**

❷ **Buffer returns an array of observable sequences**

The fact that buffers emit an array of observables is analogous to the iterator example we discussed in chapter 2 because the client of the iterator (the `for…of` loop) would receive arrays of data as each element. In that example, creating a `BufferedIterator(3)` would yield

arrays of triplets at each iteration (or at each call to `next()`). The best way to illustrate this is by using the `bufferCount()` operator.

`bufferCount()` retains a certain amount of data at a time, which you define by passing a size. Once this number is reached, the data is emitted and a new buffer started. We'll use a buffer count to display a warning message for numerical inputs that involve large quantities (say, 5 digits or more). We'll listen for changes on the amount field so that we can display a warning next to it when the amount value 5 digits long:

```
const amountTextBox = document.querySelector('#amount');
const warningMessage = document.querySelector('#amount-warning');

Rx.Observable.fromEvent(amountTextBox, 'keyup') (1)
   .bufferCount(5)                              (2)
   .map(events => events[0].target.value)       (3)
   .map(val => parseInt(val, 10))               (4)
   .filter(val => !Number.isNaN(val))           (5)
   .subscribe(amount => {
      warningMessage.setAttribute('style', 'display: inline;'); (6)
    });
```

(1) Listen for key events on the amount text box
(2) Buffer a total of 5 events at a time
(3) Extract the value of the input box. Because buffer returns an array of events, it's sufficient to just use the first event's target DOM element that received the key input
(4) Ensure the amount entered is numeric
(5) Filter out any empty numbers
(6) Display the warning to alert user he or she has entered a large quantity

As you can see a buffer is in some ways similar to a delay. But nothing relates buffers and time more than the `bufferWhen()` and `bufferTime()` operators. We'll begin with `bufferWhen()`. This operator is really useful for caching events until another observable emits a value. This operator differs slightly from `buffer()` in that it takes a selector function that creates an observable to signal when the buffer should emit. It doesn't just buffer at the pace of a subordinate observable, it calls this factory function to create the observable that dictates when it should emit, reset, and create new buffer. For example, suppose you're implementing form fields that can keep track of previously entered values, so that you could revert to a value you had entered before changing it. First, let's see a marble diagram for this entire interaction in figure 4.20:

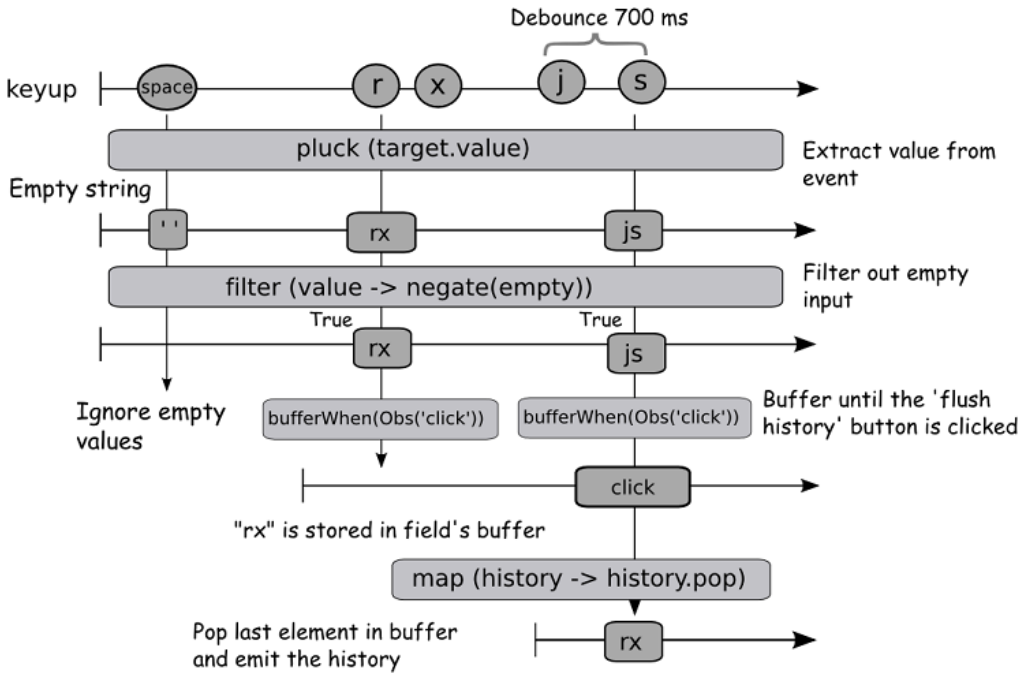

Figure 4.20 Shows a stream listening for key presses, debounced in order to capture the event a little over half a second. Any word entered is passed through a simple validation check. The data fills the buffer and only flushed when requesting the history. This observable closes off the buffer and prints the history

The goal of this program is that as values are entered into a form field, we're actually keeping them in a buffer so that you can always go back to any of them if you wished to do so. As you can see from listing 4.14 ahead, in this book we'll occasionally sprinkle a functional library called Ramda.js to replace utility functions like input validation, array manipulation, and others we would otherwise have to write ourselves (for details about installing Ramda, please visit appendix B). We also do it to show you how well RxJS interacts with functional libraries you may use (in chapter 10, "RxJS in the Wild," expect to see a lot of this interaction).

Ramda loads its arsenal of functions under a single namespace, `R`. One special function you'll come to learn and love is `R.compose()` for functional composition. Combined with RxJS that is great at managing state, this creates the perfect combination of functional and reactive programming (FRP). Here's the code to implement this business logic:

Listing 4.14 Creating form field history with bufferWhen()

```
const field = document.querySelector('.form-field');
const showHistoryButton = document.querySelector('#show-history');
const historyPanel = document.querySelector('#history');

const showHistory$ = Rx.Observable.fromEvent(showHistoryButton, 'click');
```

```
Rx.Observable.fromEvent(field, 'keyup')
.debounceTime(700)
.pluck('target', 'value')
.filter(R.compose(R.not, R.isEmpty)) ❶
.bufferWhen(() => showHistory$) ❷
.do(history => history.pop())
.subscribe(history => {
    let contents = '';
    if(history.length > 0) { ❸
       for(let item of history) {
         contents += '<li>' + item + '</li>';
       }
       historyPanel.innerHTML = contents;
    }
});
```

❶ Validate the input field by composing Ramda functions to check for non-empty
❷ Signal that the stream should clear the buffer when the history button observable emits a value
❸ Print the history next to the button

Pay attention to how the composition of `R.isEmpty()` and `R.not()` creates the behavior for "non-empty". Always remember that functional programming is powerful when we can create and combine functions with very minimal logic, that together solve complex tasks. We'll use Ramda again in the next code listing so that you can see how it benefits the readability of your code.

Finally, we'll take a look at `bufferTime()`. This operator holds on to data from an observable sequence for a specific period of time, then emits it as an observable array. You can think of this as being equivalent to a `bufferWhen()` operator combined with an `Rx.Observable.timer()`. Let's see this in action in listing 4.15. We're going to run a buffer with a timer in the background to monitor a form with two text boxes. When both have been filled in, we'll highlight the submit button in the form to indicate it's ready for submission. Essentially, the logic revolves around caching the outputs of the text fields, and when the buffer stores two valid values, we'll highlight the submit button.

Another twist we'll add to this code in preparation for the next chapters is the use of the `combineLatest()` operator to combine the outputs from two independent streams. Let's take a look:

Listing 4.15 Buffering events in for a specific period of time

```
const field1 = document.querySelector('#form-field-1');
const field2 = document.querySelector('#form-field-2');
const submit = document.querySelector('#submit');

const createField$ = elem =>  ❶
  Rx.Observable.fromEvent(elem, 'change')
    .pluck('target', 'value');

Rx.Observable.combineLatest(  ❷
    createField$(field1), createField$(field2))
.bufferTime(5000)             ❸
```

```
.map(R.compose(R.filter(R.compose(R.not, R.isEmpty)), R.flatten)) ❹
.subscribe(fields => {
    if(fields.length === 2) {
       submit.setAttribute('style', 'background-color: yellow');
    }
    else {
       submit.removeAttribute('style');
    }
});
```

❶ Function that creates observables that emit when the value of a textbox changes
❷ Combine the events emitted from both fields simultaneously (we'll learn about this operator later in the book)
❸ Buffer input for 5 seconds at a time, then flush it
❹ Compose a series of Ramda functions to normalize the data flowing through the internal buffer. Some of the details needed to understand why we need to do this stem from understanding combineLatest(). You don't need to be too concerned about this at this moment.

With combining buffers and time, it's important for you to understand how your application gets used. Buffers use temporary data structures to cache events that occur in its window of operation. Ten seconds can be a very long time if you're caching hundreds of events per second, like mouse moves or scrolls. So exert caution of not overflowing the amount of memory you have, or your user interface will become unresponsive. In the process of illustrating RxJS concepts, we introduced a new operator called `combineLatest()`, which is used to join multiple streams into one. At this point, you've seen the most frequently used operators that act on a single stream. In chapters 5 and 6, we'll begin to dive deeply into RxJS by taking-on more real world problems, and explain more advanced operators so that you can begin to work with multiple simultaneous streams.

4.6 Summary

- RxJS allows us greater control in manipulating and tracking the flow of time in an application
- Operators can interact with time to change the output of an Observable
- Time can be implicit or explicitly declared when more fine-grained control is needed.
- Implicit time manifests in the latency waiting for asynchronous HTTP calls to respond. You have no control over how long these functions take
- Explicit time is controlled by you and take advantage of JavaScript's timers
- Delay shifts the observable sequence by a due time (in milliseconds).
- Debounce emits an event from an Observable sequence only after a particular timespan (in milliseconds) has passed without it omitting any other item.
- Throttling enforces a maximum number of times a function can be called over time.
- Buffering operations borrow a lot of the same semantics as the timing operations
- RxJS features has size-based as well as time-based buffers

5

Applied Reactive Streams

This chapter covers:

- **Handling multiple observable sequences with one subscription**
- **Learning to make observable streams conformant**
- **Flattening nested observables structures**
- **Merging a collection of observables into a single output**
- **Preserving sequence order with concatenation**
- **Implementing real-world problems: search box, live stock ticker, and drag-and-drop**

In the previous chapter, we firmly rooted notion that an observable is simply a sequence of events over time. You can think about it as the orchestrator or channel through which events are pushed and transformed. So far, we've discussed how to process observable sequences in isolation for the most part, and you learned how you could apply familiar operators over all the elements within an observable in the same way that you could do with an array, irrespective of when those elements were actually emitted. Towards the end, we briefly gave you some exposure to an RxJS operator called `combineLatest()` used in the combination of different streams. The reason for this is that other than very trivial examples, most of your programming tasks will involve the use of these operators so that the events from one stream propagates and causes a reaction to happen somewhere else. This is where RxJS and the entire reactive paradigm begins to shine and set itself superior to other conventional asynchronous libraries.

Now, you have entered into new, more sophisticated territory. This chapter is our point of inflexion where we'll look at combinatorial operators similar to `combineLatest()` like `merge()`, `switch()`, and `concat()`, and others to solve real-world, complex problems. While there's still a lot to gain from using observables to handle a single event source, like a single button click, in all but the most trivial of applications you will quickly find that your logic will require more

than just a single stream to get the job done. You saw a glance of this towards the end of chapter 4 when we implemented a slightly more complicated problem, which required us to combine and buffer the output of multiple streams.

Depending on the complexity of your application, your logic will likely involve the interplay between many observables, potentially containing different types of data. For instance, a user typing keys on the search bar kicks-off an entirely different type of stream for fetching data from a server—to put it in Newton's terms, user actions may create a reaction in different parts of the application. But combining different observables into one presents a challenge because you will have to learn how to fuse them together. What if their interfaces are different? This actually happens quite a bit, especially when modeling complex state machines such as user interfaces or mashing up data from remote locations. Thus, in this chapter we'll focus on a very important principle in functional programming called *flattening a data type*, which stems from the need to project other observables carrying needed data into a single source. After learning this technique, you'll be prepared to tackle all sorts of complicated flows. Let's begin with the idea that all data sources, whether static or dynamic, can be treated in the exact manner under the observable programming model, because *everything is a stream*.

5.1 One for all, and all for one!

Time added a new dimension to the observable which let us delay, filter or manipulate elements based on when they were dispatched. By playing with various types of operators and their corresponding selector functions, we were able to determine if, when, and how events made it downstream; so this gave your business logic full control of how data is transformed within the stream.

Up until now, we've been focusing on single streams. In this chapter we want to expand our use of operators and introduce more advanced ones that will allow you to create complex flows and combine them into a single one. This is essential for any non-trivial state machines where input taken from the user creates a ripple effect on other parts of the system in real time.

Multi-stream scenarios are very common in the real world. Generally, you will find that the more complex your system becomes, the more entangled your streams will be. This linear relation should not surprise you because complexity grows as you inject more business logic into your code. However, this is far more maintainable than the exponential complexity growth of a system that deals with asynchronous flows relying only on nested callback functions, or even solely on promises.

The combination operators you'll learn in this chapter are indispensable because if isolated streams were unable to interact with each other, it would be left up to you to create the necessary boilerplate for those streams to work together. Ugh! We'd like be left, no better off than before we started using RxJS!

REACTIVE MANIFESTO According to the Reactive Manifesto, one of the central principles of reactive architectures is *elasticity*. An elastic system is one that stays responsive under varying work load. RxJS nicely achieves this when multiple sources of data with varying input rates can be combined in different ways without you having to rewrite or refactor how your code works.

As an example of when multiple streams come into play, imagine that in addition to supporting mouse handling we also wanted to support touch interfaces as well. JavaScript already provides built-in support for touch events in most browsers; however, adding touch support to an application means introducing a second set of events and logic. Without the right architecture, you will most likely have to create a whole new set of event handlers for those as well. With your newly developed reactive mindset, you realize that those are all just different streams passing through the same channel. Whether or not you're using mouse or touch, most of the time they kick-start events that need to combine with key presses, and other HTTP calls, timed intervals, animations, and much more—complex UIs work this way.

Let's look at a quick example. Just as mouse events have `mousedown`, `mouseup`, and `mousemove`, touch events have `touchstart`, `touchend`, and `touchmove`, respectively. This means that we will, at a minimum, need to create three new streams in order to emulate the same behaviors from mouse events as with touch events, and have our code work on mobile browsers. Consider the scenario of two independent streams. The `touchend` event is probably equivalent to `mouseup`, which means that for most cases users will use it in the same way:

```
const mouseUp$ = Rx.Observable.fromEvent(document, 'mouseup');

const touchEnd$ = Rx.Observable.fromEvent(document, 'touchend');
```

Each of the streams represents the same action of completing some interaction with your application, whether its through a mouse or the screen. We know from the previous chapters that we can subscribe to each of the handlers individually as such:

```
mouseUp$.subscribe(/* Handle mouse click */);

touchEnd$.subscribe(/* Handle touch click* /);
```

However, this setup is less than ideal. For one, we now have two subscription blocks for what will most likely be identical code. Any code that must be shared between the two will now require a shared external state. Additionally, there are now two subscriptions that must be tracked, which just introduces one more area of potential memory leaks. It would be in many ways preferable if we could manage both subscriptions with a single block of code without having to worry about the synchronization of both streams, as shown in figure 5.1, especially since both observables will emit very similar events, in this case.

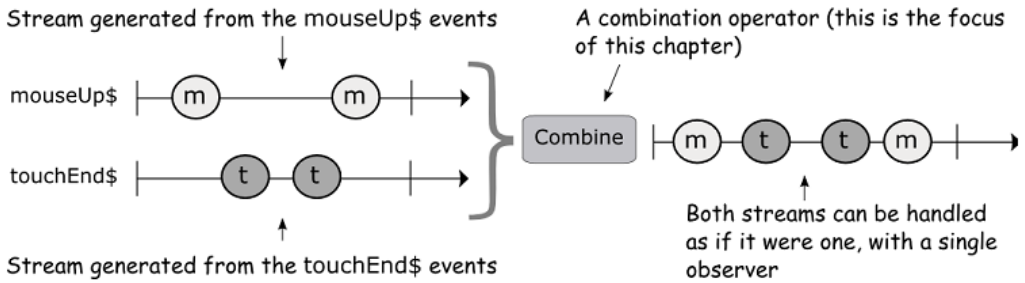

Figure 5.1 A simplified diagram of the desired behavior of ingesting two streams: mouseUp$ and touchEnd$, through a combination operator, and creating a single output block for consumption. In this chapter, you'll learn about the most important combination operators in RxJS

There are many different ways to join multiple streams into one and take advantage of using a single observer to handle them all. In this section we'll look at strategies to:

1. Interleave events by merging streams—useful for simply forwarding events from multiple streams. This is ideal for handling different types of user interaction events like mouse or touch.
2. Preserve order of events by concatenating streams—used when the order of the events emitted by multiple streams needs to be preserved.
3. Switch to the latest stream data—used when one type of event kicks-off another, such as a button click initiating a remote HTTP call, or begin a timer.

5.1.1 Interleave events by merging streams

The simplest operator that combines multiple streams together is `merge()`. It's very intuitive to understand; it's purpose is to forward the events from several streams in order of arrival into one resulting observable, like a funnel. Logically, you can think of merge as performing an OR of the two events:

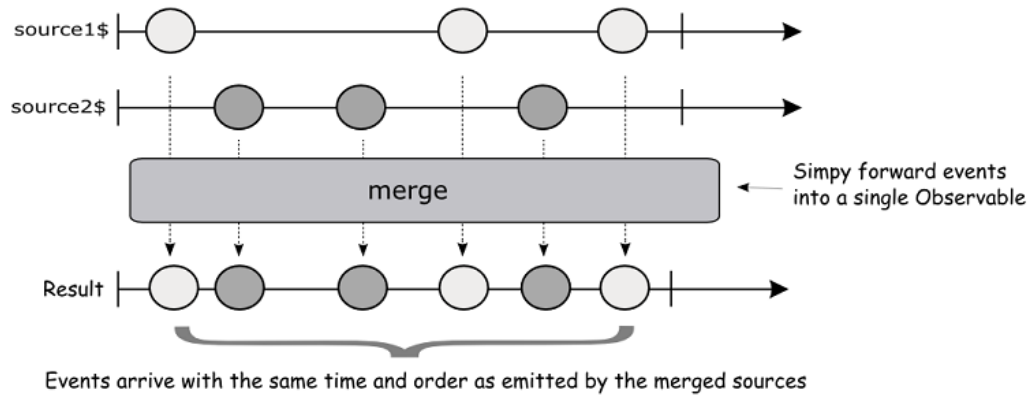

Figure 5.2 Merge operator has no logic of its own, other than simply combine events from multiple streams in the order they are emitted

We can illustrate this with a simple example:

```
const source1$ = Rx.Observable.interval(1000)
   .map(x => `Source 1 ${x}`)
   .take(3);
const source2$ = Rx.Observable.interval(1000)
   .map(y => `Source 2 ${y}`)
   .take(3);

Rx.Observable.merge(source1$, source2$)
   .subscribe(console.log);
```

The resulting stream created from merge will emit values every second alternating between sources 1 and 2. Merge has no logic of its own other than placing the events onto a single stream in the order they arrive.

In the example of mouse and touch streams, since the two types are virtually interchangeable, we can funnel both through this operator so that we can react to either one in the exact same way, or with the same observer logic. Just like most of the operators you'll learn about soon, `merge()` can be found in the static method form:

```
Rx.Observable.merge(source1$, source2$, ...)
```

Or in instance form:

```
source1$.merge(source$2).merge(...)
```

In the static form, it's a creational operator as we showed in chapter 4. In other words, you are creating a new stream from the combination of two or more observables. Using it in instance form, we say that an observable is *projected or mapped to* the source observable. Both yield the same results, and because operators are pure, both create new observables.

Merge can accept either an array or a variable number of streams that are to have their outputs merged together into one. Thus for the previous example, we can create a merged stream easily by just passing both sources into the merge operator, as shown in figure 5.3 for both static and instance forms:

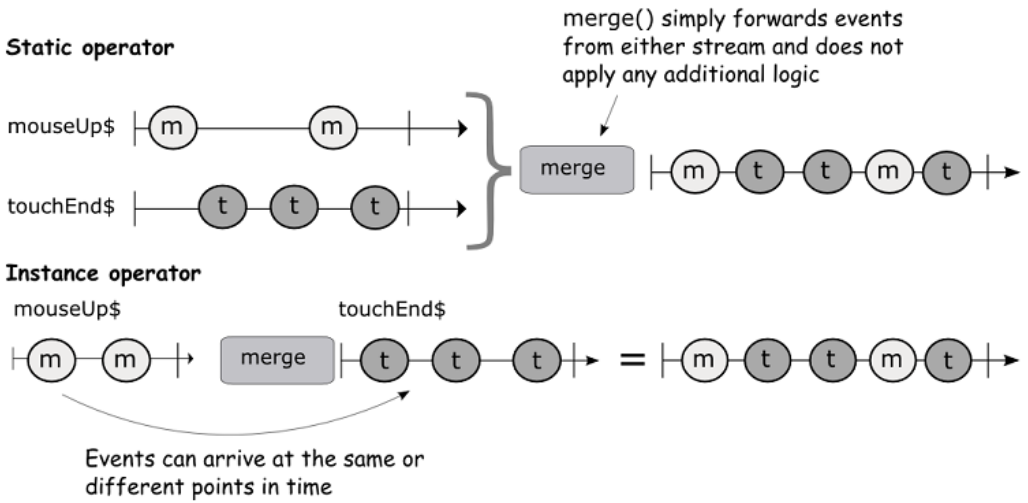

Figure 5.3 Static and instance versions of the merge operator between two streams. The results are exactly the same. Merge is the simplest operator because it only forwards events from either stream as they come.

Here's the code that combines both streams, in creational form:

```
Rx.Observable.merge(mouseUp$, touchEnd$)
    .subscribe(/* single observer to consume either event */);
```

Again, we can also write this code where a source observable merges onto another in midstream, in instance form:

```
mouseUp$.merge(touchEnd$)  ❶
    .subscribe(/*single observer to consume either event */);
```

❶ In this case, both mouseUp$ and touchEnd$ are still referred to as the source observables.

The outcome of merging two observables is that they will now appear as a single one from the point of view of any instance methods used all the way down to the observer; so all of their outputs are piped to a single output block. This is true for all of the combinatorial operators you'll learn about in this chapter. We now only need to worry about a single subscription for the two streams.

Now something to think about when merging streams is order. The merge operator will interleave events from each stream in the order the events are received; internally, RxJS does a very good job of time stamping each event that gets pushed through the observable.

Each stream, though independent, contributes to the overall output of the sequence. In more statically typed languages, the compiler will also often constrain the types that can go into a merge. This forces the input streams to have a uniform and predicable type for the output. In JavaScript, because these constraints don't exist it is much easier to merge types that may not even be compatible. However, this flexibility can result in some unexpected errors downstream when you're looking to handle one type of event differently than the other. For this, one thing you could do is check your use cases to determine the type as in listing 5.1. Again going back to the touch and click streams, suppose we need to print out the coordinates of both touch and click events:

Listing 5.1 Case matching event data resulting from merging mouse and touch streams

```
Rx.Observable.merge(mouseUp$, touchEnd$)
 .do(event => console.log(event.type))
 .map(event => {
     switch(event.type) {
        case 'touchend':
           return {left: event.changedTouches[0].clientX,
                   top: event.changedTouches[0].clientY};
        case 'mouseup':
           return {left: event.clientX,
                   top:  event.clientY};
     }
 })  .subscribe(obj =>
     console.log(`Left: ${obj.left}, Top: ${obj.top}`));
```

1. Merges the outputs of the two streams
2. Detect the type of the event and build a compatible type and construct the object accordingly

However, this sort of behavior should raise some flags as a potential code smell. One of the main reasons to combine observables is that they possess some similarities that we would like to leverage into simpler code. Introducing more boilerplate code into the mix to then have to switch on type does not serve this purpose; it simply moves the complexity to a different location.

On another note, keep in mind that in functional programming, we'll try to avoid imperative control statements, like if/else whenever possible. Instead, each event should be at least contract compatible with one another, which means that the data emitted from all of them should at least follow the same protocols or structure in order to be consumed by the same observer code. Now as to why we should avoid imperative control structures as much as possible has more to do with the notion of using RxJS in a functional style and also to continue to fluent declaration of an observable chain.

> **FUNCTIONAL VS IMPERATIVE** Generally speaking, most of the cases involving using structures like for/of or if/else can be satisfied by using RxJS operators like `filter()`, `map()`, `reduce()`, `take()`, `first()`, `last()`, and others; in other words, most of the complex tasks for which you need to implement loops and branches are just instances of a certain combination of these operators. A familiar example of this is

an array. Most uses for/of and if/else statements can be supplanted by a combination of `map()` and `filter()`.

In cases where you absolutely need to apply specific logic, you can insert operators upstream of where the merge occurs in order to create compatible types ahead of time. So, instead of forcing observers to use conditional logic to discern between different types of events (figure 5.4):

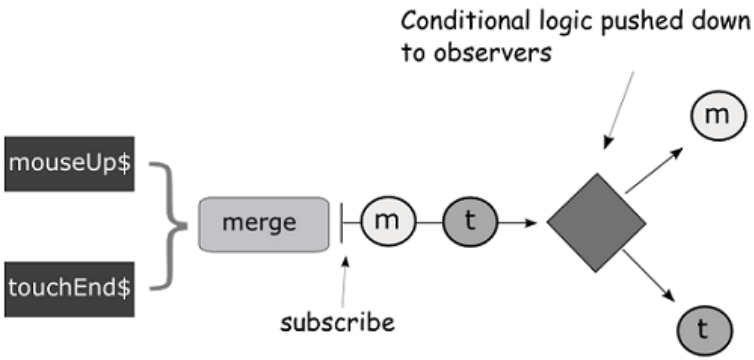

Figure 5.4 The logic decide how to handle events from the merged streams is pushed down to the observer. The observer code at this point could use if/else or switch blocks based on type.

Alternatively, you should instead make the stream data conformant before the merge, as seen in figure 5.5, to make your subscribers happier by avoiding any checks:

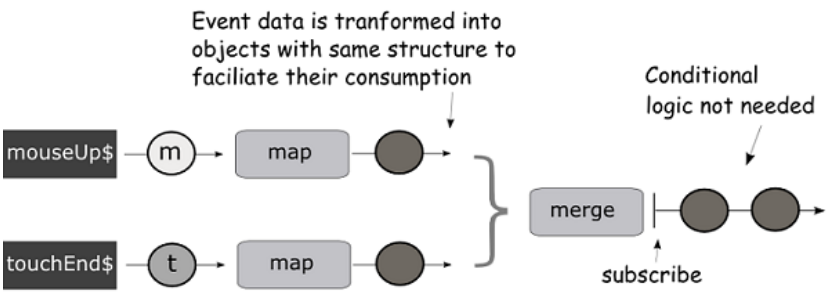

Figure 5.5 Applying an upstream transformation to both streams to normalize the data and facilitate the observer code.

Listing 5.2 shows how we can do this with our mouse clicks and touch events example. As you can imagine, the touch interface does not exactly correspond to that of the mouse interface; so to make them work together we need to normalize the events so that we can reuse the

same functions to handle the merged stream. We do this in listing 5.2 by creating conformant streams or streams that emit data with similar structure:

Listing 5.2 Normalizing event data upstream become merging the streams

```
const conformantMouseUp$ = mouseUp$.map(event => ({          ❶
  left: event.clientX,
  top: event.clientY
}));

const conformantTouchEnd$ = touchEnd$.map(event => ({        ❶
  left: event.changedTouches[0].clientX,
  top: event.changedTouches[0].clientY,
}));

Rx.Observable.merge(conformantMouseUp$, conformantTouchEnd$)
   .subscribe(obj =>
      console.log(`Left: ${obj.left}, Top: ${obj.top}`)); ❷
```

❶ Convert each type upstream before it is merged into the final stream
❷ Merge the converted streams, the observer logic stays the same

It may seem more verbose to create our separate streams, but as the complexity of the application grows, updating and managing a switch statement will become a tedious upkeep process. By requiring that the source observables are all conformant with the expected output interface, we can more easily expand this and continue adding more logic as needed, in case we wanted to include streams generated from, say, the `pointerup` event of a pointer device, like a pen (https://w3c.github.io/pointerevents). The pointer event commands may have a completely different interface from either the mouse click or the touch events, but we can still use it to control our application by forcing the incoming stream to conform to the interface beforehand expected by the subscribe block.

One important point about merge, however, that could be confusing to an RxJS beginner is that it will emit all of the data that's immediately present in memory from any of merged observables. The interleaving happens when events arrive asynchronously, like with `interval()` and mouse movements; but when the data is synchronously loaded it will emit all of one stream before emitting the next. Consider this code:

```
const source1$ = Rx.Observable.of(1, 2, 3);
const source2$ = Rx.Observable.of('a', 'b', 'c');
Rx.Observable.merge(source1$, source2$).subscribe(console.log);
```

From what you just learned, you might think `merge()` alternates between numbers and letters, but the result is that it iterates through all numbers first, and then all letters. This is because the data is synchronously available to emit. The same would happen whether you were passing scalar values at a time, whole arrays, or generators.

Whether data is synchronous or asynchronous, if your goal is to preserve the order of events in the combined streams, then you need to use concatenation.

5.1.2 Preserve order of events by concatenating streams

The RxJS `merge()` operator uses a naïve strategy of outputting all the observable data in the order that events are received from the source streams. However, in other scenarios we might be more interested in preserving the order of the entire observable sequences when we join them, instead of interleaving them. That is, given two observables, we might want to receive all the events from `source1$` then all the events from `source2$`. This is useful in cases when you would like to give priority to one of type of event versus another. We refer to this type of operation as a concatenation of the two streams.

Just like we can concatenate two strings or two arrays together, we can also concatenate two streams that will generate a brand new observable made from the events of both constituent observables—similar to a set union operation. The signature is almost identical to the merge operator.

```
const source$ = Rx.Observable.concat(...streams)
```

The behavior of this new observable operator is shown in figure 5.6:

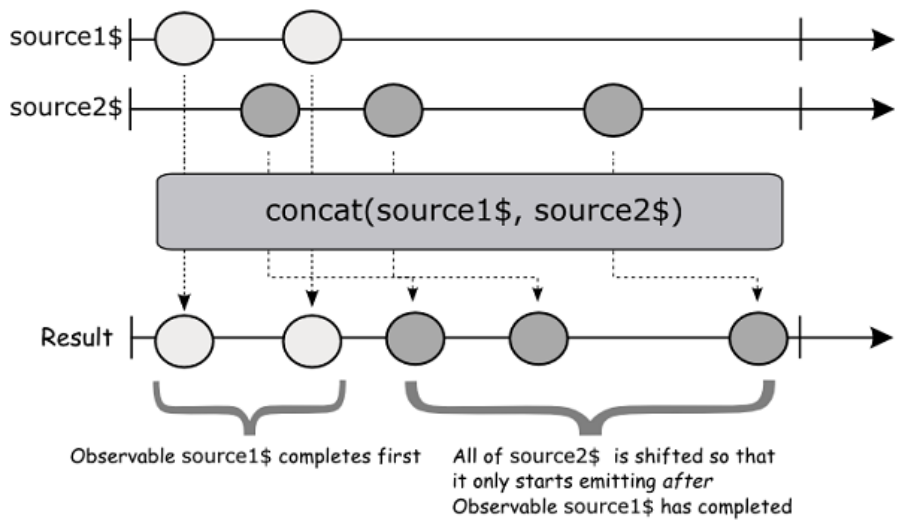

Figure 5.6 A marble diagram of the concat() operator which waits for the first Observable to complete before subscribing to the next one. In concatenation, the data from both observables are appended rather than interleaved.`

It's important to note that a merge differs from a concatenation on one key behavior: while the `merge()` operator will allow you to immediately subscribe to all of the source observables, `concat()` will only subscribe *to one observable at a time*. While it continues to manage the subscriptions to each of the underlying streams, it will only hold a single subscription at a time

and process that before the next one in order. You can see this behavior clearly with this simple example:

```
const source1$ = Rx.Observable.range(1, 3).delay(3000);  ❶
const source2$ = Rx.Observable.of('a', 'b', 'c'); ❷
const result = Rx.Observable.concat(source1$, source2$);
result.subscribe(console.log);
```

❶ First stream is delayed by 3 seconds, which means it will start emitting after 3 second
❷ Second stream is set to three letter immediately

Intuitively, because the second stream is delayed by 3 seconds you would expect to see the letter "a," "b," and "c" emitted first. But because the `concat()` keeps the order it will complete the first stream first before appending the next stream. So you would see the numbers 1, 2, and 3 before the letters:

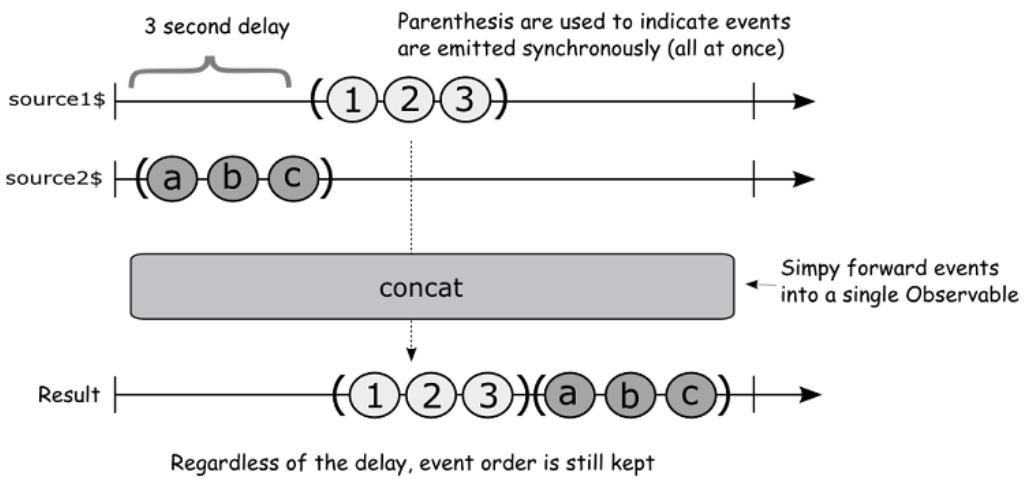

Figure 5.7 Marble diagram showing the interaction of sources with delay. Regardless of delay offsetting the entire observable sequence, the order of the events in a stream is preserved using concat

Consequently, if we were to take the example of merging the touch events and the mouse events for instance, changing from `merge()` to `concat()` would result in very different (possibly undesired) behavior:

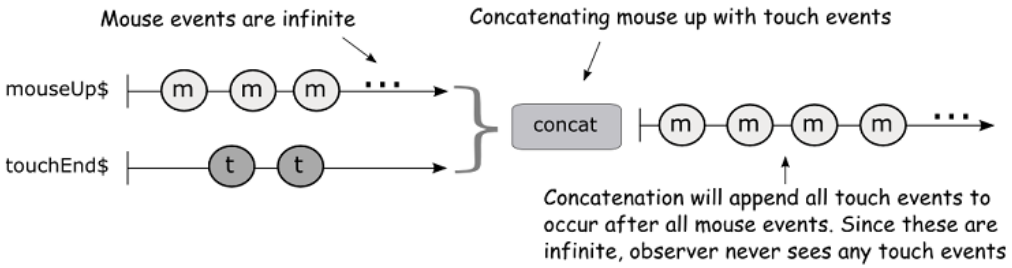

Figure 5.8 The concatenation of an infinite stream with any other will only ever emit event from the first.

Keep in mind that `concat()` begins emitting data from a second observable once the first one completes (fires the `complete()` observer method). But this actually never occurs, because all DOM events (including touch) are essentially infinite streams. In other words, you'll only ever process observables from `mouseUp$` under normal circumstances. In order to see later concatenated observables, we would need to terminate or cancel `mouseUp$` gracefully at some point in the pipeline. For instance, we could use `take()`, which you learned about in chapter 3, to have `mouseUp$` complete (made finite) after a set number of events were emitted. Say we take only the first 50 events from the mouse interface:

```
Rx.Observable.concat(mouseUp$.take(50), touchEnd$)
    .subscribe(event => console.log(event.type));
```

Voila! Now you'll see touch events after we have received first 50 mouse events. In the case of the mouse and touch streams, the concatenation operation works a little bit differently to the merge mechanism. This has to do with a basic distinction between two types of observables known as *hot* and *cold*, which we'll discuss in chapter 8. In this case, when we do begin receiving touch events they are only for events that occurred *after the end* of the mouse events, and not from the beginning offset to the end. Essentially, this use of `concat()` is the equivalent to a nested subscription of the form:

```
mouseUp$
  .take(50)
  .subscribe(
     function next(event) {
       console.log(event.type); (1)
     },
     function error(e) {
       console.log(e);
     },
     function complete() {
       touchEnd$.subscribe(  (2)
          event => console.log(event.type)
       );
     }
   );
```

(1) Handle the first 50 mouse events

❷ Start the touch stream when mouse events finish. Any touch events that occurred prior to the first 50 mouse events are missed

Which would look like this in a diagram:

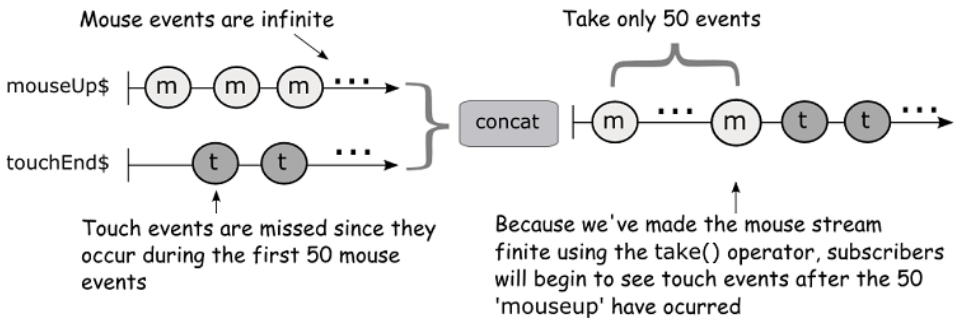

Figure 5.9 Because we've made the mouseUp$ stream finite, subscribers will see data emitted from touchEnd$ after the first 50 'mouseup' events

BEST PRACTICE We show this sample code above merely to illustrate the behavior of the `concat()` operator in this example. We're not, however, promoting that you nest subscriptions this way, which is tempting for new RxJS users. Nesting subscriptions breaks the downstream flow of data from one observable to the next, which an anti-pattern in the RxJS model. Also, not only are you duplicating your efforts by having to write the same observer code in two difference places, but you hamper RxJS' optimizations in the pipeline chain in its ability to reuse internal data structures and lazy evaluation.

As you can see, the type of strategy employed will make a huge difference on how a downstream observable will behave. It is therefore important to understand that when we combine streams there's a difference between when a stream is created and when a subscriber subscribes to it. We can use `merge()` when we want to receive the latest event from any observable as they're emitted, while `concat()` is useful if we wish to preserve the absolute ordering between observables.

As we mentioned briefly, there will be cases when you'll need to cancel out one of the observables and receive only data from another one. Let's take a look at another combinator, `switch()`.

5.1.3 Switch to the latest observable data

Both merge and concatenate propagate the input stream data forward into the pipeline. But suppose we wanted a different behavior, such as cancelling the first sequence when a new one begins emitting. Let's study this simple example:

```
Rx.Observable.fromEvent(document, 'click')  ❶
   .map(click => Rx.Observable.range(1, 3)) ❷
   .switch()  ❸
   .subscribe(console.log);
```

1. Listen for any clicks on the page
2. Map another observable to the source observable
3. Use switch to begin emitting data from the projected observable

Running this code logs the numbers 1, 2, 3 to the console after the mouse is clicked. In essence, what's happening is that when the click event occurs, this event is canceled and replaced by another observable with numbers 1 through 3. In other words, subscribers never actually see the click event come in—a switch happened.

Switch only occurs as an instance operator and it's one of the hardest ones to understand because it actually does carry a bit of logic of its own. As you can see from the code above, switch takes another observable that has been mapped to the source observable and fuses it into the source observable. The caveat is that, each time the source observable emits, it immediately unsubscribes from it and begins emitting events from the latest observable that was mapped. To show case the difference, consider what the same code would look like using `merge` instead:

```
Rx.Observable.fromEvent(document, 'click')
   .merge(Rx.Observable.range(1, 3))
   .subscribe(console.log);
```

In this case, because the source observable (click events) is not cancelled, observers will receive click events mixed with numbers from 1 through 3. Hence, this sort of behavior can be useful when one stream (like a button click, for instance) is used to initiate another stream. At this point, there's no interest in listening for the original stream's data (the button clicks). So, `switch()` is also a suitable operator for our suggestion search stream:

```
const search$ = Rx.Observable.fromEvent(inputText, 'keyup')
  .pluck('target', 'value')
  .debounceTime(1000)
  .do(query => console.log(`Querying for ${query}...`))
  .map(query =>
         sendRequest(testData, query)))
  .switch()  (1)
  .forEach(...);
```

1. Switch operator causes the observable stream to switch to the projected observable

The reason for this is that as soon as they `keyup` event fired, there is no need in really pushing that event onto the observers; instead, it's best to switch to the search results streams and push that data downstream to any subscribers, which is what gets displayed as search results. As you can see in the code above, before calling `switch()`, we mapped (or projected) an observable to the source. This terminology is important, and you'll see us repeat it quite a bit in this chapter. At this point, `switch()` will cancel the previous subscription when the new one comes in. In this way, the downstream observable is always guaranteed to have the latest data by removing operations that have become stale or no longer of interest. We can visualize this interaction with figure 5.10:

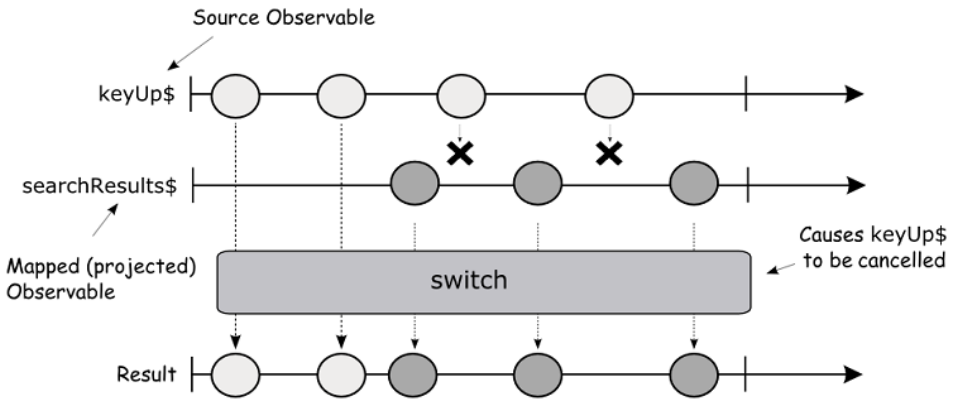

Figure 5.10 A marble diagram of the switch() operator which only allows the most recently received Observable to run.

In using the latest value, we guarantee that there will be no wasted cycles spent on data that will be overridden immediately when new data arrives, and we don't have to guard against that same data coming back out of order. Also, the observers don't need to worry handling events from key presses since those are not of interest and have been suppressed by `switch()`. Since network requests can be processed and returned out of order, in certain scenarios we could see earlier requests arrive *after* later ones. If those request were not excluded from the final stream, then we could see old data overwriting newer data which is obviously not the desired behavior.

> **DISCLAIMER** If `switch()` is turning out to be somewhat confusing to understand at this point, there's a reason for this. Internally, this operator is addressing a fundamental need in functional programs which is to flatten nested context types into a single one. In our case the Observable that's mapped to our source internally creates a nested observable object. You'll learn about this in the next section in the form of a single `swithMap()`, and everything will be clear.

These three operators: `merge()`, `concat()`, and `switch()` grouped observable data in a flat manner, each with its own different strategy. In other words, you're either receiving events from one source or another and processing them accordingly. All of these operators have a what are known as *higher-order observable* equivalents, which handle nested observables.

> **TERMINOLOGY** The term higher-order observable originates from the notion of a higher-order function, which can receive other functions as parameters. In this case, we talk about observables that receive other observables as arguments.

At this point in the book is when you'll make the leap from a novice RxJS developer to a practitioner, and the techniques we're about to learn will drastically shape the way in which

you approach reactive and functional programs. We began with the idea that all types of data are treated equally when wrapped with an observable, this also includes the observable itself.

5.2 Unwinding nested observables: the case of mergeMap

The previous section detailed how streams can have their outputs combined simultaneously. In each case, the input consisted of several streams funneled into a single output stream. Depending on the strategy of combination we can extract different behaviors from observables as you saw in the previous section; however, there are also cases that occur frequently where an observable emits other observables of its own, a situation we call *nesting the observable* that can be visualized in the following manner:

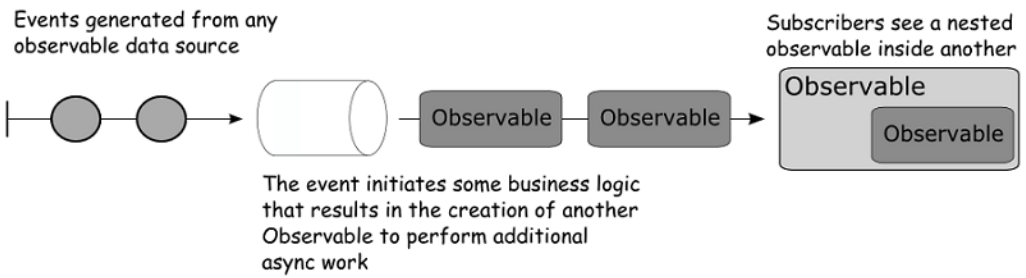

Figure 5.11 Shows a nested observable structure that occurs when subsequence observables are created within the one observable's pipeline

Nesting observables is useful when certain actions result or initiate subsequent asynchronous operations whose results need to be brought back into the source observable. Recall that when we `map()` a function to an observable, it is the result of this function (internally passed through to `subscriber.next()`) that gets wrapped into another observable and propagated through. But what happens when the function we're trying to map, also returns an observable? In other words, instead of mapping a function that returns some scalar value as we've been doing along, the mapped function returned another observable—known as *projecting an observable onto another* or *an observable of observables*. This is, by no means, a flaw in the design of RxJS; it's actually an expected and desired behavior.

In fact, this situation arises very frequently in the world of functional programming because the protocol of `map()` is that it's a *structure preserving* function (for example `Array.map()` also returns a new array). This is the contractual obligation of the implementing data type, in this case the observable, so that it retains its functor-like behavior. Again, we don't cover functors in this book because it's a more advanced functional programming topic. Suffice to say that RxJS has already implemented this for you because the `Rx.Observable` type, as you might have guessed, behaves very much like a functor.

Let's see how this can manifest itself in a real-world scenario. In chapter 4, we implemented a simple search box that streamed data from a small, static data set. In that

version, we used `map()` to transform a stream carrying a key word entered by the user into an array of search results matching said term. We'll show the relevant parts of it:

```
const search$ = Rx.Observable.fromEvent(inputText, 'keyup')
  ...
  .map(query => sendRequest(testData, query)))
  .forEach(...)
```

But considered what would have happened, if `sendRequest()` were to return an observable object, as it would if it were actually invoking an AJAX call. As you know, all operators create new observable instances; as a result, mapping this function will produce values of type:

```
Observable<Observable<Array>> (1)
```

(1) **Now subscribers need to deal with Observables<Array> directly instead of the data that's contained within it. This is not the desired behavior.**

The `search$` stream is now essentially a nested observable, as shown in figure 5.12. However, observers shouldn't be reacting to a layer of wrapped observable values, this is unnecessary exposure or lack of proper encapsulation, they should always receive the underlying data that resulted from applying all of your business rules. So we need to somehow flatten or unwind these layers into a single one:

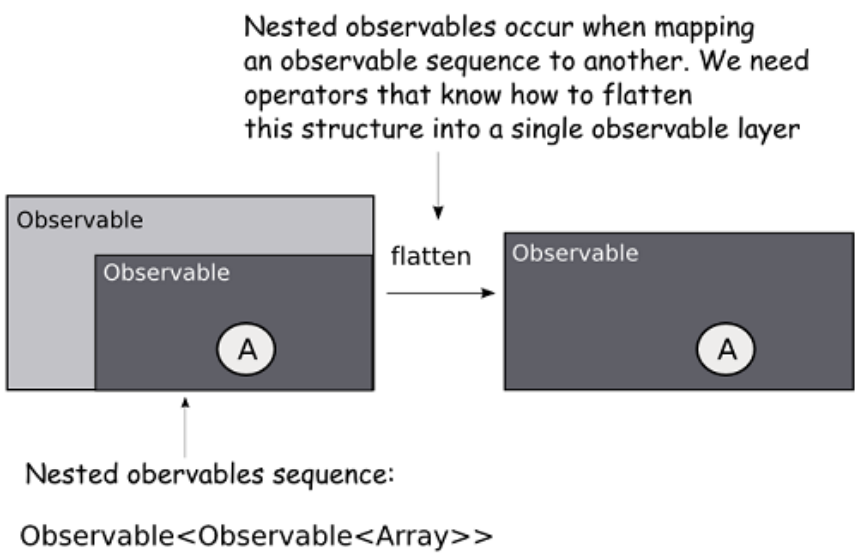

Figure 5.12 Shows that mapping an observable-returning function to a source observable yields a nested observable type. Switch can be used to flatten this structure back to a single Observable.

We can do this in two ways: either we can perform a nested subscribe, wherein we subscribe to the inner observable in the parent's subscription block (bad idea for the same reasons

discussed earlier), or we can merge the streams such that `search$` actually appears as a simple stream of search results to any subscriber (better idea). To get us there we need to learn and master the `mergeMap()` operator (previously known as `flatMap()` in RxJS 4).

Unlike, `merge()` and `concat()`, `mergeMap()` and others you'll learn about shortly have additional logic under the hood to compress the inner observable back into a single observable structure (we'll learn what this means in a bit):

```
const search$ = Rx.Observable.fromEvent(inputText, 'keyup')
  ...
  .mergeMap(query => sendRequest(testData, query))) ❶
  .forEach(...)


search$; //-> Observable<Array> ❷
```

❶ **Merge the outputs together and switch to the observable values emitted from the query results**
❷ **Subscribers of this stream will deal only with values of type T**

The same would hold if you wrapped a function that returns a non-observable value directly as part of the mapping function, like so:

```
const search$ = Rx.Observable.fromEvent(inputText, 'keyup')
  ...
  .mergeMap(query => Rx.Observable.from(queryResults(query)));
```

Both of these cases occur very often. This makes this stream much more useful in that we can now reason about the keystrokes as though they directly mapped to our search results, without worrying about the mess of asynchrony in between and nested structures. Now that we understand `mergeMap()`, let's collect all of the pieces and come up with the full solution to the search program that queries real data from Wikipedia. We will also introduce a few other helpful operators along the way. Because it involves making an AJAX call, we'll make use RxJS' version of `ajax()`, as well as `appendResults()`, `clearResults()`, and `notEmpty()` from the initial search code. Listing 5.3 combines most of the concepts you've learned up to now, including debouncing, into a single program:

Listing 5.3 Reactive search solution

```
const searchBox = document.querySelector('#search'); //-> <input>
const results = document.querySelector('#results');  //-> <ul>
const count = document.querySelector('#count');      //-> <label>

const URL = 'https://en.wikipedia.org/w/api.php
           ?action=query&format=json&list=search&utf8=1&srsearch='; ❶

const search$ = Rx.Observable.fromEvent(searchBox, 'keyup')
  .pluck('target','value')
  .debounceTime(500)
  .filter(notEmpty)
  .do(term => console.log(`Searching with term ${term}`))
  .map(query => URL + query)
```

```
  .mergeMap(query => Rx.Observable.ajax(query) ❷
        .pluck('response', 'query', 'search')
        .defaultIfEmpty([]))        ❸
  .mergeMap(R.map(R.prop('title'))) ❹
  .subscribe(arr => {
    count.innerHTML = `${arr.length} results`;
    if(arr.length === 0) {
      clearResults(results);
    }
    else {
      appendResults(arr, results);
    }
  });
```

❶ Wikipedia's API URL
❷ Mapping an observable-returning function and flatting it (or merging it) into the source observable.
❸ If the result of the AJAX call happens to be an empty object, convert it to an empty array by default
❹ Extract all title properties of the resulting response array

Here's how running this program looks with a little CSS:

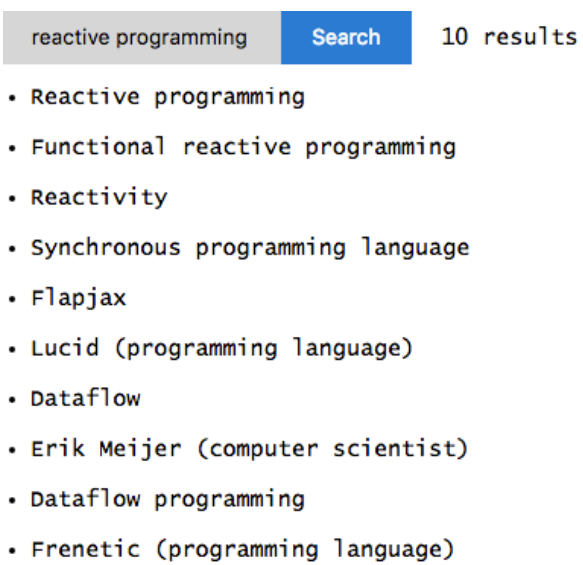

Figure 5.13 Reactive search program querying for "reactive programming"

We just implemented our first sample program, and despite the added complexity we still hold true to all the principles of immutability and side-effect-free coding. We'll continue to tackle problems of different types as you get comfortable with the RxJS operators. This will prepare for chapter 10, which will show you and complete end-to-end application that integrates RxJS into a React, to create a fully reactive solution.

Where does the notion of flattening data come from?

The notion of flattening data comes from the world of arrays when you need to reduce the levels of a multi-dimensional array. For example:

```
[[0, 1], [2, 3], [4, 5]].flatten(); //-> [0, 1, 2, 3, 4, 5]
```

Now, JavaScript's doesn't actually have an Array flatten method, but you can easily use reduce with subsequence array concatenation to achieve the same effect:

```
[[0, 1], [2, 3], [4, 5]].reduce(function(a, b) {
  return a.concat(b);
}, []); //-> [0, 1, 2, 3, 4, 5]
```

As you can see, flattening is achieved by repeated concatenation at each nested level of the multidimensional array. For this reason, `flatten()` also goes by the name `concatAll()`.

Instead of rolling your own, use a functional library like Ramda.js to achieve this:

```
R.flatten([[0, 1], [2, 3], [4, 5]]); //-> [0, 1, 2, 3, 4, 5]
```

If you come from an object-oriented background, you might not be accustomed to flattening data structures as a core operation. The reason for doing this is because observables manage and control the data that flows through them—this idea is known as *containerizing data*. There are various benefits to doing this, some of which you've already been exposed to such as: enforcing immutability and side effect free code, data handling abstractions to support many event types seamlessly, and fluent operator chaining declaratively using higher-order functions. You want to have all of these benefits regardless of how complex or deep your observable graph is.

Working with nested observables is analogous to inserting an array into another array, and expecting `map()` and `filter()` to work with just the data inside of them. Now, instead of you having to deal with the nested structure explicitly, you allow operators to take care of this for you so that you can reason about your code better. Graphically, the notion of flattening observables really means extracting the value from a nested observable and consolidating the nested structure so that the user consumers only ever see one level. This process is shown in figure 5.14:

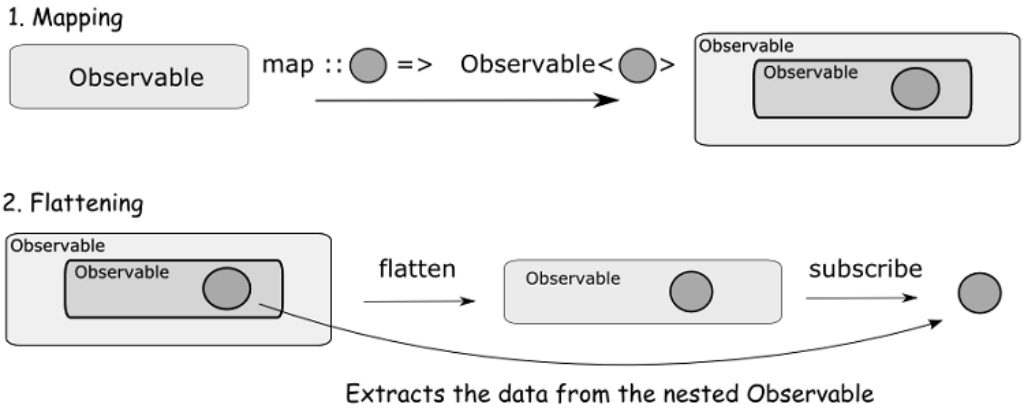

Figure 5.14 Shows the process of mapping a function onto an Observable, resulting in a nested Observable, then flattening to extract its value. We do this, so that subscribers only deal with the processed data. Here, flatten can be any one the RxJS operators such as switch() or merge().

We'll show an example that uses simple arrays so that you can see more clearly the difference between flattening an array or leaving it as is. Consider some simple code that fills in numbers into an array:

```
var source = [1, 2, 3];

source.map(value => Array(value).fill(value));  ❶
//-> [[1],[2,2],[3,3,3]]

source.map(value => Array(value).fill(value)).concatAll(); ❷
//-> [1,2,2,3,3,3]
```

❶ **Mapping a function that returns an array onto an array results in a multidimensional array**
❷ **After a call to concatAll (or flatten), the resulting array is single dimensional, and much easier to work with**

Wrapping your head around the idea of Observables propagating other Observables may take some time, but by the end of the book, we guarantee it will come to you naturally. This software pattern is at the root of the functional programming paradigm and implemented data types known as *monads*. In simple terms, a monad exposes an interface with three simple requirements that observables satisfy: a unit function to lift values into the monadic context (`of()`, `from()`, and the like), a mapping function (`map()`), and a map-with-flatten (`flatMap()` or `mergeMap()`).

More information about monads

We don't cover monads in this book, but we encourage to learn more about because it's a central pillar of functional programming. If you're interested in learning more, please read chapter 5 of *Functional Programming in JavaScript (Manning, 2016)* by Luis Atencio.

In the real world, nested streams occur so frequently that for most of the combinatorial operators there exists a set of joint operators so that we don't have to use two every time when it comes to switching, merging, and concatenating. We introduced `mergeMap()` in this section, and now we're going to use it to tackle more complex problems.

5.3 Mastering asynchronous streams

Recall that for our initial implementation of our stock ticker widget in chapter 4, we used a simple random number generator to model the variability of our made-up company RxCorp and its stock price in the market. In this chapter, we'll integrate with a real stock service Ideally, such as Yahoo Finance, to fetch data for a given symbol. Our task at hand here is to use the Yahoo Finance APIs to query for a company's real stock price in real time so that the user sees updates pushed to the application reflecting changes in the stock's value. This API has a very simple to use API that respond with comma-separated values (CSV). To start off, let's use one symbol, Facebook (FB). This time, instead of using RxJS' `ajax()`, we'll show you how to plug in promise-based AJAX calls. The process is roughly outlined in figure 5.15:

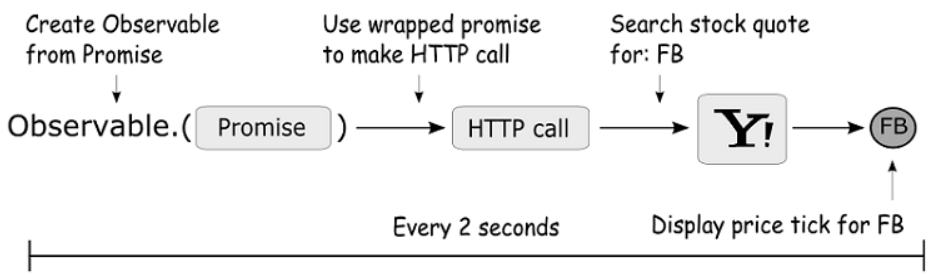

Figure 5.15 Using a promise call to the Yahoo web service for Facebook's stock data.

We're going to approach this stream by tackling its individual components. The first we'll need is a stream that uses a promise to query stock data via a AJAX:

Listing 5.4 The request quote stream

```
const csv = str => str.split(/,\s*/); (1)

const webservice = 'http://download.finance.yahoo.com
                    /d/quotes.csv?s=$symbol &f=sa&e=.csv'; (2)

const requestQuote$ = symbol =>
     Rx.Observable.fromPromise(
       ajax(webservice.replace(/\$symbol/, symbol))) (3)
     .map(response => response.replace(/"/g, ''))
     .map(csv); (4)
```

(1) Helper function that creates an array from a CSV string
(2) Yahoo Finance REST API link and requesting output format to be CSV

❸ Use the promise based ajax() to query the service
❹ Clean-up and parse the CSV output

The CSV response string emitted by the Yahoo API is designed needs to be cleaned up and parsed into a string. What we did here was create a function that returns an observable capable of fetching data from a remote service and publishing the result. We'll combine this with a 2-second interval to poll and provide the real-time feed:

```
const twoSecond$ = Rx.Observable.interval(2000);
```

Now we have two isolated streams that need to be put together, or mapped one to the other. We can use `mergeMap()` to accomplish this. I can create a function that takes any stock symbol and creates a stream that requests stock quote data every 2 seconds. We'll call this resulting observable function `fetchDataInterval$`:

Listing 5.5 Mapping one stream into another

```
const fetchDataInterval$ = symbol => twoSecond$
     .mergeMap(() => requestQuote$(symbol));
```

All that's left to do now is call this function with any stock symbol, in this case "FB."

```
fetchDataInterval$('FB')
  .subscribe(([symbol, price])=>
      console.log(`${symbol}, ${price}`)
  );
```

Notice how declarative, succinct, and easy to read this code is. At a glance it describes the process of taking a two-second poll and mapping a function to fetch stock data—as simple as that. This works well for a single item, but scaling out to multiple items, it's common to lift the collection of stock symbols to search for into an observable context and then map other operations to it. Listing 5.6 shows that how to use `mergeMap()` again to fetch quotes for companies: Facebook (FB), Citrix Systems (CTXS), and Apple Inc. (AAPL).

Listing 5.6 Updating multiple stock symbols

```
const symbols$ = Rx.Observable.of('FB', 'CTXS', 'AAPL');

const ticks$ = symbols$.mergeMap(fetchDataInterval$);

ticks$.subscribe(
  ([symbol, price]) => {
       let id = 'row-' + symbol.toLowerCase();
       let row = document.querySelector(`#${id}`);
       if(!row) {
          addRow(id, symbol, price); ❶
       }
       else {
          updateRow(row, symbol, price); ❶
       }
  },
  error => console.log(error.message));
```

❶ For brevity, we won't body of these functions that just manipulate HTML and either create the proper rows or updates them. You can visit the code repository to get all of the details

Essentially, the semantic meaning of `mergeMap()` is to transform the nature of the stream (map) by merging a projected observable. This operator is incredibly useful because now you can use it to orchestrate a complex business process containing multiple observable levels. The program flow looks like the flow in figure 5.16:

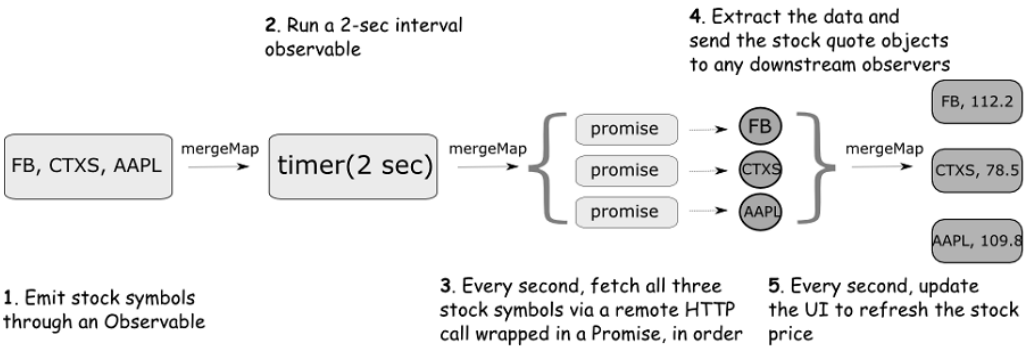

Figure 5.16 Steps to implement the stock ticker widget with multiple symbols, and a refresh interval of one second. Every step involves the used of nested observables that get merged or concatenated accordingly as the information get transformed into different types of streams. At the end, subscribers will only see the scalar values representing the stock symbol and their respective prices.

This program is simple and high-level, but actually accomplishes a lot:

1. We start by lifting the stock symbols involved in our component into a stream so that we can begin to fetch their quote data. This technique of lifting or wrapping a scalar value into an observable is beneficial because you can initiate asynchronous processes involving these values. Also, you unlock the power of RxJS when involving blocks of code related to these values.
2. We map an interval (every 2 seconds) to this observable, so that we cycle through the stock symbols every minute. This gives it the appearance of real-time
3. At each interval, we will execute AJAX calls for each stock symbol against the Yahoo web service
4. Finally, we map functions that strip out unnecessary company data and leave just the stock symbol and price, before emitting it to subscribers

The end result is that the subscribers see the following data-pairs emitted every two seconds:

Figure 5.17 A stocks table that updates with real stock symbols every 2 seconds.

There's room for improvement here, because in cases when there's not a whole lot of fluctuation in a company's stock price, we don't want to bother updating the DOM unnecessarily. One optimization we can do is to allow the stream to flow down to the observer only when a change is detected, by means of a filtering operator called `distinctUntilChanged()`. First, we'll show you how this operator works, and then we'll including it into our code. With a simple of input:

```
Rx.Observable.of('a', 'b', 'c', 'a', 'a', 'b', 'b')
  .distinctUntilChanged()
  .subscribe(console.log);
```

`distinctUntilChanged()` belongs in the category of filtering operators and emits a sequence of distinct contiguous elements. So, the fifth "a" is skipped because it's the same as the previous one, and so on for the seventh "b." You can visualize this mechanism in figure 5.18:

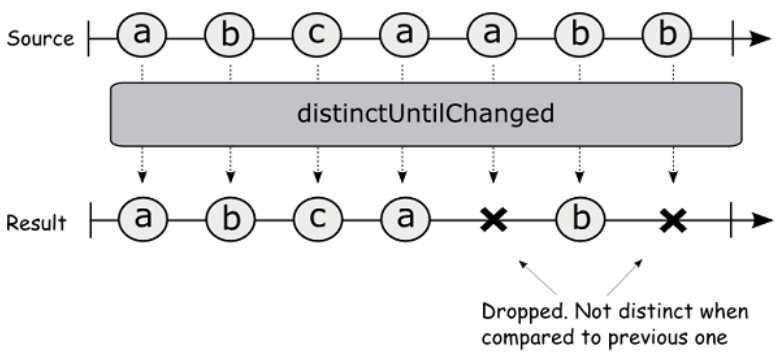

Figure 5.18 distinctUntilChanged() returns an Observable that emits all items emitted by the source Observable that are distinct by comparison from the previous item.

This is perfect for our task at hand. Adding this feature to our stream involves using it with a key selector function, a function that instructs the operator what to use as the property to compare:

```
const fetchDataInterval$ = symbol => twoSecond$
     .mergeMap(() => requestQuote$(symbol))
     .distinctUntilChanged(([symbol, price]) => price); ❶ ;
```

❶ **Perform a distinct based on price, so that the DOM will only get updated when the price changes.**

Now we'll only update the DOM strictly on a price change, which is a lot more optimal than doing it naively everything 2 seconds.

We were able to combine observable flows and solve this problem without creating a single external variable, conditionals, or for-loops. Because this task involves the combination of multiple streams: iterating through the symbols, a timed interval, and remote HTTP requests with promises, I had to unwind nested streams using a combination of a couple of nested `mergeMap()` operators to convert the intricate business logic into a more flattened and linear sequence of data that observers subscribe to and easily consume. As we mentioned previously, it's much more efficient and fluent to do it this way, rather than to subscribe to each nested stream at each step. Using RxJS operators to handle this, as well as all the business logic, is always preferred over sending raw nested observables to any downstream observers.

As we can see from this example, RxJS' combinatorial operators like `mergeMap()` are about more than simply reducing the number of subscribe blocks we have to write and the number of subscriptions to keep. They also serve as an important way to craft more intricate flows to support complex business logic. By leveraging nested streams, we can think of each block within a nest as a mini application (that can be partitioned out into its own stream as just learned previously). This a clear sign of *modularity* in your code, and something that functional programs exhibit extremely well. By breaking down problems into individual, independent blocks, you can create code that is more modular and composable. Now that we've covered merging complex observable flows, let's look at a more complex example.

Aside from `mergeMap()`, there are others that work in similar manner but with a slightly different flavor driven by the behavior of the composed function whether it be: `switch()`, `merge()`, or `concat()`, which you can use depending on what you're trying to accomplish. Here's a table that describes the three joint operators we'll use in this book:

Table 5.1 A breakdown of the 3 most used RxJS joint operators

Split operators	Joint operator	Description
`map()...merge()`	`mergeMap()`	**Projects an observable-returning function to each item value in the source observable and flattens the output observable. You might know this operator by `flatMap()`, as used in previous versions of RxJS.**
`map()...concatAll()`	`concatMap()`	**Similar to `mergeMap()` with the merging happening serially; in other words, each observable waits for the previous one to complete. Why not `map()...concat()`? We'll explain this disparity shortly.**

`map()...switch()`	`switchMap()`	**Similar to `mergeMap()` as well, but emitting only values from the recently projected observable. In other words, it "switches" to the projected observable emitted the most recent values, cancelling any previous inner observables. You might know this operator by `flatMapLatest()`, as used in previous versions of RxJS. We'll come back to this operator in chapter 6.**

Now that you've mastered handling nested observables, let's move on to other types of higher-order observable combinations continuing with `concatMap()`.

5.4 Drag and drop with concatMap

We have looked at two ways of composing streams together such that they produce a single output stream. In each case, the resulting stream appears identical to an observer. Both

```
Rx.Observable.merge(source1$, source2$) ❶
```

And

```
source1$.mergeMap(() => source2$)       ❷
```

❶ **Observers see data emitted from either observable**
❷ **Observers will only see data from source2$**

result in observables which return a type compatible with either source. As we mentioned before, this is entirely up to you because JavaScript won't enforce that observables wrap values of the same type, as statically typed languages will.

However, the behavior of both approaches above are slightly different. In the former case, we are creating a set of simultaneous streams, this means that both streams are subscribed to at the same time and the resulting observable can output from either Observable—both are active *simultaneously*.

Sequential streams, on the other hand, are streams in which the output of one stream generates a new one. In the second case we presented above, the observer will not receive any events from the first stream, an observer will only see the results of the observable projected by `mergeMap()`. By combining sequential and simultaneous streams we can make fairly complex logic relatively easily.

One canonical example of a sequential stream is drag-and-drop, present in most modern web content management systems and customizable dashboards. Implementing this behavior in vanilla JavaScript is fairly difficult to get right as it involves keeping track of fast changing states with multiple targets and interaction rules. Using streams, we can implement this in a fairly streamlined manner.

To implement drag-and-drop, let's first identify the three types of mouse events that are necessary for basic drag-and-drop. We need to know first when the mouse button is clicked as this indicates that the drag has started, and the mouse button up event to determine when it has stopped, this indicates a drop. In order to track the drag, we also need to capture the

mouse move event as well. We'll only need those three events (`mouseup`, `mousemove`, and `mousedown`), each of course modeled as a stream.

A drag starts when the user clicks, and then only stops when the mouse button is released, or until the mouse up event fires. As the mouse is moved with the button down the object being dragged, it also also moves as well in a fluent manner, so we'll anticipate performing a side effect by manipulating the DOM element. When the button is released, we "drop" that object into the coordinates of the location of the mouse on the screen—thus terminating the mouse move event. As we did before, the gist is to combine these streams together representing the three mouse events emitting data.

First, let's create streams from the different types of events we're interested in as shown in listing 5.7:

Listing 5.7 Streams needed to implement drag and drop with a mouse

```
const panel = document.querySelector('#dragTarget');                    (1)

const mouseDown$ = Rx.Observable.fromEvent(panel, 'mousedown');          (2)
const mouseUp$ = Rx.Observable.fromEvent(document, 'mouseup');           (3)
const mouseMove$ = Rx.Observable.fromEvent(document, 'mousemove');       (4)
```

(1) A reference to the panel or target we want to drag (My Stocks widget)
(2) Observable for mouse down events on the target
(3) Observable for mouseup events on the drag target
(4) Observable for mousemove events over the entire page

After we have established the streams that will be used to control the drag we can now build logic around the drag. In the next step we need to actually implement the logic to handle first detecting a click on a drag target that will initiate the drag event. Then we need to have the element follow the mouse or finger around the screen until it is released and "dropped" somewhere. You've learned in this chapter how you can use the RxJS joint operators to combine and flatten multiple nested streams so that subscribers see a simple representation of the data flowing through it. So, we need an order preserving operator (the mouse is pressed, then dragged, and finally released), just like `concat()`, but also be able to flatten the observable that's pushed through it. Care to take a guess? Yes, this is the job of `concatMap()`. This operator works just like `mergeMap()`, but performs the additional logic of retaining the order of the observable sequences, instead of interleaving the events.

The logic for handling the drag is made up of a sequence of streams that emit mouse events together, as shown in listing 5.8:

Listing 5.8 Drag-and-drop stream logic

```
const drag$ = mouseDown$.concatMap(() => mouseMove$.takeUntil(mouseUp$));

drag$.forEach(event => {
  panel.style.left = event.clientX + 'px';
  panel.style.top = event.clientY + 'px';
});
```

Sorry, were you expecting more? This is really all that's required to drag the stock widget around the page. If you think about or perhaps you've implemented drag and drop before, you're probably aware that it would take a lot more code and using a lot more variables to store some transient state. This code is not only shorter, but also has a higher level of abstraction, as all side effects were pushed elegantly onto the observer.

Here we've introduced another variation of the `take()` operator called called `takeUntil()`. The name is pretty straightforward; this operator also belongs to the filtering category and allows the source observable to emit values *until* the provided *notifier observable* emits a value. The notion of a notifier observable occurs frequently in RxJS, to be used as signals to either start or stop some kind of interaction. In our case, take any `mousemove` concatenated with the `mousedown` events until a `mouseup` event is fired. This is the gist behind dragging.

Here's a simple use of `takeUntil()` so that you can fully appreciate how it works. This example starts a simple 1-second interval, which will print values to the console until the user clicks on a button. This could be useful to implement a site inactivity feature, for instance:

```
const interval$ = Rx.Observable.interval(1000);
const clicks$ = Rx.Observable.fromEvent(document, 'click');

interval$.takeUntil(clicks$) (1)
   .subscribe(
      x   => console.log(x),
      err => console.log(`Error: ${err}`),
      ()  => console.log('OK, user is back!'));
```

(1) **As soon as a click event is emitted, the interval stream is cancelled.**

The other benefit of RxJS' unified computing model is that if we were implementing drag through a touch interface, it would just be a matter of changing the event names in the stream declaration to `touchmove`, `touchstart`, and `touchend`, respectively. The business logic stays the same and code would work exactly the same!

There's a small caveat here. From our earlier discussions you might be led to believe that calling `map()...concat()` would work in a similar fashion to the split operator `concatMap()`. You might intuitively think that this code would work exactly the same way as listing 5.7:

```
const drag$ = mouseDown$.map(() =>
    mouseMove$.takeUntil(mouseUp$)).concat(); (1)
```

(1) **Using the instance operator concat() in place of concatMap()**

Unfortunately, it doesn't because there's no mechanism here to flatten the observable type running through the stream. Recall that the `concat()` instance method just takes a number of observables and concatenates them in order. It's not designed to work with an observable of observable type—a higher-order observable. For this, you need to use a variation of `concat()` that works with nested observables and also flattens them called `concatAll()` (just like we implemented with arrays before). So, the RxJS nomenclature here is a little bit inconsistent, as `concatMap()` is really `map()...concatAll()`.

Now, this code works just like listing 5.7:

```
const drag$ = mouseDown$.
   map(() => mouseMove$.takeUntil(mouseUp$)).concatAll();
```

Here's a visual of how `concatAll()` works:

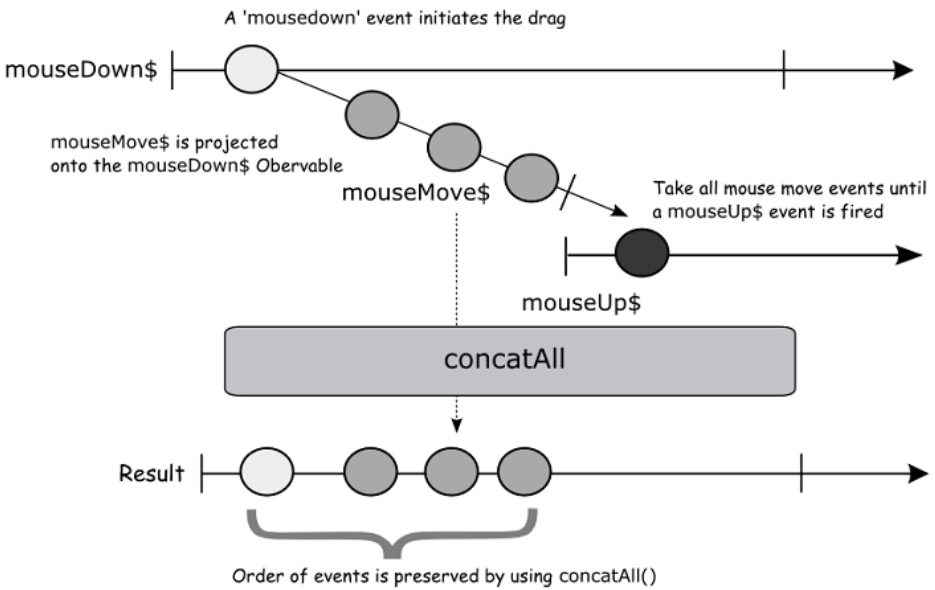

Figure 5.19 Workings of concatAll() combining three mouse events. This operator not just preserves order, but can also flatten a sequence of nested observables. Also the use of takeUntil() causes mouseMove$ to cancel as soon as mouseUp$ emits a value.

In the code repository you will find a more complete version of this example, which we simplified for the sake of highlighting the important elements. In reality, we can add any number of bells and whistles to this functionality and for our sample application we added some additional helper code for dealing with CSS. For instance, if you wanted to prevent user's from accidentally dragging a widget by allowing them to confirm the drag, you could just filter the `mouseup` stream to include this confirmation:

```
const drag$ = mouseDown$.
  concatMap(() =>
    mouseMove$.takeUntil(
      mouseUp$.filter(() => confirm('Drop widget here?'))));
```

As simple as that. The samples that we have discussed in this chapter are only a small portion of the total number of use cases we could support. As we mentioned at the outset, almost all

non-trivial applications will make use of flattened streams so understanding them a huge step toward understanding RxJS.

In this chapter we began to enter more complex territory by combining the outputs of multiple streams. This was key for us to begin implementing more real-world tasks instead of just showcasing the different operators. Initially, we looked at static streams that merely had their outputs piped together to appear as a single stream. However, as a more complex case we examined how we can nest observables within each other and then flatten the result to deliver the appropriate data to observers. We discussed how making more complex applications especially those that deal with UIs and other state machines will almost always necessitate the use of nested or merged streams. Finally, we explored a couple of practical examples of flattening and merging that helped you understand the design principle behind projecting observables.

In the next chapter we will continue expanding on this topic by examining how we can further coordinate Observables and have them work together. We'll also explore other interesting ways that observables combine using an operator called `combineLatest()`.

5.5 Summary

- Merge the outputs of several Observables into a single stream to simplify subscription logic.
- Use different merge strategies to that contain different behavior for combining streams depending on your needs.
- You can interleave stream with `merge()`, cancel and switch to a new projected observable with `switch()`, or you can preserve entire observable sequences in order by using `concat()`.
- Split operators are used to combine and flatten a series of nested observable streams.
- You can combine and project observables into a source observable using the higher-order operators such as `mergeMap()` and `concatMap()`.
- Implemented an auto-suggest search box.
- Implemented a live stock ticker with deeply nested streams.
- Implemented drag and drop feature using stream concatenation.

6

Coordinating business processes

This chapter covers:

- Synchronizing the emission of several observables
- Using observables as signaling devices
- Building complex interactions from multiple inputs
- Spawning simultaneously streams
- Streamlining your database storage operations using observables

The previous chapter examined how converting multiple observables into a single one can simplify their consumption and reduce the management overhead. This mechanism is very important because it allowed you to reuse a single subscription to handle data that's being transformed or created by the composition of multiple tasks: AJAX requests, business logic transformations, timers, and others. The various strategies for how these different types of merging operations (`merge()`, `concat()`, or `switch()`) occurred, as in whether we cared about the order of the events or canceled others, was determined by the operator itself—each had a different flavor. In doing so we also showed examples like search and drag-and-drop that use the output of one observable to signal the start or completion of another.

In this chapter, we'll continue with this theme and expand where we left off from chapter 5. You will learn that we don't always have to care about the result of an observable if we simply want to leverage the semantics of when one emits to cause some other process to begin. Furthermore, we'll explore scenarios where events from multiple streams can be aggregated and combined so that the resulting observable is emitting the sum of two observables; in other words, two streams cooperating with each other, working together towards a common a goal. To illustrate this, in this chapter we will tackle problems that involve authentication, data persistence, and stream parallelization. The interplay of using observables as a signaling device and the more interesting uses we can achieve through

joining observables forms the foundation of the more complex logic you are likely to see in the wild. In order for you to understand how observables can collaborate, you must first understand how to tap into their lifecycle.

6.1 Hooking into the observable lifecycle

The representational power of a single observable is limited. While we can obviously create a stream to represent just about any data type, a single stream can only contain logic for a single set of inputs and outputs, like the results of a series of key strokes or an individual web request. Even using the combinational operators from the previous chapters like `mergeMap()`, there is still only a single task to which the observable can be assigned without introducing side effects. Remember in the last chapter that we were able to combine mouse and touch events to support drag and drop. Trying to also support say, free-form drawing, using the same stream would be very difficult because it would no longer be clear which use case an observer should be expecting. It's important to realize that, by design, a single stream can only carry out a single task; therefore, performing multiple actions whether serially or in parallel depends on how you combine these streams.

By now, you've learned how to transform and filter data in flight, even coming from different sources. Separating tasks into loosely coupled streams is advantageous because you can compartmentalize their respective logic without bleeding state into other areas of the application—we call this *upstream compartmentalization* or conformance. You saw examples of this in the mouse and touch code when we needed to make two streams conformant to a single observer block. You could, alternatively, combine the stream data as is, and group all your business logic into the observer, or *downstream compartmentalization*. We highly recommend the former over the latter.

However, there are times where those operations are insufficient because we need several streams to interact while also maintaining the same semantically easy-to-understand flow of code that we have come to expect from RxJS. So, instead of creating separate streams and building the scaffolding to connect them ourselves, we learned in the previous chapter how to combine and map observables to other observables. We did this for real-world tasks such as our smart search, stock widget, and others. In this chapter we continue building on those techniques, and continue the theme of observables working in unison to achieve a certain goal.

6.1.1 Web hooks and the observer pattern

RxJS' `Observable` type is comparable to an `EventEmitter` data type, which we briefly mentioned in chapter 1, in that both belong to a general class of objects known as event hooks. Event hooks are quite simply just a way of targeting certain stages in an object's lifecycle with the objective of triggering further actions. When an action associated with a hook is triggered we say that the hook has *fired*. Event hooks can operate within or beyond the confines of a single application. For instance, GitHub, the most popular version control

repository for hosting code, provides access to a whole slew of external hooks that allow for multiple services to coordinate with events such as the creation of pull requests, new commits, or branches. Each time an action is performed the associated event hooks will fire and any listeners will receive those events, as seen in figure 6.1.

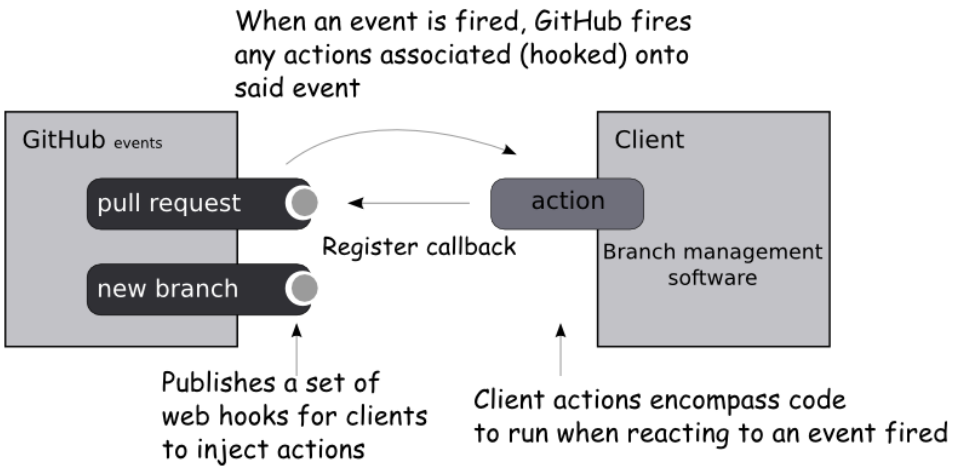

Figure 6.1 Shows a couple of well-known GitHub hooks that clients can plug logic into

In general, event hooks provide two main benefits:

1. It allows the developer of the application to retain control over what constitutes a hook, thereby maintaining final say over where and when events will be fired.
2. It allows third parties to execute arbitrary code without having to worry about detecting specific events.

It's not hard to realize that event hooks are just another instance of the observer pattern omnipresent in RxJS. Similarly, every observable also has a set of events or "hooks" in its lifecycle that can be plugged into, all of which should be familiar to you by now:

- Observable start (Subscription)
- Observable stop (Completion or Error)
- Observable next (Normal event)

6.1.2 Hooked on observables

Let's discuss each one a bit further, and offer some operators that work during these stages. The first item, the Observable start (or the "onSubscribe" hook), may not be as obvious as the other two which you have seen in some form several times up to this point, but it's also perhaps the easiest to understand. The goal of listening to when a subscription is created is to perform some action when an observable begins emitting events. Hence, the `startWith()` operator does something to this effect by prepending a value onto the front of an observable

each time it is subscribed to by an observer. So in the following code, the the number zero will appear before any other events on the console:

```
Rx.Observable.range(1, 5)
  .startWith(0)
  .subscribe(console.log);
//-> 0,1,2,3,4,5
```

The `startWith()` operator is really a `concat()` (in reverse) of all of the values provided to it with the source stream following, in that order. In fact, it's leveraging the subscription behavior to inject events before others are received. This can be trivially implemented as a custom operator just like in chapter 3. Listing 6.1 shows we might implement it:

Listing 6.1 Hooking into the start of a stream

```
function startWith(value) {
  return Rx.Observable.create(subscriber => {    ❶
    let source = this;
    try {
       subscriber.next(value); ❷
    }
    catch(err) {
       subscriber.error(err);
    }
    return source.subscribe(subscriber); ❸
  });
};

Rx.Observable.prototype.startWith = startWith;
```

❶ Use the factory method to create the stream
❷ Always emit the value before anything else
❸ Emit the rest of the stream

This operator makes sure that every time the stream is subscribed to, it will always produce that value first. Now, normally you wouldn't re-invent your own `startWith()` because RxJS already implements the necessary hooks for you; so all you have to do is inject any function that you want. But this serves to show how extensible observables are.

On the opposite side of the spectrum, we can also think of the completion event as its own kind of event. It occurs when invoking the observer's `complete()` method when an observable finishes and before the subscription is disposed of. As you've seen all along, this hook is used to perform all of your side effect logic, operations outside the scope of the observable, such as logging to the console, printing data to the screen, writing to a database, and others.

> **RXJS.FINALLY** RxJS' error handling mechanism also introduces the `finally()` operator. Semantically similar to the traditional finally block in JavaScript, this operator is the absolute last step in the observable lifecycle, regardless of whether any errors occurred. The function passed to the finally operator will be executed when the observable is shutting down for any reason, so even if the observable terminates with an

exception the block will still be run. This gives you the opportunity to recover from errors and clean up any necessary resources. We will cover error handling and all of the wonderful things you can do to recover from errors in chapter 7.

In addition, we can also combine logic that is tied to both the start and end of an observable. We can do this with a new operator, called `using()`. Figure 6.2 demonstrates how this operator works.

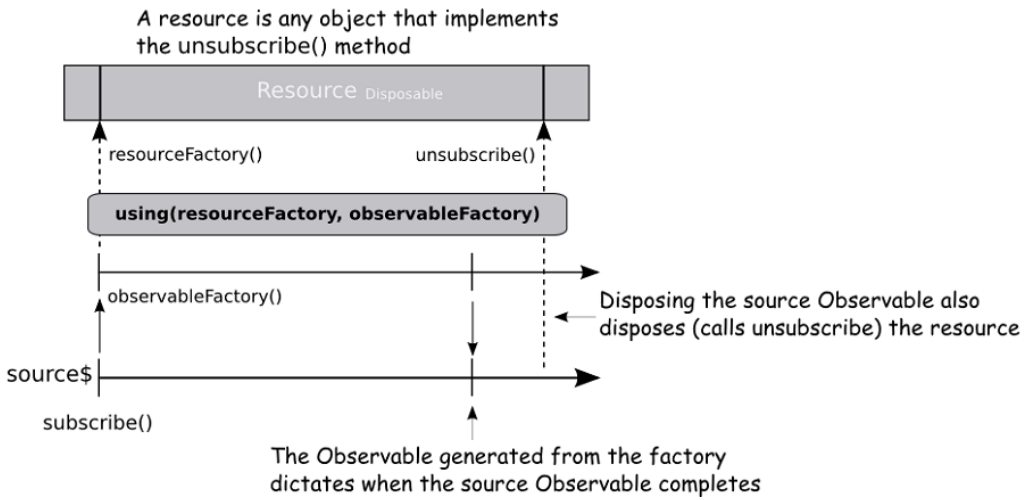

Figure 6.2 The using() operator controls the lifespan of a resource object (created via a resourceFactory()) through the lifespan of an Observable (created via an observableFactory())

This operator (`using()`) takes two parameters, a function that creates a disposable resource (like an object) and a function that creates an observable. Both of these functions are known as *factory functions* in RxJS parlance. The resource is tied to the lifecycle of the observable created by this function, so that when the latter is disposed of, so is the resource. When an observer subscribes to the observable returned from `using()`, the first factory function is invoked to create an instance of the resource. The resource is then passed to the second factory function as a parameter, that second factory returns the actual observable that will be subscribed to. Disposing of the resource is as simple as disposing of the stream through normal means. When the subscription goes through the disposal process it will also attempt to dispose of the resource that was created by the resource factory. The completion of the observable will also attempt to dispose the resource, whichever comes first. Essentially, what you're doing is linking the lifespan of an object *using* an observable.

Here's an example that would help you understand how it works. Consider an arbitrary disposable resource object, called `DisposableResource`:

```
class DisposableResource {
  constructor(value) {
     this.value = value;
     this.disposed = false;
  }

  getValue() {
     if (this.disposed) {
        throw new Error('Object is disposed');
     }
     return this.value;
  }

  unsubscribe() {  ❶
     if (!this.disposed) {
        this.disposed = true;
        this.value = null;
     }
     console.log('Disposed');
  }
}
```

❶ **A disposable resource must provide an implementation for the unsubscribe() behavior**

We can tie the behavior of this object and its state with the lifespan of any observable with `using()` as follows:

```
const source$ = Rx.Observable.using(  ❶
  ()=> new DisposableResource(42),
  resource => Rx.Observable.interval(1000)
);

const subscription = source$.subscribe(
    next => console.log(`Next: ${next}`),
    err  => console.log(`Error: ${err}`),
    ()   => console.log('Completed'),
);

//...
subscription.unsubscribe();  ❷
```

❶ **Using receives two parameters: a resource object and an observable**
❷ **Seconds later we unsubscribe from the source, which will unsubscribe from the observable managing the resource as well as the resource itself.**

Running this code will begin emitting interval 20 values every second. Before this time, seconds later we unsubscribe from the source. This cleans up the resource observable as well as the `DisposableResource` instance by calling its unsubscribe method.

The idea behind it is that often we will have resources that are completely subject to the lifespan of the observable. In order to hook into this stage, the only requirement is that the object you plug in must be "disposable-like," which is to say that it must declare an `unsubscribe()` method, in order to be cleaned up properly if needed. As an example, suppose you wanted to manage the login session of the user through an observable. When the user

logs in, we can create a session token that could be stored in the cookies that keeps track of the authenticated user session. However, when the user logs out or closes the window, the session needs to be deleted. The closing of the browser itself signals an event that we can listen for, so we could use observables for this.

First we would need to create an object that would manage the lifecycle of the session token, very similar to our code sample above. Upon construction, it will set the session to expire in 24 hours as shown in listing 6.2:

Listing 6.2 SessionDisposable object implementing the dispose functionality

```
class SessionDisposable {
  constructor(sessionToken) {
    this.token = sessionToken;
    this.disposed = false;
    let expiration = moment().add(1, 'days').toDate(); (1)
    document.cookie = `session_token=${sessionToken};
        expires=${expiration.toUTCString()}`;    (2)
    console.log('Session created: ' + this.token);
  }

  getToken() {
     return this.token;
  }

  unsubscribe() { (3)
     if (!this.disposed) {
        this.disposed = true;
        this.token = null;
        document.cookie = 'session_token=; expires=Thu, 01 Jan 1970
            00:00:00 GMT';
          console.log('Ended session! This object has been disposed.');
     }
  }
}
```

(1) Create a cookie with an expiration date 24 hours from now. Using the popular moment.js library to manipulate dates easily (installation instructions found in appendix)
(2) Add the cookie
(3) Clear the cookie

The most important aspect to note from this class is the declaration of the `unsubscribe()` method, so that objects of this type conform to the `Disposable` specification.

> **DISPOSE OR UNSUBSCRIBE** The terms *dispose* and *unsubscribe* are interchangeable. The notion of disposing was used predominantly in RxJS 4 so the terminology became part of the RxJS jargon. RxJS 5 changes this to unsubscribe, yet it's still easier to say disposable rather than "unsubscribable."

If you have a Java or C# background, this is analogous to saying that your class implements the `Disposable` interface. The logic here is simple; it only resets the value of the token and sets the time back to the epoch (01/01/1970) so that the browser deletes it before closing.

Now, let's tie our `SessionDisposable` object to a stream. For this example we'll use the `using()` operator to construct an observable sequence that depends on an object whose lifetime is tied to the resulting observable sequence's lifetime; in other words, we're making one stream dependent on another one. This operator takes a factory function which creates the `SessionDisposable` object and a second method which makes it available to the stream.

Here's how we can use `using()` manage a session token available for the duration of a countdown, shown in listing 6.3:

Listing 6.3 Managing a temporary session token with using()

```
function generateSessionToken() { ❶
     return 'xyxyxyxy'.replace(/[xy]/g, c => {
         return Math.floor(Math.random() * 10);
     });
}

const $countDownSession = Rx.Observable.using(
   () => new SessionDisposable(generateSessionToken()), ❷
   () => Rx.Observable.interval(1000)
     .startWith(10)
     .scan(val => val - 1)
     .take(10)
);

$countDownSession.subscribe(console.log); ❸
```

❶ **Simple function to generate a random session token**
❷ **Attach the session to the lifespan of this Observable**
❸ **When this subscription completes, the subordinate session disposable token will be disposed and the cookie deleted**

With this code, say your user's login state is now tied into the lifetime of the session, such that when the user closes the window or logs out (thereby unsubscribing from the subscription) they're also logged out of the application. This kind of pattern is used a lot in e-e-commerce sites where you need to perform some action within a certain time span.

This kind of coordination takes advantage of the hooks to create side effects around the observable, but we can do much more by incorporating more streams in to the mix.

About the Rx.Observable.using() operator

The using operator actually comes from C#, where a using(resource){ … } block is a synchronous construct that manages the life time of a resource by invoking the garbage collector and cleaning up the resource once the inner block (between the curly braces) exits.

Using a disposable object allowed us to tap into RxJS' unsubscription mechanism. It turns out that we can solve lots of different uses cases when observables coordinate with others through signals.

Another form of coordination is when multiple streams join together to produce a result. You already learned in the previous chapters about combination operators such as `merge()`, `switch()`, `concat()`, and we briefly introduced `combineLatest()`. When talking about signaling and coordination, `combineLatest()` can have many practical effects. Let's spend more time understanding how important this operator is when it comes to parallel streams.

6.2 Joining parallel streams with combineLatest and forkJoin

Building asynchronous flows is a difficult endeavor, because each possible permutation of the data's arrival times must be accounted for, which you can never guarantee. Using the browser's multiple connections, some of the data can be retrieved in parallel. But when data is causally linked, it needs to be fetched serially. In this section, we'll show you how to use RxJS to coordinate between observable streams that can originate from independent and dependent actions. That is to say, that a stream's data can come from events causally linked to other data sources (dependent), or can be run in parallel with other streams (independent). There are ways to do this with plain vanilla JavaScript using our long lost friends, callbacks and promises. Consider writing an event handler for a button that, when clicked, queries two remote data sources for data using an `ajax()` function with a callback:

```
button.addEventListener('click', () => {
   let result1, result2 = {};
   ajax('/source1', data => {
      result1 = data;
   });

   ajax('/source2', data => {
      result2 = data;
   });

   setTimeout(() => {
     processResults(result1, result2);
   }, arbitraryWaitTimeMs); ❶
});
```

❶ **Would need to be long enough so that both AJAX calls have enough time to finish**

Of course, the chances of this working are slim to none because you can never predict how long both AJAX calls would take. Other options include writing custom waiting routines based on `setInterval()` and semaphores, yet they're extremely convoluted and invasive to the business logic. This code would allow both AJAX requests to happen in parallel, but waiting and combining the results later is difficult without proper wait semantics. Inserting sleep functions into a single-threaded JavaScript code is frowned upon, as browsers may deem your scripts unresponsive. In order for this to work, you have to sacrifice parallelism and nest your AJAX calls:

```
button.addEventListener('click', () => {
   ajax('/source1', result1 => {
      ajax('/source2', result2 => {
```

```
        processResults(result1, result2);
    });
  });
});
```

And just in case you're thinking about this, it's important to mention that also using observables this way goes against the RxJS philosophy—it's an anti-pattern. What we mean to say is that you could think of nesting `subscribe()` blocks like this:

```
click$.subscribe(() => {
  source1$.subscribe(result1 => {
      source2$.subscribe(result2 => {
       processResults(result1, result2);
      });
  });
});
```

But just like the snippet above this would lead us to the very familiar callback hell we're trying to stir way from in the first place. In addition, this solution would only apply if the inner observables were eagerly loaded, which is not the case. Consequently, the inner subscriptions would only begin executing only after the outer subscription had produced its first value. This makes our code blocks dependent and tightly coupled to each other, not parallel, as shown in figure 6.3:

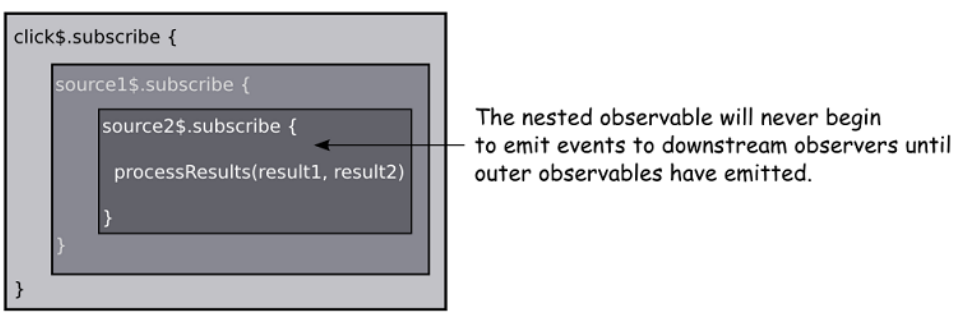

Figure 6.3 Shows the issue with nested subscription blocks.

In the previous chapter we learned that we can project observables using operators such as `mergeMap()` and `switchMap()`, but these won't work either for two reasons. Respectively:

1. Nesting observable implies causality, that a source observable dictates how the other executes. The checking and savings streams are independent of each other, and
2. We need to preserve the data from all of the observables at the same time without cancelling any of them.

Note that nesting callbacks does eliminate the need for an arbitrary wait time. However, in either case you won't be able to perform tasks in parallel, which is what we're trying to do here. Given how frequently this occurs in JavaScript, there are other libraries that address it like Async.js (https://github.com/caolan/async), which you can use to write code of the form:

```
button.addEventListener('click', () => {
     async.parallel([ ❶
        ajax('/source1'),
        ajax('/source2')
     ],
      (err, ([result1, result2])) => {  ❷
        processResults(result1, result1);
     });
});
```

❶ Runs all functions in parallel
❷ The final callback in async.parallel() receives an array with the outcome of both AJAX calls

Luckily, we don't need to include another library, because RxJS is the right tool for the job. In this section, we'll explore an operator called `combineLatest()` that applies very similar parallel semantics to Async.js but in-line with the RxJS operator philosophy. To frame this problem more concretely, consider two streams that access two popular APIs to shorten a URL. One makes an AJAX request to Bitly (https://app.bitly.com), and the other to Google Shortener (https://goo.gl). For the sake of discussion, we will kick these streams off using another stream that monitors a URL text box field—debounced for efficiency, of course.

We would want both to be able to run in parallel, and we might be able to use operators that we learned in the previous chapter, like `merge()` and `concat()`, depending on whether we care about order preservation. However, because both streams are not causally linked (as in we can't just chain them together one after the next), we can't use these operators. For example, `concat()` would force each result to emit in order one after another rather than in parallel, while `merge()` would only allow us to consider a single emission at a time downstream instead of collectively.

Rather, we have stipulated that both tasks must run in parallel, but only emit results when all of them have emitted, which may be at any point in the future. We tried using callbacks, let's see if promises fair better.

6.2.1 Limitations of using promises

Certainly, we need a new pattern or set of operators to accomplish our goal, such that we can execute all of the statements in parallel while also being able to gather them collectively when they have all completed. Promise libraries that follow the Promise/A+ protocol include a collection operation called `Promise.all()`, which creates a new promise that awaits the completion of all the promises, or rejects with the error of the first one to reject. Let's use this method here, but instead of using callbacks, we'll use a version of `ajax()` that returns promises that wrap the HTTP request:

```
button.addEventListener('click', () => {
   Promise.all(ajax('/source1'), ajax('/source2'))  ❶
       .then(([result1, result2]) => {  ❷
          processResults(result1, result2);
     });
});
```

1. Executes all promises and waits for all to complete
2. Processes the joined value, an array, returned from the call to .all(), which is destructured and passed into a method that knows how to render account details.

Already we see that the use of promises helps our code not only in indentation and readability, but also because we're using a mechanism that knows to emit the value only when all have arrived, in parallel, effectively. But the more these types have to be mixed and matched, the more difficult this becomes as each variation will require a more intricate solution. In the code above, we're mixing two fundamentally different paradigms: event-driven listeners with the more functional promises. Nevertheless, it does achieve parallelism and moves the needle in the right direction.

We already know the desired traits of the new operator we need. It should provide the fluent API design of promises plus the parallelism semantics of Async.js. This operator should take multiple sources, like the static `merge()` operator, but at the same time it should be able to *combine* and emit the collective result from all inputs as an event of its own. Let's introduce `combineLatest()`.

6.2.2 Combining parallel streams

Whereas operators such as `merge()`, `concat()`, and `switch()` combined a series of observables (or an array of them) to output a single observable, `combineLatest()` gives you a way to emit and capture events from multiple sources at the same time. This operator creates an observable whose values are calculated from the *latest values of each of its input observables*. This operator is ideal for situations where you need to spawn to long running processes in parallel and then used the combined result. For example, suppose we wanted to use third-party services to shorten URLs. Because both streams act independently, we could use both services in parallel and then present the user with both outputs. This is the task we'll tackle in this section.

Before we begin developing the solution to this problem, let's briefly introduce you to `combineLatest()` with a simple example. The data emitted is very similar to how buffering worked in chapter 3. In other words, the output is an array that combines the latest data from all of input observables—same semantics as `Promise.all()` or `async.parallel()`. Here's a quick example to showcase how this operator works. We'll combine the output of two streams: one emits numbers letters every second, and the other numbers every second:

Listing 6.4 Synchronizing streams with combineLatest()

```
const letter$ = Rx.Observable.interval(1000) ❶
  .map(num => String.fromCharCode(65 + num))
  .map(letter => `Source 1 -> ${letter}`);

const number$ = Rx.Observable.interval(1000)
  .map(num => `Source 2 -> ${num}`);

Rx.Observable.combineLatest(letter$, number$)
   .take(5)
```

```
    .subscribe(console.log);
```

❶ Emits A, B, C, ... every second
❷ Emits 0, 1, 2, 3, ... every second

Running this code prints:

```
["Source 1 -> A", "Source 2 -> 0"]
["Source 1 -> B", "Source 2 -> 0"] ❶
["Source 1 -> B", "Source 2 -> 1"] ❷
["Source 1 -> C", "Source 2 -> 1"]
["Source 1 -> C", "Source 2 -> 2"]
```

❶ Source 1 emits "B" with the latest value in source 2 "0"
❷ Source 2 emits "1" with the latest value in source 1 "B"

Here we have two independent streams that emit every second, one letters starting with "A," and the other numbers starting at zero. Each emission will cause a collective emission of the latest value present in the stream. So after the first emission A -> 0, each result alternates emitting the latest from the other stream. In other words, when Source 1 emits "B," it sends the result with the latest value in Source 2 at that time "0." Then when Source 2 emits the next value, "1," it sends the result with the latest value in the stream at that time "0." In summary, an emission from any stream in the combination causes all of the to publish their latest value, all sent to the observer via an array.

In this simple case, both data sources were asynchronous intervals. With synchronous data sources, we have to be careful because RxJS will immediately run through the events of the first source stream, and combine its latest value with the latest value of the combined stream:

```
const letter$ = Rx.Observable.from(['a', 'b', 'c']);
const number$ = Rx.Observable.from([1, 2, 3]);
Rx.Observable.combineLatest(letter$, number$).subscribe(console.log);
```

Instead of paring up each number with a letter. Running this code will output a very different result:

```
   ["c", 1]
   ["c", 2]
   ["c", 3]
```

Now that you understand how this operator works, let's jump into our task. Again, we want to spawn parallel AJAX calls to shorten some URL. The user is expected to type a valid URL into a text box; when the user removes focus from it, it will kick-off these independent streams. So, we're mixing one casually linked stream with two parallel streams, which should suggest the use or `mergeMap()` (or `switchMap()`) and `combineLatest()`, respectively.

> **CAUSALITY** Generally, causal streams (one depends on the other) are combined using `mergeMap()` or `switchMap()`, while independent streams are combined using `combineLatest()` and others we'll learn about shortly

Reason about this problem this way—thinking in streams, we can come up with the following program for a URL shortener stream that uses both Bitly and Google:

Listing 6.5 Combining multiple URL shortener streams

```
const urlField = document.querySelector('#url');

const url$ = Rx.Observable.fromEvent(urlField, 'blur')
   .pluck('target', 'value')
   .filter(isUrl)          (1)
   .switchMap(input => (2)
       Rx.Observable.combineLatest(bitly$(input), goog$(input)))   (3)
   .subscribe(([bitly, goog]) => {
     console.log(`From Bitly: ${bitly}`);
     console.log(`From Google: ${goog}`)
  });
```

(1) Checks usin a regex that the input provided matches a valid URL (omitted for brevity)
(2) Project an observable that will emit results when both subordinate streams emit
(3) Combine the latest events of both services

To run this program, type any URL into the input field and we will use these providers to shorten this URL. So, for https://www.manning.com/books/rxjs-in-action, the output is:

```
From Bitly: http://bit.ly/2dkHUau
From Google: https://goo.gl/plTbDG
```

These all resolve to the original link (so feel free to share it on your favorite social media!). Of course, we don't understand exactly how `bitly$` and `goog$` work, but the abstraction provided by RxJS means we can still reason about this code as is, from its declarative nature. Let's create a simple graph to visualize what's happening:

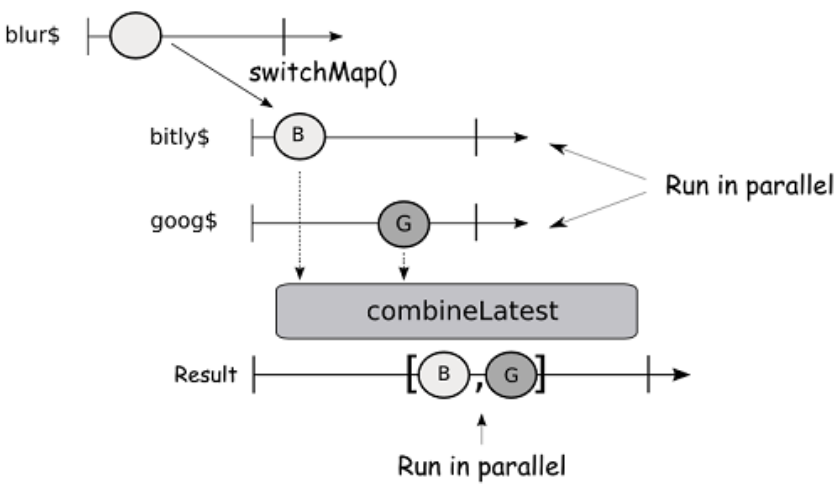

Figure 6.4 Shows the workings of combineLatest(). This operator outputs an array containing the latest value from all of its input observables.

Fortunately, `combineLatest()` allows you to provide a selector function that makes the stream more conformant, so that you can avoid the direct array access, which can be tedious and error prone when you only need one of the results. This selector function receives as arguments the data emitted from each subordinate observable. So, using a selector function that measures the length, we can get the shorter of the URLs computed:

```
const url$ = Rx.Observable.fromEvent(urlField, 'blur')
   .pluck('target', 'value')
   .filter(isUrl)
   .switchMap(input =>
         Rx.Observable.combineLatest(bitly$(input), goog$(input),
           (b, g) => b.length > g.length ? b : g))     (1)
    .subscribe(shortUrl => {
      console.log(`The shorter URL is: ${shortUrl}`);
    });
```

(1) **Using a selector function to pick the data from the stream that emits the shorted URL.**

For the sake of completing this example, let's finish implementing each individual stream because they pack another interesting technique used to deal with third-party APIs that work with callbacks. We'll implement both services as functions that accept a URL and return a stream used to shorten it. We'll start with `bitly$`. When you open a Bitly account, you'll need to find the following information in order to make remote web API request:

```
const API = 'https://api-ssl.bitly.com'; (1)
const LOGIN = '<YOUR LOGIN>';            (2)
const KEY = '<YOUR GENERATED KEY>';      (2)
```

(1) **Bitly's Web API URL**
(2) **You can obtain these fields from your profile settings**

Listing 6.6 Shows the code observable function used to shorten this URL:

Listing 6.6 Bitly URL shortener stream

```
const ajaxAsObservable = Rx.Observable.bindCallback(ajax); (1)

const bitly$ = url => Rx.Observable.of(url)
   .filter(R.compose(R.not, R.isEmpty))
   .map(encodeURIComponent)
   .map(encodedUrl =>     (2)
    `${API}/v3/shorten?longUrl=${encodedUrl}&login=${LOGIN}&apiKey=${KEY}`)
   .switchMap(url => ajaxAsObservable(url).map(R.head))     (3)
   .filter(obj => obj.status_code === 200 && obj.status_txt === 'OK')
   .pluck('data', 'url'); (4)
```

(1) **Bind the function's callback internally to the observer's next function**
(2) **Builds the API path**

❸ Invoke an AJAX call against Bitly with the longUrl to shorten
❹ Extract the URL property

For starters, we need to explain the first line in listing 6.6, which you haven't encountered before. It's a fact that many JavaScript APIs, particularly for Node.js, still use callback functions very heavily. Just as RxJS works well with promises, it's also important to be able to *adapt* callback based APIs into RxJS. The way to do this is by internally binding the callback as the observer's `next()` method and publishing that value as an observable to continue the chain.

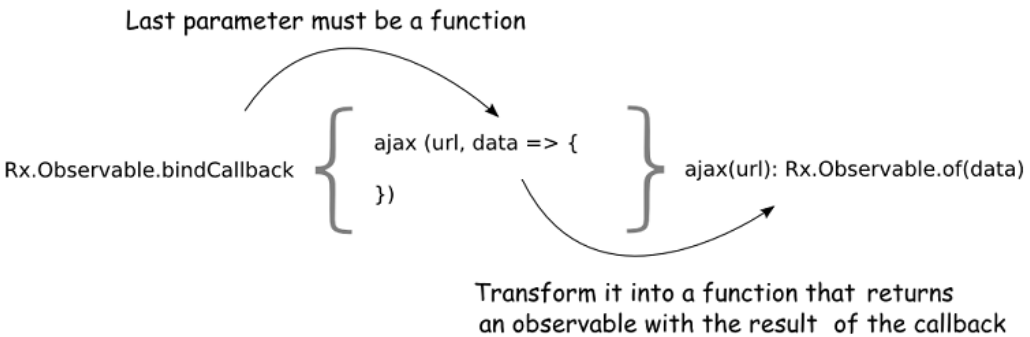

Figure 6.5 Bind callback transforms any function f(x, callback) into a function g, such that when called as g(x) will output an observable with the result of the callback

This way, when the bound `ajax()` function is invoked with the URL argument, it will execute and the result intended for the callback is proxied into a new observable. Because we're returning an observable, we use `switchMap()` to project it and replace the source stream. This is the only new part, everything else should be straightforward.

Furthermore, working with Google's URL shortener is very similar, except that for reasons of security and authentication, it's best to use their JavaScript client APIs instead of making raw request (details about installing this library can be found in Appendix A). Just like Bitly, Google's service expects you to have a Google account, this particular API enabled, and that you have generated a security OAuth2 token. This client API library `gapi` gives you access to many of Google's Web APIs, and it works partially with callbacks and promises. So, integrating it into RxJS involves wrapping those promisified method calls to configure the library and pushing it downstream as you set up to make the shorten call.

Listing 6.7 Google URL shortener stream

```
const GKEY = '<YOUR-GENERATED-OAUTH-KEY>'; ❶

const gAPILoadAsObservable = Rx.Observable.bindCallback(gapi.load); ❷

const goog$ = url => Rx.Observable.of(url)
   .filter(R.compose(R.not, R.isEmpty))
```

```
   .map(encodeURIComponent)
   .switchMap(() => gAPILoadAsObservable('client')) ❸
   .do(() => gapi.client.setApiKey(GKEY)) ❹
   .switchMap(() =>  ❺
       Rx.Observable.fromPromise(gapi.client.load('urlshortener', 'v1')))
   .switchMap(() =>  ❻
       Rx.Observable.fromPromise(gapi.client.urlshortener.url.insert(
         {'longUrl': example_url}))
   )
   .filter(obj => obj.status === 200)
   .pluck('result', 'id');
```

❶ Use your OAuth2 token generated through the Google APIs console
❷ Bind the callback into the load method so that we can integrate it into the observable
❸ Load the client library
❹ Pass the generated token
❺ Load the URL shortener API
❻ Shorten the URL and insert it into your personal list of URLs shortened

As you can see, we were able to compartmentalize both services as individual observables, only to embed them into an orchestrating observable using `combineLatest()` to run those services in parallel in reaction to the URL field changing. Here's that code once more:

```
const url$ = Rx.Observable.fromEvent(urlField, 'blur')
   .pluck('target', 'value')
   .filter(isUrl)
   .switchMap(input => ❶
       Rx.Observable.combineLatest(bitly$(input), goog$(input)))  ❷
   .subscribe(([bitly, goog]) => {
     console.log(`From Bitly: ${bitly}`);
     console.log(`From Google: ${goog}`)
  });
```

This code reveals that spawning and joining streams is made a first-class citizen in RxJS. To nail this point home, let's look at an operator called `forkJoin()`.

6.2.3 More coordination with fork-join

RxJS provides an operator called `forkJoin()`, in many ways similar to `combineLatest()`, in charge of running multiple observable sequences in parallel and collecting their last element. At the time of this writing, most modern browsers allow you to make up to 10 requests for data simultaneously, and `forkJoin()` takes advantage of this to maximize throughput. For the stock ticker widget, this operator is advantageous because we can look up simultaneous stock symbols, and then add them all up to reflect the grand total of the user's entire stock portfolio. Let's take a look at this example. Here's an outline of the steps:

1. Create a function that uses an observable to fetch the stock data for a company's symbol with price
2. Iterate through the user's preferred stock symbols: FB (Facebook), APPL (Apple), and CTXS (Citrix)
3. Use `forkJoin()` to spawn these simultaneous processes and join the result

4. Add the final result

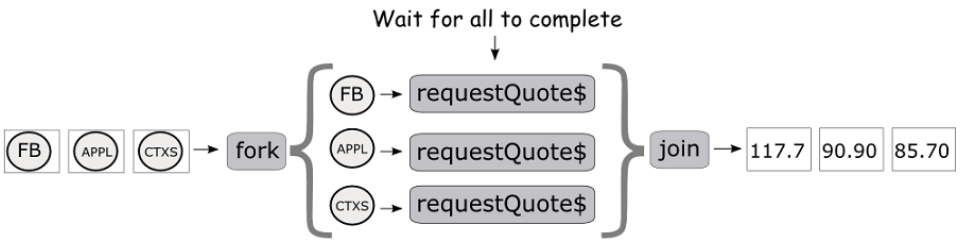

Figure 6.6 Shows that the fork operation spawns several requests, waits for them to complete, and emits when all streams have completed. The result is an array mapping to the output of each stream

To implement it, the first thing we'll do is to reuse the function that fetches a stock symbol's price `requestQuote$()` from our stock ticker widget in chapter 5.

```
const requestQuote$ = symbol =>
     Rx.Observable.fromPromise(
       ajax(webservice.replace(/\$symbol/, symbol)))
     .map(response => response.replace(/"/g, ''))
     .map(csv);
```

There are so many things we can do and it all depends on your needs. In this case, we're optimizing for parallelism. One of the things we did was make the stream conformant in that it only returns the price property of the fetched company symbol as a numerical float.

Remember from previous chapters, that the user has set to fetch stock information for three companies, namely:

```
const symbols = ['FB', 'AAPL', 'CTXS'];
```

To compute the total price, we need to query for each of these symbols in parallel, and add up the joined result. For this we'll use `forkJoin()`. I could pass each request observable one by one:

```
Rx.Observable.forkJoin(
   requestQuote$('FB'),
   requestQuote$('AAPL'),
   requestQuote$('CTXS')
);
```

This is very clean and declarative. Preferably, use our functional programming skills to map this function over the `symbols` array, as shown in listing 6.8:

Listing 6.8 Using forkJoin to fetch simultaneous stock symbols

```
Rx.Observable.forkJoin(symbols.map(requestQuote$))
 .map(data => data.map(arr => parseInt(arr[1]))) (1)
 .subscribe(allPrices => {
    console.log('Total Portfolio Value: ' +
        new USDMoney(allPrices.reduce(add).toLocaleString()));
 });
```

❶ Read the price amount only

Just like `combineLatest()`, `forkJoin()` will return an array with all stock prices all at once. The subscriber receives the array and reduces it with a simple `const add = (x, y) -> x + y;` function to produce the final result, which at the time of this run is:

```
"Total Portfolio Value: USD 293.25" ❶
```

❶ Total value subject to change depending on market conditions

As you can see this flow is very declarative, all immutable, and uses functional expressions to obtain the final answer. A simple look at the browser's console reveals that all simultaneous processes began at the same time:

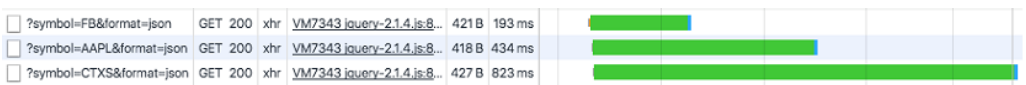

Figure 6.7 Browser's view of network traffic shows how the remote HTTP requests all start at the same time. The forkJoin() operator spawns these requests, and waits for all to emit before emitting its result

`forkJoin()` and `combineLatest()` are very similar, yet each imparts its own flavor. Aside from the former being strictly a static factory method and the latter used interchangeably, both differ in the criteria with which they emit their values. `forkJoin()` emits *only* the latest values from each of the input observables. So, if a sequence emits 5 values, it will sit there and wait for the last one (certainly expect some level of in-memory caching occurring here).

```
Rx.Observable.forkJoin(
  Rx.Observable.of(42),
  Rx.Observable.interval(1000).take(5)) ❶
.subscribe(console.log);  //-> [42,4]
```

❶ It will hold on to 42 for about 5 seconds and then emit the last value seen from all streams

On the other hand, `combineLatest()` is closer to a merge in the sense that it will emit values for the latest values when *any* of its input observables emits, namely:

```
Rx.Observable.combineLatest(
  Rx.Observable.of(42),
  Rx.Observable.interval(1000).take(5))
.subscribe(console.log);

//-> [42, 0]
     [42, 1]
     [42, 2]
     [42, 3]
     [42, 4]
```

As you saw in these examples, asynchronous data may arrive at any time, which makes coordination difficult to implement without a tool like RxJS. This is particularly important when synchronizing data operations into a database, for instance. Let's see how RxJS fairs with these kinds of problems.

6.3 Building a reactive database

When data sources are expected to arrive at different times or are tied to different source events, it can become difficult to properly coordinate them. As you saw earlier, operators like `combineLatest()` and `forkJoin()` both implement a joining pattern that one way or another waits for input observables to complete before emitting a value. This is incredibly powerful and the sort of behavior you'll find in sophisticated concurrency frameworks. We can also find plenty of uses cases of this pattern in backend systems, especially when dealing with data persistence.

The use case we'll tackle here is a simple banking transaction system that keeps track of all transactions as a user withdraws money from their account. Thinking reactively here we should recognize instances of join patterns as reacting to some action triggers another to occur. In this case, we'll need to join together or sequence a set of database calls to reflect a withdraw action and a transaction record being created. Around this problem domain, let's implement a few tasks such as loading all of a user's transactions from the database.

A common problem with sophisticated client side applications is loading all of the data from the backend into the browser, an environment that utilizes a limited amount of memory. Some architectures load the data as needed; this is called *progressive loading*. However, this doesn't work well if an application has high demands for performance or needs to work without internet. Most modern applications are expected to work this way. Another approach is to bypass the browser's memory and load the data into persistent storage. Let's go over the technology we'll be using.

IndexedDB is a great and relatively under-utilized web standard for client-side databases. It takes what was traditionally a server side process of storing data efficiently in some structured manner, and allows those same types of operations for the web. Unfortunately, the standard has a less than straight-forward interface. So for this example we will use an abstracted library modeled after the more popular CouchDB library, called PouchDB, which is more readable and handles browser differences (please visit the Appendix installation instructions).

The benefit of using PouchDB, like most modern asynchronous JavaScript APIs we interacted with earlier, is that it uses promises to model all of its asynchronous operations, which means we can use `Rx.Observable.fromPromise()` to adapt all of the API calls if we wanted to use observables, which is exactly what we'll do because we're smarter about preferring observables to regular promises. For instance the output of `PouchDB.put()`, a promise, method can be converted to an observable:

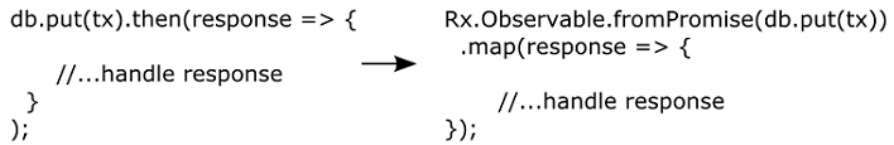

Figure 6.8 Adapting the callback-based API into an Observable

We can use RxJS to move this static, persistent data into flows of asynchronous operations that compose or cascade the outcome of one into the next seamlessly. Hence, RxJS becomes our query language, treating data as constantly moving and changing infinitely. Keep in mind that PouchDB is a schema-less document store, so this means we don't need to define and create schema before writing data to its tables. We'll start with a simple example that loads a set of banking transactions into the document store. Constructing an instance of the database is as simple as:

```
const txDb = new PouchDB('transactions');

R.isNil(txDb.then); //-> false      (1)
R.is(Function, txDb.then) //-> true (1)
```

(1) Demonstrating that the PouchDB APIs use promises behind the scenes

This database stores transaction documents in JSON form. A transaction has the following structure:

Listing 6.9 Transaction class

```
class Transaction {
  constructor(name, type, amount, from, to = null) {
     this.name = name;   (1)
     this.type = type;
     this.from = from;
     this.to   = to;
     this.amount = amount;
  }

  name() {
    return this.name;
  }

  from() {
    return this.from;
  }

  to() {
    return this.to;
  }

  amount() {
    return this.amount;
  }
```

```
  type() {
    return this.type;
  }
}
```

❶ Normally we would store the instance member fields using the underscore convention (this._name); however, this causes validation issues with PouchDB

Next, we'll populate our database with a few transaction records that represent a user transferred money from one account to another.

6.3.1 Populating a database reactively

The code to create and store several transactions involves looping through `Transaction` objects (whether they come from a locally stored array or from a remote HTTP call), date stamping each transaction with an RxJS timestamp, and posting it to the database as shown in figure 6.9:

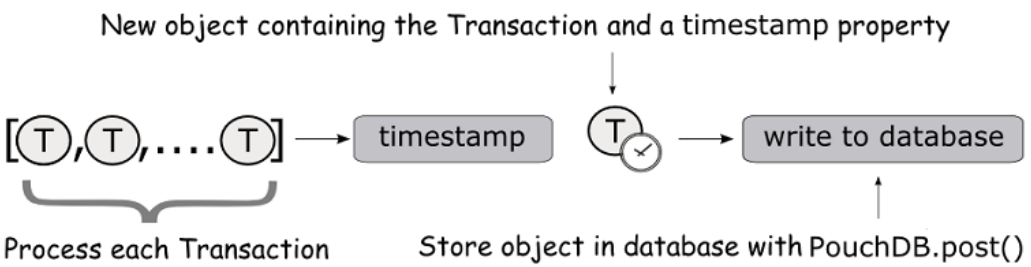

Figure 6.9 Steps to populate data into local storage using streams

We'll start by artificially populating the database with this data set:

```
function getTransactionsArray() {
  return [
   new Transaction('Brendan Eich', 'withdraw', 500, 'checking'),
   new Transaction('George Lucas', 'deposit',  800, 'savings'),
   new Transaction('Emmet Brown', 'transfer', 2000, 'checking', 'savings'),
   new Transaction('Bjarne Stroustrup', 'transfer', 1000, 'savings', 'CD'),
  ];
}
```

Listing 6.10 shows this in action. We'll create two streams one in charge of performing the database operation and the other processing the input:

Listing 6.10 Populating the database

```
const writeTx$ = tx => Rx.Observable.of(tx)
  .timestamp() ❶
  .map(obj => Object.assign({}, obj.value, { ❷
                date: obj.timestamp
```

```
            })
   )
  .do(tx => console.log(`Processing transaction for: ${tx.name}`))
  .mergeMap(datedTx => Rx.Observable.fromPromise(txDb.post(datedTx))); ❸

Rx.Observable.from(getTransactionsArray()) ❹
  .concatMap(writeTx$) ❺
  .subscribe(
    rec => console.log(`New record created: ${rec.id}`),
    err => console.log('Error: ' + err),
    ()  => console.log('Database populated!')
  );
```

❶ Attaching a timestamp to each emitted item that indicated when it was emitted. The resulting object has two properties obj.value which points to the emitted object (transaction) and obj.timestamp which contains the time the event was emitted
❷ Using ES6 Object.assign() to create a copy of the transaction object with the additional date property. Preserves immutability
❸ Post the object into the database by wrapping the PouchDB.post() operation with an Observable. This assigns the stored document a unique _id
❹ Reading the transaction objects from a local array
❺ Join the stream to process and create the new transaction document

Before we get into the details of this code, it's important to note that we were able to process and manipulate a set of objects and store them in a database, all in an immutable manner; this is very compelling and reduces the probability of bugs. Listing 6.10 involves multiple steps and new concepts:

1. We know we'll need to modify the transaction objects to include a date of when the transaction was processed. This is typical of any banking application as most transactions are sorted based on date. Because functional programs are immutable, instead of mapping a function to the transactions array and modify the object's internal structure directly, we can use JavaScript's ES6 `Object.assign()` to immutably create or set a new property into the object, leaving the original intact—we want our code to be as stateless as possible.
2. Now we retrieve the transaction data into an array. Given RxJS' unifying model of computation, we could easily retrieve data from a local array, or we could just as easily have fetched it with a remote HTTP call, as such:

   ```
   Rx.Observable.fromPromise(ajax('/transactions'))
      .timestamp()
      ...
   ```

3. We use the `Object.assign()` function to add `date` onto the transaction object iterated over passing the generated RxJS `timestamp()` operator. This operator creates an object with a timestamp and value property, containing the original object's data.

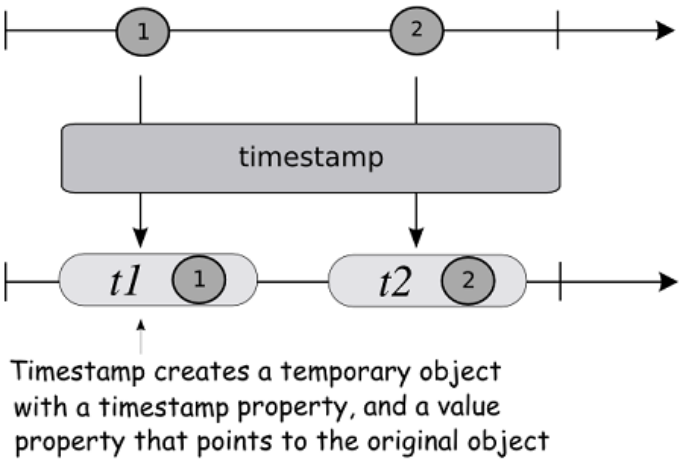

Figure 6.10 RxJS timestamp operator

4. Create each transaction object using the `post()` method of the PouchDB object. This object also sets a random generated key in the database table. While this method call inevitably creates a side effect in our application (writing to a database), it's one that's managed by RxJS and isolated to its own function—the rest of the code remains pure. As we said earlier, because PouchDB exposes a thenable API, we can wrap observables over it, creating our reactive database.
5. Finally, because the call to `post()` returns an observable we use `mergeMap()` to flatten the projected observable.

Running this code prints the following:

```
"Processing transaction for: Brendan Eich"
"New record created: 4F7404AF-10D2-8438-AEAB-CC21CDC23810"
"Processing transaction for: George Lucas"
"New record created: A9ACE7FE-85DB-484E-AA74-B47A7F4D32B1"
"Processing transaction for: Emmet Brown"
"New record created: DD469ACA-BC5C-A5C6-8E4A-0FB544C62231"
"Processing transaction for: Bjarne Stroustrup"
"New record created: B5C8B8C7-127B-11C7-A90E-64D79C8315E2"
"Database populated!"
```

Another benefit of wrapping observables over the database API is that all side effects are pushed down downstream to observers instead of each `Promise.then()` call. It's nice to keep your business logic pure as much as possible and side effects isolated.

Depending on the size of the transaction objects, when storing thousands of them in an array, we could end up with very large memory footprints. Of course, we would like to avoid keeping all of that data directly in memory, which is why we leverage the browser's database to store this data within it but persisted out of memory. To make this example simple we used a small array. Most likely you'll also want to transactions created locally as well as data

coming in remotely. Can you guess which operator we need? Correct! We can use RxJS' `merge()` to plug in all of data from multiple sources:

```
Rx.Observable.merge(  ❶
    getTransactionsArray(),
    Rx.Observable.fromPromise(ajax('/transactions')))
  .concatMap(writeTx$)
  .subscribe(
    rec => console.log(`New record created: ${rec.id}`),
    err => console.log('Error: ' + err),
    ()  => console.log('Database populated!')
  );
```

❶ **Merging the output from both local and remote streams**

The rest of the code continues to work exactly the same way. Brilliant! The asynchronicity of code is seamless in reactive programming!

And in the event that the remote HTTP call response is not an array, remember we can make the observable conformant just like we discussed above, and push some logic upstream like this. It's typical of remote calls to return an object with a status and a payload. So if you're response object is something like:

```
{
   status: 'OK',
   payload: [{name: 'Brendan Eich', ...}]
}
```

You can make it conformant as you inject it into the stream:

```
Rx.Observable.merge(
    getTransactionsArray(),
    Rx.Observable.fromPromise(ajax('/transactions'))
      .mergeMap(response => Rx.Observable.from(response.payload)) ❶
  )
  .concatMap(writeTx$)
  ...
```

❶ **Converts the JSON response object into an array that gets combined with the other transaction records and pushed through the stream**

Moreover, databases are full of optimizations to improve read and write speed. We can further help these optimizations by performing bulk operations whenever possible.

6.3.2 Writing bulk data

The previous code samples create single bank transaction records at a time. We can optimize this process with bulk operations. Bulk operations write an entire set of records with a single post request. Naturally, the PouchDB operation `bulkDocs()` takes an array. We talked about how much memory used to build this set before, and this is completely in your control using RxJS buffers.

The `buffer()` operator which you saw back in chapter 4 would come in very handy here when we're not just processing a handful of transactions, but hundreds of them. Let's optimize listing 6.10 with listing 6.11:

Listing 6.11 Optimizing write operations using bulk writes

```
Rx.Observable.from(getTransactionsArray())
  .bufferCount(20) ❶
  .timestamp()     ❷
  .map(obj => {
      return obj.value.map(tx => {  ❸
        return Object.assign({}, tx, {
            date: obj.timestamp
          })
      })
  })
  .do(txs => console.log(`Processing ${txs.length} transactions`))
  .mergeMap(datedTxs =>
        Rx.Observable.fromPromise(txDb.bulkDocs(datedTxs))) ❹
  .subscribe(
    rec => console.log('New records created'),
    err => console.log('Error: ' + err),
    ()  => console.log('Database populated!')
  );
```

❶ Buffer 20 transactions at a time
❷ Timestamp the entire set of items once
❸ Loop within each set and assign a date to each transaction object
❹ Perform bulk operation on passing the entire buffer

To support this optimization, we had to make a few adjustments. After collecting 20 objects with `bufferCount(20)`, the data passing through the stream is now an array instead of a single record.

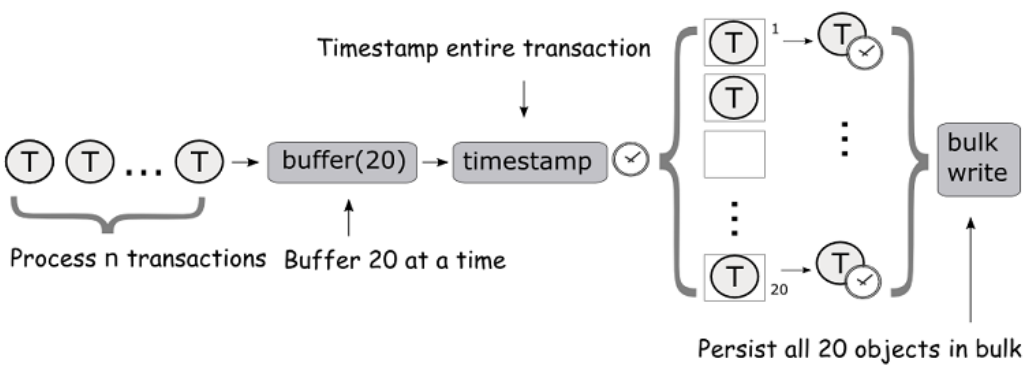

Figure 6.11 Flow followed to add items in bulk (in this case, 20 at a time)

Alternatively you could have also buffered for a certain period of time with `buffer(Rx.Observable.interval(500))`, this decision will depend on the amount of data your application will process. In this case, each record will be kept in a buffer for 500 milliseconds, at which point it will be released and all the records can be written in bulk to the database.

However, there is a problem with just using a count or time based buffer. If the user attempts to navigate away from the page while the data is being cached, we could potentially lose anything that is waiting in the buffer, up to 20 transactions in this case, which will never get saved. To fix this let's introduce another observable to trigger a buffer write. Buffers also support signaling, so that it the emission can occur in response to the execution of some browser hook, such as the closing of the window. To implement this we can use the `bufferWhen()` operator with an observable that's smart enough to support both use cases: to cache the results for a specific period of time or emit before the browser closes:

```
Rx.Observable.from(getTransactionsArray())
  .bufferWhen(() =>          ❶
    Rx.Observable.race(      ❷
      Rx.Observable.interval(500),
      Rx.Observable.fromEvent(window, 'beforeunload')) ❸
   )
   ...
```

❶ **Buffers events from the source Observable until the provided Observable emits**
❷ **Created an Observable that mirrors the first Observable to emit a value of the ones provided to it. In this case, it will emit after half a second, or if the window closes, whichever comes first.**
❸ **Hooking into the browser closing event. Because the contents within the buffer storage are emitted as a single array object and processed synchronously, there's no danger of the browser shutting down the buffer gets processed.**

`bufferWhen()` instead of taking an observable to trigger the *start of* each new buffer, accepts a closing selector method that is re-invoked every time the buffer is closed and the resulting observable is used to determine when the next buffer should close. Using this we can create a signal observable that has a whole host of possible constraint states. Now that you know how to get data into the database, let's join with a query that can count the total number of records.

6.3.3 Joining related database operations

All of the local store operations, whether you're using IndexedDB directly or PouchDB, happen asynchronously, but with RxJS you can treat your operations almost as if they were synchronous due to the abstraction that it poses over the latency involved in database calls. To illustrate this, we'll chain together an operation to insert a record, followed by an operation that retrieves the total count.

PouchDB is a map/reduce database, so in order to query the data, you must first define how the projection or the mapping function works. This object is called a *design document*, which you need to include as part of the query. For our purpose, we'll keep it simple and

count number of transactions performed. So our design document, we'll call it `count`, looks like this:

```
const count = {
  map: function (doc) {
    emit(doc.name); ❶
  },
  reduce: '_count' ❷
};
```

❶ Count the number of users
❷ Using the reduce PouchDB aggregate operator _count

Now listing 6.12 shows how we can join two queries with a single stream declaration:

Listing 6.12 Two queries in a single stream declaration

```
Rx.Observable.from(getTransactionsArray())
  .switchMap(writeTx$) ❶
  .mergeMap(() => Rx.Observable.fromPromise(
     txDb.query(count, {reduce: true}))) ❷
  .subscribe(
    recs  => console.log('Total: ' + recs.rows[0].value), ❸
    error => console.log('Error: ' + error),
    ()    => console.log('Query completed!')
  );
```

❶ Post a single transaction
❷ Run a reduction query to count the total number of documents in the table
❸ Print the total value

PouchDB also has some reduction operations on its own, and you understand what a reduction is because you're a experienced functional programmer by now. Aside from `_count`, you can perform other reductions such as `_sum`, and `_stats`. Let's go over another example that mixes all of what we've learned thus far. It performs a withdraw from the account database and creates a new transaction document.

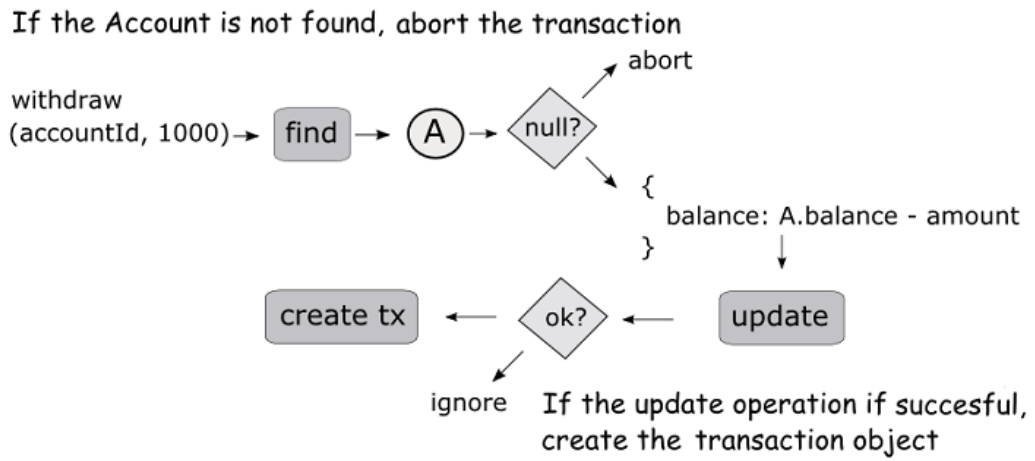

Figure 6.12 Shows the backend workflow that takes place when a withdraw operation occurs. First we find the account by ID, if it results in a valid object, we subtract the withdraw amount and update the account.

First we'll need to seed a set of user accounts with the following structure (again, we'll keep our domain simple):

Listing 6.13 The Account class

```
class Account {
  constructor(id, name, type, balance) {
     this._id = id;       (1)
     this.name = name;
     this.type = type;
     this.balance = balance;
  }

  id() {
    return this._id;
  }

  name() {
    return this.name;
  }

  type() {
    return this.type;
  }

  balance() {
    return this.balance;
  }
}
```

(1) The _id field is used to tell PouchDB's put() method to use our provided ID instead of generating a new one. We can use PouchDB's get() method to query by this ID

Similarly, we'll create a few different types of accounts for our user Emmet Brown:

```
const accounts = [
     new Account('1', 'Emmet Brown', 'savings', 1000),
     new Account('2', 'Emmet Brown', 'checking', 2000),
     new Account('3', 'Emmet Brown', 'CD', 20000),
  ];
```

To populate a new document store:

```
const accountsDb = new PouchDB('accounts');
```

Because you're already familiar with creating databases and populating it, we'll jump right into our `withdraw()` function, which returns an observable responsible for creating the flow to query and update multiple databases:

Listing 6.14 Withdraw function

```
function withdraw$({name, accountId, type, amount}) {    ❶
   return Rx.Observable.fromPromise(accountsDb.get(id)) ❷
     .do(doc => console.log(
         doc.balance < amount ?
          'WARN: This operation will cause an overdraft!' :
          'Sufficient funds'
      ))
     .mergeMap(doc => Rx.Observable.fromPromise(        ❸
        accountsDb.put({
         _id: doc._id,
         _rev: doc._rev,
         balance: doc.balance - amount
        }))
     )
     .filter(response => response.ok)  ❹
     .do(() =>
         console.log('Withdraw suceeded. Creating transaction document'))
     .concatMap(() => writeTx$(
          new Transaction(name, 'withdraw', amount, type)));   ❺
}
```

❶ Unpack the input into the parameters needed for the transaction
❷ Retrieve the existing account info
❸ Update the user balance
❹ Only continue if the DB update was successful
❺ Create the transaction record and return it

You can run this code by passing it an operation object literal:

```
withdraw$({
  name: 'Emmet Brown',
  accountId: '3',
  type: 'checking',
  amount: 1000
})
.subscribe(
   tx    => console.log(`Transaction number: ${tx.id}`),
   error => console.log('Error: ' + error),
```

```
   ()    => console.log('Operation completed!!')
);
```

Which will generate the following output:

```
"Sufficient funds"
"Withdraw suceeded. Creating transaction document"
"Processing transaction for: Emmet Brown"
"Transaction number: DB6FF825-C7O3-0F1A-B86O-DA6B1138F723"
"Operation completed!!"
```

As you can see, because the API of PouchDB uses promises, it's easy to integrate you're your database code with your business logic, all wrapped and coordinated via the observable operators. While database calls are a form of a side effect, is one we're willing to take in practice, and rely on the unidirectional flow of streams to streamline the use of this shared state. But wrapping API calls is not the only thing you can do with PouchDB. In addition, you can build support for a reactive database.

6.3.4 Reactive databases

PouchDB is an Event Emitter, which means it exposes a set of events or hooks for you to plug in logic into certain phases of its lifecycle. Just like GitHub exposes hooks to tap into when branches are created, you can add event listeners that fire when databases are created and destroyed.

This is very important in browser storage where the lifespan of a database is temporary as it can be destroyed and re-created at any point in time. So before we begin adding any documents, it will be good to do so under the context of database created hook.

Using the `Rx.Observable.fromEvent()` operator you can transform any event emitter into an observable sequence. So, hooking into the database creation event looks like the following:

```
Rx.Observable.fromEvent(txDb, 'created')
  .subscribe(
    () => console.log('Database to accept data!')
  );
```

So, adding this check into our streams is very easy. All you need to do is key off of that hook, to perform all of your logic. This is somewhat similar to waiting for the document to be ready, before executing any of our JavaScript code. The withdraw operation would look like this:

```
Rx.Observable.fromEvent(txDb, 'created') ❶
  .switchMap(() =>
            withdraw$({
             name: 'Charlie Brown',
             accountId: '1',
             type: 'checking',
             amount: 1000
            })
 )
 ...
```

❶ **React to the 'created' event**

In this chapter you saw how we can bring together multiple distinct sub-systems, and build coherent state machines from them. Each example brought out a small piece of functionality which could be independently attached to and handled. The combinatorial operators allow use to join each stream while still maintaining the same separation of concerns that we had achieved with single streams. Notice that so far, none of the code we've written has accounted for error or exceptions. What if the there's an error inserting a record in a database? What if there's an exception happening when we call some third-party function? In the next chapter we will take these same concepts and see how to make our applications more fault tolerant against the unexpected.

6.4 Summary

- Synchronized email verification with Observables to process an out-of-flow user interaction with a 3rd party.
- Joined parallel URL shortening services with `combineLatest()` and spawn multiple observable sequences with `forkJoin()`.
- Used buffering to improve the performance database queries.
- Used observables to control the lifespans of non-observables like user sessions.
- Reactive databases allow you to orchestrate business flows involving permanent storage

7

Error Handling with RxJS

This chapter covers

- The issues with imperative error–handling schemes
- Using functional data types to abstract exception handling
- Using Observable operators to handle exceptions
- Different strategies for retrying Observable sequences

Until now, we've only explored the happy-path use cases involving the use of RxJS to tackle lots of different use cases. We suspect that at some point you've probably asked yourself: "What would happen if a remote HTTP call failed while fetching data for my stock quote widget?" Observers see the outcome of combining and transforming sequences of streams that you use to map your business logic to. But if an exception occurs mid-stream, what will the observer see at that point? These are some very valid and important questions, but it was important that you first understood and learned to think reactively with ideal scenarios; now we're going to sprinkle a dose of the real-world onto our code. And the brutal reality is that software will likely fail at some point during its execution.

Many issues can arise in software where data inadvertently becomes `null` or `undefined`, exceptions are thrown, or network connectivity is lost, to name a few. Our code needs to account for the potential of any of these issues occurring, which unavoidably creates complexity. In other words, we can't escape errors, but we can learn how to deal with them. One strategy that developers often use is to scatter error handling code around every function call. We do this to make our code more robust and fault tolerant, but it has the detrimental effect of making it even more complex and harder to read.

In this chapter, you'll learn that the key to elegant error-handling in RxJS is done in part by the effective use of observables and by following proper functional programming principles, as you've seen all along. Given a properly constructed observable stream, the next step is to

learn about the different RxJS observable operators that you can plug in to respond to any adversity. Before we get started, it's important to understand that you need to put aside the imperative error-handling techniques you're accustomed to like try/catch, in favor of a functional approach as implemented in RxJS.

7.1 Common error-handling techniques

JavaScript errors can occur in many situations, especially when an application fails to communicate with a server when invoking an AJAX call. Also, third-party libraries that you load into your project can also have functions that throw exceptions to signal special error conditions. Hence, we always need to be prepared for the worst and design with failure in mind, instead of letting it become an afterthought and regretting it later.

In the imperative world, exceptions are typically handled with the common try/catch idiom, which occurs frequently with synchronous code. Conversely, in the asynchronous world—remote HTTP calls and event emitters—you're required to interface with functions that delegate failures to callback functions. And recently with JavaScript ES6, many libraries have switched to using promises to wrap their asynchronous computations. Let's examine each of these cases individually.

7.1.1 Error-handling with try/catch

JavaScript's default exception-handling mechanism is geared toward throwing and catching exceptions through the popular try/catch block, also pervasive in most modern programming languages. Here's a sample:

```
try {
   someDangerousFunction();
}
catch (error) {
   // statements to handle any exceptions
   console.log(error.message);
}
```

As you know, the purpose of this structure is to surround a piece of code that you deem to be unsafe. Upon throwing an exception, the JavaScript runtime abruptly halts the program's execution and creates a stack trace of all the function calls leading up to the problematic instruction. Specific details about the error, such as the message, line number, and filename, are populated into an object of type `Error` and passed into the `catch` block.

> **RXJS 5 EXCEPTIONS** RxJS 5 has a number of improvements over its previous version. One of them involves the simplification of the internal mechanisms of RxJS, resulting in a stack trace that's much easier to parse.

The `catch` block becomes a safe haven so that you can potentially recover your program. However, with your knowledge about observables, you can see how this imperative style of

dealing with exceptions is structurally very different to what we've done so far. So, adding try/catch to your RxJS code, to provide error handling logic to a stream would look like the following:

```
try {
   const data$ = Rx.Observable.fromPromise(ajax('/data'))
       .subscribe(console.log);
}
catch(error) {
   console.log(`Error processing stream: ${error.message}`);
}
```

Now imagine having to merge and combine multiple streams together, each with its own type of failure, you can quickly see how this pattern couldn't possibly support this if you need to wrap each stream with their own try/catch. With asynchronous functions, the common JavaScript pattern is to provide the error callback alongside the success callback.

7.1.2 Delegating errors to callbacks

As is common with asynchronous functions in many JavaScript libraries, there's typically a function that responds to the success case, and one that handles errors. This is necessary because asynchronous functions are unpredictable in terms of if and when they return or if errors occur. Until now, we purposely avoided talking about the error cases when using an asynchronous function like `ajax()`. We've been using this function all along, as a kind of black box, that always ran correctly. You could use it in two different ways: with callbacks or with promises. Let's peek under the hood of this function using callbacks in listing 7.1:

Listing 7.1 Function ajax() with success and error callbacks

```
const ajax = function (url, success, error) {
   let req = new XMLHttpRequest();      (1)
   req.responseType = 'json';
   req.open('GET', url);
   req.onload = function() {
      if(req.status == 200) {
         let data = JSON.parse(req.responseText);
         success(data);                 (2)
      }
      else {
         req.onerror();
      }
   }
   req.onerror = function () {
      if(error) {
         error(new Error('IO Error')); (3)
      }
   };
   req.send();
};
```

(1) **Initialize an XmlHttpRequest object used to fetch data remotely**
(2) **On success, parse the data as JSON and invoke the success() callback**

❸ On error, what the error message into an exception object

Using this function, code that would require multiple nested sequences of HTTP calls such as when mashing up different sources of data, would look like listing 7.2:

Listing 7.2 Imperative error handling with asynchronous code

```
ajax('/data',
   data => {
        for (let item of data) {
          ajax(`/data/${item.getId()}/info`,
               dataInfo => {
                 ajax(`/data/images/${dataInfo.img}`,
                   showImage,
                   error => {  ❶
                      console.log(`Error image: ${error.message}`);
                   });
              },
              error => {       ❷
                 console.log(`Error each data item: ${error.message}`);
              });
   },
   error => {                  ❸
        console.log(`Error fetching data: ${error.message}`);
   }
});
```

❶ Handle the inner most HTTP call
❷ Handle second level HTTP call
❸ Handle the first, outermost HTTP call

Looking at this code from just a structural point of view, you can picture it as nested code blocks such as the ones shown in figure 7.1:

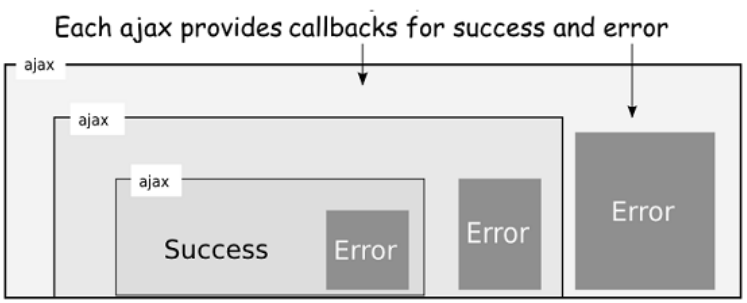

Figure 7.1 Imperative asynchronous error handling tends to nest when making a series of asynchronous calls.

Indeed, while our code is more fault tolerant, all we've done here is exacerbate the problem of having to parse this nested "pyramid of doom," which we spoke about in chapter 1. As a

result, the JavaScript ES6 specification introduced promises, which elegantly streamline the invocation of a sequence of asynchronous function to tackle cases such as this.

7.1.3 Errors and promises

The `Promise.then()` function acts as the mapping function (similar to `Rx.Observable.map()`) used to project (or map) another promise to a source promise. This is the reason why we decided to "promisify" `ajax()`, and it's what we've been using for most of the examples as a much superior form to its callback counterpart. Here's the code for that:

Listing 7.3 Promisifed ajax()

```
const ajax = function (url) {
    return new Promise(function(resolve, reject) {  (1)
        let req = new XMLHttpRequest();
        req.responseType = 'json';
        req.open('GET', url);
        req.onload = function() {
            if(req.status == 200) {
               let data = JSON.parse(req.responseText);
               resolve(data);  (2)
            }
            else {
               reject(new Error(req.statusText));     (3)
            }
        };
        req.onerror = function () {
           reject(new Error('IO Error'));  (3)
        };
        req.send();
    });
 };
```

(1) Create and return the HTTP call wrapped in a Promise
(2) Promise is resolved if the data is fetched successfully
(3) Promise is rejected in case a failure occurred while performing the remote request

Much like observables, you can chain multiple asynchronous calls by mapping new promises to a source promise. Then you can use the `Promise.catch()` operator, to implement an error handling strategy that answers to any of the rejected promises or ones that throw exceptions, as such:

```
ajax('/data')
   .then(...)
   .catch(error => console.log(`Error fetching data: ${error.message}`))
```

Because `catch()` itself returns a Promise, you can implement specific errors by inserting multiple asynchronous calls to `catch()` in series, just like this:

```
ajax('/data')
   .then(item => ajax(`/data/${item.getId()}/info`))
   .catch(error => console.log(`Error fetching data: ${error.message}`))
```

```
    .then(dataInfo => ajax(`/data/images/${dataInfo.img}`))
    .catch(error => console.log(`Error each data item: ${error.message}`))

       .then(showImg);
       .catch(error => console.log(`Error image: ${error.message}`))
```

Arguably, in comparison to figure 7.1, this statement resembles a much easier structure to parse:

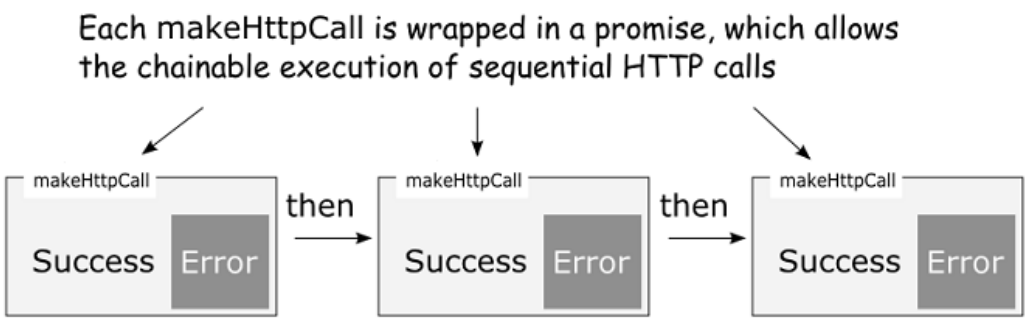

Figure 7.2 Promises allow you to chain subsequent asynchronous calls, each with their own success and error (catch) callbacks.

If the first `ajax()` fails, the first `catch()` operator runs before jumping into the next promise in the chain. Each catch can be thought of as a recovery block for the previous Promise, the Promise allows us resume processing in some known state.

> **CONTINUOUS CATCH** The code example above introduces a small bug that we ignored to prevent cluttering up the code. Since catches are also part of the continuation, the handler method can either return a value or another promise, if no value is returned then an `undefined` value will be passed to the next continuation block.

Just like with a synchronous try/catch we can either continue by recovering from the error, in this case by returning a non-error value or we can re-throw the error. In the catch block that is done by either returning a `Promise.reject()` or by throwing within the callback method. But because you're basically just transferring control from one promise to the next, it's more typical to just implement a single, global `catch()` operator (this is essentially equivalent to an overarching try/catch block over your entire function body). In this case, when any promise fails the `catch()` operator is run and the entire body of code is exited:

```
ajax('/data')
   .then(item => ajax(`/data/${item.getId()}/info`))
   .then(dataInfo => ajax(`/data/images/${dataInfo.img}`))
   .then(showImg)
   .catch(error => console.log(error.message));
```

Certainly promises gets us closer to where we want to be. Unfortunately, all of these approaches limit your ability to make your code responsive and reactive; in other words, you can't easily return a default value in case a request failed, or perhaps retry a rejected promise. You can get around passing default values down the chain by introducing side effects in your code. And you can implement retries with the help of third-party libraries, such as Q.js (https://github.com/kriskowal/q). But more importantly, recall from our earlier discussions that promises model single asynchronous values, not a deluge of them, which is the type of problems we solve when combining functional and reactive programming. Let's examine in more detail the reasons that make these imperative error handling mechanisms incompatible with a reactive application.

7.2 Incompatibilities between imperative error-handling techniques and functional and reactive code bases

The structured mechanism of throwing and catching exceptions in imperative JavaScript code has many drawbacks when used in a functional or reactive style. In general, functions that throw exceptions

- Can't be composed or chained like other functional artifacts.
- Violate the principle of pure functions that advocates a single, predictable value because throwing exceptions constitutes another exit path from your function calls.
- Cause side effects to occur because an unanticipated unwinding of the stack impacts the entire system beyond just the function call or the stream declaration.
- Violate the principle of non-locality because the code used to recover from the error is distanced from the originating function call. When an error is thrown, a function leaves the local stack and environment. For instance:

```
try {
     let record = findRecordById('123');

     ... potentially many lines of code in between
}
catch (e) {
     console.log('ERROR: Record not found!');

     // Handle error here
}
```

- Put a great deal of responsibility on the caller to declare matching `catch` blocks to manage specific exceptions instead of just worrying about a function's single return value.
- Are hard to use asynchronously. The try/catch idiom is effective when enclosing synchronous code, where errors are syntactically bounded by the enclosing `try` blocks. This code is predictable and not affected by time and latency. Asynchronous functions, on the other hand, are unpredictable and typically provide an error callback mechanism to give the user control back of the program.

- Are hard to use when multiple error conditions create nested levels of exception-handling blocks:

```
var record = null;
try {
   record = findRecordByName('RecordA');
}
catch (e) {
   console.log('ERROR: Cannot locate record by name');

   try {
      record = findRecordById('123');
   }
   catch (e) {
      console.log('ERROR: Record is no where to be found!');
   }
}
```

After reading all of these statements, you're probably asking yourself: "Is throwing exceptions completely off the table?" We certainly don't believe so. In practice, they can never be off the table because there are many factors outside of your control that you may need to account for like system or environmental errors, or calls to third-party code that's outside of your control.

We're not recommending you don't use exceptions at all because they do serve a purpose—just use them for *truly exceptional* conditions. When you need to use exceptions or deal with errors, the functional approach, rather, is to allow functional data types to abstract it away your main business logic; this prevents you from creating side effects or code that becomes hard to maintain.

7.3 Understanding the functional error-handling approach

The functional approach to error handling is actually quite simple. As we mentioned before, we won't get too deep into any functional topics in this book, so we'll provide a simplistic view of this approach that will serve to better understand the design of RxJS' error handling mechanism. The goal here is to reify or make a first-class citizen the idea of a wrapper around a function or body of code that has the potential of throwing an exception. If you think about it for a second, that's really what you've been doing all along when you use a try/catch block. Function `findRecordById()` can throw an exception in the event that a database record is not found:

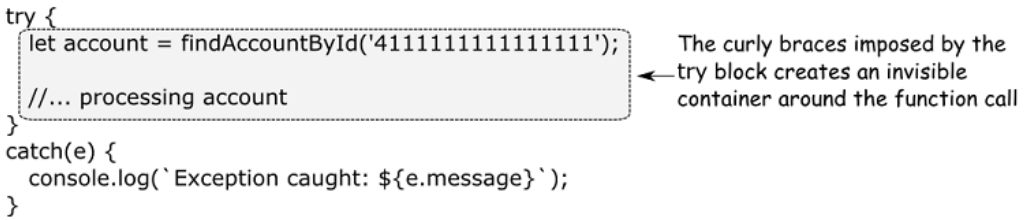

Figure 7.3 A try/catch block creates an invisible section that protects any body of code

The `try` block creates an invisible enclosure around the function call so that you can implement all your error handling logic inside the `catch` block. In the functional world, we'll reify this container with a data type called `Try`.

> **NOTE** This data type is a common pattern in functional programming that we introduce here merely as a theoretical construct, which will help you understand how Observables implement this later.

Here's how this data type would work:

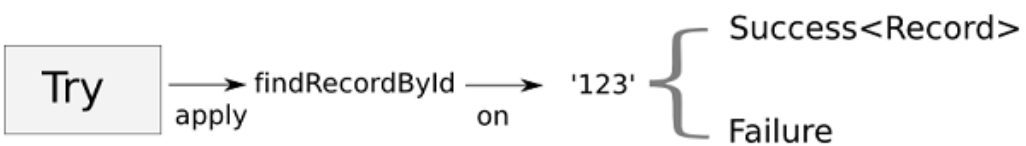

Figure 7.4 Use a data type called Try to make errors a first-class citizen of your application. This can be used to wrap any value, and then safely apply or map functions to it. If a function invocation is successful (no exceptions produced), a data called Success is returned; otherwise, an object of Failure is returned

We can use this type to apply or map a function to a certain value. This is equivalent to simply invoking the function with that parameter. With this extra plumbing, `Try` allows us to provide the necessary abstraction to return an object of type `Success` if an record object is found, or an object of type `Failure` signaling that something unexpected occurred otherwise:

```
Try.of(findRecordById('123')); //-> Success(Record)
Try.of(findRecordById('456')); //-> Failure
Try.of(findRecordById('xxxxx'))
   .getOrElse(new Record(...));   //-> Default value
```

Now, just like any functional data type, suppose `Try` also had a `map()` operator, which we can use to perform any action on the resolved object, if one is found:

```
Try.of(findRecordById('123')).map(processRecord);
```

Using `Try` as the return type of your functions is actually quite useful, because not only are you protecting the value it returns from a possible `null` access, but also you're letting your

users know that this particular function might produce an invalid result—it's self-documenting. This is why other languages such as Scala, Java, and Haskell one way or another provide native APIs for this data type.

For the purpose of our discussion, we'll show some of the pieces of `Try` in listing 7.4, as well as its derived types `Success` and `Failure`:

Listing 7.4 Internals of the Try functional data type

```
class Try {
    constructor(val) {  ❶
      this._val = val;
    }

    static of(val) {  ❷
       if(val === null || val.constructor === Error
          || val instanceof Error) {
                return new Failure(val);
          }
       return new Success(val);
    }

    map(fn) {  ❸
       try {
          return Try.of(fn(this._val));
       }
       catch (e) {
          return Try.of(e);
       }
    }
}

class Success extends Try { ❹

   getOrElse(anotherVal) {
     return this._val;
   }

   getOrElseThrow() {
      return this._val;
   }
}

class Failure extends Try { ❺

  map(fn) {
    return this;
  }

  getOrElse(anotherVal) {
     return anotherVal;
  }

  getOrElseThrow() {
    if(this._val !== null) {
       throw this._val;
    }
```

```
  }
}
```

1. Create a new instance of this data type
2. If Try a successful computation, wrap the result in a Success; otherwise, wrap the result in a Failure
3. Map applies a function to a value with internal try/catch logic, and returns an instance of Try, to continue chaining more operations (this is analogous to Rx.Observable.map())
4. Success represents a successful computation, with a method to get the value
5. Failure represents a function that resulted in an exception being thrown. Any subsequent mapping operations are skipped

> **SYNTAX** Listing 7.4 uses the class syntax in ES6 to model the `Try` data type. We use classes only because they're syntactically shorter that using functions and object prototypes. As you probably know by now, classes are nothing more than syntactic sugar over JavaScript's existing prototype-based inheritance. Whether you decide to implement this using function or class syntax is entirely up to you.

Listing 7.4 shows just a few of the key details of this functional data type. `Try` models two scenarios:

- If an instance of `Try<Record>` represents a successful computation, it is an instance of `Success<Record>` internally that is used to continue the chain.
- If, on the other hand, it represents a computation in which an error has occurred, it is an instance of `Failure<Error>`, wrapping an `Error` object or an exception.

What we accomplish with this is a simple data type that allows us to pipeline, or chain operations on objects, catching exceptions along the way, without impacting our business logic and hiding away the imperative try/catch structure. Here's how you can use it:

```
let record = Try.of(findRecordById('123'))
   .map(processRecord)
   .getOrElse(new Record('123', 'RecordA'));
```

Arguably this code is much more readable and has fewer side effects than:

```
var record = null;
try {
   record = findRecordById('123');
}
catch (e) {
   record = new Record('123', 'RecordA');
}
processRecord(record);
```

In the functional case, if the initial find operation were to fail, nothing in this logic would actually change because the error would be propagated internally via `Failure` instances, finally resulting in the `getOrElse()` function that creates and returns a default record object. This simple design pattern is really powerful, because it abstracted error handling completely from our business logic so that our functions only worry about writing code to solve my task at hand, while remaining side effect free. Let's show the workings of this with a diagram in figure 7.5:

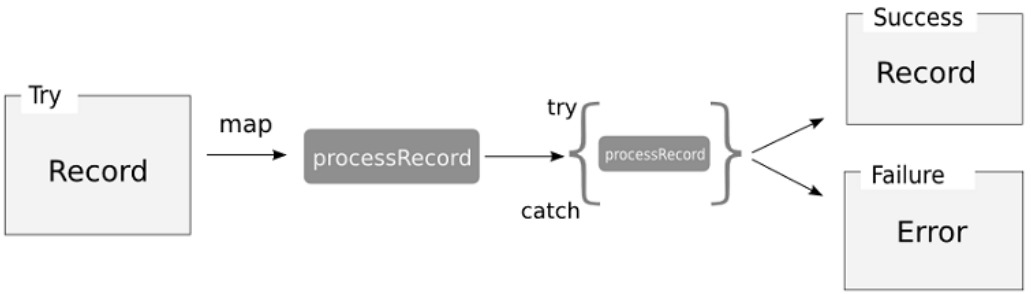

Figure 7.5 Mapping a function to a Try returns the result wrapped in a Success type, or an exception wrapped in a Failure

Does this discussion about propagation of change and the mapping of functions ring a bell? That's right! The observable data type works exactly the same way, and now we'll see how it implements its own exception handling operators.

7.4 The RxJS way of dealing with failure

Just like observables abstract data flow and processing, they also abstract errors and exception handling as well. RxJS' observable type provides several strategies for you to manage the errors that could arise mid-stream. In this section, you'll learn about:

- Propagating errors to observers
- Catching errors and reacting accordingly
- Retrying a failed operation for a fixed number of times
- Reacting to failed retries

7.4.1 Errors propagated downstream to observers

In chapter 2 we mentioned that at the end of the observable stream is a subscriber waiting to pounce on the next event to occurs. This subscriber implements the `Observer` interface, consisting of three methods: `next()`, `error()`, and `complete()`.

In general, *errors don't escape the observable pipeline*. They are contained and guarded to prevent side effects from happening—much like `Try`, as shown in figure 7.6:

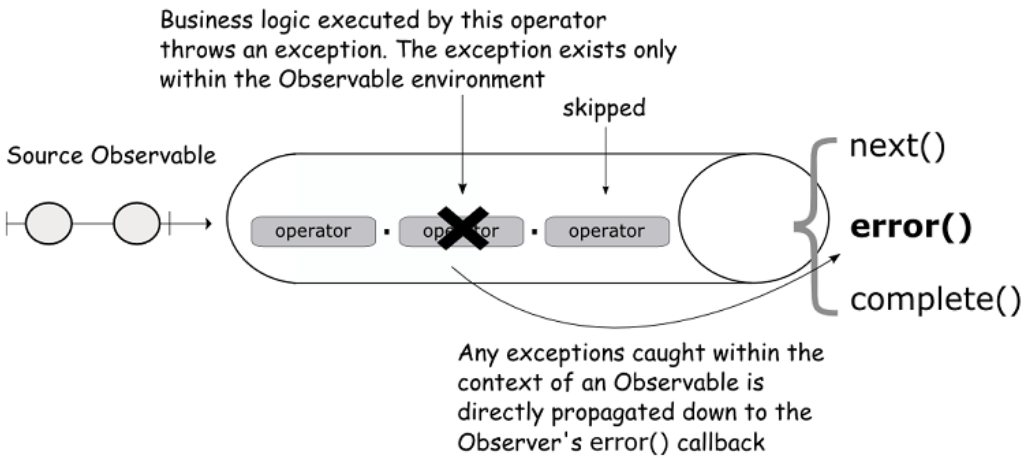

Figure 7.6 Errors that occur within some operator are not allowed to escape the context of the Observable; rather, errors can handled within the pipeline (as we'll see later); otherwise, the Observer's error() function is called

Errors that occur at the beginning of the stream or in the middle are propagated down to any observers, finally resulting in a call to `error()`. Here's a quick example to illustrate this concept:

Listing 7.5 Calling the error method on observers when an exception is thrown

```
const computeHalf = x => Math.floor(x / 2);
Rx.Observable.of(2,4,5,8,10)
   .map(num => {
      if(num % 2 !== 0) {
        throw new Error(`Unexpected odd number: ${num}`); ❶
      }
      return num;
   })
   .map(computeHalf)
   .subscribe(
       function next(val) {
          console.log(val);
       },
       function error(err) {
          console.log(`Caught: ${err}`); ❷
       },
       function complete() {
          console.log('All done!');
       }
    );
```

❶ Our business logic spits out an exception somewhere in mid-stream
❷ Without any exception handlers (discussed later in this chapter), any errors are automatically propagated down to the observers

Running this code prints:

```
1
2
"Caught: Error: Unexpected odd number: 5"
```

You can consider this approach as being very similar in structure to an overarching try/catch block. The important aspect to note from this example is how the observable data type is essentially acting like a `Try` by preventing the exception to leak out from the stream's context. Because there's no way to recover, the first exception that fires will result in the entire stream being cancelled. Think of parsing data from a network call, we would want to obviously skip parsing the object if the network call was unsuccessful. The error is pushed down to any subscribes so that they can perform any side effects such as showing an alert popup, a modal dialog, etc. Most of the time, though, you'll want to catch and recover from the error that occurred. To make understanding the different recovery strategies easier, we'll continue using this simple numerical example form listing 7.5 as our theme.

7.4.2 Catching and reacting to errors

Most of the time, you'll want to catch and recover from any errors so that your application is always responsive and resilient—one of the main requirements of being reactive is always being responsive.

> **REACTIVE MANIFESTO** One of the main principles of reactive systems is the notion of *resiliency*, which states that systems should stay responsive in the face of failure. Reacting to errors using RxJS operators is one way to work towards this goal.

The basic error handling mechanism that RxJS provides is the `catch()` operator, used to intercept any error in the Observable and give you the option to handle by returning a new Observable or, again, by propagating it down to observers in case there's recoverability path, as shown in figure 7.7:

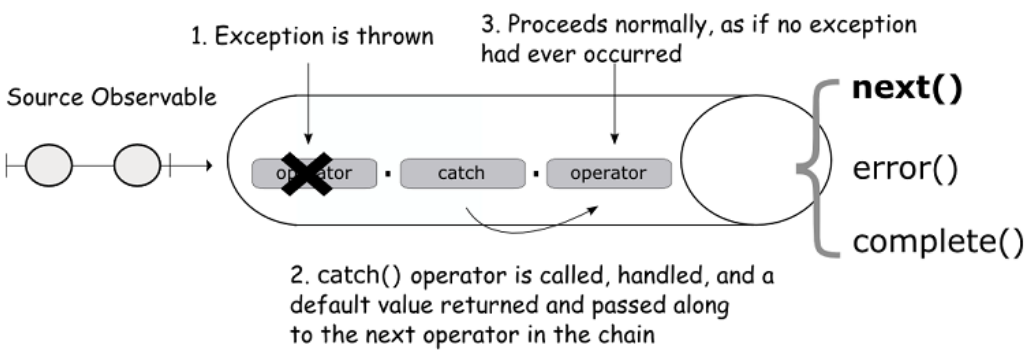

Figure 7.7 Exception caught in an operator upstream and caught using the catch() operator

Just like with regular try/catch usage, you want to place the `catch()` operator close to the segment of code that might fail. `catch()` allows us to insert a default value in place of the event that caused the error; any subsequent operators in the chain will never actually know that an exception ever occurred. Imagine if we experienced a login error to the server or had a problem accessing our local DB. The catch could be used to capture that error and inject a default or in memory value into the stream without the downstream being any the wiser!

Marble diagrams can be used to show error handling in a stream as well, just like with any other operator. Here's the example of a stream that rejects odd numbers and returns even instead:

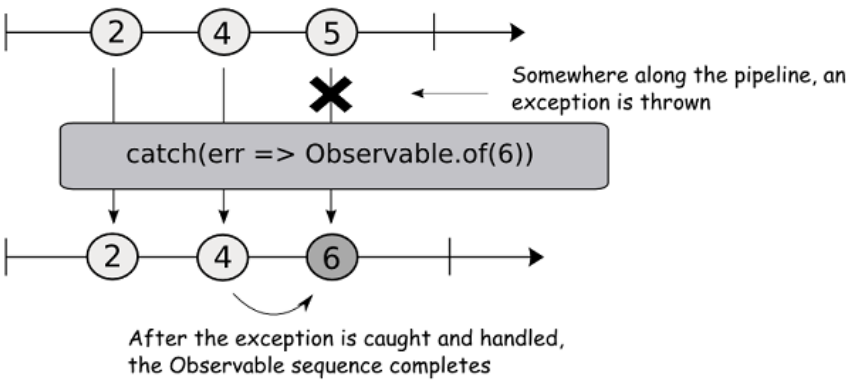

Figure 7.8 Illustrating error handling using marble diagrams

Here's the code for this:

Listing 7.6 Recovering from an exception using catch()

```
Rx.Observable.of(2,4,5,8,10)
   .map(num => {
      if(num % 2 !== 0) {
        throw new Error(`Unexpected odd number: ${num}`);
      }
      return num;
   })
   .catch(err => Rx.Observable.of(6)) ❶
   .map(n => n / 2)
   .subscribe(
       function next(val) {
          console.log(val);
       },
       function error(err) {
          console.log(`Received error: ${err}`); ❷
       },
       function complete() {
          console.log('All done!');
       }
    );
```

1. Catch intercepts the error and return an observable in its place
2. In this case, because the exception is caught and handled, the error method on the observer is never executed

Running this code now prints:

```
1
2
3
"All done!"
```

As you can see the stream continues to be cancelled when the exception occurs, but now we're at least able to recover. Some errors, however, might be intermittent and shouldn't halt the stream. For instance, a server is unavailable for a short period of time due to a planned outage. In cases like this, you may want to retry your failed operations.

7.4.3 Retrying failed streams for a fixed number of times

The `catch()` operator is passed a function that takes an error argument (shown in listing 7.6) as well as the source observable that was caught, which you can return to tell the source observable to retry from the beginning. Let's take a look:

```
Rx.Observable.of(2,4,5,8,10)
   .map(num => {
      if(num % 2 !== 0) {
        throw new Error(`Unexpected odd number: ${num}`);
      }
      return num;
   })
   .catch((err, source) => source)  (1)
   ...
```

1. Returning the original observable, which will begin to emit the entire observable sequence, starting with the first value, 2

This operation can be dangerous when the exception is unavoidable or not transient because we've now basically entered into an infinite loop; there is no condition in the business logic that will change for the error to disappear. Essentially what's occurring is the following:

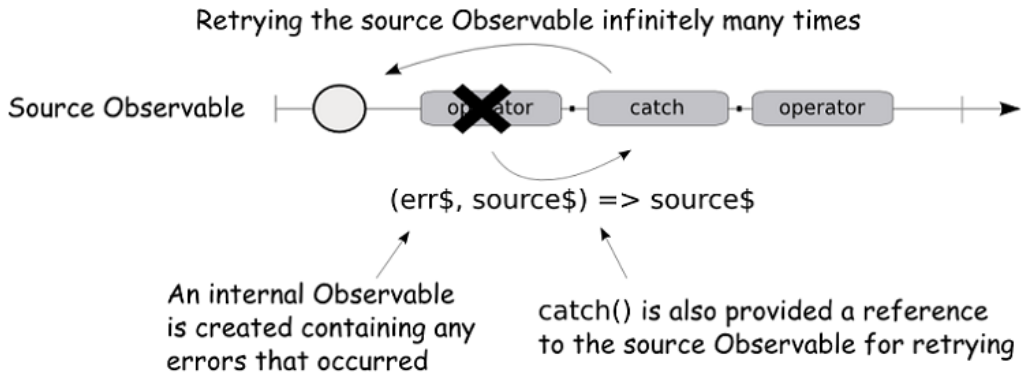

Figure 7.9 The catch() operator is provided an Observable sequence populated with any errors that occurred as well as the source Observable, which you can use to retry the sequence from the beginning.

Another place this can occur is when using promises. A promise can emit two types of errors: either an unexpected exception is thrown during the body of the computation, or it becomes unfulfilled and gets rejected. Because promises are *not retriable* artifacts, dereferencing the value of a promise will return its fulfilled value or error, as the case may be, always. The following code creates a big problem:

```
const requestQuote$ = symbol =>
     Rx.Observable.fromPromise(
       ajax(webservice.replace(/\$symbol/, symbol)))
     .catch((err$, promise$)=> promise$) (1)
     .map(response => response.replace(/"/g, ''))
     .map(csv);
```

(1) **Use the selector function to reiterate the execution of this promise stream. Bad idea!!**

Just like in figure 7.9 above, if the server we're trying to access is offline, the exception thrown would also create an infinite loop and exhaust the main thread, because you would just be retrying the same exception (failed promise) over and over again. We'll see how to solve this problem in a bit.

RxJS provides more intuitive ways of retrying via the `retry()` operator, which essentially combines this notion of catching and re-executing the source observable into one function. Here's a simple example:

```
Rx.Observable.of(2,4,5,8,10)
   .map(num => {
      if(num % 2 !== 0) {
        throw new Error(`Unexpected odd number: ${num}`);
      }
      return num;
   })
   .retry(3) (1)
   .subscribe(
```

```
    num => console.log(num),
    err => console.log(err.message)
  );
```

❶ Repeats this sequence a total of 3 more times (a total of 4) if there is an error before giving up and letting the exception propagate down to the observer

Running this code will print a sequence of numbers 2 and 4 a total of 4 times, before printing "Unexpected odd number: 5." So unless you're dealing with a transient failure that you know will resolve itself somehow, avoid catching and returning the same sequence or the equivalent retry operation with an empty argument. In order to ensure you don't lock up the UI or cause infinite loops to occur, you should always use `retry()` with a fixed number. You could also combine the two very elegantly. We can re-attempt the operation 3 more times and then catch the exception to fallback to a default value:

```
Rx.Observable.of(2,4,5,8,10)
  .map(num => {
    if(num % 2 !== 0) {
      throw new Error(`Unexpected odd number: ${num}`);
    }
    return num;
  })
  .retry(3)
  .catch(err$ => Rx.Observable.of(6)) ❶
  .subscribe(
    num => console.log(num),
    err => console.log(err.message));
```

❶ Instead of propagating the error down, we can use this placeholder value

Again the effect of this is that the sequence would be repeated a total of 4 times before the catch block executes emitting our default value 6 and then completing the sequence. Notice that using a default value with catch doesn't simply replace the value in the sequence and allows it to continue. After an exception occurs, the Observable is terminated at that point.

Now that we've talked about catch/retry, you're probably thinking it would be appropriate to embed `retry` into our stock ticker code, so that if the server were to fail due to a restart or a small outage, we could at least retry to fetch stock information:

```
const requestQuote$ = symbol =>
    Rx.Observable.fromPromise(
      ajax(webservice.replace(/\$symbol/, symbol)))
    .retry(3)
    .map(response => response.replace(/"/g, ''))
    .map(csv); ❹
```

However, there's a small caveat here. Recall from our previous discussions that promises have no retry capability (you don't have second chances with promises). Unlike promises, *streams are retriable artifacts*, so we can easily get around this limitation by wrapping the promise observable into another stream that is retriable—again creating a higher-order observable. Effective what you want to do is apply the retry function to an outer observable that wraps the

inner promise. We can use `mergeMap()` to flatten it back into a single stream, so placing the retry at `fetchDataInterval$` solves this problem:

```
const fetchDataInterval$ = symbol => twoSecond$
     .mergeMap(() => requestData$(symbol) ❶
          .distinctUntilChanged((previous, next) => {
               ...
          }))
     .retry(3);
```

❶ **requestData$ invokes the promise. This source observable is an outer observable that we can use to make the promise observable retry 3 more times**

This code will cause the promise internally to re-instantiate and retry three more times if it encounters an exception or a rejection, which is really nice. Keep in mind that it will al become a single observable layer once `mergeMap()` projects `requestData$(symbol)` onto the source. Looking at this technique from the point of view of the nature of a stream as a retriable type is one way, but there's a bit more to understand that's happening behind the scenes. We'll come back to this solution in the next chapter in the context of hot observables.

Another way of implementing retries effectively is to add a backoff strategy, which introduces some wait time in between retry actions.

7.4.4 Reacting to failed retries

Using retries with backoff is an effective way you can implement larger retry values without overloading the server. Examples of a backoff strategy are: constant, linear, exponential, and random (also known as jitter). The exponential and linear are more commonly used, but in any case the goal is to use progressively longer waits between retries for consecutives periods of time. RxJS allows us to accomplish this using the `retryWhen()` operator. `retryWhen()` takes a notifier observable argument (an internal Observable object that contains any errors that occurred during the execution of the stream, just like with `catch()`) and repeats the source observable that errors at the pace of when this notifier emits values. For instance, you can say: "retry after 3 seconds," as shown in figure 7.10:

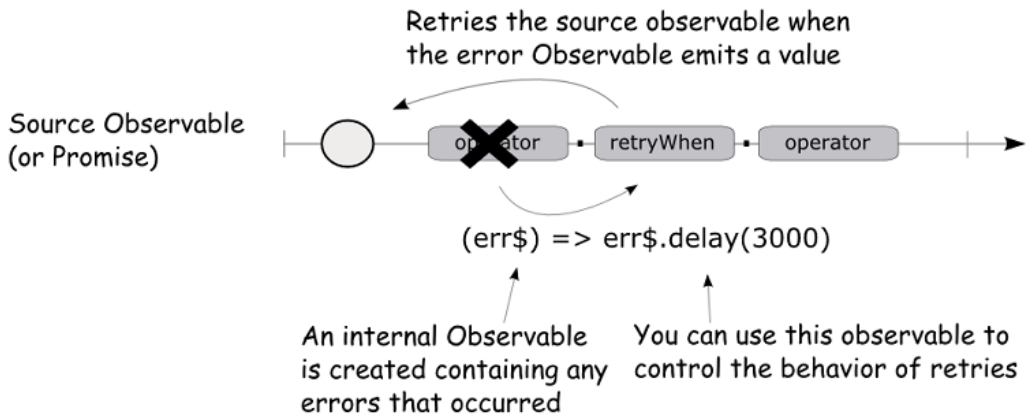

Figure 7.10 Implementing retries with a constant wait of 3 seconds in between retries

In other words, if the provided error observable emits a value, the retry action is executed. So you can use this observable to control when and how retries should take place; it's common to use timer observables to accomplish this. Let's go back to our numbers example to see this clearly:

```
Rx.Observable.of(2,4,5,8,10)
   .map(num => {
      if(num % 2 !== 0) {
        throw new Error(`Unexpected odd number: ${num}`);
      }
      return num;
   })
   .retryWhen(errors$ => errors$.delay(3000)) ❶
   ...
```

❶ Using the delay operator to plug in a 3-second delay between each error value is emitted

This will retry the observable sequence from the start and three seconds apart and repeat the numbers indefinitely, or until the operation that threw the exception becomes successful:

```
1
2
// 3 seconds wait...
1
2
// 3 seconds wait...
...
// and so on
```

We can also use `retryWhen()` to implement a fixed number of retries by keeping track of the number of times the source observable has been retried. Remember we can use `scan()` to emit values at every accumulated interval:

```
const maxRetries = 3;

Rx.Observable.of(2,4,5,8,10)
   .map(num => {
      if(num % 2 !== 0) {
        throw new Error(`Unexpected odd number: ${num}`);
      }
      return num;
   })
   .retryWhen(errors$ =>
        errors$.scan((errorCount, err) => {
            if(errorCount >= maxRetries) {
                throw err;
            }
            return errorCount + 1;
        }, 0)
    )
    ...
```

Running this code prints the same result as above, with the difference that instead of running indefinitely, it will retry for `maxRetries` time, and then error calling the `error()` method on the observers. A more effective retry strategy used in cases where remote requests are being made is a linear backoff, which alleviates the overall load on the server. This technique is readily implemented in most major modern websites; the first retry action occurs immediately, and subsequent actions occur after a certain period of time in between, which linearly backoff (times get bigger with each retry), as shown in figure 7.11:

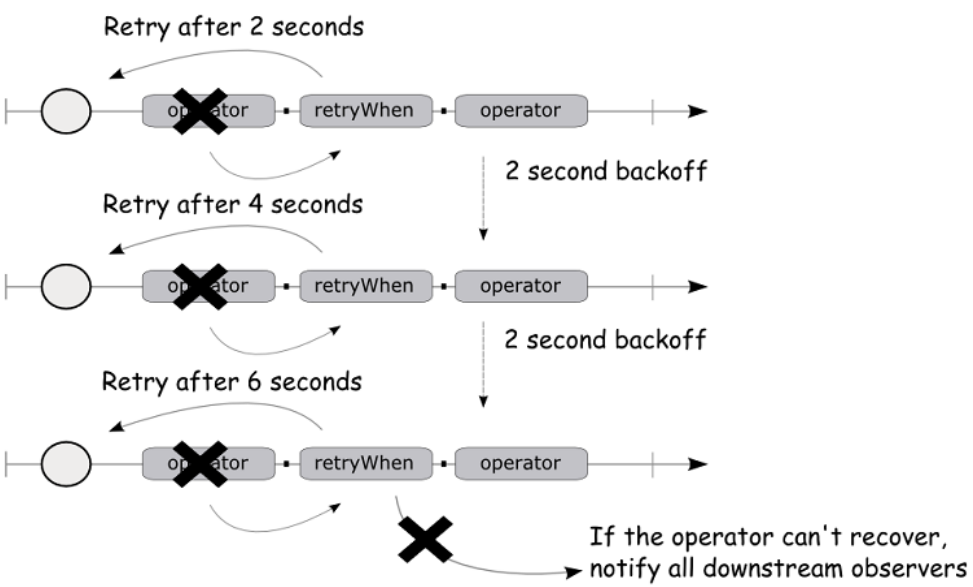

Figure 7.11 After each retry the time between grows linearly. It starts with a 2 second wait, then it invokes the next retry call after 4 seconds, then after 6 seconds, and so on.

Before we get into the code that implements this, we'll introduce a new operator called `zip()`. This operator merges the specified observable sequence into one by using a selector function (a function that you provide to instruct `zip()` how to format the events emitted) whenever all of the observable sequences have emitted values at a corresponding index. This operator is frequently used in functional programming to merge two corresponding arrays; for instance, `zip()` is implemented in Ramda.js:

```
const records = R.zip(
   ['RecordA', 'RecordB', 'RecordC'],  ❶
   ['123', '456', '789']
);
//=> [['RecordA', '123'],
      ['RecordB', '456'],
      ['RecordC', '789']
     ]
```

❶ zip combines both arrays into a multidimensional array, associating each value at the corresponding key

This works with streams just as well, as shown in the following marble diagram:

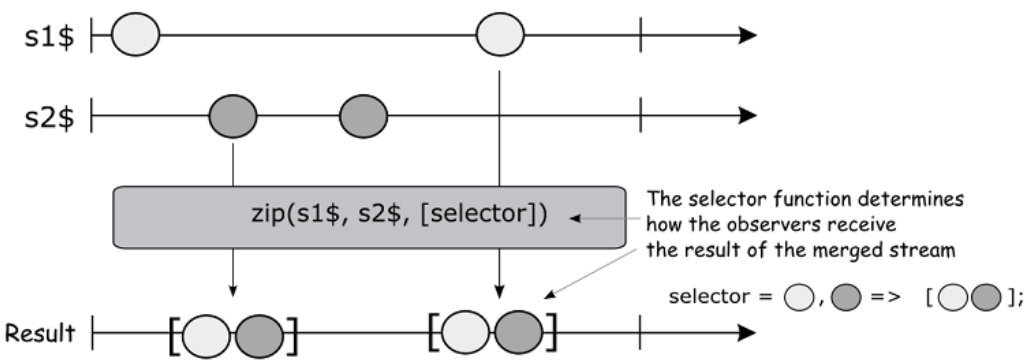

Figure 7.12 Internal workings of zip with streams. Both stream events are combined at each index irrespective of the time either event occurs.

In some ways, `zip()` works like `combineLatest()`, except that the former matches the index of the corresponding events one-to-one like as shown in 7.11; whereas the latter just combines the latest values when any of the observable emits a value. Here's a simple numerical example illustrating this difference:

```
const s1$ = Rx.Observable.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
const s2$ = Rx.Observable.of('a', 'b', 'c', 'd', 'e');

Rx.Observable.zip(s1$, s2$).subscribe(console.log);
//-> [1, "a"]
     [2, "b"]
     [3, "c"]
     [4, "d"]
```

```
     [5, "e"]

   Rx.Observable.combineLatest(s1$, s2$).subscribe(console.log);
//-> [9, "a"]
     [9, "b"]
     [9, "c"]
     [9, "d"]
     [9, "e"]
```

As you can see, `zip()` sticks to the array definition and merges both events and matches the corresponding indexes between both streams. In this case `s1$` continues to emit more values, but because `s2$` doesn't `zip()` ignores them—both have to emit events. On the other hand, `combineLatest()`, just merges the latest event of `s1$` with whatever is the latest value emitted by `s2$`.

Now that you know understand how to use `zip()`, we'll implement a linear backoff that retries the first time after a second, the second after 2 seconds, and so on in listing 7.7:

Listing 7.7 Implementing a linear backoff retry for our stock ticker stream

```
const maxRetries = 3;
Rx.Observable.of(2,4,5,8,10)
   .map(num => {
      if(num % 2 !== 0) {
        throw new Error(`Unexpected odd number: ${num}`);
      }
      return num;
   })
   .retryWhen(errors$ =>
       Rx.Observable.range(0, maxRetries) ❶
         .zip(errors$, val => val)        ❷
         .mergeMap(i =>                   ❸
           Rx.Observable.timer(i * 1000)
                   .do(() => console.log(`Retrying after ${i} second(s)...`)))
    )
   .subscribe(console.log);
```

❶ Returning an observable that will emit a maxRetries number of events. So, if maxRetries is 3, it will emit events 3 - 0 = 3 times.

❷ Zip is used to combine 1 to 1 values from the source observable (range), with the error observable. We're passing a selector function that basically returns the value of the first argument passed to it. This is known in functional programming as the *identity* function

❸ Merge map with a timer observable based on the number of attempts. This is what allows us to emulate the backoff mechanism.

With this retry strategy, the stream will attempt to run for a fixed number of times (given by `maxRetries`), with linearly-incrementing time of one second in between before finally giving up. Because we're not throwing the exception, the stream just halts execution on every retry, generating the following output:

```
2
4
Retrying after 0 second(s)...
2
```

```
4
Retrying after 1 second(s)...
2
4
Retrying after 2 second(s)...
2
4
```

While this is pretty advanced, we can also bundle throwing the exception if your goal is to signal the unrecoverable condition on your last retry action. We can do this by implementing some conditional logic within the projected observable returned from `mergeMap()`. In this case, if we've reached the last retry, we'll project an observable with an exception; otherwise, we'll project the timer, just as before.

Listing 7.8 Fixed count, linear back off and throwing exception if error persists

```
const maxRetries = 3;

Rx.Observable.of(2,4,5,8,10)
   .map(num => {
      if(num % 2 !== 0) {
        throw new Error(`Unexpected odd number: ${num}`);
      }
      return num;
   })
   .retryWhen(errors$ =>
       Rx.Observable.range(0, maxRetries + 1)
         .zip(errors$, (i, err) => ({'i': i, 'err': err})) (1)
         .mergeMap( ({i, err}) => {                        (2)
             if(i === maxRetries) {
               return Rx.Observable.throw(err);            (3)
             }
             return Rx.Observable.timer(i * 1000)
               .do(() =>
                    console.log(`Retrying after ${i} second(s)...`));
         })
    )
   .subscribe(
     console.log,
     error => console.log(error.message)
   );
```

(1) Using a selector function that combines events from both zipped streams into a single object
(2) Destructuring the parameter to extract the attempt count and the last error object that occurred.
(3) Because this code is inside a mergeMap() operator, it expects we return an observable object. So, I can use throw() operator to create an observable that safely wraps an exception object (throwing the error would also work, but this approach is more elegant).

Running this code prints the following:

```
2
4
Retrying after 0 second(s)...
2
4
```

```
Retrying after 1 second(s)...
2
4
Retrying after 2 second(s)...
2
4
Unexpected odd number: 5
```

Using if/else here is not the most functional way of writing code, but it's acceptable in practice given that the scope is internal to the pipeline. However, if you're looking for a more pure approach that uses more lambda expressions and keeps the functional programming spirit high, RxJS provides `Rx.Observable.if(condition, then$, else$)` that evaluates a given condition function, and either returns the `then$` observable or the `else$`, respectively. We'll use this to refactor just that segment of code:

```
...
.retryWhen(errors$ =>
     Rx.Observable.range(1, maxRetries)
       .zip(errors$, (i, err) => ({'i': i, 'err': err}))
       .mergeMap(({i, err}) =>
          Rx.Observable.if(() => i <= maxRetries - 1, ❶
             Rx.Observable.timer(i * 1000)            ❷
               .do(() => console.log(`Retrying after ${i} second(s)...`)),
             Rx.Observable.throw(err))                ❸
         )
    )
    ...
```

❶ Use the if() operator (also called functional combinator) to select between two streams depending on the evaluation of the condition function.
❷ If the condition returns true, project this observable; otherwise, project the observable created in the else block.
❸ Otherwise, use throw() to propagate an exception downstream to subscribers

> **BEST PRACTICE** The `zip()` operator can be very useful in cases when you need to spread out a stream synchronously over time, just as we did here in the previous code samples. It's not recommended when coordinating asynchronous streams that emit at different times—`combineLatest()` is the operator of choice in these cases. The reason for this is that `zip()` pairs the events 1-to-1, so it's effective when the asynchronous streams its operating over emit values with similar time intervals, which you can't control all of the time. So, if you're pairing a mouse-move observable that emits rapidly, for example, with an AJAX call that emits every few seconds, you can easily cause its internal, unbounded buffer to overflow and your application to crash.

Last but not least, in order to be at feature parity with the imperative world of try/catch/finally, RxJS provides the `finally()` operator. Just like the `do()` operator, this operator mirrors the source observable and invokes a specified void function after the source observable terminates by invoking the observer's `complete()` or `error()` methods. So, the expectation is that `finally()` could perform some kind of side effect, if need be. Such as performing any clean up actions. This is perfect for our stock ticker widget, which shows a

counter of the last time the stock quotes for updated. In this case, we can add another subscription to the `twoSecond$` observable for updating the last updated date:

```
const lastUpdated = document.querySelector('#last-updated');

const updateSubscription = twoSecond$.subscribe(() => {
   lastUpdated.innerHTML = new Date().toLocaleTimeString();
});
```

Remember that you can have a list of subscribers for the same event, so separating the logic for updating different portions of the site keeps the code under the observer nice and simple. So, if you had, say, three components that needed to change as a result of a stream emitting events, you can attach three observers and update the different portions of the site accordingly. So, we have two subscribers: the one we just showed you, and another used to fetch the stock data, to which we'll add error handling code. If case the web service call made in the `fetchDataInterval$` observable were to fail (returning a 500 HTTP response code, for example), the `catch()` operator will react and return a default value for that stock quote section:

Listing 7.9 Stock ticker with error handling

```
const requestQuote$ = symbol =>
      Rx.Observable.fromPromise(
        ajax(webservice.replace(/\$symbol/, symbol)))

      .map(response => response.replace(/"/g, ''))
      .map(csv)
      .catch(() =>
         Rx.Observable.of([new Error('Check again later...'), 0]))
      .finally(() => {
         updateSubscription.unsubscribe();
      });
```

1. **Adding catch() to handle the exception potentially thrown from requestQuote$**
2. **In the event an error occurs, cancel the twoSecond$ interval observable through its subscription object**

The other code we added was the `finally()` operator, which fires when a stream completes or when it errors. Because we are running a 2-second interval, we don't expect a completion, but in the event of an error, we should also clean up the interval and by cancelling the subscription, so that the updated time shown reflects the last quoted update received before the error occurred. This process can be visualized in the following graph:

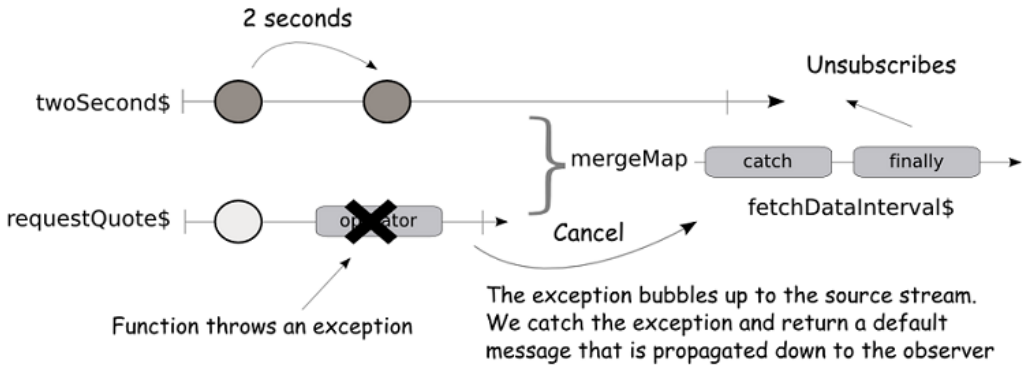

Figure 7.13 Use of finally to clean-up and cancel any outstanding streams

And now you need to make a small adjustment to the `tick$` observable, to know how to handle an error. We can use our `Try` functional data type to handle this and if a failure did occur, delegate the exception to the error callback of the observer. Here is that code once more with the new addition:

```
ticks$
 .map(([symbol, price]) => [Try.of(symbol).getOrElseThrow(), price]) ❶
 .subscribe(
   ([symbol, price]) => {

       let id = 'row-' + symbol.toLowerCase();
       let row = document.querySelector(`#${id}`);
       if(!row) {
        addRow(id, symbol, price);
       }
       else {
        updateRow(row, symbol, price);
       }
   },
   error => console.log(error.message));
```

❶ Before the data is handed down to the subscriber, Try can inspect it and decide if the data flowing in is an exception that needs to be thrown

If this service were to fail (or your internet disconnects), you will see "Check again later…" printed in the console.

As you can see, RxJS provides a comprehensive set of error handling operations that allow you to easily retry an entire observable sequence when an error is detected in the pipeline. However, we made a really big assumption about the nature of the observable sequences. That is, the observables that we created and retried in this chapter belong to a category known as "cold observables." Cold observables are passive (dormant) and only emit values when subscribed to: an array of numbers, a promise, intervals, etc. In other words, retrying a

cold observable basically just re-subscribes to it and requests that emit its values again. In the next chapter, we'll learn to create and handle the different types of observables: cold and hot.

7.5 Summary

- Imperative error handling has many drawbacks that make it incompatible for you to use it in a functional style of programming.
- Value containers provide a fluent, expressive mechanism for transforming values immutably.
- The `Try` wrapper is a functional data type used to consolidate and abstract exception handling so that you can map functions to value in sequence.
- RxJS implements many useful and powerful operators that allow you to catch and retry failed operations in a way that doesn't break the flow of the stream and the declarative nature of an RxJS stream declaration.
- RxJS provides operators such as: `catch()`, `retry()`, `retryWhen()`, and `finally()` that you can combine to create very sophisticated error handling schemes.

8

Heating up observables

This chapter covers:

- The difference between hot and cold observables
- Working with Websockets and event emitters via RxJS
- Sharing streams with multiple subscribers
- Understanding that hot and cold pertains to how the producer is created
- Sending events as unicast or multicast to one or multiple subscribers

As you know, an observable function is a lazy computation, which means the entire sequence of operators that encompass the observable declaration won't begin executing until an observer subscribes to it. But have you ever thought about what happens to all of the events that occur before the subscription happens? For instance, what does a mouse move observable do with all of the mouse events, or a WebSocket that has received a set of messages? Do these, potentially, very important occurrences get lost in the ether? The reality is that these active data sources won't wait for subscribers to listen to begin emitting events, and it's really important for you to know how to deal with this situation. Earlier, we briefly called out this idea that observables come in two different flavors: hot and cold. This is not a simple topic to grasp; it's probably one of the most complex in the RxJS world, which is why we wanted to dedicate an entire chapter to it.

In this chapter, we'll take closer look at hot and cold observables, how they differ, the benefits both have, and how you can take advantage of this in your own code. Up until now we've only had a single subscriber to a stream, and we even looked at combining multiple streams funneled through one observer in chapter 6. Now, we take the opposite approach as we look into sharing a single observable sequence with multiple observers. For instance, we can take a simple numerical stream and share its values with multiple subscribers, or take a single web socket message and broadcast it to multiple subscribers. This is very different to

the single stream subscriber cases we've dealt with. First, let's begin by understanding the differences between hot and cold observables.

8.1 Introducing hot and cold observables

Think about when you turn on your TV set and switch to the channel of your favorite show. If you catch the show 10 minutes after it began, will it start from the beginning? Restarting the show would mean that cable companies broadcast independent streams to every subscriber—which would be great. Instead, the same content is broadcast out to all subscriber. So unless you have a recording device, which we can associate with a buffer operator, you can't replay content that happened in the past.

A TV's livestream is equivalent to a hot observable. RxJS divides observable sources into one of two categories, hot or cold. These categories determine the behavioral characteristics not just of subscription semantics but also of the entire lifetime of the stream. An observable's temperature also affects whether the stream and the producer are managed together or separately, which can greatly affect resource utilization (we'll get to this shortly). We classify observables as either hot or cold based on the nature of the data source that it's listening to. Let's begin with the cold observables.

8.1.1 Cold Observables

In simple terms, a cold observable is one that doesn't begin emitting *all of its values* until an observer subscribes to it. Cold observables are typically used to wrap bounded data types such as numbers, range of numbers, strings, arrays, and HTTP requests, as well as unbounded types like generator functions. These resources are known as *passive* in the sense that their declaration is independent of their execution. This also means that these observables are truly lazy in their creation and execution.

Now, this isn't news to you, since that's what we've defined an observable to be all along—so what's the catch? Being cold means that each new subscription is creating a *new independent stream* with a "new starting" point for that stream. This means that subscribers will independently receive the same exact set of events always, from the beginning. Another way to conceptualize it: when creating a cold observable you are actually creating a plan or recipe to be executed later—repeatedly, top to bottom. The recipe itself is just a set of instructions (operators) that tells the JavaScript runtime engine how to combine and cook the ingredients (data); cold observables begin emitting events only when we choose to start cooking.

> **PURE OBSERVABLES** Observables are pure when they abide by the functional programming principles of a pure function, which is immutable, side-effect free, and repeatable. We have talked about the first two principles at length in this book, and now we tag on this third quality. In order to support a desirable functional property known as *referential transparency*, functions must be repeatable and predictable, which means that invoking a function with the same arguments *always* yields the same result. The same holds for cold observables when viewed simply as functions that produce (return) a set of values.

From a pure functional programming perspective, we think of cold observables as behaving very much like functions. A function can be reasoned a lazy or to-be-computed value that is returned when you invoke it, only when needed (languages with lazy evaluation, like Haskell, work this way). Similarly, observable objects won't run until subscribed to, and you can use the provided observers to process their return values. You can visualize this resemblance in figure 8.1:

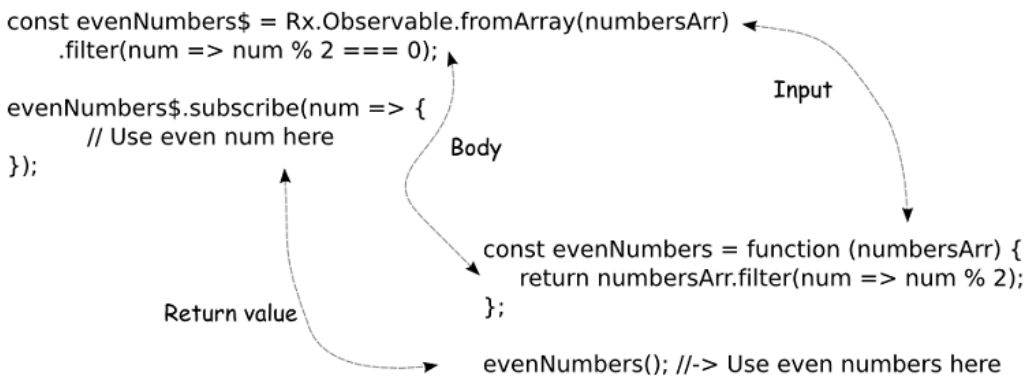

Figure 8.1 A cold observable can be thought of as a function that takes input, or the data that is to be processed, and based on this return an output to the caller.

Furthermore, the declaration of a cold observable frequently begins with static operators such as `of()` or `from()`, and timing observables `interval()` and `timer()` also behave cold. Here's a quick example:

```
const arr$ = Rx.Observable.from([1,2,3,4,5,6,7,8,9]);

const sub1 = arr$.subscribe(console.log);  ❶

// ... moments later ... //
const sub2 = arr$.subscribe(console.log);  ❷
```

❶ **Every subscribers gets their own independent copy of the same data no matter when the subscription happens**
❷ **sub2 could have subscribed moments later, yet it still receives the entire array of elements.**

With cold observables, all subscribers, no matter what point in time the subscription occurred, will observe the same of events. Another example is the `interval()` operator. Each time a new subscription occurs a *brand new* interval instance is created like in figure 8.2:

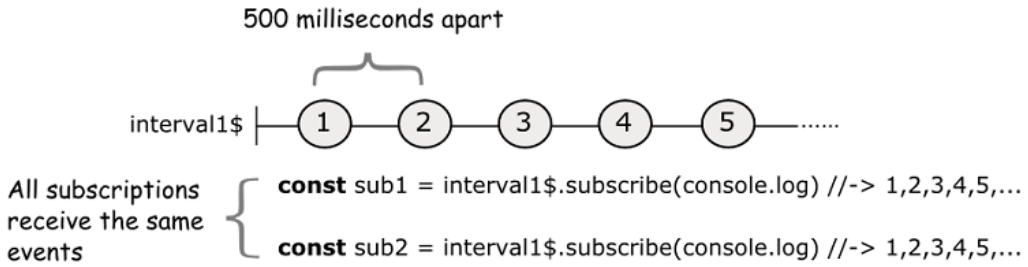

Figure 8.2 A cold observable is like an object factory, which can be used to create a family of subscriptions that will receive their own independent copy of all of the events pushed through it

Interval acts like a *factory for timers*, where each timer operates on its own schedule and each subscription can, therefore, be independently created and cancelled as needed. Cold observables can be likened to a factory function that produces stream instances according to a template (its pipeline) for each subscriber to consume fully. Listing 8.1 demonstrates that I can use the same observable with two subscribers that listen for even and odd numbers only, respectively:

Listing 8.1 Interval factory

```
const interval$ = Rx.Observable.interval(500);

const isEven = x => x % 2 === 0;

interval$
  .filter(isEven)
  .take(5)
  .subscribe(x => { ❶
      console.log(`Even number found: ${x}`);
   });

interval$
  .filter(R.compose(R.not, isEven))
  .take(5)
  .subscribe(x => { ❶
      console.log(`Odd number found: ${x}`);
   });
```

❶ Two subscriptions for the same observable interval$

In the example in 8.1, the streams are independently created from one another. Each time a new subscriber subscribes, the logic in the pipeline is re-executed from scratch. So these streams all receive the same numbers and don't affect each other's progress—they're isolated as shown in figure 8.3:

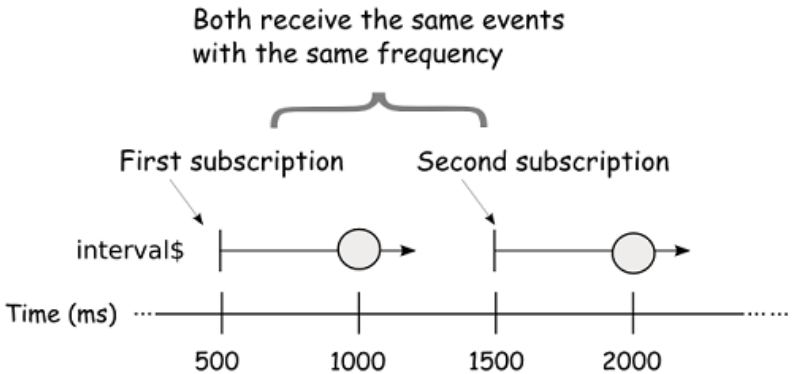

Figure 8.3 Re-subscribing to an interval Observable yields two independent sequences with the same events happening with the same frequency; hence, this is a cold observable

DEFINITION A cold observable is one that, when subscribed to, emits the entire sequence of events to any active subscribers.

Now, we're ready to heat things up and look closely at the other side of the coin: the hot observables.

8.1.2 Hot observables

Streams do not always start when we want them to, nor can we reasonably expect that we would always want every event from every observable. It may be the case that by delaying the subscription, we are deliberately avoiding certain events as an implicit version of `skip()`.

Hot observables are ones which produce events regardless of the presence of subscribers—they are *active*. In the real world, hot observables are used to model events like clicks, mouse movement, touch, or any other exposed via event emitters. This means that, unlike the cold counterpart where each subscription triggers a new stream, subscribers to hot observables tend to only receive the events that are emitted *after* the subscription is created, as shown in figure 8.4:

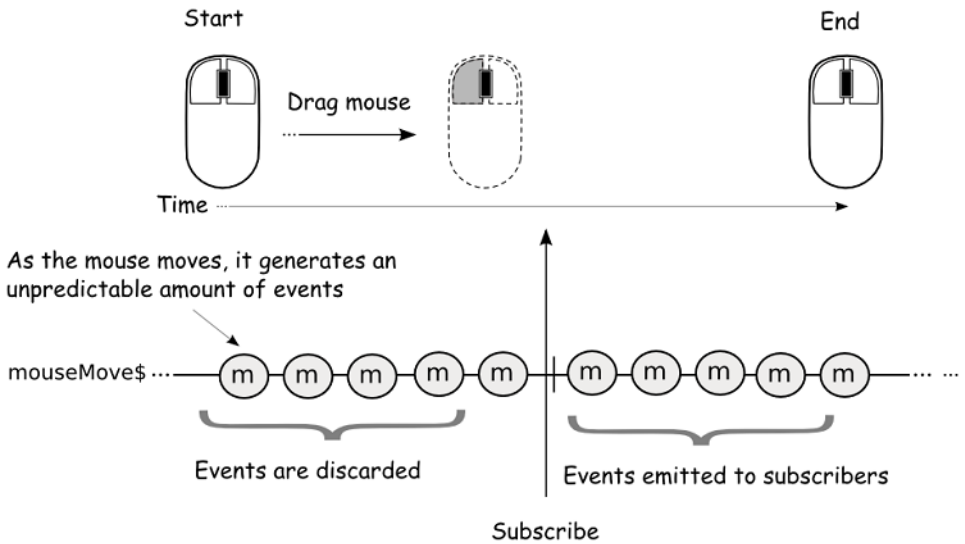

Figure 8.4 A mouse move handler generates unbounded events that can be captured as soon as the HTML document loads; however, these events will be ignored until the stream has been created and observer subscribed to it.

A hot observable continues to remain lazy in the sense that without a subscriber, the events are simply emitted and ignored. Only when an observer subscribes to the stream is when the pipeline of operators begins to do its job and data flows downstream.

This type of stream is often more intuitive to many developers because it closely mirrors behaviors they are already familiar with in promises and event emitters.

> **PROMISES AND HTTP CALLS** While a conventional HTTP request is cold, it isn't when a promise is used to resolve it. As you'll learn in a bit, a promise of any type represents a hot observable because it's not re-executed after it's been fulfilled.

Due to the unpredictable and unrepeatable nature of the data that hot observables emit, we can reason that they are not completely pure, from a theoretical perspective. After all, reacting to an external stimulus, like when a button is clicked, can be considered a form of side effect dependent on the behavior of some other resource, like the DOM, or simply time. Nevertheless, from the point of view of the application and your code, all observables can be considered pure.

Unlike cold observables that create independent copies of the data source to emit to every subscriber, hot observables will share the same subscription to all observers that listen to it, as shown in figure 8.5. Therefore, we conclude that a hot observable is one that, when subscribed to, emits the on-going sequence of events from the point of subscription and not from the beginning.

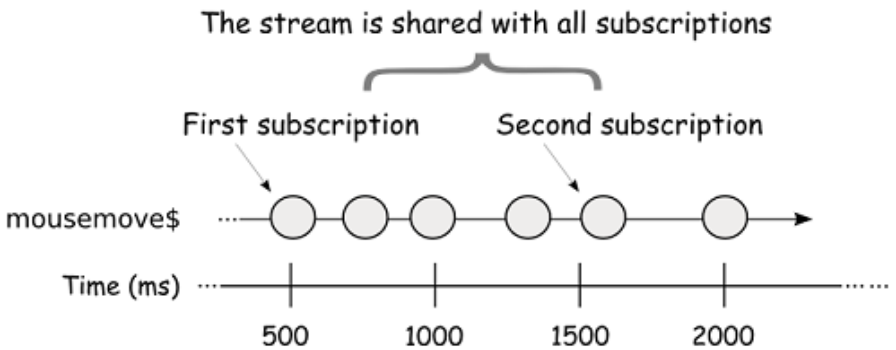

Figure 8.5 Hot observables share the same stream of events to all subscribers. Each subscriber will start receiving events currently flowing through the stream after they subscribe.

Whether an observable is hot or cold is partly related to the type of source the it's wrapping. For instance, in the case of any mouse event handler, short of creating some new mechanism for handling mouse events, an observable is merely abstracting the existing `addEventListener()` call for a given emitter. As a result, the behavior of mouse event observables is contingent on the behavior of the system's handling of mouse events. We can further categorize this source as *natively hot*, because the source is determining the behavior. We can also make sources hot, programmatically, using operators as well and we'll discuss this further in the sections to come.

On the other hand, observables that wrap either a static data source (array) or use generated data (via a generator function) are typically cold, which means they do not begin producing values without a subscriber listening to them. This is intuitive because, like iteration, stepping through a data source requires a consumer or a client.

A key selling point for using RxJS is that it allows us to build logic independently of the type of data source we need to interact with—we called this a unifying computing model. A source can emit zero to thousands of events unpredictably fired at different times. Nevertheless, the abstraction provided by the observable type means that we don't have to worry about these peculiarities when building the logic inside the stream, or within the context of the observable. This interface abstracts the underlying implementation out of sight and out of mind. For the most part.

HOT OBSERVABLE Hot observables are ones which produce events regardless of the presence of subscribers.

In general, it is better to use cold observables wherever possible because they are *inherently stateless*. This means that each subscription is independent from every other subscription, so there is less shared state to worry about, from an internal RxJS perspective, since we know a new stream is starting on every subscription.

8.2 A new type of data source: WebSockets

In chapter 4, we mentioned that time ubiquitously exists in observables—in hot observables to be exact. This disparity between when a data source begins emitting events and when the subscriber starts listening can lead to issues. Think about that TV show that you have switched to mid-program. In such contexts, unless you've watched the show before tuning in, you will always miss some context or plot that was presented at the beginning. In the same vein, you can imagine a simple messaging system using a protocol like WebSockets, or any other event emitter for that matter. In these cases, missing any messages can be critical to the proper functioning of your application. So, if a subscription to a hot observable occurs after a critical message packet arrives, then those instructions might be lost. We haven't talked about using Websockets with RxJS, so let's begin by briefly understanding what they are and how RxJS can help us handle these asynchronous message flows.

8.2.1 A brief look at WebSockets

Aside from binding to DOM events and AJAX calls, RxJS can also just as easily bind to web sockets. WebSockets is an asynchronous communication protocol that provides faster and more efficient lines of communication from client to server than traditional HTTP. This is useful for highly interactive applications like live chats, streaming services, or games. Like HTTP, WebSockets run on top of a TCP connection; however, the advantage is that information can be passed back and forth while keeping the connection open (taking advantage of the browser's multiplexing capabilities and the keep-alive feature). The other benefit is that servers can send content to the browser without it explicitly requesting it.

Figure 8.6 shows a simplified view of a WebSocket communication. It begins with a "handshake," which bridges the world of HTTP to WS. At this time, details about the connection and security are discussed to pave the way for a secure, efficient communication between the parties involved:

The steps taken between the client and the server are:

- Establish a socket connection between parties for the initial handshake
- Switch or upgrade the communication protocol from regular HTTP to socket-based protocol
- Send messages in both directions (known as full duplex)
- Both server or client can issue a disconnect.

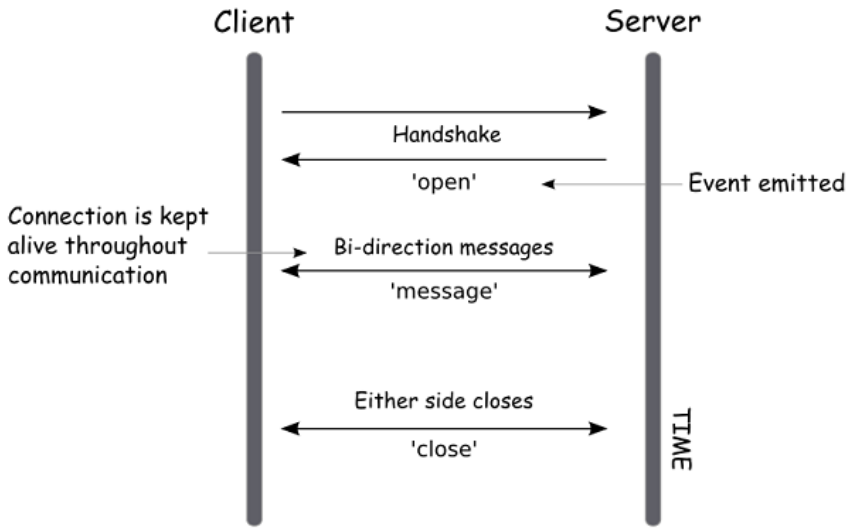

Figure 8.6 Websocket communication diagram shows communicating begins with a handshake where client and server negotiate the terms of the connection. Afterwards, communication flows freely until one of the parties closes.

The crucial point of this process is the initial handshake that negotiates the upgrade process. The upgrade needs to happen because WebSockets use the same ports 80 and 443 for HTTP and HTTPs, respectively (443 is used by WebSockets Secure protocol). Routing requests through the same ports is advantageous because firewalls are typically configured to allow this information to flow freely. At a very high-level, this process happens with an initial secure request by the client and a proper response by the WebSocket-supporting server, shown in figure 8.7:

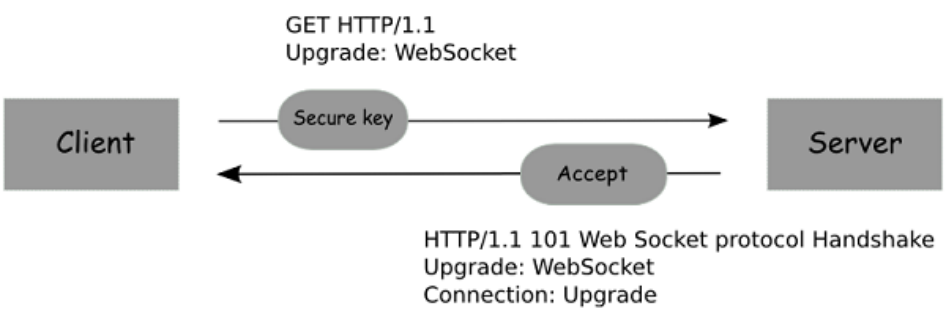

Figure 8.7 The handshake negotiation process begins with the client's GET request containing a secure key and instructions for the server to attempt to upgrade to a message-based WebSocket connection. If the server understands WebSockets, it will respond with a unique hash confirming the protocol upgrade

From the point of view of asynchronous event messaging, you can think of WebSockets as an event emitter for client-server communication. Now, we also haven't really talked about server-side RxJS, but not much changes. We can use RxJS to easily support both sides of the coin, starting with the server.

8.2.2 A simple WebSocket server in Node.js

In the spirit of a JavaScript book, we'll write our server using Node.js. But you can use any other platform of your liking such as Python, PHP, Java, and any other platform with a socket API. Our server side will be a simple TCP application listening on port 1337 (chosen arbitrarily) using the Node.js `WebSocket` API. WebSockets negotiate with the HTTP server to be the vehicle to send and receive messages. Once the server receives a request, it will response back with the message "Hello Socket."

Listing 8.2 Simple RxJS Node.js server

```
const Rx = require('rxjs/Rx');      ❶
const WebSocketServer = require('websocket').server; ❷
const http = require('http');       ❸

// ws port
const server = http.createServer(); ❹
server.listen(1337);

// create the server
wsServer = new WebSocketServer({
    httpServer: server
});

Rx.Observable.fromEvent(wsServer, 'request')         ❺
   .map(request => request.accept(null, request.origin))
   .subscribe(connection => {
       connection.sendUTF(JSON.stringify({ msg:'Hello Socket' })); ❻
   });
```

❶ **Import the RxJS core APIs**
❷ **Import the WebSocket server library**
❸ **Import the HTTP library**
❹ **Instantiate an HTTP server to begin listening on port 1337**
❺ **React to the request received event**
❻ **Send a JSON object packet once a connection is established.**

You can run the server with Node.js using the CLI (for details about setting up RxJS on the server, please visit the Appendix section):

```
> node server.js
```

Now that our server is up and listening, we can build the client again using RxJS—one library to rule them all!

8.2.3 WebSocket client

Modern browsers come equipped with the WebSocket APIs, which allows for an interactive communication with a server. Using these APIs you can send messages to a server and receive event-driven responses (server-push) without you having to explicitly poll for data, which is what you would do with regular HTTP requests. Listing 8.3 shows a simple WebSocket connection using RxJS:

Listing 8.3 Websocket client with RxJS

```
const websocket = new WebSocket('ws://localhost:1337'); (1)

Rx.Observable.fromEvent(websocket, 'message') (2)
  .map(msg => JSON.parse(msg.data)) (3)
  .pluck('msg')  (4)
  .subscribe(console.log);
```

(1) Connect to port 1337 using the WebSocket (ws) protocol
(2) Listen for the 'message' events used to transmit messages between client and server
(3) Parse the serialized string into a JSON object.
(4) Read the message

WebSockets are a form of loosely decoupled communication between two entities, in this case a browser and a server, that act as event emitters. This goes back to the idea of using a familiar computing model for everything. In essence, with RxJS the details of setting up event listeners for WebSocket communication are completely abstracted and removed from your code.

> **REACTIVE MANIFESTO** A good design principle of reactive systems is that they communicate using asynchronous message passing, in order to establish a boundary between loosely coupled components[12]. This allows not only for components to evolve independently, but also delegates failures as messages. This allows for non-local components to be able to react to errors appropriately.

Just like any event emitter, using RxJS with WebSockets creates a hot observable, which means it won't reenact all of the messages emitted upon subscription but merely begin pushing any thereafter. This can be tricky because subscribing to a hot observable a little too late can create loss of data. Consider this slight variation of listing 8.3:

```
Rx.Observable.timer(3000)  (1)
  .mergeMap(() => Rx.Observable.fromEvent(websocket, 'message'))
  .map(msg => JSON.parse(msg.data))
  .pluck('msg')

  .subscribe(console.log);
```

[12] http://www.reactivemanifesto.org/

❶ **Wrap the socket object after 3 second delay**

In the above stream, we tied the subscription to the WebSocket after a given amount of time (3 seconds). We arbitrarily set this wait time in order to simulate a delay such as one caused by page initialization—essentially we created a time dependency. If you recall the WebSocket handshake diagram, as soon as the socket fires the "open" event, this hot observable can begin emitting events. Thus if the opening of the socket occurs before the page has completed its initialization step, then the observable would potentially miss any events emitted in the intervening period.

As an added complication, time dependencies are not necessarily deterministic either. In the above scenario we added a simple 3 second time out, but your page load or initialization logic could be much more complex. It could be affected by any number of variables, from whether the application was loaded from a cache, how many resources the user's system has available for processing requests, how much network latency is present, or even how much animation is on the page, because they all can change when the page initialization occurs or when the WebSocket connects and begins sending events.

Certainly a dependency on time can be significant in an observable's behavior, to the point of breaking the nice functional quality that cold observables posses. Ideally, every subscriber to a cold observable should see the same sequence of events replayed before them; however, this isn't always the case, as there's a big difference between re-subscribing and replaying when side effects are at play.

8.3 The impact of side effects on a re-subscribe or a replay

As you saw from the previous use case, hot observables for the most part follow a strict you-snooze-you-lose policy, which means you can't replay the contents of a hot observable by simply re-subscribing to it, as you can with cold observables (there are ways of doing it, but this isn't the default behavior). Now, this doesn't mean that all cold observables behave this way, especially when you introduce side effects into your code. Before we discuss this further, you need to understand the difference between re-subscribing and re-playing in RxJS:

- A replay is about reemitting the *same sequence* of events to each subscriber—in effect *replaying* the entire sequence. Caution must be taken when attempting to replay sequences because they require using potentially lots of memory (often with unbounded buffers) to store the contents of a stream that is to be re-emitted at a later time. For obvious reasons, we recommend against doing this with streams like mouse clicks or any other infinite event emitter.

  A good example of replay semantics is promises. Replaying the observable created from a promise by means of a retry or simply attaching new subscribers does not cause the fulfilled promise to invoke again, but simply return the value or the error, as the case may be.

- A re-subscribe recreates the same pipeline and re-executes the code that produces

events. While the results emitted by the producer will be implementation dependent, if your observable pipeline remains pure, then you can guarantee that subscribers will all receive the same events for the same input produced.

8.3.1 Replay vs. re-subscribe

The difference is subtle, but important. In essence, it's really about whether the pipeline (your business logic) gets re-executed or not when another subscriber starts listening. Most of the canned observable factory methods: `create()`, `interval()`, `range()`, `from(Array|scalar|generator*)`, and others are cold by default. Here's a diagram illustrating the differences between these mechanisms:

A replay emits the same output to all subscribers without invoking the operator sequence as shown in figure 8.8:

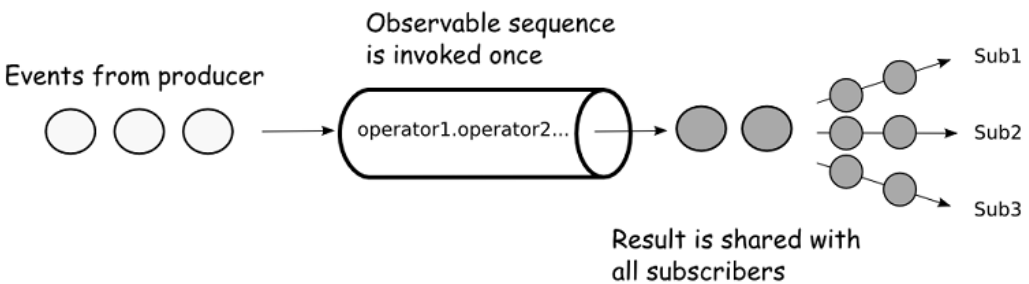

Figure 8.8 When replaying, the output emitted by an observable sequence is simply shared or broadcasted to all subscribers

While a cold re-subscribe (figure 8.9) invokes the sequence of operators that lead to the result for every subscriber:

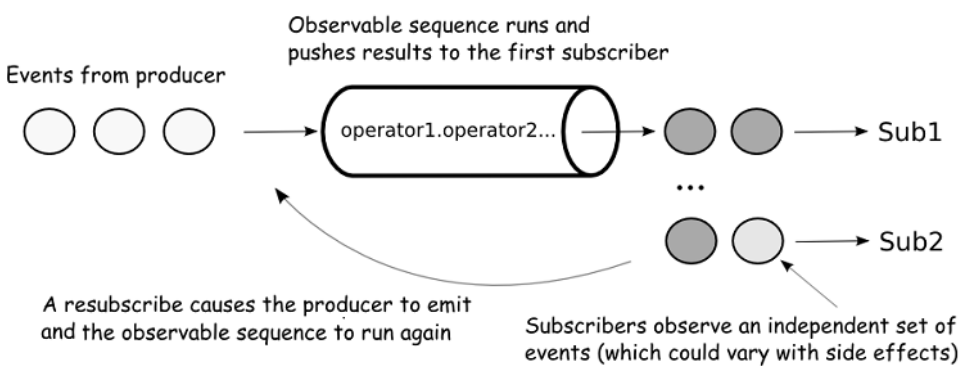

Figure 8.9 A re-subscribe causes the producer and the observable sequence to execute. If the operator sequence has side effects, then new subscribers could see different results.

We'll show a simple example showcasing both scenarios and the impact a side effect can have. For this we'll build custom observables whose behavior depends on the time of day (a side effect). In this case, we'll emit events until the time reaches 10:00 pm, at this point they fire the complete signal.

8.3.2 Replaying the logic of a stream

To showcase how a replay works, consider a body of time-sensitive code wrapped in a promise that would emit a value of "Success!" before 10:00 pm or throw an exception if executed after. The first observer that subscribes before 10:00 pm will cause the promise to execute and resolve. Any observers that subscribe any time later will simply receive the same value without invoking the body of the promise again. The business logic is simply ignored:

```
const p = new Promise((resolve, reject) => {
  setTimeout(() =>{
    let isAtAfter10pm = moment().hour() >= 20;          ❶
    if(isAtAfter10pm) {
       reject(new Error('Too late!'));
    }
    else {
       resolve('Success!');
    }
  }, 5000);
});

const promise$ = Rx.Observable.fromPromise(p);

promise$.subscribe(val => console.log(`Sub1 ${val}`));  ❷

// ... after 10 pm ...//
promise$.subscribe(val => console.log(`Sub2 ${val}`));  ❸
```

❶ Use moment.js to check if the current time is 10:00 pm.
❷ First subscriber executes the promise
❸ Second subscriber will emit the same value, regardless of the time it subscribed, because it won't run the code within the body of the promise

Regardless of the time of the subscription, any observers that subscribe to this stream receive the same value, whether it's a success or failure. This cold observable behaved predictably, but this is only due to the way promises work. Querying the result of a fulfilled promise, always outputs the same value. So, the body of the promise in this case doesn't actually run when the second subscribe occurs—this is a replay and the `fromPromise()` static operator is hot, now let's look at the case of a re-subscribe.

8.3.3 Re-subscribing to a stream

Now, consider this custom observable that emits numbers every second with, again, two observers subscribed to it at different times. The first subscription "Sub1" happens before

10:00 pm and will immediately begin receiving events, while the second "Sub2" happens after and is terminated immediately:

```
"Sub1 Starting interval..." ❶
"Sub1 Next 0"
"Sub1 Next 1"
"Sub1 Next 2"

"Sub2 Starting interval..." ❷
```

❶ Subscription occurs before 10 pm and begins to receive values
❷ Subscription occurs after, so observer never sees the numbers emitted

Let's see this code. The reason this behaves differently compared to the code in the previous section is because the `create()` factory operator is cold by default:

```
const interval$ = Rx.Observable.create(observer => {
  let i = 0;

  observer.next('Starting interval...');
  let intervalId = setInterval(() => {

    let isAtAfter10pm = moment().hour() >= 20; ❶

    if(isAtAfter10pm) {
      clearInterval(intervalId);  ❷
      observer.complete();
    }

    observer.next(`Next ${i++}`);
  }, 1000);
});

   // ... before 10 pm ... //
   const sub1 = interval$.subscribe(val => console.log(`Sub1 ${val}`)); ❸

// ... after 10 pm ... //
const sub2 = interval$.subscribe(val => console.log(`Sub2 ${val}`)); ❹
```

❶ Use moment.js to check if the current time is 10:00 pm.
❷ Stop emitting events
❸ Subscriber sub1 begins listening
❹ After 10 pm, sub2 subscribes but ends immediately

Re-subscribing to the above stream (with `sub2`) would create a new, independent stream; however, it won't just blindly propagate the same values again. Instead it re-invokes the logic so that different subscribers would receive (or not) events based on when they subscribed. If the time reaches 10:00 pm before subscriber `sub2` has a chance to listen, it won't receive any events at all. So the fact that the observable begins emitting events when subscribed to indicates that it's a cold observable. However, because we have a side effect in our code, preventing it from replaying the sequence, the results that subscribers see might be significantly different:

SIDE EFFECT ALERT As we mentioned before, the use of *time* directly in your own code is clearly a sign of side effect because, intuitively, it is global to your application and ever changing. This is why the hot observables are the lesser pure form when compared to cold observables which should re-emit (or replay) all of the items to any subscribers.

In the same vein, consider an operator you saw in the last chapter, `retry()`, which performs a re-subscribe on an observable when an error occurs. Let's revisit using it as part of our stock ticker stream. Using `retry()` directly on the promise seemed like a good idea:

```
const requestQuote$ = symbol =>
     Rx.Observable.fromPromise(
       ajax(webservice.replace(/\$symbol/, symbol)))
     .retry(3)
     .map(response => response.replace(/"/g, ''))
     .map(csv);
```

We might expect to retry a web request if it fails, resulting in up to 4 requests sent to the server before an error is finally served. However, we talked about a small caveat in that you may have then been surprised to see that the network debugger only shows a single request being executed. Why? Let's explain this in the context of hot and cold observables.

This goes back to our examination of how sources affect the behavior of observables. A promise is an eager data type (read hot observable), which means that upon creation it can only ever resolve or reject and will do so even without listeners. Promises are not retriable. So once it's in one of those two states it stays there, and every new handler will simply receive either the resolved value or the error that caused the rejection. Because promises do not have retry, any attempt to retry through `fromPromise()` is futile because it actually just replays whatever the final state of the promise is, by design. Recall that to get around this limitation, we wrapped the creation of the Promise in another observable, which is retriable, so that the operation contained inside it (the promise) could be retried. That's why we moved it out to its outer observable:

```
const fetchDataInterval$ = symbol => twoSecond$
     .mergeMap(() => requestQuote$(symbol)) ❶
     .retry(3) ❷
     .catch(err => Rx.Observable.throw(
        new Error('Stock data not available. Try again later!')));
```

❶ The inner observable that's projected onto the two second stream
❷ This is the observable that retries the failed stream 3 more times; otherwise, catch and propagate the error downstream

Remember that the reason this worked is because we actually created a new promise within the `mergeMap()` operator and because the retry is re-subscribing to the projected observable and not to the one created directly in `fromPromise()`. The re-subscription rebuilds the pipeline and re-executes it for the single value that we passed into it; so now you have a complete and deep understanding of why this technique works. Essentially what we did was change the temperature of this observable, so that it essentially behaves cold.

To summarize, re-subscribing to a cold observable creates independent event channels for each subscriber, which means each observer creates its own copy of the producer. In most situations this is desirable. For instance, if you use observables to process a set of objects originating from a generator function, you'll definitely want to work on copies of this producer (created via the cold `from()` static operator), instead of sharing it. On the other hand, replaying can be effective and save you precious computing cycles when your intent is to broadcast or share the output of an observable sequence to multiple observes.

In practice, when the producer resource is expensive to create, such as a remote HTTP call or a WebSocket, then sharing it is a smart thing to do. Let's examine how we can heat up observables to accomplish this.

8.4 Changing the temperature of an Observable

The re-subscription mechanism of cold observables is easy to reason about because each stream carries its own copy of the producer spawning a new pipeline back up to the source. This happens in RxJS by default when wrapping synchronous data sources like scalars and arrays, but also through custom observables containing asynchronous such as remote event emitters, AJAX calls, or WebSockets that are created within the observable context. In all of these cases you deal with cold observables. From a functional point of view, this is the purest solution since there's no data being shared and the observable acts like a template (or a recipe as we mentioned previously) for creating data:

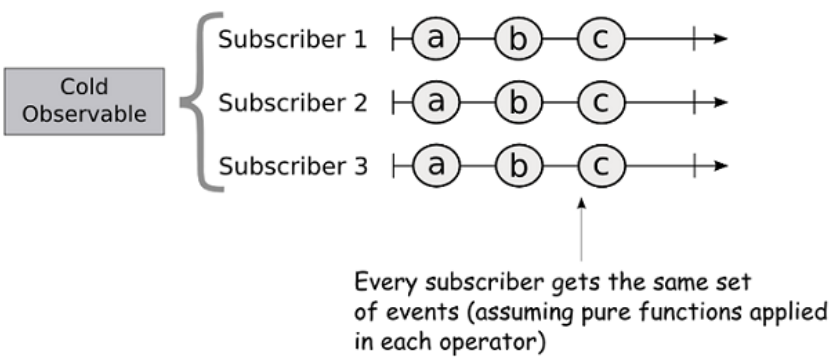

Figure 8.10 New subscribers to cold observables fork the event sequence and obtain their own copy

However, when resources are scarce, doing this can pose significant problems. In practice, it's beneficial to spawn one HTTP request, have a single event emitter instance, or create one WebSocket connection that many observers can share, instead of one for each. In section 8.3, we showed you that wrapping an external socket (created outside of the observable context) represents a hot observable. The WebSocket object in this case is the producer of data and it's

important to understand that the scope in which it's created is the ultimate "thermometer," so to speak, to measure whether an observable is considered hot or cold.

8.4.1 Producers as thermometers

Ben Lesh, who is the project lead for RxJS 5, wrote an interesting piece on Medium.com[13] that explains hot and cold observables from the perspective of the producer (i.e WebSockets, eager HTTP requests via promises, and event emitters). He articulates it very eloquently as follows:

> ***COLD is when your observable creates the producer***
>
> ***HOT is when your observable closes over the producer***

In this article, he treats producers as a generic object of type Producer, which represents any object capable of emitting data asynchronously—without necessarily being iterated over. Let's explain the terminology he uses. A cold observable "creates the producer" means that it's created within the scope of the observable context. For instance:

```
const cold$ = new Rx.Observable(observer => {
   const producer = new Producer(); (1)

   // ...Observer listens to producer,
   //    producer pushes events to the observer...

   producer.addEventListener('some-evet', e => observer.next(e));

   return () => producer.dispose();
});
```

(1) **The lifecycle of the producer entity (a generic object) is bound to that of the observable's**

When this stream object is garbage collected, the underlying producer object gets collected with it. Likewise, when the stream is disposed of, it will invoke the mechanism to discard the producer as well. The other implication is that anything that subscribes to `cold$` will obtain its own copy of the producer object, as we've mentioned before. This one-to-one communication between a producer and a consumer (observer) is referred to as *unicast*.

> **UNICAST** **In the world of computer networking, a unicast transmission involves the sending of messages to a single network destination identified by a unique source address.**

On the other hand, hot observables "close over the producer object" that is created or activated outside of the observable context. In this case the lifecycle of the event emitter source is independent of that of the observable's. The term "closes over" derives from the idea

[13] https://medium.com/@benlesh/hot-vs-cold-observables-f8094ed53339#.966re47vq

that the producer object is accessible through the closure formed around the observable declaration.

```
const producer = new Producer(); (1)

const hot$ = new Rx.Observable(observer => {

   // ...Observer listens to producer,
   //    and pushes events onto it...
   producer.addEventListener('some-event', e => observer.next(e)); (1)

   // producer gets disposed of outside of Observable context
   });
```

(1) **Producer object is in scope through the closure formed around the observable declaration**

From our functional programming discussion, you can see that this is not pure since the observable object (or function) is accessing external data directly, which is a side effect. In practice, though, the benefit of doing this is that the producer is now shared by all subscribers to `hot$`, and emits data to all of them—a model known as *multicast*.

> **MULTICAST** In computer networking, multicast refers to a one-to-many form of communication where information is addressed to multiple destinations from a single source.

Figure 8.11 shows a visual explaining the difference between both modes of message passing:

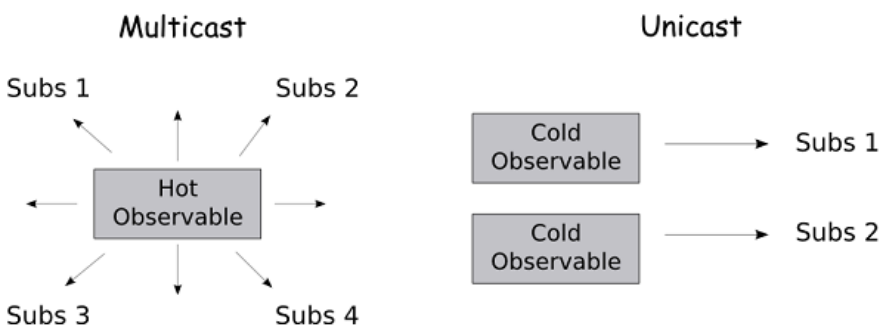

Figure 8.11 Multicast transmits messages from one source to many destinations; unicast sends dedicated messages to one destination address

To sum up, the multicast mode is the process behind hot observables, while unicast messaging occurs with cold observables. Depending on your needs, you can use RxJS to make hot observables cold for isolated, dedicated connections, or vice-versa for shared access to resources. Let's look at examples of each scenario.

8.4.2 Making a hot observable cold

So far, our de facto mechanism for fetching remote data has always involved using promises. Without you realizing it, you've already had to convert observable types before; we did this in chapter 5 so that we could use promises to fetch fresh stock data. Now we can discuss that technique more in-depth and frame the problem this way: if promises are hot because the value they emit is shared amongst all subscribers and are not repeatable or retriable, how can we use it to fetch new stock quotes? Wouldn't the stock price be the same all the time? Again, this has everything to do with where the observable was instantiated. If we execute the promise request globally (eagerly) as in this simple code:

```
const futureVal = new Promise((resolve, reject) => {

   const value = computeValue();

   resolve(value);
});

const promise$ = Rx.Observable.fromPromise(futureVal);

promise$.subscribe(console.log);  (1)

promise$.subscribe(console.log);  (2)
```

(1) Begin invoking the promise
(2) After the first invocation of the promise resolves, all subsequent subscriptions will just resolve to the same value

The same value (or error) is essentially broadcasted to all subscribers. To make this observable cold, we move the instantiation of the promise within the observable context through `ajax()`. Here's a snippet of that code once more:

```
const requestQuote$ = symbol =>
   Rx.Observable.fromPromise(ajax(...))

...

const fetchDataInterval$ = symbol => twoSecond$
     .mergeMap(() => requestQuote$(symbol)
   ...
```

(1) ajax() instantiates a new promise within the Observable context

In essence this is analogous to what you just learned, which is to move the source or the producer of events into the observable context:

```
const coldPromise$ = new Rx.Observable(observer => {
   const futureVal = new Promise((resolve, reject) => {  (1)

      const value = computeValue();

      resolve(value);
   });
```

```
   futureVal.then(result => {
       observer.next(result); ❷
       observer.complete();   ❸
   });
});

coldPromise$.subscribe(console.log); ❹

coldPromise$.subscribe(console.log); ❹
```

❶ Shove the instantiation of the promise into the Observable
❷ Emit the value from the promise
❸ Because we're only expecting a single value, complete the stream
❹ Both subscribers will invoke the internal promise object

We can apply this same principle to WebSockets as well. In section 8.3.3 we used an observable to wrap a global, shared WebSocket object. Here it is once more:

```
const websocket = new WebSocket('ws://localhost:1337'); ❶

const sub = Rx.Observable.fromEvent(websocket, 'message')
  .map(msg => JSON.parse(msg.data))
  .pluck('msg')
  .subscribe(console.log);

websocket.onclose = () => sub.unsubscribe(); ❷
```

❶ Globally shared object
❷ Socket's lifecycle is managed outside of the observable sequence

If you instead wanted to have dedicated connections to each subscriber, you could enclose the activation of the producer within the observable context. This way, every subscriber of the stream, will create its own socket connection:

```
const ws$ = new Rx.Observable(observer => {
  const socket = new WebSocket('ws://localhost:1337'); ❶
  socket.addEventListener('message', e => observer.next(e));
  return () => socket.close();
});

const sub1 = ws$.map(msg => JSON.parse(msg.data))
   .subscribe(msg => console.log(`Sub1 ${msg}`));

const sub2 = ws$.map(msg => JSON.parse(msg.data))
   .subscribe(msg => console.log(`Sub2 ${msg}`)); ❷
```

❶ Not global, but instantiated with every observer
❷ Gets its own dedicated socket connection

This also applies to the RxJS static factory operators such as `create()`, `from()`, `interval()`, `range()`, and others. As mentioned previously, because they are cold by default, you may

want to make them hot with the intention of sharing their content and avoiding not only duplicating your efforts, but also avoiding duplicating resources. Let's look at doing that next.

8.4.3 Making a cold observable hot

In this section, we'll apply the reverse logic. To make a cold observable hot, we need to focus on how they emit data and how subscribers access this data. Circling back to our stock ticker widget, this means that we must move the source of events (stock ticks) away from the observable pipeline. You would want to do something like this if you had stock data being pushed into different parts of the application and we wanted them all in sync with the same event timer. In this case, simply subscribing with multiple observers won't do the trick.

```
const sub1 = ticks$.subscribe( (1)
   quoteDetails => updatePanel1(quoteDetails.symbol, quoteDetails.price)
);

const sub1 = ticks$.subscribe( (1)
   quoteDetails => updatePanel2(quoteDetails.symbol, quoteDetails.price)
);
```

(1) **sub1 and sub2 will use their own 2 second intervals, which could get out of sync, and also they fetch their own copies of stock data. We could optimize this so that the HTTP response can be parsed once and consumed by multiple observers**

The other problem with this is resource usage. With two subscribers, if you were to open the developer console, you'll notice that for each refresh action (2 seconds apart) there would not be one, but two separate requests being sent to the server with each observer. This means that every subscriber would have its own independent interval stream which would incur the expensive cost of establishing the same remote connection multiple times to fetch data against the Yahoo API, as seen in figure 8.12:

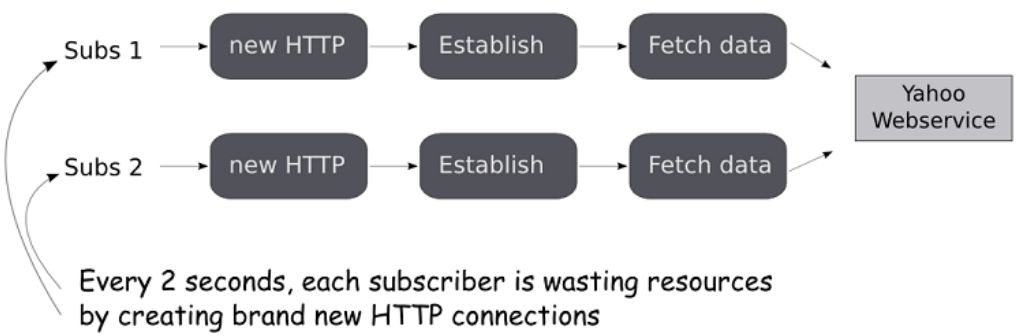

Figure 8.12 With two cold subscriptions, each one is responsible for initiating and allocating resources to make AJAX calls against the same service

Now, why create new HTTP connections, when we could just share this data downstream to multiple subscribers? Again, the answer lies in how the source of stream is activated: as a globally accessible resource or within the observable context—we can't stress that enough. We can expose our service that fetches stock data as a resource that multiple subscribers can observe. But how can we make AJAX calls hot? There can be many ways of doing this, but one idea involves using event emitters with an internal polling mechanism, so that in conjunction with the hot `Rx.Observable.fromEvent()` observable we can pump stock data to all subscribers; this works just like a WebSocket, broadcasting stock ticks independently of when a subscriber exists. Let's take a look at an example of this in listing 8.4

Listing 8.4 Stock ticker as event emitter

```
class StockTicker extends EventEmitter { ❶

  constructor(symbol) {
    super();
    this.symbol = symbol;
    this.intId = 0;
  }

  tick(symbol, price) { ❷
    this.emit('tick', symbol, price);
  }

  start() {
    this.intId = setInterval(() => { ❸
    const webservice =
      `http://finance.yahoo.com/d/quotes.csv?s=${this.symbol}&f=sa&e=.csv`;

      ajax(webservice).then(csv).then(
        ([symbol, price]) => {
         this.tick(symbol.replace(/\"/g, ''), price);
      });
    }, 2000);
  }
  stop() {
    clearInterval(this.intId);
  }
}

const ticker = new StockTicker('FB');
ticker.start(); ❹

const tick$ = Rx.Observable.fromEvent(ticker, 'tick',
        (symbol, price) => ({'symbol': symbol, 'price': price}))
   .catch(Rx.Observable.throw(new Error('Stock ticker exception')));

const sub1 = ticks$.subscribe( ❺
   quoteDetails => updatePanel1(quoteDetails.symbol, quoteDetails.price)
);

const sub2 = ticks$.subscribe( ❺
   quoteDetails => updatePanel2(quoteDetails.symbol, quoteDetails.price)
);
```

1. event emitter to encapsulate a Stock Ticker
2. This represents the 'tick' event
3. Every 2 seconds, poll the stock service for new price data
4. Start the event emitter
5. sub1 and sub2 will share the data from the same source of events

As you can see from the code above, whether we subscribe to the source or not, it will still continue to fetch and send out price ticks. The big difference here being that we've decoupled the subscription from the activation of the event source, essentially now a hot observable. Due to this decoupling, the observable is also removed from the lifecycle of the event source (i.e. `start()` and `stop()`), just like with WebSockets earlier. Thus, when all subscribers unsubscribed, the event emitter will continue to send data—it's just that no one is there to listen. This is expected and most of the hot observables you'll find in the wild (DOM elements, database, or the file system) won't actually emit a complete signal, as their lifecycle extends that of the observable.

Certainly we shouldn't expect that to make cold observables hot we need to have them depend on global producers all the time. We want the best of both worlds, and certainly the nice functional qualities of cold observables. There are many benefits to encapsulating the event source, and have it be managed through the observable's lifecycle (this also ensures less possibilities of memory leaks as all resources are collected and disposed of when it's completely unsubscribed from). On the other hand, we also don't want to duplicate our efforts and instead share the events from a single source to multiple subscribers.

8.4.4 Creating hot-by-operator streams

So far, we've seen that the process of converting a cold stream hot is to place the activation of the producer resource within the context of observable. But this isn't the only way. Fortunately, RxJS provides convenient operators that does just that called `share()`. It's so named because it shares a single subscription to a stream amongst multiple subscribers (kind of like the old days of DirecTV, where a single satellite feed could operate multiple TVs in the same house). This means that we can place this operator right after a set of operations whose results should be common, and the subscribers to each of them will all get the same stream instance (without replaying the pipeline). Just as important, this operator takes care of the management of the underlying streams state such that upon the first subscriber subscribing, the underlying stream is also subscribed to, and when all the subscribers stop listening (either through error or cancellation), the underlying subscription is disposed of as well. Brilliant! When a new subscriber comes in, the source is reconnected and the process is started over again. Here's a quick example that shows the same result shared without replaying the entire sequence:

```
const source$ = Rx.Observable.interval(1000)
  .take(10)
  .do(num => {
      console.log(`Running some code with ${num}`);
   });
```

```
const shared$ = source$.share(); ❶

shared$.subscribe(createObserver('SourceA')); ❷

shared$.subscribe(createObserver('SourceB')); ❸

function createObserver(tag) { ❹
    return {
        next: x => {
            console.log(`Next: ${tag} ${x}`);
        },
        error: err => {
            console.log(`Error: ${err}`);
        },
        complete: () => {
            console.log('Completed');
        }
    };
}
```

❶ Convert the cold observable to hot
❷ When the number of observers subscribed to published observable goes from 0 to 1, we connect to the underlying observable sequence.
❸ When the second subscriber is added, no additional subscriptions are added to the underlying observable sequence. As a result the operations that result in side effects are not repeated per subscriber.
❹ Helper method to create a simple observer for standard out

Once the observable in front of `share()` is subscribed to, it's for all intents and purposes, hot; *this is* known as *hot-by-operator,* and it's the best way to heat up a cold observable. Running this code illustrates that a single subscription is shared to the underlying sequence:

```
"Running some code with 0"
"Next: SubA 0"
"Next: SubB 0"
"Running some code with 1"
"Next: SubA 1"
"Next: SubB 1"

... and so on...

"Completed"
"Completed"
```

Pitfall: sharing with a synchronous event source

The `share()` operator is very useful in many cases where the subscribers will subscribe at different points in time but are somewhat tolerant of data loss. Since it can be used following any Observable, it is sometimes confusing to newcomers who might be tempted to do the following:

```
const source$ = Rx.Observable.from([1,2,3,4])
  .filter(isEven)
  .map(x => x * x)
```

```
  .share();
source$.subscribe(x => console.log(`Stream 1 ${x}`));
source$.subscribe(x => console.log(`Stream 2 ${x}`));
```

The above code is often seen as an easy efficiency win for those new to reactive programming. If the pipeline executes for each subscription, then it makes sense that by adding the share operator we can force it to only execute once and both Observers can use the results. As the console will tell you however, this does not appear to occur. Instead, only stream 1 seems to get executed. The reason for this is twofold. The first is scheduling, which we will gloss over for right now as it is covered in a later chapter. In basic terms, subscribing to a synchronous source like an array will execute and complete before the second subscribe statement is even reached. The second reason is that `share()` has introduced state into our application. With it, the first subscription always results in the Observable beginning to emit and so long as at least one subscriber continues to listen it will continue to emit until the source completes. If you are not careful, this kind of behavior can become a subtle bug.

When dealing with Observables that run immediately like those in the example above, this can result in only a single subscriber receiving the events.

Let's apply this to our original stock ticker stream so that we can avoid making unnecessary HTTP calls with each subscriber that subscribes to the stream; instead we only make one call that broadcasts out to any subscribers. Making our stock ticker stream hot is as simple as adding `share()` at the end of it:

```
const ticks$ = symbol$.mergeMap(fetchDataInterval$).share();

const sub1 = ticks$.subscribe( (1)
   quoteDetails => updatePanel1(quoteDetails.symbol, quoteDetails.price)
);

const sub1 = ticks$.subscribe( (1)
   quoteDetails => updatePanel2(quoteDetails.symbol, quoteDetails.price)
);
```

(1) **Just like with the global event emitter sample, all subscribers will receive the same tick data. We'll show a more complete version of this code in a bit**

And now we've officially completed the stock ticker code. Here's the full rendition with all of the parts added from the operators you learned about in chapters 5 through 8. Let's recap a bit:

- Chapter 5: you learned how to use higher-order observables and flattening operators such as `mergeMap()`
- Chapter 6: you learned to coordinate multiple observables with `combineLatest()`
- Chapter 7: you added fault tolerance to your streams with retry and error handling
- Chapter 8: you learned how to convert a cold observable into a hot observable that shares its event data with many subscribers. To show this we'll have 2 subscribers updating different parts of the site.

Let's put all of this together in listing into listing 8.6 and enhance our stock widget with the ability to track the price and day's change for all stocks (we'll omit the CSS and HTML code, which you can find in the GitHub repository). With these changes our UI is updated like this:

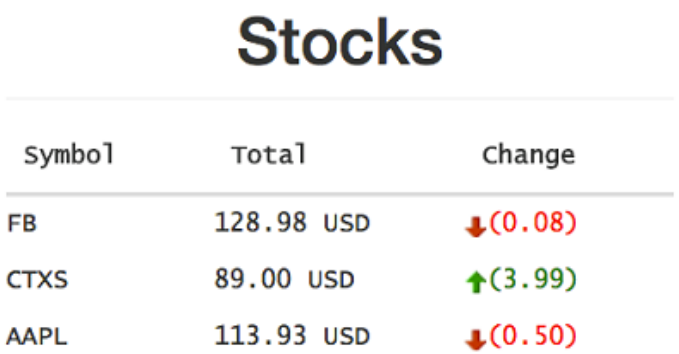

Figure 8.13 Screenshot of the live HTML stock ticker component as it ticks every 2 seconds and also includes additional logic to compute the stock's price change from the opening price (prices are subject to market conditions)

We'll make several changes to the code we started in listing 5.6 since we'll combine the stock ticket stream with another stream against the same service to read the company's stock opening price. Subtracting the current price with the gives us the next change. These will be two independent subscriptions parting from the commonly shared `tick$` stream, which is now hot. Listing 8.5 shows us the complete code and make use of a lot of the techniques you've learned about so far.

Listing 8.5 Complete stock ticker widget with change tracking

```
const csv = str => str.split(/,\s*/);                    ❶
const cleanStr = str => str.replace(/\"|\s*/g, '');

const webservice = 'http://download.finance.yahoo.com/d/quotes.csv
     ?s=$symbol&f=$options&e=.csv';                       ❷

const requestQuote$ = (symbol, opts = 'sa') =>           ❸
   Rx.Observable.fromPromise(
       ajax(webservice.replace(/\$symbol/,symbol)        ❹
          .replace(/\$options/, opts)))
       .retry(3)
       .catch(err => Rx.Observable.throw(
           new Error('Stock data not available. Try again later!')))
       .map(cleanStr)                                    ❺
       .map(data => data.indexOf(',') > 0 ? csv(data) : data); ❻


const twoSecond$ = Rx.Observable.interval(2000);         ❼

const fetchDataInterval$ = symbol => twoSecond$
```

```
    .mergeMap(() => requestQuote$(symbol)
         .distinctUntilChanged((previous, next) => {        ❽
             let prevPrice = parseFloat(previous[1]).toFixed(2);
             let nextPrice = parseFloat(next[1]).toFixed(2);
             return prevPrice === nextPrice;
         }));


const symbol$ = Rx.Observable.of('FB', 'CTXS', 'AAPL');       ❾

const ticks$ = symbol$.mergeMap(fetchDataInterval$).share(); ❿

ticks$.subscribe( ⓫
  ([symbol, price]) => {
     let id = 'row-' + symbol.toLowerCase();
     let row = document.querySelector(`#${id}`);
     if(!row) {
      addRow(id, symbol, price);
     }
     else {
      updateRow(row, symbol, price);
     }
  },
  error => console.log(error.message));

ticks$            ⓬
  .mergeMap(([symbol, price]) =>
    Rx.Observable.of([symbol, price])
      .combineLatest(requestQuote$(symbol, 'o')))
  .map(R.flatten) ⓭
  .map(([symbol, current, open]) => [symbol, (current - open).toFixed(2)])
  .do(console.log)
  .subscribe(([symbol, change]) => {
      let id = 'row-' + symbol.toLowerCase();
      let row = document.querySelector(`#${id}`);
      if(row) {
        updatePriceChange(row, change);
      }
    },
    error => console.log(`Fetch error occurred: ${error}`)
   );

const updatePriceChange = (rowElem, change) => {
   let [,, changeElem] = rowElem.childNodes;
   let priceClass = "green-text", priceIcon="up-green";
   if(parseFloat(change) < 0) {
       priceClass = "red-text"; priceIcon="down-red";
   }
   changeElem.innerHTML =
     `<span class="${priceClass}">
        <span class="${priceIcon}">
           (${parseFloat(Math.abs(change)).toFixed(2)})
        </span>
      </span>`;
};
```

❶ **Helper function to split a string into a comma separated set of values (CSV)**

❷ The Yahoo finance web service to use with additional options used to query for the pertinent data
❸ Function used to create a stream that invokes a promise that fetches data from the web service, and parses the result into CSV
❹ Apply options to the request URI. s = symbol; a = asking price; o = open
❺ Clean string from any white spaces and unnecessary characters
❻ Convert output to CSV
❼ Our two second observable used to drive the execution of the real-time poll.
❽ Only propagate stock price values when they have changed. This avoids unnecessary code from executing when price values remaining the same every 2 seconds
❾ Stock symbols to render data for. These can be any symbols and keep in mind that to see live changes, you must run the program during market hours
❿ Share the stock data with all subscribers
⓫ First subscription. Creates all necessary rows and updates the price amount in USD
⓬ Second subscription. A conformant stream that combines the price of the stock at the open, so that it can compute the change amount for the day. It appends the next change price in the stock
⓭ Use Ramda to flatten the internal array of data passing through, making it easier to parse

The `share()` method is a useful and powerful shortcut to implement these very complex examples with a relatively small amount of lines of code. But to understand what's really happening under the hood we need to dive a little deeper and explore the domain of an observable variety called `ConnectableObservable`.

8.5 Connecting one observable to many observers

We mentioned previously that in RxJS and the networking world, a single point to point transmission is known as unicast, while the one-to-many transmission is known as multicast. As you saw from example above, `share()` is a multicast operator, but there are other flavors or specializations of it all derived from a single generic function known as `multicast()`. In practice, typically you'll never actually use `multicast()` directly, but one of its specializations.

> **RXJS 5 IMPROVEMENT** It's important to recall that the RxJS 5 team has done a great job at cutting the API surface pertaining to the set of operators used to share and publish values by about 75%.

Understanding all of the specializations for multicasting (also known as the multicasting operators in RxJS parlance) can require a whole book of its own. We won't be using them much in this book, and you won't need them in your initial exploration of RxJS. When the time arises, `share()` is all you need (no pun intended). Nevertheless, it's important to be aware that RxJS gives you a lot more control over the amount and types of data to emit when you need it. So we'll spend some time talking about the most common ones:

- Publish
- Publish with replay
- Publish last

8.5.1 Publishing the stream data

The first specialization is the operator `publish()`. This is the vanilla multicast specialization. The idea of it is to create an observable (like `share()`) which allows a single subscription to be distributed to several subscribers. The difference between these operators is one of simplicity versus control. Where `share()` automatically managed the subscription and unsubscription of the source stream based solely on the number of subscribers, `publish()` is slightly more low-level. Using the same `source$` stream as before:

```
const source$ = Rx.Observable.interval(1000)
  .take(10) ❶
  .do(num => {
      console.log(`Running some code with ${num}`);
   });
const published$ = source$.publish();

published$.subscribe(createObserver('SubA'));

published$.subscribe(createObserver('SubB'));
```

❶ Making our stream finite

Now when we run this code, there is no output, as if the stream is just sitting idle. The issue that we are encountering is that `publish()` returns a derivation of observables known as a `ConnectableObservable`. This new type requires a more explicit initiation step than `share()`. Where the latter connected on first subscription, the former requires another call to build the underlying subscription. We can start the source observable by calling `connect()` on the resulting `ConnectableObservable`:

```
published$.connect(); ❶
```

❶ Can be invoked at any point after the call to publish()

> **CAUTION** We made our stream finite in the code sample above for a very important reason. `connect()` is a low level operator, which means we're bypassing all of the nice subscription management logic available in the core RxJS creational operators. In other words, it's not a managed subscription. `connect()` can be a very powerful tool, but it's up to you to ensure that the stream is unsubscribed from at some point, otherwise you'll cause memory leaks.

As soon as we call the `connect()` method, our observable will act just like the one in the previous example. You can visualize this process with figure 8.14:

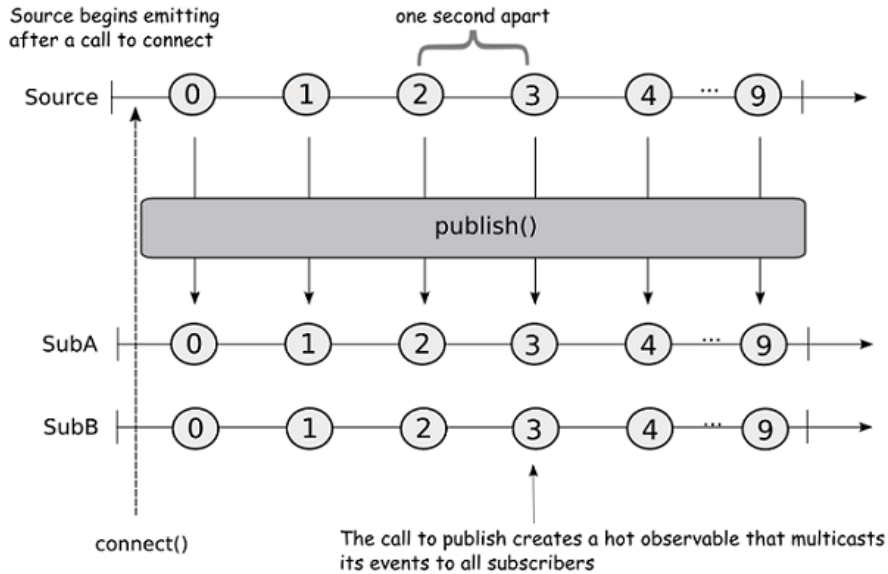

Figure 8.14 Publish creates a hot observable whereby all subscribers begin receiving the events as soon as the call to connect is made. Otherwise, the source stream behaves like a cold observable sitting idle until connect() is called. The share() operator would yield the same results except that the call to connect() would be done internally by the library

If you were to examine `ConnectableObservable`, you would see an interface roughly shaped like this:

```
interface ConnectableObservable<T> extends Observable<T> {
  connect() : Subscription
  refCount(): Observable<T>
}
```

NOTE Observables which share subscriptions are generally called hot and those that don't cold; however, we also sometimes refer to observables which only start emitting events after the first subscription (a quality seen only in cold observables) and thereafter share their data to all subscribers as "warm."

These are two important concepts to understand:

- The `connect()` method returns a `Subscription` instance which represents the shared underlying subscription. Unsubscribing from it will result in both subscribers no longer receiving any events. You saw how to use connect in the example above.
- The `refCount()` method is named after the garbage collection concept known as "ref counting" or reference counting. The point is that it returns an observable sequence that stays "connected" to the source as long as there's a least one active subscription. Does this sound familiar? It should because the `share()` operator we discussed at moment ago is nothing more than an alias for `publish.refCount()`.

Now you understand why `share()` is just a shortcut for what's really happening under the hood. RxJS creates warm observables that multicast their values out to all connected subscribers managed through the connectable observable, which keeps a count of all active subscribers. When all subscribers have unsubscribed, it will unsubscribe from the source observable.

The use of the internal ref counting is crucial to the efficiency of RxJS. As we mentioned earlier, an important difference between hot and cold observables is when their lives start and when they can be considered to have ended. Hot observables can produce events in the absence of observers, while cold observables do not become active until it has a subscription. A result of this is that hot streams will often have much longer lifespans than their cold counterparts. Where a cold stream will shut down when the subscriber shuts down, a hot can continue running after the end of a subscription. This can have important implications depending on the source. If a hot observable is unmanaged, that is, its state is not being maintained by `refCount()`, then its state (and the associated resources) can easily be forgotten about and cause a memory leak—which is what happens when a connectable observable emits infinitely many events. While a majority of the library introduced operators such as `Rx.Observable.fromEvent()`, which are intrinsically wrapping hot sources and taking care to manage their own disposed state. If you ever find yourself creating a hot observable explicitly with one of the `multicast()` family of operators, it is worth also asking yourself when those streams will be destroyed and how many will be created. It may also be helpful to identify where and when that stream would be disposed of and explicitly unsubscribe from it to avoid taking up unnecessary resources. This is especially important in single page applications with multiple views where it is easy to create streams within a view without properly disposing of them.

Publish is just one flavor of hot observable, there are several others that can be useful depending on the desired behavior on subsequent subscription. Suppose we wanted to have a moving window of past values to be emitted to all observables. Earlier we talked about the differences between replaying the results of a sequence and re-subscribing to execute the entire sequence again. We can mix that concept with publish.

8.5.2 Publish with replay

We could use another specialization of multicast called `publishReplay()` to emit the last 1, 10, 100 or all of the most recent values to all subscribers (obviously this is another case of a warm observable). This operator uses several parameters to determine the characteristics of a buffer to maintain. And as with any of the buffering operators we learned about in chapter 4, we caution you again that the use of buffers can be dangerous when replaying entire sequences and the the buffer grows indefinitely. You can see this clearly if you were to inspect the signature of this operator:

```
publishReplay(bufferSize = Number.POSITIVE_INFINITY,
              windowTime = Number.POSITIVE_INFINITY)
```

This operator is analogous to the RxJS 4 `shareReplay()` operator, which had the same issue. So, using `publishReplay()` with empty arguments can be dangerous. Here's an example of this operator. Unlike the publish example, to showcase the use of this operator, we have to simulate a subscriber coming at a later time.

```
const source$ = Rx.Observable.interval(1000) ❶
  .take(10)
  .do(num => {
      console.log(`Running some code with ${num}`);
   });

const published$ = source$.publishReplay(2); ❷

published$.subscribe(createObserver('SubA')); ❸

setTimeout(() => {
  published$.subscribe(createObserver('SubB'));  ❹
}, 5000)

published$.connect();
```

❶ Begin a counter that pushes integers every second, starting at zero. It creates a side effect via do() to show that it's running
❷ Create a publish replay observable that can store two past events and re-emit them to any new subscribers that connect
❸ Subscriber A connects subscribers immediately and it begin receiving events from count 0
❹ Subscribing 5 seconds later, subscriber B should begin receiving events starting with the number 4 approximately, yet because we're replaying the last 2 elements, it receives numbers 2 and 3 at the moment subscriber A receives 3. In other words, at the moment it subscribers it receives 2 elements in the buffer, the current element and the one before it.

Running this code would print the following output. What you will notice here is that as soon as the second subscriber comes it will first make sure to emit the last events in the stream (current and previous); afterwards, both streams will replay the same events.

```
"Running some code with 0"
"Next: SubA 0"
"Running some code with 1"
"Next: SubA 1"
"Running some code with 2"
"Next: SubA 2"
"Running some code with 3"
"Next: SubA 3"
"Next: SubB 2"
"Next: SubB 3"  ❶
"Running some code with 4"
"Next: SubA 4"  ❷
"Next: SubB 4"
"Running some code with 5"
"Next: SubA 5"
"Next: SubB 5"
...
"Next: SubA 9"
"Next: SubB 9"
"Completed"
```

```
"Completed"
```

1. SubB will begin receiving the last 2 events (previous and current)
2. Both subscribers receive the same data

Here's a diagram of what's happening in the code sample above:

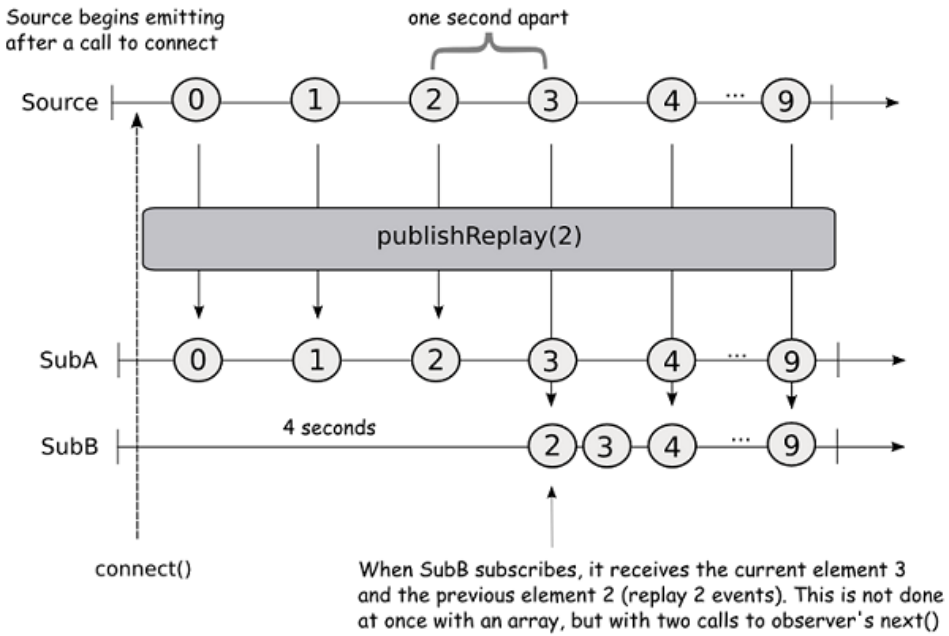

Figure 8.15 Publish replay with 2 will emit the last 2 elements in the buffer (current and previous) at the moment the subscriber subscribers to the stream. Afterwards, both streams receive the latest value emitted

Alternatively, when you want only the last value to be emitted, `publishLast()` will do the trick.

8.5.3 Publish last

Publish is simple to understand. It returns a connectable observable sequence that shares a single subscription containing only the last notification. This operator is analogous to `last()` (except non-blocking) in that it multicasts the last observable value from a sequence to all subscribers. Following our simple example once more:

```
const published$ = source$.publishLast();

published$.subscribe(createObserver('SubA'));

published$.subscribe(createObserver('SubB'));
```

```
published$.connect();
```

Running this code prints out:

```
"Running some code with 0"
"Running some code with 1"
...
"Running some code with 9"
"Next: SubA 9"
"Next: SubB 9"
"Completed"
"Completed"
```

There are many overloaded specializations of `multicast()` in RxJS. You can find them all here, starting with: https://github.com/ReactiveX/rxjs/blob/master/src/operator/publish.ts. If you inspect these operators, what you'll find is that they delegate most of their work to entities called "subjects."

Further reading

Theoretically subjects can be used to supplant virtually all of your observable needs, and the possibilities are endless. Because this is an advanced topic, in this book we prefer sticking to the managed observables created via RxJS factory methods, but if you would like to read on further, we recommend you begin with the RxJS manual:

http://xgrommx.github.io/rx-book/content/subjects/index.html

Overall, RxJS provides hot and cold observables because the designers recognized there are different types of sources that must often be addressed, with different needs. There is no "one size fits all" solution to every problem, but `share()` will get you a long way ahead. However, to that end, there is also a steeper learning curve to mount in order to fully understand the library, especially the topic on multicasting operators. We hope this chapter has demystified some of the strangeness around these different observable types. But when all else fails, here's a good analogy to keep in mind (taken from the Reactive Extensions GitHub project[14]):

- A cold observable is like watching a movie.
- A hot observable is like watching live theater.
- A hot observable replayed is like watching a play on video.

These lessons will be useful moving into the next chapter when we discuss how we can test our Rx pipelines.

[14] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/creating.md

8.6 Summary

- A cold observable is passive in that it waits until a subscriber is listening to execute an an individual pipeline for each subscriber. Cold observables manage the lifecycle of the event producer.
- Hot observables are active and can begin emitting events regardless of whether subscribers are listening. Hot observables close over the producer of events, so its lifecycle is independent of the source.
- Event emitters such as WebSockets and DOM elements are examples of hot observables
- Events from hot observables will be lost if no one is listening, while cold Observables will always rebuild their pipeline every subscription.
- `share()` makes observers use the same underlying source stream, and disconnects when all of the subscribers stop listening. This operator can be used to make a cold observable hot, or at least "warm."
- Using operators such as `publish()`, `publishReplay()`, and `publishLast()` are create multicast observables.