Covers RxJS 5

SAMPLE CHAPTER

RxJS IN ACTION

Paul P. Daniels
Luis Atencio

FOREWORD BY Ben Lesh

MANNING

RxJS in Action

by Paul P. Daniels

Luis Atencio

Chapter 1

brief contents

1 Thinking reactively

This chapter covers

- Comparing asynchronous JavaScript with callback- and `Promise`-based solutions
- Using streams to model static, dynamic, and time-bound data
- Using observable streams to handle unbounded data in a functional manner
- Thinking reactively to deal 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. These tiny pinpricks of data light up the internet and pass ubiquitously through semiconductors scattered across the planet. A deluge of data propagates from any connected device. What does this have 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 your 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 (UI) to become unresponsive. 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 modern web users.

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 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.

Although 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 haven't 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 we're trying to solve involves data that lives not in one but in several different remote locations.

The solution lies not only in which library to use but which paradigm best suits these types of problems. In this book, you'll first 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, you'll learn about the principles that lead to thinking reactively as well as the current solutions, their drawbacks, and how RxJS improves on them. With this new-found mindset, you'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, autocompleting 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 you work through the most important features of RxJS. Finally, all these new techniques will come together to end your journey with a full-scale web application using a hybrid React/Rx architecture.

The goal of this chapter is to give a broad view of the topics you'll be learning about in this book. We'll focus on looking at the limitations of the current solutions and point you to the chapters that show how RxJS addresses them. Furthermore, you'll learn how to shift your mindset to think in terms of *streams*, also known as *functional sequences of events*, which RxJS implements under the hood through the use of 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,

which also scales to any amount of data. Understanding the differences between these two worlds is crucial, so let's begin there.

1.1 Synchronous vs. 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 your 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 doesn't 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 not acceptable, so you need to take advantage of asynchronous execution so that the application is always responsive. The main issue 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 put the burden on your users to wait for their processes to complete. Many systems still work this way today, such as ATMs, point of sale systems, and other dumb terminals. Writing code this way is much easier to grasp, maintain, and debug; unfortunately, because of JavaScript's single-threaded nature, any long-running tasks such as waiting for an AJAX call to return or a database operation to complete shouldn't be done synchronously. Doing so creates an 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 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.

In this case, the program makes a blocking call to process 1, which means it must wait for it to return control to the caller, so that it can proceed with process 2. This might work well for kiosks and dumb terminals, but browser UIs should never be implemented this way. Not only would it create a terrible user experience (UX), 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 on the server to respond:

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

Loading server-side data synchronously halts program execution. The nature of the data isn't important right now; it's some generic sample data pertaining to your application.

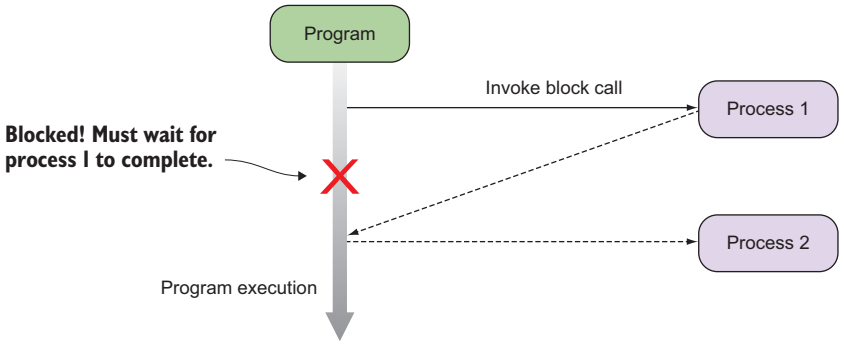

Figure 1.1 A program that invokes two processes synchronously. A process in this case can be as simple as a function call, an I/O process, or a network transaction. When process 1 runs, it blocks anything else from running.

A better approach would be to invoke the HTTP call and perform other actions while you're waiting on the response. Long-running tasks aren't 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 provides callback functions.

1.1.2 Non-blocking code with callback functions

Using functions as callbacks has been a staple of JavaScript development for years. They're used in everything from mouse clicks and key presses to handling remote HTTP requests or file I/O. 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 is ready for use. In the meantime, your application can continue carrying out any other task, as shown in figure 1.2.

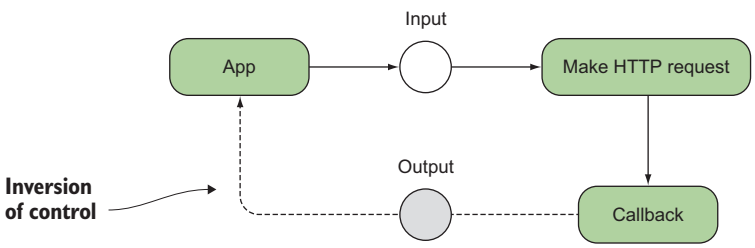

Figure 1.2 Callback functions in JavaScript create an inversion of control where functions call the application back, instead of the other way around.

Unlike the previous code that makes a blocking HTTP call that you must wait for, using callbacks with asynchronous (AJAX) requests 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 this case, the runtime calls you (or returns control to you) via the function handler when the data is ready to be processed; hence, the term *callback*. Look at this alternative:

```
ajax('/data',                    <-- No explicit return value
  items  => {                    <-- Declaration of callback function
     items.forEach(item => {
        // process each item      <-- All processing is carried out within the callback body
     });                            after the data has been fetched from the server.
});
beginUiRendering();              <-- This function begins immediately after AJAX is called.
```

Callback functions allow you to invoke code asynchronously, so that the application can return control to you later. This allows the program to continue with any other task in the meantime. In this code sample, the HTTP function runs in the background and immediately returns control to the caller to begin rendering the UI; it handles the contents of the items only *after* it has completely loaded. This behavior is ideal because 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 I/O-bound work like fetching data from the web or a database. The reason this works is that I/O processes are typically much slower than any other type of instruction, so we allow them to run in the background because they're not dependent on processor cycles to complete.

> **SYNTAX CHECK** In the code sample in section 1.1.2, the second parameter of `ajax()` is the callback function. In that code, as in many parts of the book, we use the ECMAScript 6 lambda expression syntax,[1] which offers a terser and more succinct way of invoking functions. Also called *arrow functions*, lambda expressions behave somewhat similarly to an anonymous function call, which you're probably familiar with. The subtle difference has to do with what the keyword `this` refers to. On 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 does this mean? State can be understood simply as a snapshot of all the information stored into

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

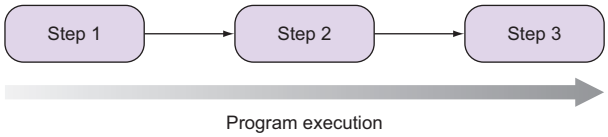

Figure 1.3 Synchronous code is a step-by-step sequential execution of statements where each step depends on the previous one to run.

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, as shown in figure 1.3.

In this model, it's easy to determine at any point what the states of the variables are and what will occur next, which is why it's easy to write and debug. But when tasks have different wait times or complete at different times, it's 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 grows not only vertically but also horizontally.

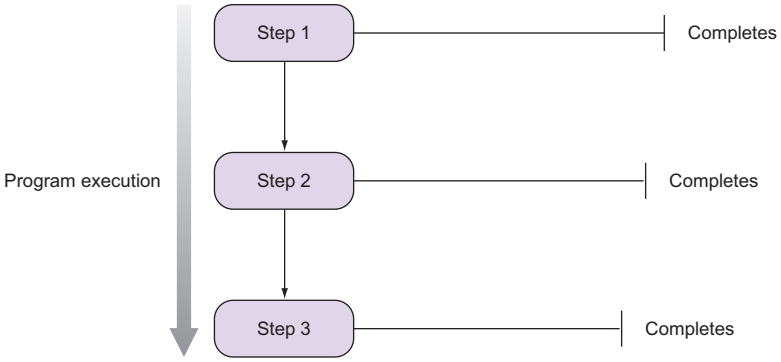

Figure 1.4 In asynchronous execution, steps that are invoked in sequence need not terminate all at the same time. So there's absolutely no guarantee that you can rely on the data from step 1 to be available in step 2, for example.

As of now, if steps 1, 2, and 3 were independent tasks, then executing them in any order wouldn't be a problem. But if these were functions that shared any global state, then their behavior would be determined by the order in which they were called or by the global state of the system. These conditions we refer to as *side effects*, which you'll learn more about in chapter 2; they involve situations where you need to read or modify an external resource like a database, the DOM, the console, and others. Functions with side effects can perform unreliably when run in any arbitrary order. In functional and reactive programming, you'll learn to minimize them by using *pure functions*, and you'll learn in this book that 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 complete depending on the nature of the work. The main issue is how we can guarantee that these steps run in the correct order. As you've probably done many times before, the proper way to achieve this 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 the application's runtime resembles figure 1.5.

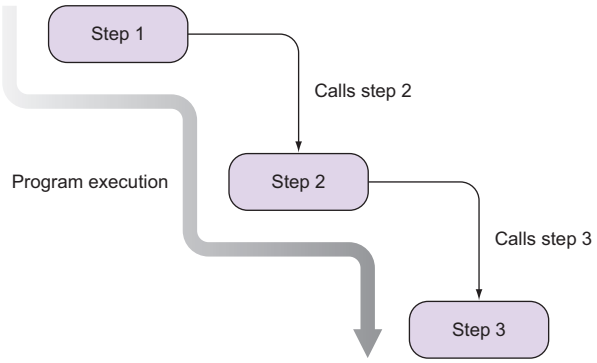

Figure 1.5 In order to guarantee the proper order of steps and asynchronous invocation takes place, we use callback functions to transfer control of the application once a long-running operation terminates.

Undoubtedly, this nested control flow is much harder to reason about than the synchronous, straight-line model of figure 1.4. 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 it 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 these systems rely on callbacks.

1.1.4 Are callbacks out of the picture?

The short answer is no. Using a paradigm to tackle event-based or asynchronous code isn't necessary when you're dealing with simple interactions with users or external services. If you're writing a simple script that issues a single remote HTTP request, RxJS is a bit of overkill, and callbacks remain the perfect solution. On the other hand, a library that mixes functional and reactive paradigms 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 the need to orchestrate the execution of several business processes that consume several microservices, data mashups, or perhaps the implementation of features of a rich UI made up of several widgets on the page that interact with each other.

Consider the task of loading data from the client originating from different remote server-side endpoints. To coordinate among them, you'd need several nested AJAX requests where each step wraps the processing of the data residing within each callback body in the logic of invoking the next step, as you saw previously in figure 1.5. Following is a possible solution for this, which requires the use of three composed callback functions to load datasets that potentially live in the same host or different hosts, together with its related meta-information and files:

```
ajax('<host1>/items',                          <-- Loads all items you want to display
   items => {
      for (let item of items) {
      ajax(`<host2>/items/${item.getId()}/info`,   <-- For each item, loads additional meta-information
         dataInfo => {
         ajax(`<host3>/files/${dataInfo.files}`,   <-- For each meta record, loads associated files
             processFiles);
       });
    }
});
beginUiRendering();
```

Now although you might think this code looks trivial, if continuing this pattern, we'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. It isn't simply aesthetics—making sure that separate asynchronous operations are synchronized is hard enough without also having difficult-to-read code. There's 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 aren't aware that there's latency in those calls, so they'll always march ahead no matter what, which can cause some really unpredictable and hard-to-diagnose bugs. In these situations, you can improve matters by creating closures around your asynchronous functions, managed by using `forEach()` instead of the loop:

```
ajax('<host1>/items',
   items => {
      items.forEach(item => {          <-- The forEach() method of arrays will properly scope each item object into the nested HTTP call.
         ajax(`<host2>/items/${item.getId()}/info`,
         dataInfo => {
         ajax(`<host3>/files/${dataInfo.files}`,
         processFiles);
       });
     });
});
```

This is why in RxJS—and FP in general, for that matter—all loops are virtually eliminated! Instead, in chapters 4 and 5 you'll 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 popular mechanisms for asynchronous event-based architectures. The DOM, for instance, is probably one of the most widely known event emitters. On a server like Node.js, certain kinds of objects periodically produce events that cause functions to be called. In Node.js, the `EventEmitter` class is used to implement APIs for things like WebSocket I/O or file reading/writing so that if you're iterating through directories and you find a file of interest, an 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; see figure 1.6.

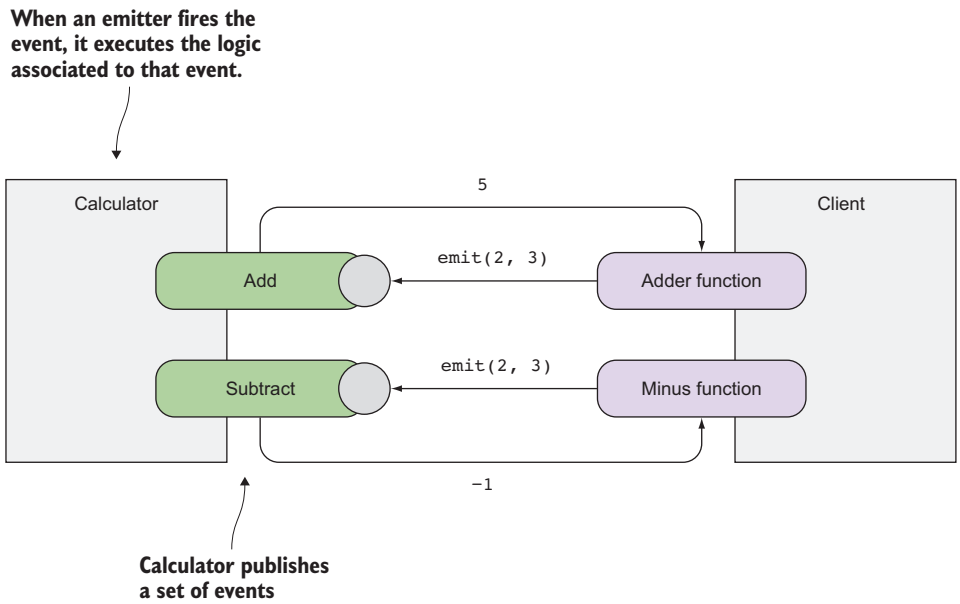

Figure 1.6 Node emitter object representing a simple calculator, which exposes two events: `add` and `subtract`

Here's some code for the calculator `add` and `subtract` events:

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

class Calculator extends EventEmitter {}

const calc = new Calculator();

calc.addListener('add', (a, b) => {
  calc.emit('result',  a + b);
});
calc.addListener('subtract', (a, b) => {
  calc.emit('result', a - b);
```

Loads the events module

Creates a custom emitter

Handles the add event

```
});

calc.addListener('result', (result) => {
 console.log('Result: ' + result);
});

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

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 problems associated with using callbacks to handle 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 these types of issues. With the help of patterns emerging from FP, 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 introduced `Promises` to represent any asynchronous computation that's expected to complete in the future. With `Promises`, you can chain together a set of actions with future values to form a *continuation*.[2] A continuation is just a fancy term for writing callbacks and has a lot to do with the principle of Inversion of Control we referenced earlier. A continuation (a callback) allows the function to decide what it should do next, instead of indiscriminately waiting for a return value. They're used heavily when iterating over arrays, tree structures, `try/catch` blocks, and, of course, asynchronous programming. So, the code you saw earlier—

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

—is known to be continuation-passing style (CPS), because none of the functions are explicitly waiting for a return value. But as we mentioned, abusing this makes code hard to reason about. What you can do is to make continuations first-class citizens and actually define a concrete interpretation of what it means to "continue." So, we introduce the notion of `then`: "Do X, `then` do Y," to create code that reads like this:

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

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

The key term "then" suggests time and sequence.

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. A `Promise` is considered to be fulfilled when its underlying operation completes, at which point subscribers will receive the computed result. Because we 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 in our programs. Different `Promise` implementations exist based on the Promises/A+ protocol (see https://promisesaplus.com/), and it's designed to provide some level of error handling and continuations via the `then()` methods. Here's how you can tackle the same example if you assume that `ajax()` returns `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);
    )
  );
```

This looks similar to the previous statement! Being a more recent addition to the language with ES6 and inspired in FP design, `Promises` are more versatile and idiomatic than callbacks. Applying these functions declaratively—meaning your code expresses the *what* and not the *how* of what you're trying to accomplish—into `then` blocks allows you to express side effects in a pure manner. We can refactor this to be more declarative by pulling out each function independently

```
let getItems = () => ajax('<host1>/items');
let getInfo  = item => ajax(`<host2>/data/${item.getId()}/info`);
let getFiles = dataInfo => ajax(`<host3>/data/files/${dataInfo.files}`);
```

and then use `Promises` to stitch together our asynchronous flow. We use the `Promise.all()` function to map an array of separate `Promises` into a single one containing an array of results:

```
getItems()
  .then(items => items.map(getInfo))
  .then(promises => Promise.all(promises))
  .then(infos => infos.map(getFiles))
  .then(promises => Promise.all(promises))
  .then(processFiles);
```

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

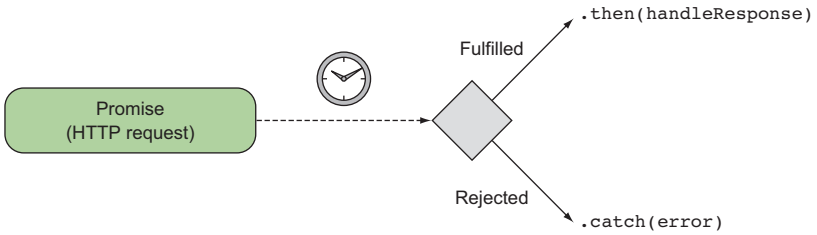

Figure 1.7 `Promises` create a flow of calls chained by `then` methods. If the `Promise` is fulfilled, the chain of functions continues; otherwise, the error is delegated to the `Promise catch` block.

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. Also, they lack 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 isn't honored through the `Promise` interface. These limitations reduce their usefulness and force developers to write some of the cancellation logic themselves or seek other libraries.

Collectively, `Promises` and event emitters solve what are essentially the same problems in slightly different ways. They have different use cases (`Promises` for single-value returns like HTTP requests and event emitters for multiple-value returns like mouse click handlers), mostly because of their own implementation constraints, not because the use cases are so different. The result 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 aren't the only reasons that we, as developers, need to worry about 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.

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, or others, yet they all fall short one way or another. We believe that it's not a matter of just choosing a library,

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

but choosing the right paradigm for the job. By combining functional and reactive programming paradigms, RxJS will help you address the following issues:

- Familiar control flow structures (like `for` and `while` loops) with asynchronous functions don't work well together because they're not async aware; that is, they're oblivious of wait time or latency between iterations.
- Error-handling strategies become easily convoluted when you begin nesting `try/catch` blocks within each callback. In chapter 7, we'll approach error handling from a functional perspective. Also, if you want to implement some level of retry logic at every step, this will be incredibly difficult even with the help of other libraries.
- Business logic is tightly coupled 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 reusable and modular components in order to have loosely coupled business logic that can be maintained and unit tested independently. We'll cover unit testing with RxJS in chapter 9.
- You want to avoid excessive use of closures, but functions in JavaScript create a closure around the scope in which they're declared. 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 FP addresses this problem. 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 a few loops and conditional `if-else` statements into the mix, and you'll regret the day a bug occurs that impacts this functionality.
- 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's 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 third-party libraries.
- One good quality of responsive design is to always throttle a user's interaction with any UI components, so that the system isn't 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 their local scope, which breaks the stability of your entire program.

- It's rare to be concerned about memory management in JavaScript applications, especially client-side code. After all, the browser takes care of most of these low-level details. But as UIs become larger and richer, we can begin to see that lingering event listeners may cause memory leaks and cause the size of the browser process to grow. It's true that this was 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 in the right direction (and RxJS integrates with `Promises` seamlessly if you feel the need to do so). But what you really need is a solution that abstracts out the notion of latency away from your code while allowing you to model your solutions using a linear sequence of steps through which data can flow over time, as shown in figure 1.8.

In essence, you need to combine the ability to decouple functionality like event emitters with the fluent design pattern of `Promises`, all into a single abstraction. Moreover, you 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 these questions: How can you write code as a linear sequence of steps that acts only after some event has occurred in the future? How do you combine it with other code that might have its own set of constraints? Your desire for synchronicity isn't just about convenience; it's what you're used to. Unfortunately, most of the common language constructs that you use in synchronous code aren't 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, emerging as what's known as the *Reactive Extensions*—we have arrived!

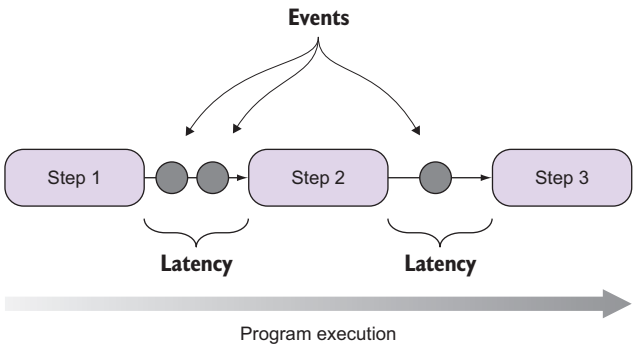

Figure 1.8 RxJS can treat asynchronous data flows with a programming model that resembles a simple chain of sequential steps.

1.4 The Reactive Extensions for JavaScript

Reactive Extensions for JavaScript (RxJS) is an elegant replacement for callback or `Promise`-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, RxJS is also inherently robust and easy to test with a vibrant community to support it. The power of RxJS derives from being built on top of the pillars of functional and reactive programming, as well as a few popular design patterns such as observer and iterator that have been used successfully for years. Certainly, RxJS didn't invent these patterns, but it found ways to use them within the context of FP. We'll discuss FP and its role in RxJS further in the next chapter; 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 and propagation

Whether you deal with thousands of key presses, 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 I/O operations such as reading a file, reading a socket, or requesting data from an HTTP server. For instance, Node.js implements readable, writable, and duplex streams for doing just this. In the RP world, we expand the definition of a stream to mean *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:

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

a = 100;
c = ?
```

You can easily predict the value of `c` in this case: 42. The fact that we changed `a` didn't 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:

Creates a stream initialized with the value 20

```
A$ = [20];
B$ = [22];
```

Creates a stream initialized with the value 22

```
C$ = A$.concat(B$).reduce(adder); //-> [42]

A$.push(100);
C$ = ?
```

Concatenates both streams and applies an adder function to get a new container with 42

Pushes a new value into A$

First, we'll explain some of the notation we use here. Streams are containers or wrappers of data very similar to arrays, 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 of the main contributors of RxJS and Finnish.

We created two streams, `A$` and `B$`, with one numerical value inside each. Because they're not primitive objects in JavaScript or have a plus (`+`) overloaded operator, we need to symbolize addition by 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 is represented by `C$`.

> **ARRAY EXTRAS** JavaScript ES5 introduced new array methods, known as the array extras, which enable some level of native support for FP. These include `map`, `reduce`, `filter`, `some`, `every`, and others.

What happens to `C$` if the value `100` is pushed onto `A$`? In an imperative program, nothing will actually happen except that `A$` will have an extra value. But in the world of streams, where there's change propagation, if `A$` receives a new value (a new event), this state is pushed through any streams that it's 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 part changes. Now let's see how RxJS implements this concept.

1.4.2 Introducing the RxJS project

RxJS 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, because 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 website for the Reactive Extensions project (http://reactivex.io/), you'd 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'll demystify this concept 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 breakdown 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 are the data sources, which produce various forms of data to be consumed by an application. And on the right are the data consumers, the entities that subscribe to (or listen for) these events and will do something with data they receive, such as present it on a chart or save it to a file. In the middle is the data pipeline. During this middle step, data that's 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.

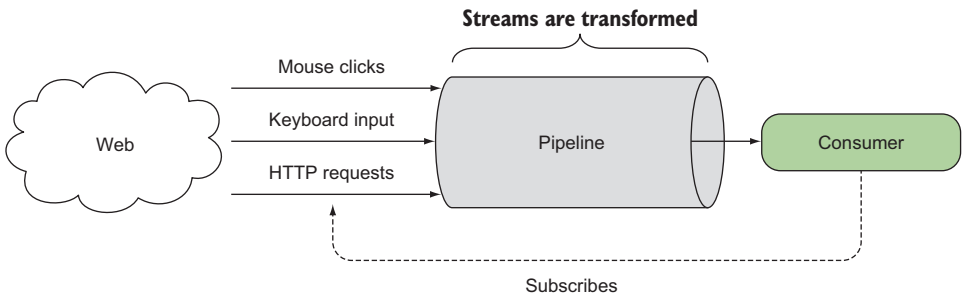

Figure 1.9 A generic data-processing pipeline deals with a constant stream of asynchronous data, moving it from a producer (for example, a user clicking the mouse) to a consumer (code that reacts to the click). The pipeline will process data before it's passed to the consumer for consumption.

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 FP will come into play, as you'll learn about in chapter 2).

DEFINITION 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 change. A stream is an 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 a single integer to bytes of data received from a remote HTTP call. RxJS provides lightweight data types to subscribe to 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 is one of the central topics of this book. At this

[4] You can relate this to the popular pipes and filter design pattern.

point, we haven't talked about any specific RxJS objects; for now, we'll assume that an abstract data type, a container called `Stream`, exists. You 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`, which execute their operations as soon as they're 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 => {
      console.log(val); //-> prints 42
   }
);
```

Using a simple function that will be called with each event in the stream

Observer pattern

Behind RxJS is a fine-tuned 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. But the rudimentary observer pattern has its drawbacks because of memory leaks related to improper disposal of observers. You can learn more about this 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.

a. Gamma, Helm, Johnson, and Vlissides (Addison-Wesley, 1977, Oxford University Press).

Furthermore, you can extend this example to a sequence of numbers

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

or even arrays:

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

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

In this example, 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's where your business logic will be executed, as outlined in figure 1.10.

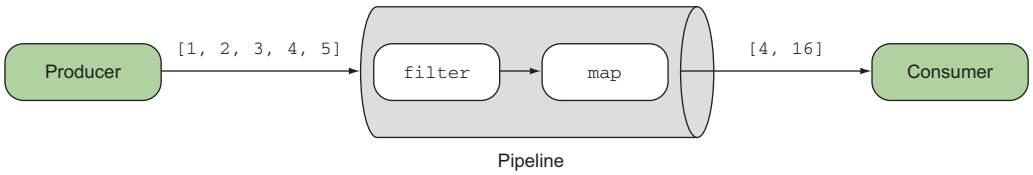

Figure 1.10 A simple producer (an array of numbers) that emits events linearly. These events are submitted through the pipeline and transformed. The final data is then sent to all subscribers to be consumed.

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 dealing 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 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, you can see that streams are analogous to a real-world monthly magazine subscription. Your subscription to the magazine is actually a

collection of magazines that are separated by time; that is, there are 12 magazines annually, but you receive only one every month. Upon receiving a magazine, you usually perform an action on it (read it or throw it away). There are additional cases that you can also consider, such as the time between magazine deliveries being zero, whereby you would receive all the magazines at once, or there might be no magazines (and someone would be getting an angry email). In all these cases, because you perform the action only upon receiving the magazine, you can think of this process as reactive (because you're *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 receive updates only when you happen to be near a magazine stand rather than every time a new magazine becomes available.

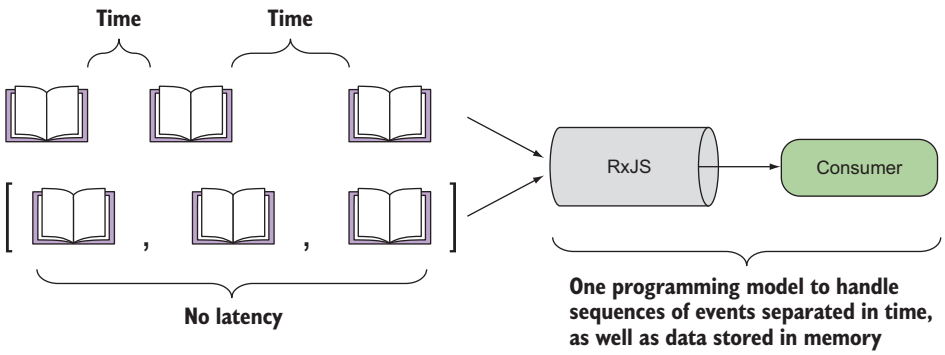

Figure 1.11 Not only does RxJS handle sequential events, but using the same programming model, it can just as easily work with asynchronous events (bound by time). This means that the same level of reasoning applied to linear programs can also be applied to non-linear programs with latency and wait times.

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, or handling real-time services like RSS and Twitter feeds. Following the same examples as before, with RxJS you can consume a stream of time-based asynchronous sequences of events, just as you did with normal synchronous data:

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

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

These types 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, you 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 you're 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 saw some examples of these earlier, and now we'll introduce them more formally:

- Producers
- Consumers
- Data pipeline
- Time

PRODUCERS

Producers are the sources of your data. A stream must always have a producer of data, which will be the starting point for any logic that you'll 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's *able to be observed*.

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

> **TC-39 OBSERVABLE SPEC** The use of observables has proven to be so successful from the previous version of the library (RxJS 4) that a proposal has been made to include it 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, you 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, you now have a stream, and it's at this point that the stream begins to push events; we'll refer to a consumer as an *observer*.

Streams travel only 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 of how to

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

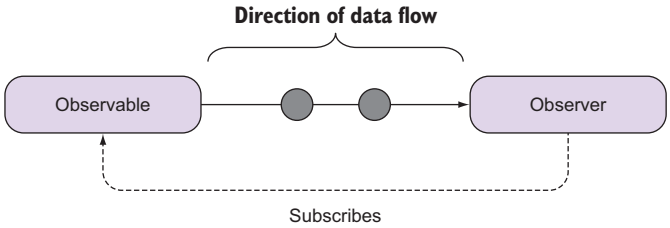

Figure 1.12 Events always move from observables to observers and never the other way around.

think in streams will mean understanding how to think about parts of an application as upstream or downstream to determine the direction in which 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.

For instance, a keyboard event handler would be upstream because it would only produce events, not consume them, whereas 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, you have effectively created a stream. Now what can you do with this data? All of that happens within the data pipeline.

DATA PIPELINE

One advantage of RxJS is that you 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) comes into play. Manipulating data en route means that you 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

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

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

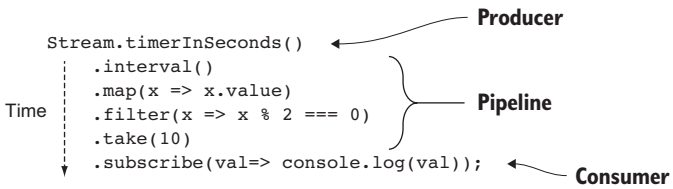

Figure 1.13 Sample code highlighting the different components of a stream

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

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

was drastically improved using `Promises`:

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

Streams can also compose other streams.

Remember that the `Stream` object here is merely an 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. But RxJS doesn't obligate you to use only a single paradigm; it's often the combination of paradigms 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 units of abstraction, and the intricacy of the paradigm comes from the interactions that arise when they interact with one another. In a similar fashion, FP places *behavior* at the center of all things, with functions as the main unit of work. Reactive programming, on the other hand, requires you 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 into the same code base? The beauty behind all this is that you can use all of them together. Many prominent figures in our industry have attested to this. In other words, RxJS doesn't force on you a certain style of development or design pattern to use—it is *unopinionated*. 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. The library provides many operators for such operations baked directly into it. It will even support unusual design patterns such as those you'll see when you use a library like React or Redux (which you'll see in the last chapter).

In practice, you can use OOP to model your domain and use a powerful combination of reactive and FP (a combination known as functional reactive programming) to drive your behavior and events. When it comes to managing events, you'll soon begin to see an important theme 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 by FP. 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, you can also fluently create pipelines of logic that sound very much like spoken sentences like this: "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 free of side effects, with a powerful arsenal of operators and transformations that allow you to elegantly compose your business logic with asynchronous operations. We chose to keep the code abstract for now as we work through some of the new concepts. But we'll quickly ramp up to a comprehensive theoretical and practical understanding of the library, so 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 code 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 they 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 originate 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.

JAVASCRIPT/WEB DEVELOPMENT

RxJS IN ACTION

Daniels • Atencio

On the web, events and messages flow constantly between UI and server components. With RxJS, you can filter, merge, and transform these streams directly, opening the world of data flow programming to browser-based apps. This JavaScript implementation of the ReactiveX spec is perfect for on-the-fly tasks like autocomplete. Its asynchronous communication model makes concurrency much, much easier.

RxJS in Action is your guide to building a reactive web UI using RxJS. You'll begin with an intro to stream-based programming as you explore the power of RxJS through practical examples. With the core concepts in hand, you'll tackle production techniques like error handling, unit testing, and interacting with frameworks like React and Redux. And because RxJS builds on ideas from the world of functional programming, you'll even pick up some key FP concepts along the way.

What's Inside

- Building clean, declarative, fault-tolerant applications
- Transforming and composing streams
- Taming asynchronous processes
- Integrating streams with third-party libraries
- Covers RxJS 5

This book is suitable for readers comfortable with JavaScript and standard web application architectures.

Paul P. Daniels is a professional software engineer with experience in .NET, Java, and JavaScript. **Luis Atencio** is a software engineer working daily with Java, PHP, and JavaScript platforms, and author of Manning's *Functional Programming in JavaScript*.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/rxjs-in-action

"Important information you need to know in order to become an effective reactive programmer."
—From the Foreword by Ben Lesh
Project lead, RxJS 5

"Covers the subject thoroughly and with great accessibility."
—Corinna Cohn, Fusion Alliance

"All you need to really understand streaming!"
—Carlos Corutto, Globant

"Learn to leverage the power of RxJS to build a reactive and resilient foundation for your applications."
—Thomas Peklak, Emakina CEE

ISBN-13: 978-1-61729-341-2
ISBN-10: 1-61729-341-5
54999
9 781617 293412

$49.99 / Can $65.99 [INCLUDING eBOOK]