Learning Reactive Programming with Java 8

Whether you are a Java expert or at a beginner level, you'll benefit from this book, because it will teach you a brand new way of coding and thinking.

The book starts with an explanation of what reactive programming is, why it is so appealing, and how we can integrate it in to Java. It continues by introducing the new Java 8 syntax features, such as lambdas and function references, and some functional programming basics. From this point on, the book focuses on RxJava in depth. It goes through creating Observables, transforming, filtering, and combining them, and concurrency and testing to finish with extending the library itself.

This book is a definite tutorial in RxJava filled with a lot of well-described examples. It explains reactive programming concepts in plain and readable language, without scientific formulas and terms.

Who this book is written for

If you are a Java developer that knows how to write software and would like to learn how to apply your existing skills to reactive programming, this book is for you.

What you will learn from this book

- Discover what reactive programming is and how you can benefit from it
- Get to grips with the new functional features of Java 8 and some functional theory
- Create RxJava Observable instances from virtually any data source
- Transform, filter, and accumulate your data using various RxJava operators
- Combine multiple data sources in one dataset, using custom logic
- Write concurrent programs with ease, scheduling actions on various workers
- Learn about unit testing asynchronous RxJava logic
- Extend RxJava by writing your own operators and factory methods

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'An Introduction to Reactive Programming'
- A synopsis of the book’s content
- More information on Learning Reactive Programming with Java 8
About the Author

Nickolay Tsvetinov is a professional all-round web developer at TransportAPI—Britain’s first comprehensive open platform for transport solutions. During his career as a software developer, he experienced both good and bad and played with most of the popular programming languages—from C and Java to Ruby and JavaScript. For the last 3-4 years, he's been creating and maintaining single-page applications (SPA) and the backend API architectures that serve them. He is a fan of open source software, Rails, Vim, Sinatra, Ember.js, Node.js, and Nintendo. He was an unsuccessful musician and poet, but he is a successful husband and father. His area of interest and expertise includes the declarative/functional and reactive programming that resulted in the creation of ProAct.js (http://proactjs.com), which is a library that augments the JavaScript language and turns it into a reactive language.
Preface

Reactive programming has been around for decades. There has been a few implementations of reactive programming from the time Smalltalk was a young language. However, it has only become popular recently and it is now becoming a trend. Why now you ask? Because it is good for writing fast, real-time applications and current technologies and the Web demand this.

I got involved in it back in 2008, when the team I was part of was developing a multimedia book creator called Sophie 2. It had to be fast and responsive so we created a framework called Prolib, which provided objects with properties which could depend on each other (in other words, we implemented bindings for Swing and much more—transformations, filtering, and so on). It felt natural to wire the model data to the GUI like this.

Of course, this was far away from the functional-like approach that comes with RX. In 2010, Microsoft released RX and, after that, Netflix ported it to Java—RxJava. However, Netflix released RxJava to the open source community and the project became a huge success. Many other languages have their port of RX and many alternatives to it. Now, you can code using reactive programming on your Java backend and wire it to your RxJava's frontend.

This book tries to explain to you what reactive programming is all about and how to use it with RxJava. It has many small examples and it explains concepts and API details in small steps. After reading this book, you will have an idea of RxJava, functional programming, and the reactive paradigm.
What this book covers

Chapter 1, An Introduction to Reactive Programming, will introduce you to the concept of reactive programming and will tell you why you should learn about it. This chapter contains examples that demonstrate how RxJava incorporates the reactive programming concept.

Chapter 2, Using the Functional Constructions of Java 8, will teach you how to use the new lambda constructions of Java 8. It will explain some functional programming concepts and will show you how to use them with RxJava in your reactive programs.

Chapter 3, Creating and Connecting Observables, Observers, and Subjects, will show you the basic building blocks of the RxJava library called the Observables. You will learn the difference between 'hot' and 'cold' Observables and how to subscribe to and unsubscribe from them using a subscription instance.

Chapter 4, Transforming, Filtering, and Accumulating Your Data, will walk you through the basic reactive operators, which you will learn how to use to achieve step-by-step computations. This chapter will give you an idea of how to transform the events the Observables emit, how to filter only the data we need, and how to group, accumulate, and process it.

Chapter 5, Combinators, Conditionals, and Error Handling, will present you with more complex reactive operators, which will allow you to master observable chaining. You will learn about the combining and conditional operators and how the Observables interact with each other. This chapter demonstrates the different approaches to error handling.

Chapter 6, Using Concurrency and Parallelism with Schedulers, will guide you through the process of writing concurrent and parallel programs with RxJava. This will be accomplished by the RxJava Schedulers. The types of Schedulers will be introduced and you will come to know when and why to use each one of them. This chapter will present you with a mechanism that will show you how to avoid and apply backpressure.

Chapter 7, Testing Your RxJava Application, will show you how to unit test your RxJava applications.

Chapter 8, Resource Management and Extending RxJava, will teach you how to manage the resources used as data sources by your RxJava applications. We will write our own Observable operators here.
An Introduction to Reactive Programming

Nowadays, the term **reactive programming** is trending. Libraries and frameworks in various programming languages are emerging. Blog posts, articles and presentations about reactive programming are being created. Big companies, such as Facebook, SoundCloud, Microsoft, and Netflix, are supporting and using this concept. So we, as programmers, are starting to wonder about it. Why are people so excited about reactive programming? What does it mean to be reactive? Would it be helpful in our projects? Should we learn how to use it?

Meanwhile, Java is popular with its multi-threading, speed, reliability, and good portability. It is used for building a wide variety of applications, from search engines, through databases to complex web applications running on server clusters. But Java has bad reputation too—it is very hard to write both concurrent and simple applications using only the built-in tools, and programming in Java requires writing a lot of boilerplate code. Also, if you need to be asynchronous (using futures, for example), you can easily get into "callback hell", which actually holds true for all programming languages.

In other words, Java is powerful and you can create great applications with it, but it won't be easy. The good news is that there is a way to change that, using the reactive style of programming.

This book will present **RxJava** ([https://github.com/ReactiveX/RxJava](https://github.com/ReactiveX/RxJava)), an open source Java implementation of the reactive programming paradigm. Writing code using RxJava requires a different kind of thinking, but it will give you the power to create complex logic using simple pieces of well-structured code.
In this chapter, we will cover:

- What reactive programming is
- Reasons to learn and use this style of programming
- Setting up RxJava and comparing it with familiar patterns and structures
- A simple example with RxJava

**What is reactive programming?**

Reactive programming is a paradigm that revolves around the propagation of change. In other words, if a program propagates all the changes that modify its data to all the interested parties (users, other programs, components, and subparts), then this program can be called **reactive**.

A simple example of this is Microsoft Excel. If you set a number in cell A1 and another number in cell 'B1', and set cell 'C1' to \textit{SUM(A1, B1)}; whenever 'A1' or 'B1' changes, 'C1' will be updated to be their sum.

Let's call this **the reactive sum**.

What is the difference between assigning a simple variable \( c \) to be equal to the sum of the \( a \) and \( b \) variables and the reactive sum approach?

In a normal Java program, when we change 'a' or 'b', we will have to update 'c' ourselves. In other words, the change in the flow of the data represented by 'a' and 'b', is not propagated to 'c'. Here is this illustrated through source code:

```java
int a = 4;
int b = 5;
int c = a + b;
System.out.println(c); // 9

a = 6;
System.out.println(c);
// 9 again, but if 'c' was tracking the changes of 'a' and 'b',
// it would've been 6 + 5 = 11
```

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at \url{http://www.packtpub.com}. If you purchased this book elsewhere, you can visit \url{http://www.packtpub.com/support} and register to have the files e-mailed directly to you.
This is a very simple explanation of what "being reactive" means. Of course, there are various implementations of this idea and there are various problems that these implementations must solve.

**Why should we be reactive?**

The easiest way for us to answer this question is to think about the requirements we have while building applications these days.

While 10-15 years ago it was normal for websites to go through maintenance or to have a slow response time, today everything should be online 24/7 and should respond with lightning speed; if it's slow or down, users would prefer an alternative service. Today slow means unusable or broken. We are working with greater volumes of data that we need to serve and process fast.

HTTP failures weren't something rare in the recent past, but now, we have to be fault-tolerant and give our users readable and reasonable message updates.

In the past, we wrote simple desktop applications, but today we write web applications, which should be fast and responsive. In most cases, these applications communicate with a large number of remote services.

These are the new requirements we have to fulfill if we want our software to be competitive. So in other words we have to be:

- **Modular/dynamic:** This way, we will be able to have 24/7 systems, because modules can go offline and come online without breaking or halting the entire system. Additionally, this helps us better structure our applications as they grow larger and manage their code base.
- **Scalable:** This way, we are going to be able to handle a huge amount of data or large numbers of user requests.
- **Fault-tolerant:** This way, the system will appear stable to its users.
- **Responsive:** This means fast and available.

Let's think about how to accomplish this:

- **We can become modular if our system is event-driven.** We can divide the system into multiple micro-services/components/modules that are going to communicate with each other using notifications. This way, we are going to react to the data flow of the system, represented by notifications.
- **To be scalable means to react to the ever-growing data, to react to load without falling apart.**
Reacting to failures/errors will make the system more fault-tolerant.

To be responsive means reacting to user activity in a timely manner.

If the application is event-driven, it can be decoupled into multiple self-contained components. This helps us become more scalable, because we can always add new components or remove old ones without stopping or breaking the system. If errors and failures are passed to the right component, which can handle them as notifications, the application can become more fault-tolerant or resilient. So if we build our system to be event-driven, we can more easily achieve scalability and failure tolerance, and a scalable, decoupled, and error-proof application is fast and responsive to users.

The Reactive Manifesto (http://www.reactivemanifesto.org/) is a document defining the four reactive principles that we mentioned previously. Each reactive system should be message-driven (event-driven). That way, it can become loosely coupled and therefore scalable and resilient (fault-tolerant), which means it is reliable and responsive (see the preceding diagram).

Note that the Reactive Manifesto describes a reactive system and is not the same as our definition of reactive programming. You can build a message-driven, resilient, scalable, and responsive application without using a reactive library or language.

Changes in the application data can be modeled with notifications, which can be propagated to the right handlers. So, writing applications using reactive programming is the easiest way to comply with the Manifesto.
Introducing RxJava
To write reactive programs, we need a library or a specific programming language, because building something like that ourselves is quite a difficult task. Java is not really a reactive programming language (it provides some tools like the java.util.Observable class, but they are quite limited). It is a statically typed, object-oriented language, and we write a lot of boilerplate code to accomplish simple things (POJOs, for example). But there are reactive libraries in Java that we can use. In this book, we will be using RxJava (developed by people in the Java open source community, guided by Netflix).

Downloading and setting up RxJava
You can download and build RxJava from Github (https://github.com/ReactiveX/RxJava). It requires zero dependencies and supports Java 8 lambdas. The documentation provided by its Javadoc and the GitHub wiki pages is well structured and some of the best out there. Here is how to check out the project and run the build:

```
$ git clone git@github.com:ReactiveX/RxJava.git
$ cd RxJava/
$ ./gradlew build
```

Of course, you can also download the prebuilt JAR. For this book, we'll be using version 1.0.8.

If you use Maven, you can add RxJava as a dependency to your pom.xml file:

```
<dependency>
  <groupId>io.reactivex</groupId>
  <artifactId>rxjava</artifactId>
  <version>1.0.8</version>
</dependency>
```

Alternatively, for Apache Ivy, put this snippet in your Ivy file's dependencies:

```
<dependency org="io.reactivex" name="rxjava" rev="1.0.8" />
```

If you use Gradle instead, update your build.gradle file's dependencies as follows:

```
dependencies {
  ...
  compile 'io.reactivex:rxjava:1.0.8'
  ...
}
```
An Introduction to Reactive Programming

The code examples and programs accompanying this book can be built and tested with Gradle. It can be downloaded from this Github repository: https://github.com/meddle0x53/learning-rxjava.

Now, let's take a peek at what RxJava is all about. We are going to begin with something well known, and gradually get into the library's secrets.

Comparing the iterator pattern and the RxJava Observable

As a Java programmer, it is highly possible that you've heard or used the Iterator pattern. The idea is simple: an Iterator instance is used to traverse through a container (collection/data source/generator), pulling the container's elements one by one when they are required, until it reaches the container's end. Here is a little example of how it is used in Java:

```java
List<String> list = Arrays.asList("One", "Two", "Three", "Four", "Five"); // (1)

Iterator<String> iterator = list.iterator(); // (2)

while(iterator.hasNext()) { // 3
    // Prints elements (4)
    System.out.println(iterator.next());
}
```

Every java.util.Collection object is an Iterable instance which means that it has the method iterator(). This method creates an Iterator instance, which has as its source the collection. Let's look at what the preceding code does:

1. We create a new List instance containing five strings.
2. We create an Iterator instance from this List instance, using the iterator() method.
3. The Iterator interface has two important methods: hasNext() and next(). The hasNext() method is used to check whether the Iterator instance has more elements for traversing. Here, we haven't begun going through the elements, so it will return True. When we go through the five strings, it will return False and the program will proceed after the while loop.
4. The first five times, when we call the `next()` method on the `Iterator` instance, it will return the elements in the order they were inserted in the collection. So the strings will be printed.

In this example, our program consumes the items from the `List` instance using the `Iterator` instance. It pulls the data (here, represented by strings) and the current thread blocks until the requested data is ready and received. So, for example, if the `Iterator` instance was firing a request to a web server on every `next()` method call, the main thread of our program would be blocked while waiting for each of the responses to arrive.

RxJava's building blocks are the observables. The `Observable` class (note that this is not the `java.util.Observable` class that comes with the JDK) is the mathematical dual of the `Iterator` class, which basically means that they are like the two sides of the same coin. It has an underlying collection or computation that produces values that can be consumed by a consumer. But the difference is that the consumer doesn't "pull" these values from the producer like in the `Iterator` pattern. It is exactly the opposite; the producer 'pushes' the values as notifications to the consumer.

Here is an example of the same program but written using an `Observable` instance:

```java
List<String> list = Arrays.asList("One", "Two", "Three", "Four", "Five"); // (1)
Observable<String> observable = Observable.from(list); // (2)

observable.subscribe(new Action1<String>() { // (3)
    @Override
    public void call(String element) {
        System.out.println(element); // Prints the element (4)
    }
});
```

Here is what is happening in the code:

1. We create the list of strings in the same way as in the previous example.
2. Then, we create an `Observable` instance from the list, using the `from(Iterable<? extends T> iterable)` method. This method is used to create instances of `Observable` that send all the values synchronously from an `Iterator` instance (the list in our case) one by one to their subscribers (consumers). We'll look at how the values are sent to the subscribers one by one in Chapter 3, Creating and Connecting Observables, Observers, and Subjects.
3. Here, we can subscribe to the Observable instance. By subscribing, we tell RxJava that we are interested in this Observable instance and want to receive notifications from it. We subscribe using an anonymous class implementing the Action1 interface, by defining a single method — call(T). This method will be called by the Observable instance every time it has a value, ready to be pushed. Always creating new Action1 instances may seem too verbose, but Java 8 solves this verbosity. We'll learn more about that in Chapter 2, Using the Functional Constructions of Java 8.

4. So, every string from the source list will be pushed through to the call() method, and it will be printed.

Instances of the RxJava Observable class behave somewhat like asynchronous iterators, which notify that there is a next value their subscribers/consumers by themselves. In fact, the Observable class adds to the classic Observer pattern (implemented in Java — see java.util.Observable, see Design Patterns: Elements of Reusable Object-Oriented Software by the Gang Of Four) two things available in the Iterable type.

- The ability to signal the consumer that there is no more data available. Instead of calling the hasNext() method, we can attach a subscriber to listen for a 'OnCompleted' notification.
- The ability to signal the subscriber that an error has occurred. Instead of try-catching an error, we can attach an error listener to the Observable instance.

These listeners can be attached using the subscribe(Action1<? super T>, Action1<Throwable>, Action0) method. Let's expand the Observable instance example by adding error and completed listeners:

```java
List<String> list = Arrays.asList("One", "Two", "Three", "Four", "Five");

Observable<String> observable = Observable.from(list);
observable.subscribe(new Action1<String>() {
    @Override
    public void call(String element) {
        System.out.println(element);
    }
},
new Action1<Throwable>() {
    @Override
    public void call(Throwable t) {
        System.err.println(t); // (1)
    }
},
```
new Action0() {
    @Override
    public void call() {
        System.out.println("We've finished!"); // (2)
    }
});

The new things here are:

1. If there is an error while processing the elements, the Observable instance will send this error through the call(Throwable) method of this listener. This is analogous to the try-catch block in the Iterator instance example.

2. When everything finishes, this call() method will be invoked by the Observable instance. This is analogous to using the hasNext() method in order to see if the traversal over the Iterable instance has finished and printing "We've finished!".

This example is available at GitHub and can be viewed/downloaded at https://github.com/meddle0x53/learning-rxjava/blob/master/src/main/java/com/packtpub/reactive/chapter01/ObservableVSIterator.java.

We saw how we can use the Observable instances and that they are not so different from something familiar to us—the Iterator instance. These Observable instances can be used for building asynchronous streams and pushing data updates to their subscribers (they can have multiple subscribers). This is an implementation of the reactive programming paradigm. The data is being propagated to all the interested parties—the subscribers.

Coding using such streams is a more functional-like implementation of Reactive Programming. Of course, there are formal definitions and complex terms for it, but this is the simplest explanation.

Subscribing to events should be familiar; for example, clicking on a button in a GUI application fires an event which is propagated to the subscribers—handlers. But, using RxJava, we can create data streams from anything—file input, sockets, responses, variables, caches, user inputs, and so on. On top of that, consumers can be notified that the stream is closed, or that there has been an error. So, by using these streams, our applications can react to failure.
An Introduction to Reactive Programming

To summarize, a stream is a sequence of ongoing messages/events, ordered as they are processed in real time. It can be looked at as a value that is changing through time, and these changes can be observed by subscribers (consumers), dependent on it. So, going back to the example from Excel, we have effectively replaced the traditional variables with "reactive variables" or RxJava's `Observable` instances.

**Implementing the reactive sum**

Now that we are familiar with the `Observable` class and the idea of how to use it to code in a reactive way, we are ready to implement the reactive sum, mentioned at the beginning of this chapter.

Let's look at the requirements our program must fulfill:

- It will be an application that runs in the terminal.
- Once started, it will run until the user enters `exit`.
- If the user enters `a:<number>`, the `a` collector will be updated to the `<number>`.
- If the user enters `b:<number>`, the `b` collector will be updated to the `<number>`.
- If the user enters anything else, it will be skipped.
- When both the `a` and `b` collectors have initial values, their sum will automatically be computed and printed on the standard output in the format `a + b = <sum>`. On every change in `a` or `b`, the sum will be updated and printed.

The source code contains features that we will discuss in detail in the next four chapters.

The first piece of code represents the main body of the program:

```java
ConnectableObservable<String> input = from(System.in); // (1)

Observable<Double> a = varStream("a", input); (2)
Observable<Double> b = varStream("b", input);

ReactiveSum sum = new ReactiveSum(a, b); (3)

input.connect(); (4)
```
There are a lot of new things happening here:

1. The first thing we must do is to create an Observable instance, representing the standard input stream (System.in). So, we use the from(InputStream) method (implementation will be presented in the next code snippet) to create a ConnectableObservable variable from the System.in. The ConnectableObservable variable is an Observable instance and starts emitting events coming from its source only after its connect() method is called. Read more on it in Chapter 3, Creating and Connecting Observables, Observers, and Subjects.

2. We create two Observable instances representing the a and b values, using the varStream(String, Observable) method, which we are going to examine later. The source stream for these values is the input stream.

3. We create a ReactiveSum instance, dependent on the a and b values.

4. And now, we can start listening to the input stream.

This code is responsible for building dependencies in the program and starting it off. The a and b values are dependent on the user input and their sum is dependent on them.

Now let's look at the implementation of the from(InputStream) method, which creates an Observable instance with the java.io.InputStream source:

```java
static ConnectableObservable<String> from(final InputStream stream) {
    return from(new BufferedReader(new InputStreamReader(stream))); // (1)
}

static ConnectableObservable<String> from(final BufferedReader reader) {
    return Observable.create(new OnSubscribe<String>() { // (2)
        @Override
        public void call(Subscriber<? super String> subscriber) {
            if (subscriber.isUnsubscribed()) { // (3)
                return;
            }
            try {
                String line;
                while (!subscriber.isUnsubscribed() &&
                        (line = reader.readLine()) != null) { // (4)
                    if (line == null || line.equals("exit")) { // (5)
                        break;
                    }
                }
```

```
An Introduction to Reactive Programming

```java
subscriber.onNext(line); // (6)
}
}
catch (IOException e) { // (7)
subscriber.onError(e);
}
if (!subscriber.isUnsubscribed()) // (8)
subscriber.onCompleted();
}
}).publish(); // (9)
```

This is one complex piece of code, so let’s look at it step-by-step:

1. This method implementation converts its InputStream parameter to the BufferedReader object and to calls the from(BufferedReader) method. We are doing that because we are going to use strings as data, and working with the Reader instance is easier.

2. So the actual implementation is in the second method. It returns an Observable instance, created using the Observable.create(OnSubscribe) method. This method is the one we are going to use the most in this book. It is used to create Observable instances with custom behavior. The rx.Observable.OnSubscribe interface passed to it has one method, call(Subscriber). This method is used to implement the behavior of the Observable instance because the Subscriber instance passed to it can be used to emit messages to the Observable instance's subscriber. A subscriber is the client of an Observable instance, which consumes its notifications. Read more about that in Chapter 3, Creating and Connecting Observables, Observers, and Subjects.

3. If the subscriber has already unsubscribed from this Observable instance, nothing should be done.

4. The main logic is to listen for user input, while the subscriber is subscribed. Every line the user enters in the terminal is treated as a message. This is the main loop of the program.

5. If the user enters the word exit and hits Enter, the main loop stops.

6. Otherwise, the message the user entered is passed as a notification to the subscriber of the Observable instance, using the onNext(T) method. This way, we pass everything to the interested parties. It's their job to filter out and transform the raw messages.

7. If there is an IO error, the subscribers are notified with an onError notification through the onError(Throw)able method.
8. If the program reaches here (through breaking out of the main loop) and the subscriber is still subscribed to the Observable instance, an OnCompleted notification is sent to the subscribers using the onCompleted() method.

9. With the publish() method, we turn the new Observable instance into ConnectableObservable instance. We have to do this because, otherwise, for every subscription to this Observable instance, our logic will be executed from the beginning. In our case, we want to execute it only once and all the subscribers to receive the same notifications; this is achievable with the use of a ConnectableObservable instance. Read more about that in Chapter 3, Creating and Connecting Observables, Observers, and Subjects.

This illustrates a simplified way to turn Java's IO streams into Observable instances. Of course, with this main loop, the main thread of the program will block waiting for user input. This can be prevented using the right Scheduler instances to move the logic to another thread. We'll revisit this topic in Chapter 6, Using Concurrency and Parallelism with Schedulers.

Now, every line the user types into the terminal is propagated as a notification by the ConnectableObservable instance created by this method. The time has come to look at how we connect our value Observable instances, representing the collectors of the sum, to this input Observable instance. Here is the implementation of the varStream(String, Observable) method, which takes a name of a value and source Observable instance and returns an Observable instance representing this value:

```java
public static Observable<Double> varStream(final String varName, Observable<String> input) {
    final Pattern pattern = Pattern.compile("^s*" + varName + "s*[:|=]\s*(-?\d+\.?\d*)$\"); // (1)
    return input.map(new Func1<String, Matcher>() {
        public Matcher call(String str) {
            return pattern.matcher(str); // (2)
        }
    }).filter(new Func1<Matcher, Boolean>() {
        public Boolean call(Matcher matcher) {
            return matcher.matches() && matcher.group(1) != null; // (3)
        }
    }).map(new Func1<Matcher, Double>() {
        public Double call(Matcher matcher) {
            return Double.parseDouble(matcher.group(1)); // (4)
        }
    });
}
```
The `map()` and `filter()` methods called on the `Observable` instance here are part of the fluent API provided by RxJava. They can be called on an `Observable` instance, creating a new `Observable` instance that depends on these methods and that transforms or filters the incoming data. Using these methods the right way, you can express complex logic in a series of steps leading to your objective. Read more about this in Chapter 4, Transforming, Filtering, and Accumulating Your Data. Let's analyze the code:

1. Our variables are interested only in messages in the format `<var_name>: <value>` or `<var_name> = <value>`, so we are going to use this regular expression to filter and process only these kinds of messages. Remember that our input `Observable` instance sends each line the user writes; it is our job to handle it the right way.
2. Using the messages we receive from the input, we create a `Matcher` instance using the preceding regular expression as a pattern.
3. We pass through only data that matches the regular expression. Everything else is discarded.
4. Here, the value to set is extracted as a `Double` number value.

This is how the values `a` and `b` are represented by streams of double values, changing in time. Now we can implement their sum. We implemented it as a class that implements the `Observer` interface, because I wanted to show you another way of subscribing to `Observable` instances—using the `Observer` interface. Here is the code:

```java
public static final class ReactiveSum implements Observer<Double>
{
    // (1)
    private double sum;

    public ReactiveSum(Observable<Double> a, Observable<Double> b) {
        this.sum = 0;
        Observable.combineLatest(a, b, new Func2<Double, Double, Double>() {
            // (5)
            public Double call(Double a, Double b) {
                return a + b;
            }
        }).subscribe(this); // (6)
    }

    public void onCompleted() {
        System.out.println("Exiting last sum was : " + this.sum); // (4)
    }
}
```
public void onError(Throwable e) {
    System.err.println("Got an error!"); // (3)
    e.printStackTrace();
}

public void onNext(Double sum) {
    this.sum = sum;
    System.out.println("update : a + b = " + sum); // (2)
}

This is the implementation of the actual sum, dependent on the two Observable instances representing its collectors:

1. It is an Observer interface. The Observer instance can be passed to the Observable instance's subscribe(Observer) method and defines three methods that are named after the three types of notification: onNext(T), onError(Throwable), and onCompleted. Read more about this interface in Chapter 3, Creating and Connecting Observables, Observers, and Subjects.

2. In our onNext(Double) method implementation, we set the sum to the incoming value and print an update to the standard output.

3. If we get an error, we just print it.

4. When everything is done, we greet the user with the final sum.

5. We implement the sum with the combineLatest(Observable, Observable, Func2) method. This method creates a new Observable instance. The new Observable instance is updated when any of the two Observable instances, passed to combineLatest receives an update. The value emitted through the new Observable instance is computed by the third parameter—a function that has access to the latest values of the two source sequences. In our case, we sum up the values. There will be no notification until both of the Observable instances passed to the method emit at least one value. So, we will have the sum only when both a and b have notifications. Read more about this method and other combiners in Chapter 5, Combinators, Conditionals, and Error Handling.

6. We subscribe our Observer instance to the combined Observable instance.

Here is sample of what the output of this example would look like:

Reactive Sum. Type 'a: <number>' and 'b: <number>' to try it.
a:4
b:5
update : a + b = 9.0
So this is it! We have implemented our reactive sum using streams of data.

Summary

In this chapter, we went through the reactive principles and the reasons we should learn and use them. It is not so hard to build a reactive application; it just requires structuring the program in little declarative steps. With RxJava, this can be accomplished by building multiple asynchronous streams connected the right way, transforming the data all the way through its consumer.

The two examples presented in this chapter may look a bit complex and confusing at first glance, but in reality, they are pretty simple. There are a lot of new things in them, but everything will be explained in detail in the following chapters.

If you want to read more about reactive programming, take a look at Reactive Programming in the Netflix API with RxJava, a fine article on the topic, available at http://techblog.netflix.com/2013/02/rxjava-netflix-api.html. Another fine post introducing the concept can be found here: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754.

And these are slides about reactive programming and RX by Ben Christensen, one of the creators of RxJava: https://speakerdeck.com/benjchristensen/reactive-programming-with-rx-at-qconsf-2014.

In the next chapter, we are going to talk about some of the concepts of functional programming and their implementation in Java 8. This will give us the basic ideas needed in the rest of the chapters and will help us get rid of Java verbosity when writing reactive programs.
Where to buy this book

You can buy Learning Reactive Programming with Java 8 from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.