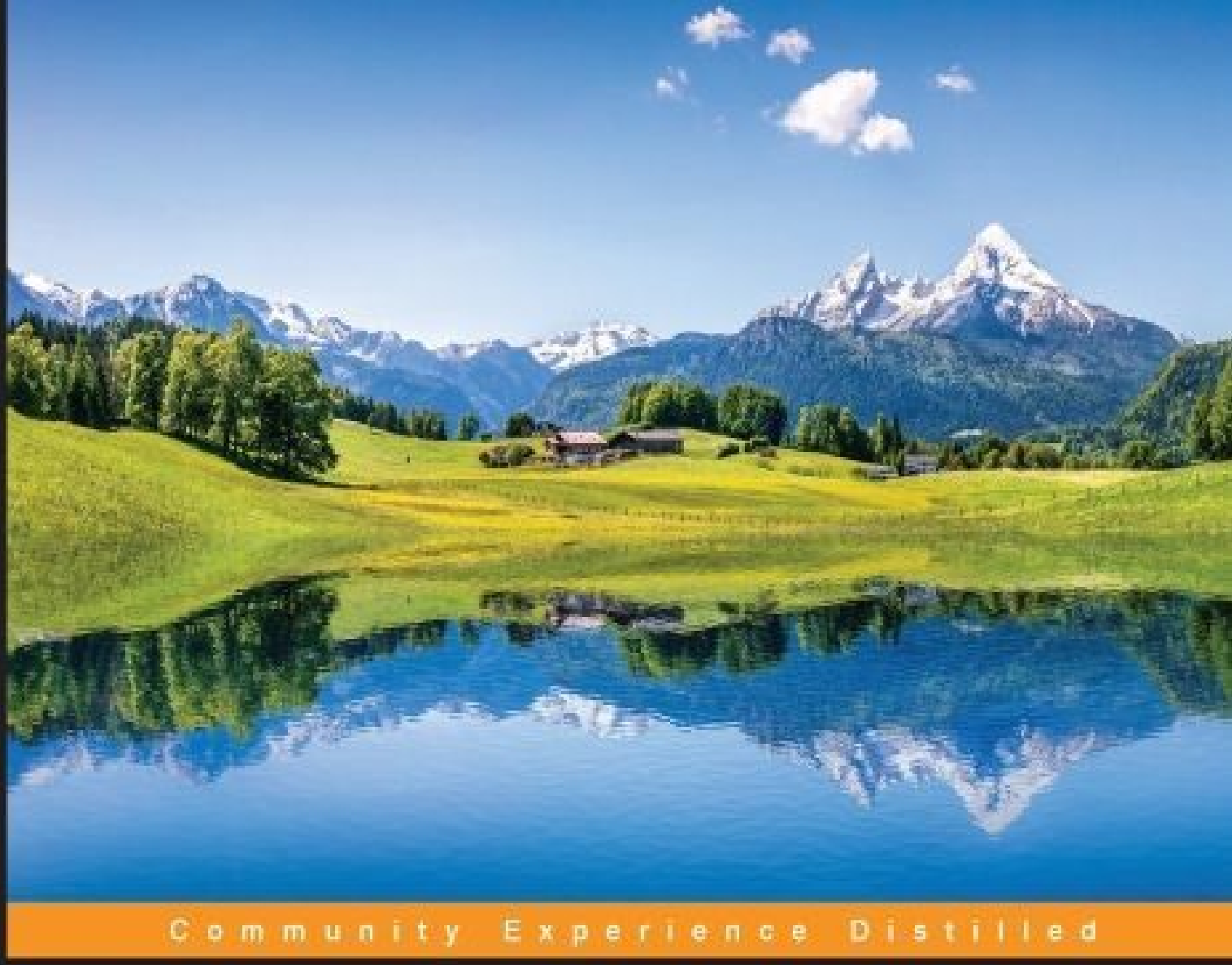


Community Experience Distilled

RxJava Essentials

Learn reactive programming to create awesome Android and Java apps

Ivan Morgillo

[PACKT] open source
PUBLISHING
community experience distilled

RxJava Essentials

Table of Contents

RxJava Essentials

RxJava Essentials

First published: May 2015

Production reference: 1220515

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78439-910-8

www.packtpub.com

Credits

Author

Ivan Morgillo

Reviewer

David Moten

Commissioning Editor

Ashwin Nair

Acquisition Editor

Harsha Bharwani

Content Development Editor

Prachi Bisht

Technical Editor

Mrunal M. Chavan

Copy Editor

Stuti Srivastava

Project Coordinator

Sanjeet Rao

Proofreaders

Stephen Copestake

Safis Editing

Indexer

Priya Sane

Production Coordinator

Shantanu N. Zagade

Cover Work

Shantanu N. Zagade

About the Author

Ivan Morgillo was just a kid with a C64 and some basic skills before becoming an engineer a few years later. After working as an embedded systems consultant for Italtel and Telecom Italia, he moved to Android. He worked as a consultant for Deltatre, Mondia Media, and Clevertech.

He currently runs a mobile and embedded applications development company, Alter Ego Solutions, contributing to open source projects and still working on his Android projects over the weekend.

I want to thank Francesca, my beloved, for her constant support and infinite love; Sasa Sekulic, my company cofounder, for his perpetual trolling; and Fabrizio Chignoli, who changed my life, teaching me what reactive programming really is. This book is dedicated to my family and my friends. I couldn't have been where I am without their love.

About the Reviewer

David Moten has been programming in Java since 2001, largely with the Australian government. He learned Scala in 2007, which made him strongly appreciate the functional style, and he was delighted to discover RxJava when he went looking for functional libraries for Java in early 2014. He loves a bit of math when it turns up, and he is enjoying learning the tricks of the concurrent programming trade by contributing to the RxJava project on GitHub.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Preface

In a world where there is a smartphone in every pocket, designing and building applications that can run smoothly and provide a user experience that users will enjoy is the only way to go. The reactive programming style with RxJava will help you beat Android Platform limitations to create astonishing Android Apps.

This book will be a practical journey, from the basics of reactive programming and Observer pattern concepts, to the main feature of RxJava, which will be accompanied by practical code examples and a real-world app.

I'll show you how to create an Observable from "scratch", from a list, or from a function that we already have in our codebase. You will learn how to filter an Observable sequence to create a new sequence, containing only the values we want; you will also learn how to apply a function to an Observable and how to concatenate, merge, or zip Observables. I'll show you how to enjoy RxAndroid Schedulers to overcome the threading and concurrency hell in Android.

The book will wind up with a practical example of RxJava combined with Retrofit to easily communicate with a REST API.

What this book covers

Chapter 1, *RX – from .NET to RxJava*, takes initial steps into the reactive world. We will compare the reactive approach with the classic approach, and will explore the similarities and differences between them.

Chapter 2, *Why Observables?*, gives an overview of the Observer pattern, how it's implemented and extended by RxJava, what an Observable is, and how Observables relate to Iterables.

Chapter 3, *Hello Reactive World*, uses what we have learned so far to create our first reactive Android app.

Chapter 4, *Filtering Observables*, dives into the essence of an Observable sequence: filtering. We will also learn how to select only the values we want from an emitting Observable, how to obtain a finite number of values, how to handle overflow scenarios, and a few more useful tricks.

Chapter 5, *Transforming Observables*, shows how to transform Observable sequences to create sequences that can fit our needs.

Chapter 6, *Combining Observables*, digs into combining functions, and we are going to learn how to work with multiple Observables simultaneously when we create the Observable we want.

Chapter 7, *Schedulers – Defeating the Android MainThread Issue*, shows you how to work with multithreading and concurrent programming using RxJava Schedulers. We will create network operations, memory accesses, and time-consuming tasks in a reactive way.

Chapter 8, *REST in Peace – RxJava and Retrofit*, teaches you how Retrofit by Square can be used with RxJava to create an REST client efficiently and effectively.

What you need for this book

To run the examples in this book, you will need a standard Android development environment:

- Android Studio or IntelliJ IDEA
- The Android SDK
- The Java JDK

As you are approaching RxJava as a pure Java developer, you will obviously need your preferred Java editor and a standard Java JDK environment. Some of the figures used in this book are taken from http://rxmarbles.com/ and http://reactivex.io/.

Who this book is for

If you are an experienced Java developer, reactive programming will give you a new way to approach scalability and concurrency in your backend systems, without forcing you to switch to change the programming languages. This book will help you learn the core aspects of RxJava and will also help you overcome the limitations of the Android platform to create event-driven, reactive, and smooth Android applications.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "As you can see, `zip()` has three parameters: the two Observables and `Func2`, as expected."

A block of code is set as follows:

```
public Observable<List<User>> getMostPopularSOusers(int
howmany) {
        return mStackExchangeService
              .getMostPopularSOusers(howmany)
                .map(UsersResponse::getUsers)
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread());
    }
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
public Observable<List<User>> getMostPopularSOusers(int
howmany) {
        return mStackExchangeService
              .getMostPopularSOusers(howmany)
                .map(UsersResponse::getUsers)
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread());
    }
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "We will just need a fancy progress bar and a **DOWNLOAD** button."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <feedback@packtpub.com>, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files emailed directly to you.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/9108OS_GraphicsBu

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

Chapter 1. RX – from .NET to RxJava

Reactive programming is a programming paradigm based on the concept of an asynchronous *data flow*. A data flow is like a river: it can be observed, filtered, manipulated, or merged with a second flow to create a new flow for a new consumer.

A key concept of reactive programming is events. Events can be awaited, can trigger procedures, and can trigger other events. Events are the only proper way to map our real world to our software: we open a window if it's too hot inside. In the same way, we want the `Total` cell to update when we change some values in our spreadsheet (the propagation of the changes) or our robot to turn around the moment it reaches the wall (reaction to events).

Nowadays, one of the most common scenarios for reactive programming is UIs: we have mobile apps that have to react to network calls, user touch input, and system alerts. In this world, software has to be event-driven and reactive because real life is event-driven.

Microsoft Reactive Extensions

Functional reactive programming is an idea from the late 90s that inspired Erik Meijer, a computer scientist at Microsoft, to design and develop the Microsoft Rx library.

Rx is a *reactive extension* for Microsoft .NET. Rx provides an easy way to create asynchronous, event-driven programs using Observable sequences. A developer can model an asynchronous data stream using Observables, query the Observables using LINQ syntax, and easily manage concurrency with Schedulers.

Rx makes well-known concepts, such as the **push approach**, easy to implement and consume. In a reactive world, we can't just wait for a function result, a network call, or a database query to return and pretend that the user won't notice or even complain about it. Every moment we wait for something, we lose the opportunity to do other things in parallel, provide a better user experience, and free our software from the chains of *sequential, blocking* programming.

.NET Observable relates to .NET Enumerable according to the following table:

.NET Observable	Single return value	Multiple return values
Pull/Synchronous/Interactive	T	`IEnumerable<T>`
Push/Asynchronous/Reactive	`Task<T>`	`IObservable<T>`

The push approach reverses the problem: instead of asking for a result and waiting, the developer simply asks for a result and gets notified when the result is available. The developer provides a clear sequence of reactions to the events that are going to happen. For every event, the developer provides a reaction; for example, the user is asked to log in and submit a form with his username and password. The application executes the network call for the

login and states what is going to happen:

- Show a success message and store the user's profile
- Show an error message

As you can see with the push approach, the developer doesn't wait for the result. He will be notified when the result arrives. In the meantime, he can do whatever he wants:

- Show a progress dialog
- Store the username and password for future logins
- Preload something he knows would take some time the moment the login succeeds

Landing in the Java world – Netflix RxJava

In 2012, at Netflix, they started to realize that their architecture was having a hard time to properly adapt to the amount of users they had. They decided to redesign their architecture to reduce the number of REST calls. Instead of having dozens of REST calls and letting the client process the data as needed, they decided to create a single optimized REST call based on the clients' needs.

To achieve this goal, they decided to go *reactive*, and they started to port .NET Rx to the JVM. They didn't want to target just Java; they decided to target the JVM instead, having the possibility to provide a new tool for every JVM language on the market: Java, Closure, Groovy, Scala, and so on.

With a post on the Netflix tech blog in February 2013, Ben Christensen and Jafar Husain showed RxJava to the world for the first time.

The key concepts were:

- Easy concurrency to better use their server's power
- Easy conditional asynchronous execution
- A proper way to escape the callback hell
- A reactive approach

As for .NET, RxJava Observable is the *push* equivalent of Iterable, which is *pull*. The pull approach is a block-and-wait approach: the consumer pulls values from the source, blocking the thread until the producer provides new values.

The push approach works on subscription and reaction: the consumer subscribes to new values' emissions; the producer pushes these new values when they are available, and notifies the consumer. At this point, the consumer, well, consumes them. The push approach is clearly more flexible, because from a logical and practical point of view, the developer can simply ignore if

the data he needs comes synchronously or asynchronously; his code will still work.

What's different in RxJava

From a pure Java point of view, the RxJava Observable class extends the classic Gang of Four Observer pattern concept.

It adds three missing abilities:

- The producer can now signal that there is no more data available: the `onCompleted()` event
- The producer can now signal that an error occurred: the `onError()` event
- RxJava Observables can be composed instead of nested, saving the developer from the callback hell

Observables and Iterables share a similar API; lots of operations that we can perform on Iterable can be performed on Observable too. Of course, due to the "flow" nature of Observables, we don't have equivalents for methods such as `Iterable.remove()`.

Pattern	Single return value	Multiple return values
Synchronous	`T getData()`	`Iterable<T> getData()`
Asynchronous	`Future<T> getData()`	`Observable<T> getData()`

From a semantic point of view, RxJava is .NET Rx. From a syntactical point of view, Netflix takes care of porting every Rx method, keeping in mind Java code conventions and basic patterns.

Summary

In this chapter, we took the initial steps into the reactive world. We learned how Rx was born, from Microsoft .NET to Netflix RxJava. We learned the similarities and differences of the reactive approach compared with the classic approach.

In the next chapter, we will learn what Observables are, and how we can create them to bring reactive programming into our daily coding life.

Chapter 2. Why Observables?

In object-oriented architectures, the developer works hard to create a set of decoupled entities. In this way, entities can be tested, reused, and maintained without interfering with the whole system. Designing this kind of system brings a tricky *side effect*: maintaining consistency between related objects.

The first example of a pattern created to solve this issue was in the Smalltalk Model-View-Controller architecture. The user interface framework provided a way to keep UI elements separated from the actual object containing the data, and, at the same time, it provided a handy way to keep everything in sync.

The Observer pattern is one of the most famous design patterns discussed in the popular *Design Patterns: Elements of Reusable Object-Oriented Software* by The Gang of Four. It's a behavioral pattern and it provides a way to bind objects in a one-to-many dependency: when one object changes, all the objects depending on it are notified and updated automatically.

In this chapter, we are going to have an overview of the Observer pattern, how it's implemented and extended by RxJava, what an Observable is, and how Observables relate to Iterables.

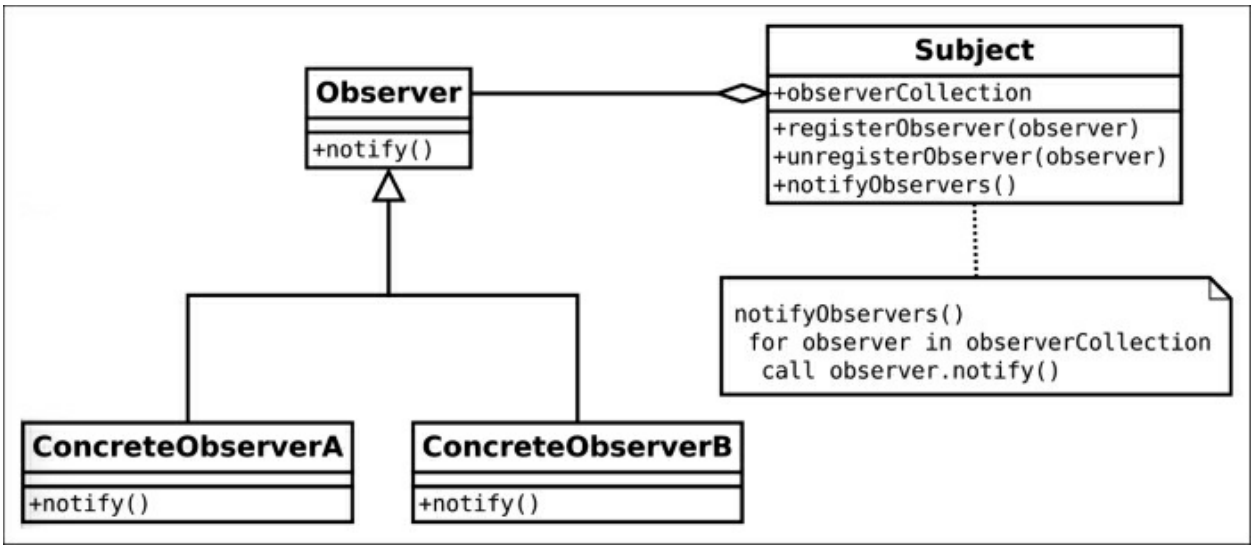
The Observer pattern

Nowadays, the Observer pattern is one of the most common software design patterns on the scene. It's based on the concept of *subject*. A subject is a particular object that keeps a list of objects that want to be notified when the subject changes. These objects are called *Observers* and they expose a notification method that the subject invokes when its state changes.

In the previous chapter, we saw the spreadsheet example. Now we can expand the example, showing a more complex scenario. Let's think about a spreadsheet containing the accounting data. We could represent this data as a table, as a 3D-histogram, or as a pie chart. Every one of these representations will depend on the same set of data being displayed. Every one of these representations will be an Observer, depending on one single subject, maintaining all the information.

The 3D-histogram class, the pie chart class, the table class, and the class maintaining the data are perfectly decoupled: they can be used and reused independently, but they can work together too. The representation classes don't know each other, but they act like they do: they know where to find the information they need to show, and they know that they have to update their data representation the moment the data changes are notified.

The figure here depicts how the Subject/Observer relationship is a one-to-many relationship:



The previous figure shows that one single subject can serve three Observers. Obviously, there is no reason to limit the number of Observers: a subject could have an infinite number of Observers if necessary, and every one of them will be notified when the subject state changes.

When do you use the Observer pattern?

The Observer pattern is the perfect fit for any of these scenarios:

- When your architecture has two entities, one depending on the other, and you want to keep them separated to change them or reuse them independently
- When a changing object has to notify an unknown amount of related objects about its own change
- When a changing object has to notify other objects without making assumptions about who these objects are

The RxJava Observer pattern toolkit

In the RxJava world, we have four main *players*:

- Observable
- Observer
- Subscriber
- Subjects

Observables and Subjects are the two "producing" entities. Observers and Subscribers are the two "consuming" entities.

Observable

When we have to execute something asynchronously with a lite level of complexity, Java provides classic classes, such as `Thread`, `Future`, `FutureTask`, `CompletableFuture`, to approach the problem. When the level of complexity goes up, these solutions tend to become messy and hard to maintain. Most of all, they cannot be chained.

RxJava Observables were designed to solve these issues. They are flexible and easy to use, they can be chained, and they can work on a single result routine or, even better, on sequences. Whenever you want to emit a single scalar value or a sequence of values, or even an infinite value stream, you can use an Observable.

The Observable life cycle contains three possible events that we can easily compare to Iterable life cycle events. The next table shows how the Observable async/push approach relates to the Iterable sync/pull one:

Event	**Iterable (pull)**	**Observable (push)**
Retrieve the data	`T next()`	`onNext(T)`
Discover the error	throws `Exception`	`onError(Throwable)`
Complete	`!hasNext()`	`onCompleted()`

With Iterable, the consumer synchronously pulls values from the producer and the thread is blocked until these values arrive. By contrast, with Observable, the producer asynchronously pushes values to the *Observer* whenever values are available. This approach is more flexible because even if values arrive synchronously or asynchronously, the consumer can act according to expectations in both scenarios.

To properly replicate the same Iterable interface, the RxJava Observable class enhances the base semantic of the Observer pattern by the Gang of Four, and

introduces two new abilities:

- `onCompleted()`, that is, the ability to notify the Observer that there is no more data coming from the Observable
- `onError()`, that is, the ability to notify the Observer that an error occurred

Keeping in mind these two abilities and the preceding table, we know that Observables and Iterables have a comparable abilities set but a different data flow direction: push versus pull.

Hot and cold Observables

From an *emission* point of view, there are two different types of Observables: **hot** and **cold**. A hot Observable, typically, starts emitting items as soon as it is created, so any Observer who subscribes to this Observable may start observing the sequence somewhere in the middle. A cold Observable waits until an Observer subscribes to it before it begins to emit items, so such an Observer is guaranteed to see the whole sequence from the beginning.

Creating an Observable

The Observable class provides the ways discussed in the upcoming sections to create an Observable.

Observable.create()

The `create()` method gives the developer the ability to create an Observable from scratch. It takes an `OnSubscribe` object, which extends `Action1`, as a parameter and executes the `call()` function when an Observer subscribes to our Observable:

```
Observable
        .create(new Observable.OnSubscribe<Object>() {
            @Override
            public void call(Subscriber<? super Object>
subscriber) {

            }
        });
```

Tip

Downloading the example code

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

`Observable` communicates with the Observer using the `subscriber` variable and by calling its methods according to the conditions. Let's see a real-world example:

```
Observable<Integer> observableString = Observable.create(new
Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> observer) {

        for (int i = 0; i < 5; i++) {
            observer.onNext(i);
        }

        observer.onCompleted();
    }
});

Subscription subscriptionPrint =
observableString.subscribe(new Observer<Integer>() {
    @Override
    public void onCompleted() {
        System.out.println("Observable completed");
    }

    @Override
    public void onError(Throwable e) {
        System.out.println("Oh no! Something wrong
happened!");
    }

    @Override
```

```
    public void onNext(Integer item) {
        System.out.println("Item is " + item);
    }
});
```

The example is kept simple on purpose, because even if it's the first time you see RxJava in action, I want you to be able to figure out what's going on.

We created a new `Observable<Integer>` item that executes a `for` loop on five elements, emitting them one by one, and then it completes.

On the other side, we subscribe to the Observable, obtaining a `Subscription` object. The moment we subscribe, we start to receive integers and we print them one by one. We don't know how many integers we will receive. Indeed, we don't need to know because we provided behaviors for every possible scenario:

- If we receive an integer, we print it
- If the sequence ends, we print a closed sequence message
- If an error occurs, we print an error message

Observable.from()

In the previous example, we created a sequence of integers and emitted them one by one. What if we already have a list? Can we save the `for` loop but still emit items one by one?

In the following code example, we create an Observable sequence from a list we already had:

```
List<Integer> items = new ArrayList<Integer>();
items.add(1);
items.add(10);
items.add(100);
items.add(200);

Observable<Integer> observableString = Observable.from(items);
Subscription subscriptionPrint =
observableString.subscribe(new Observer<Integer>() {
    @Override
    public void onCompleted() {
```

```
        System.out.println("Observable completed");
    }
    @Override
    public void onError(Throwable e) {
        System.out.println("Oh no! Something wrong
happened!");
    }

    @Override
    public void onNext(Integer item) {
        System.out.println("Item is " + item);
    }
});
```

The output is absolutely the same as that in the previous example.

The `from()` creator can create an Observable from a list/array, emitting every object contained in the list/array one by one or from a Java `Future` class, emitting the result value of the `.get()` method of the `Future` object. Passing `Future` as parameter, we can even specify a timeout value. The Observable will wait for a result from `Future`; if no result comes before the timeout, the Observable will fire the `onError()` method and notify the Observer that something was wrong.

Observable.just()

What if we already have a classic Java function and we want to transform it in an Observable? We could use the `create()` method, as we already saw, or we can avoid a lot of boilerplate code using something like this:

```
Observable<String> observableString =
Observable.just(helloWorld());

Subscription subscriptionPrint =
observableString.subscribe(new Observer<String>() {
    @Override
    public void onCompleted() {
        System.out.println("Observable completed");
    }

    @Override
    public void onError(Throwable e) {
```

```
        System.out.println("Oh no! Something wrong
happened!");
    }

    @Override
    public void onNext(String message) {
        System.out.println(message);
    }
});
```

The `helloworld()` function is super easy, like this:

```
private String helloWorld() {
    return "Hello World";
}
```

However, this could be any function we want. In the previous example, the moment we create the Observable, `just()` executes the function and when we subscribe to our Observable, it emits the returned value.

The `just()` creator can take between one and nine parameters. It will emit them in the same order as they are given as parameters. The `just()` creator also accepts lists or arrays, like `from()`, but it won't iterate on the list, emitting every value, it will just emit the whole list. Usually, it's used when we want to emit a defined set of values, but if our function is not time-variant, we can use `just()` to have a more organized and testable code base.

As a final note about the `just()` creator, after emitting the value, the Observable terminates normally. For the previous example, we will have two messages on the system console: `"Hello World"` and `"Observable completed"`.

Observable.empty(), Observable.never(), and Observable.throw()

If for any reason we need an Observable emitting nothing but terminating normally, we can use `empty()`. We can use `never()` to create an Observable emitting nothing and never terminating, and we can use `throw()` to create an Observable emitting nothing and terminating with an error.

Subject = Observable + Observer

A *subject* is a magic object that can be an Observable and an Observer at the same time: it acts as a bridge connecting the two worlds. A subject can subscribe to an Observable, acting like an Observer, and it can emit new items or even pass through the item it received, acting like an Observable. Obviously, being an Observable, Observers or other subjects can subscribe to it.

The moment the subject subscribes to the Observable, it will trigger the Observable to begin emitting. If the original Observable is *cold*, this can have the effect of making the resulting subject a *hot* Observable variant of the original *cold* Observable.

RxJava provides four different types of subjects:

- `PublishSubject`
- `BehaviorSubject`
- `ReplaySubject`
- `AsyncSubject`

PublishSubject

`PublishSubject` is the basic `Subject` object. Let's see our classic Observable Hello World achieved with a `PublishSubject`:

```
PublishSubject<String> stringPublishSubject =
PublishSubject.create();

Subscription subscriptionPrint =
stringPublishSubject.subscribe(new Observer<String>() {
    @Override
    public void onCompleted() {
        System.out.println("Observable completed");
    }

    @Override
    public void onError(Throwable e) {
        System.out.println("Oh no! Something wrong
happened!");
```

```
    }

    @Override
    public void onNext(String message) {
        System.out.println(message);
    }
});

stringPublishSubject.onNext("Hello World");
```

In the previous example, we created a new `PublishSubject`, emitting a `String` value with its `create()` method, and then we subscribed to `PublishSubject`. At this point, no item has been emitted yet, so our Observer is just *waiting*, without blocking or consuming resources. It's just there, ready to receive values from the subject. Our Observer would wait forever if the subject never emits a value. Once again, no worries: the Observer knows what to do in every scenario. The *when* does not concern us because this is reactive: the system will react. We don't care *when* it will react. We only care what's going to happen when it reacts.

The last line of code shows the manual emission of our `"Hello World"` string, which triggers the Observer `onNext()` method and lets us print the `"Hello World"` message to the console.

Let's see a more complex example. Say we have a `private` Observable and it's not accessible from the outside. This Observable emits values during its lifetime. We don't really care about these values, we only care about their termination.

First of all, we create a new `PublishSubject` that reacts on its `onNext()` method and is accessible from the outside world:

```
final PublishSubject<Boolean> subject =
PublishSubject.create();

subject.subscribe(new Observer<Boolean>() {
    @Override
    public void onCompleted() {

    }
```

```
    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onNext(Boolean completed) {
        System.out.println("Observable completed!");
    }
});
```

Then, we create the `private` Observable, which is only accessible to the subject:

```
Observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        for (int i = 0; i < 5; i++) {
            subscriber.onNext(i);
        }
        subscriber.onCompleted();
    }
}).doOnCompleted(new Action0() {
    @Override
    public void call() {
        subject.onNext(true);
    }
}).subscribe();
```

The `Observable.create()` method contains our familiar `for` loop, emitting numbers. The `doOnCompleted()` method specifies what is going to happen when the Observable terminates: emit `true` on the subject. Finally, we subscribe to the Observable. Obviously, the empty `subscribe()` call just starts the Observable, ignoring any emitted value, completed or error event. We need it like this for the sake of the example.

In this example, we created an entity that can be connected to Observables and can be observed at the same time. This is extremely useful when we want to create separation, abstraction, or a more observable point for a common resource.

BehaviorSubject

Basically, `BehaviorSubject` is a subject that emits the most recent item it has observed and all subsequent observed items to each subscribed item:

```
BehaviorSubject<Integer> behaviorSubject =
BehaviorSubject.create(1);
```

In this short example, we are creating a `BehaviorSubject` that emits `Integer`. `BehaviorSubject` needs an initial value due to the fact that it's going to emit the most recent value the moment an Observe subscribes.

ReplaySubject

`ReplaySubject` buffers all items it observes and replays them to any Observer that subscribes:

```
ReplaySubject<Integer> replaySubject = ReplaySubject.create();
```

AsyncSubject

`AsyncSubject` publishes only the last item observed to each Observer that has subscribed when the Observable completes:

```
AsyncSubject<Integer> asyncSubject = AsyncSubject.create();
```

Summary

In this chapter, we learned what an Observer pattern is, why Observables are so important in programming scenarios nowadays, and how we can create Observables and subjects.

In the next chapter, we will create our first Android app based on RxJava, learn how to retrieve data to populate `ListView`, and explore how to create a reactive UI based on RxJava.

Chapter 3. Hello Reactive World

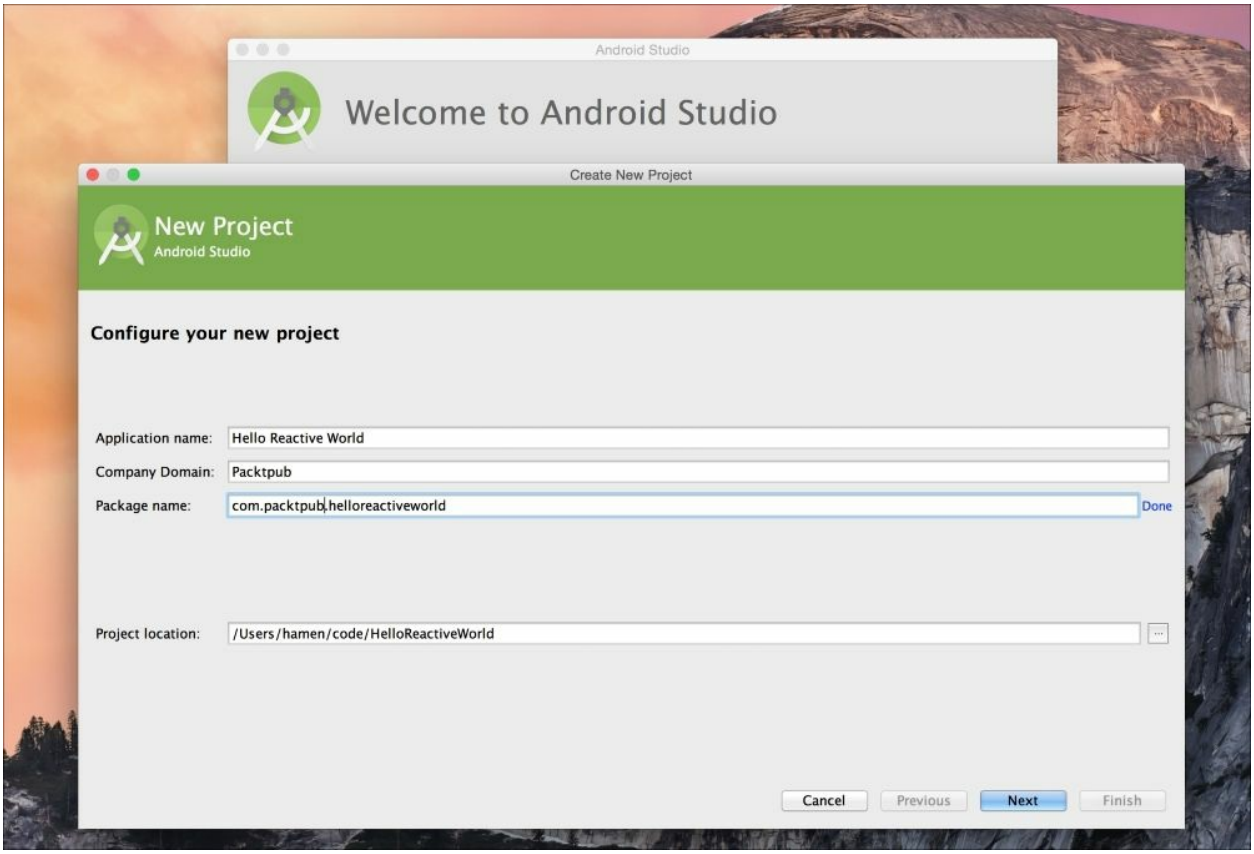
In the previous chapter, we had a quick theoretical overview of the Observer pattern. We also took a look at creating Observables from scratch, from a list, or from an already existing function. In this chapter, we are going to use what we learned to create our first reactive Android app. First of all, we are going to set up the environment, importing required libraries and useful libraries. Then, we are going to create a simple app, containing a few `RecyclerView` items, populated using RxJava, in a few different flavors.

Start the engine!

We are going to use IntelliJ IDEA/Android Studio for this project, so the screenshots should look familiar to you.

Let's dive in and create a new Android project. You can create your own project or import the one provided with the book. It's up to you to choose your preferred setup.

If you want to create a new project with Android Studio, as usual, you can refer to the official documentation at http://developer.android.com/training/basics/firstapp/creating-project.html:



Dependencies

Obviously, we are going to use **Gradle** to manage our dependencies list. Our `build.gradle` file will look like this:

MainActivity.java × | FirstExampleFragment.java × | app × | fragment_first_example.xml ×

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath 'me.tatarka:gradle-retrolambda:2.5.0'
    }
}

repositories {
    mavenCentral()
}

apply plugin: 'com.android.application'
apply plugin: 'me.tatarka.retrolambda'

android {
    compileSdkVersion 21
    buildToolsVersion "21.1.2"

    defaultConfig {
        applicationId "com.packtpub.apps.rxjava_essentials"
        minSdkVersion 16
        targetSdkVersion 21
        versionCode 1
        versionName "1.0"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }

    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }

    lintOptions {
        disable 'InvalidPackage'
    }

    packagingOptions {
        exclude 'META-INF/services/javax.annotation.processing.Processor'
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:support-v4:21.0.3'
    compile "com.android.support:appcompat-v7:21.0.3"
    compile 'com.android.support:recyclerview-v7:21.0.0'
    compile 'com.android.support:cardview-v7:21.0.3'

    compile 'org.projectlombok:lombok:1.14.8'
    compile 'com.jakewharton:butterknife:6.0.0'

    compile 'io.reactivex:rxandroid:0.24.0'
}
```

As you can see, we are importing RxAndroid. RxAndroid is an enhancement of RxJava, specifically designed for Android.

RxAndroid

RxAndroid is part of the RxJava family. It's based on RxJava 1.0.x, and it adds a few useful classes to the vanilla RxJava. Most of all, it adds specific Schedulers for Android. We will deal with Schedulers in Chapter 7, *Schedulers – Defeating the Android MainThread Issue*.

Utils

Being pragmatic, we also imported Lombok and Butter Knife. Both of them will help us avoid a lot of boilerplate code in our Android app.

Lombok

Lombok uses annotations to generate tons of code for you. We will use it mostly to generate getter/setter, `toString()`, `equals()`, and `hashCode()`. It comes with a Gradle dependency and an Android Studio plugin.

Butter Knife

Butter Knife uses annotations to save us from `findViewById()` and setting click listeners' pain. As for Lombok, we can import the dependency and install the Android Studio plugin for a better experience.

Retrolambda

Finally, we imported Retrolambda, because even if we are working with Android and its Java 1.6 support, we want to use Java 8 Lambda functions to cut down boilerplate code.

Our first Observable

In our first example, we are going to retrieve the list of the installed apps and populate a `RecyclerView` item to show them. We also have a fancy **pull-to-refresh** feature and a progress bar to notify the user that the task is ongoing.

First of all, we create our Observable. We need a function that retrieves the installed apps' list and provides it to the Observer. We are emitting items one by one and then grouping them into one single list, to show the flexibility of the reactive approach:

```
import com.packtpub.apps.rxjava_essentials.apps.AppInfo;

private Observable<AppInfo> getApps() {
    return Observable.create(subscriber -> {
        List<AppInfoRich> apps = new ArrayList<>();

        final Intent mainIntent = new
Intent(Intent.ACTION_MAIN,  null);
        mainIntent.addCategory(Intent.CATEGORY_LAUNCHER);

        List<ResolveInfo> infos = getActivity()
.getPackageManager().queryIntentActivities(mainIntent, 0);
        for (ResolveInfo info : infos) {
            apps.add(new AppInfoRich(getActivity(), info));
        }

        for (AppInfoRich appInfo : apps) {
            Bitmap icon =
Utils.drawableToBitmap(appInfo.getIcon());
            String name = appInfo.getName();
            String iconPath = mFilesDir + "/" + name;
            Utils.storeBitmap(App.instance, icon, name);

            if (subscriber.isUnsubscribed()) {
                return;
            }
            subscriber.onNext(new AppInfo(name, iconPath,
appInfo.getLastUpdateTime()));
        }
        if (!subscriber.isUnsubscribed()) {
            subscriber.onCompleted();
```

```
        }
    });
}
```

The `AppInfo` object looks like this:

```
@Data
@Accessors(prefix = "m")
public class AppInfo implements Comparable<Object> {

    long mLastUpdateTime;

    String mName;

    String mIcon;

    public AppInfo(String name, String icon, long
lastUpdateTime) {
        mName = name;
        mIcon = icon;
        mLastUpdateTime = lastUpdateTime;
    }

    @Override
    public int compareTo(Object another) {
        AppInfo f = (AppInfo) another;
        return getName().compareTo(f.getName());
    }
}
```

It's important to note that we are checking the `Observer` subscription before emitting new items or completing the sequence. This makes the code more efficient, because we are not generating unnecessary items if nobody is waiting for them.

At this point, we can subscribe to this Observable and observe it. Subscribing to an Observable means that we have to provide the actions to execute when the data we need come in.

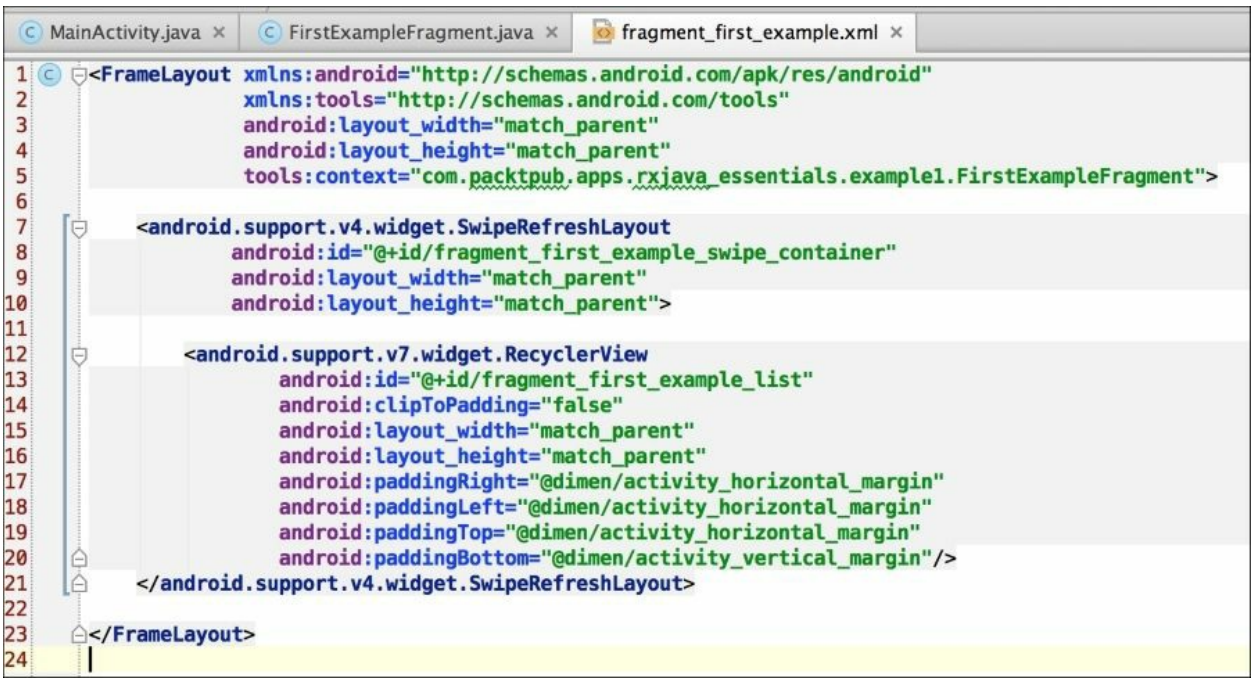
What is our scenario right now? Well, we are showing a spinning progress bar, waiting for the data. When the data comes in, we have to hide the progress bar, populate the list, and, eventually, show the list. Now, we know what to do

when everything is fine. What about an error scenario? In case of error, we just want to show an error message as `Toast`.

Using Butter Knife, we get the reference to the list and the pull-to-refresh element:

```
@InjectView(R.id.fragment_first_example_list)
RecyclerView mRecyclerView;

@InjectView(R.id.fragment_first_example_swipe_container)
SwipeRefreshLayout mSwipeRefreshLayout;
```

We are using Android 5's standard components: `RecyclerView` and `SwipeRefreshLayout`. This screenshot shows the layout file for our simple apps' list's `Fragment`:



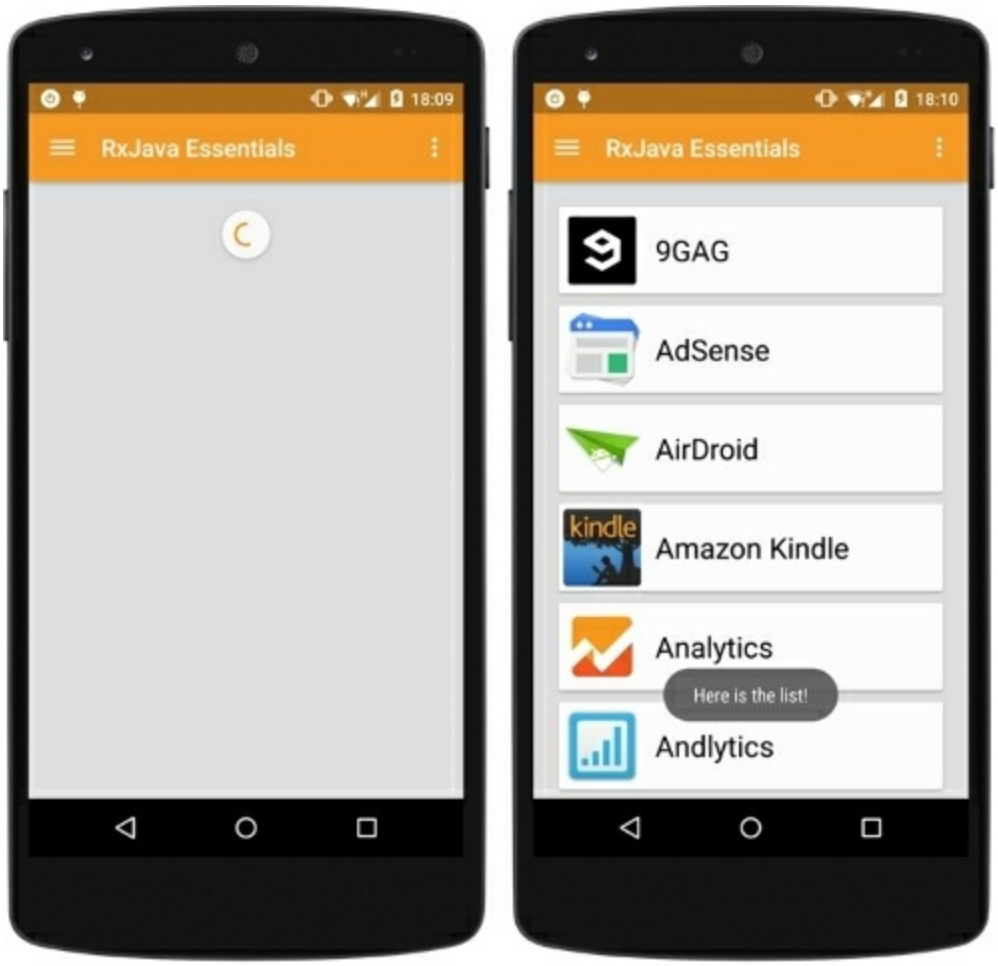
We are using a pull-to-refresh approach, so the list can come from the initial loading, or from a refresh action triggered by the user. We have the same behavior for two scenarios, so we will put our Observer in a function to be easily reused. Here is our Observer, with its success, errors, and completed behaviors:

```
private void refreshTheList() {
    getApps().toSortedList()
            .subscribe(new Observer<List<AppInfo>>() {
                @Override
                public void onCompleted() {
                    Toast.makeText(getActivity(), "Here is the
list!", Toast.LENGTH_LONG).show();
                }

                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something
went  wrong!", Toast.LENGTH_SHORT).show();
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onNext(List<AppInfo> appInfos) {
                    mRecyclerView.setVisibility(View.VISIBLE);
                    mAdapter.addApplications(appInfos);
                    mSwipeRefreshLayout.setRefreshing(false);
                }
            });
}
```

Having a function gives us the possibility of using the same block for two scenarios. We just have to call `refreshTheList()` when the fragment loads and sets `refreshTheList()` as the action to trigger when the user uses the pull-to-refresh approach:

```
mSwipeRefreshLayout.setOnRefreshListener(this::refreshTheList)
;
```

Our first example is now complete, up, and running!


18:09
RxJava Essentials
18:10
RxJava Essentials
9GAG
AdSense
AirDroid
Amazon Kindle
Analytics
Here is the list!
Andlytics

Creating an Observable from a list

In this example, we are going to introduce the `from()` function. With this particular "create" function, we can create an Observable from a list. The Observable will emit every element in the list, and we can subscribe to react to these emitted elements.

To achieve the same result of the first example, we are going to update our adapter on every `onNext()` function, adding the element and notifying the insertion.

We are going to reuse the same structure as the first example. The main difference is that we are not going to retrieve the installed applications' list. The list will be provided by an external entity:

```
mApps = ApplicationsList.getInstance().getList();
```

After obtaining the list, we only need to make it reactive and populate the `RecyclerView` item:

```
private void loadList(List<AppInfo> apps) {
    mRecyclerView.setVisibility(View.VISIBLE);

    Observable.from(apps)
            .subscribe(new Observer<AppInfo>() {
                @Override
                public void onCompleted() {
                    mSwipeRefreshLayout.setRefreshing(false);
                    Toast.makeText(getActivity(), "Here is the
list!", Toast.LENGTH_LONG).show();
                }

                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something
went  wrong!", Toast.LENGTH_SHORT).show();
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
```

```
                public void onNext(AppInfo appInfo) {
                    mAddedApps.add(appInfo);
                    mAdapter.addApplication(mAddedApps.size()
- 1,  appInfo);
                }
            });
}
```

As you can see, we are passing the installed apps list as a parameter to the `from()` function, and then we subscribe to the generated Observable. The Observer is quite similar to the Observer in the first example. One major difference is that we are stopping the spinning progress bar in the `onCompleted()` function because we are emitting every item singularly; the Observable in the first example was emitting the whole list, so it was safe to stop the spinning progress bar in the `onNext()` function.

A few more examples

In this section, we are going to show a few examples based on RxJava's `just()`, `repeat()`, `defer()`, `range()`, `interval()`, and `timer()` methods.

just()

Let's assume we only have three separated `AppInfo` objects and we want to convert them into an Observable and populate our `RecyclerView` item:

```
List<AppInfo> apps = ApplicationsList.getInstance().getList();

AppInfo appOne = apps.get(0);

AppInfo appTwo = apps.get(10);

AppInfo appThree = apps.get(24);

loadApps(appOne, appTwo, appThree);
```

We can retrieve the list like we did in the previous example and extract only three elements. Then, we pass them to the `loadApps()` function:

```
private void loadApps(AppInfo appOne, AppInfo appTwo, AppInfo
appThree) {
    mRecyclerView.setVisibility(View.VISIBLE);

    Observable.just(appOne, appTwo, appThree)
            .subscribe(new Observer<AppInfo>() {
                @Override
                public void onCompleted() {
                    mSwipeRefreshLayout.setRefreshing(false);
                    Toast.makeText(getActivity(), "Here is the
list!", Toast.LENGTH_LONG).show();
                }
                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something
went  wrong!", Toast.LENGTH_SHORT).show();
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onNext(AppInfo appInfo) {
                    mAddedApps.add(appInfo);
                    mAdapter.addApplication(mAddedApps.size()
- 1,  appInfo);
                }
```

```
                });
}
```

As you can see, the code is very similar to the previous example. This approach gives you the opportunity to think about code reuse.

You can even pass a function as a parameter to the `just()` method and you will have a raw Observable version of the existing code. Migrating from an existing code base to a new reactive architecture, this approach can be a useful point of start.

repeat()

Let's assume you want to repeat the items emitted by an Observable three times. For example, we will use the Observable in the `just()` example:

```
private void loadApps(AppInfo appOne, AppInfo appTwo, AppInfo
appThree) {
    mRecyclerView.setVisibility(View.VISIBLE);

    Observable.just(appOne, appTwo, appThree)
            .repeat(3)
            .subscribe(new Observer<AppInfo>() {
                @Override
                public void onCompleted() {
                    mSwipeRefreshLayout.setRefreshing(false);
                    Toast.makeText(getActivity(), "Here is the
list!", Toast.LENGTH_LONG).show();
                }

                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something
went  wrong!", Toast.LENGTH_SHORT).show();
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onNext(AppInfo appInfo) {
                    mAddedApps.add(appInfo);
                    mAdapter.addApplication(mAddedApps.size()
- 1,  appInfo);
                }
            });
}
```

As you can see, appending `repeat(3)` after the `just()` Observable creation call will create a sequence of nine items, every one emitted singularly.

defer()

There can be scenarios where you want to declare an Observable but you want to defer its creation until an Observer subscribes. Let's say we have this `getInt()` function:

```
private Observable<Integer> getInt() {
    return Observable.create(subscriber -> {
        if (subscriber.isUnsubscribed()) {
            return;
        }
        App.L.debug("GETINT");
        subscriber.onNext(42);
        subscriber.onCompleted();
    });
}
```

This is pretty simple and it doesn't really do much, but it will serve the purpose properly. Now, we can create a new Observable and apply `defer()`:

```
Observable<Integer> deferred = Observable.defer(this::getInt);
```

At this time, `deferred` exists, but the `getInt() create()` method hasn't been called yet: there is no `"GETINT"` in our logcat log:

```
deferred.subscribe(number -> {
   App.L.debug(String.valueOf(number));
});
```

But the moment we subscribe, `create()` gets called and we get two new lines in our `logcat` log: `GETINT` and `42`.

range()

Do you need to emit *N* integers from a specific starting number *X*? You can use `range`:

```
Observable.range(10, 3)
        .subscribe(new Observer<Integer>() {
            @Override
            public void onCompleted() {
                Toast.makeText(getActivity(), "Yeaaah!",
Toast.LENGTH_LONG).show();
            }

            @Override
            public void onError(Throwable e) {
                Toast.makeText(getActivity(), "Something went
wrong!", Toast.LENGTH_SHORT).show();
            }

            @Override
            public void onNext(Integer number) {
                Toast.makeText(getActivity(), "I say " +
number,  Toast.LENGTH_SHORT).show();
            }
        });
```

The `range()` function takes two numbers as parameters: the first one is the starting point, and the second one is the amount of numbers we want to emit.

interval()

The `interval()` function comes in very handy when you have to create a polling routine:

```
Subscription stopMePlease = Observable.interval(3,
TimeUnit.SECONDS)
        .subscribe(new Observer<Integer>() {
            @Override
            public void onCompleted() {
                Toast.makeText(getActivity(), "Yeaaah!",
Toast.LENGTH_LONG).show();
            }

            @Override
            public void onError(Throwable e) {
                Toast.makeText(getActivity(), "Something went
wrong!", Toast.LENGTH_SHORT).show();
            }

            @Override
            public void onNext(Integer number) {
                Toast.makeText(getActivity(), "I say " +
number,  Toast.LENGTH_SHORT).show();
            }
        });
```

The `interval()` function takes two parameters: a number that specifies the amount of time between two emissions, and the unit of time to be used.

timer()

If you need an Observable that emits after a span of time, you can use `timer()` like in the following example:

```
Observable.timer(3, TimeUnit.SECONDS)
        .subscribe(new Observer<Long>() {
            @Override
            public void onCompleted() {

            }

            @Override
            public void onError(Throwable e) {
            }
            @Override
            public void onNext(Long number) {
                Log.d("RXJAVA", "I say " + number);
            }
        });
```

This will emit `0` after 3 seconds, and then it will compete. Let's use `timer()` with a third parameter, like the following example:

```
Observable.timer(3, 3, TimeUnit.SECONDS)
        .subscribe(new Observer<Long>() {
            @Override
            public void onCompleted() {

            }

            @Override
            public void onError(Throwable e) {
            }

            @Override
            public void onNext(Long number) {
                Log.d("RXJAVA", "I say " + number);
            }
        });
```

With this code, you can create a version of `interval()` that starts with an initial delay (3 seconds in the previous example) and then keeps on emitting a

new number every *N* seconds (`3` in the previous example).

Summary

In this chapter, we created our first Android app enriched by RxJava. We created Observable from scratch, from existing lists, and from existing functions. We learned how to create Observables that repeat, emit on an interval, or emit after a delay.

In the next chapter, we will master filtering, and will be able to create the sequence we need out of the value sequence we receive.

Chapter 4. Filtering Observables

In the previous chapter, we learned how to set up a new Android project with RxJava and how to create an Observable list to populate `RecyclerView`. We now know how to create an Observable from scratch, a list, or an existing classic Java function.

In this chapter, we will dive into the essence of an Observable sequence: filtering. We will learn how to select only the values we want from an emitting Observable, how to obtain a finite number of values, and how to handle overflow scenarios, and a few more useful tricks.

Filtering a sequence

RxJava lets us use `filter()` to keep certain values that we don't want out of the sequence we are observing. In the previous chapter, we used the installed apps list in a few examples, but what if we want to show only the installed app whose name starts with `C`? In this new example, we will use the same list, but we will filter it, passing the proper predicate to the `filter()` function to include the values we want.

The `loadList()` function we had in the previous chapter changes like this:

```
private void loadList(List<AppInfo> apps) {
    mRecyclerView.setVisibility(View.VISIBLE);

    Observable.from(apps)
            .filter((appInfo) ->
appInfo.getName().startsWith("C"))
            .subscribe(new Observer<AppInfo>() {
                @Override
                public void onCompleted() {
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something
went  south!", Toast.LENGTH_SHORT).show();
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onNext(AppInfo appInfo) {
                    mAddedApps.add(appInfo);
                    mAdapter.addApplication(mAddedApps.size()
- 1,  appInfo);
                }
            });
}
```

We have added the following line to the `loadList()` function from the previous chapter:

```
.filter((appInfo) -> appInfo.getName().startsWith("C"))
```

After the creation of an Observable, we are filtering out every emitted element that has a name starting with a letter that is not `C`. Let's have this in Java 7 syntax in order to clarify the types here:

```
.filter(new Func1<AppInfo, Boolean>() {
    @Override
    public Boolean call(AppInfo appInfo) {
        return appInfo.getName().startsWith("C");
    }
})
```

We are passing a new `Func1` object to `filter()`, that is, a function that has just one parameter. `Func1` has an `AppInfo` object as its parameter type and it returns a `Boolean` object. The `filter()` function will return `true` only if the condition is verified. At this point, the value will be emitted and received by all the Observers.

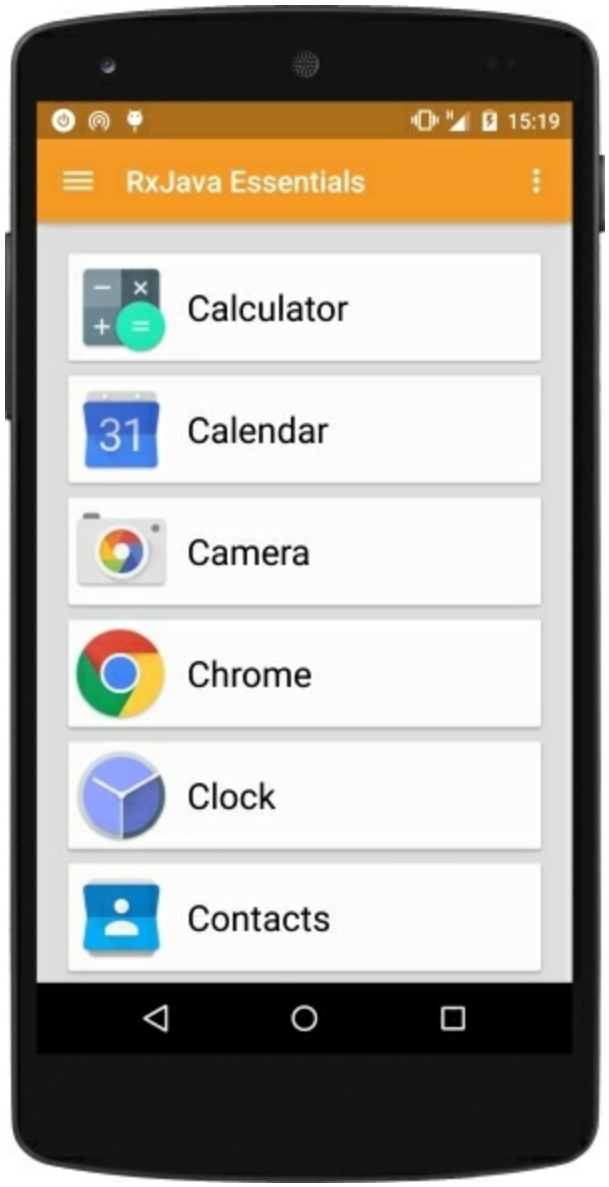
As you can imagine, `filter()` is critically useful to create the perfect sequence that we need from the Observable sequence we get. We don't need to know the source of the Observable sequence or why it's emitting tons of different elements. We just want a useful subset of these elements to create a new sequence we can use in our app. This mindset enforces the separation and abstraction skills of our coding life.

One of the most common uses of `filter()` is filtering `null` objects:

```
.filter(new Func1<AppInfo, Boolean>() {
    @Override
    public Boolean call(AppInfo appInfo) {
        return appInfo != null;
    }
})
```

This seems to be simple, and there is a lot of boilerplate code for something this simple, but this will save us from checking for `null` values in the `onNext()` call, letting us focus on the actual app logic.

The next figure shows the installed apps list, filtered by names starting with *C*:


15:19
RxJava Essentials
Calculator
Calendar
Camera
Chrome
Clock
Contacts

Let's take what we need

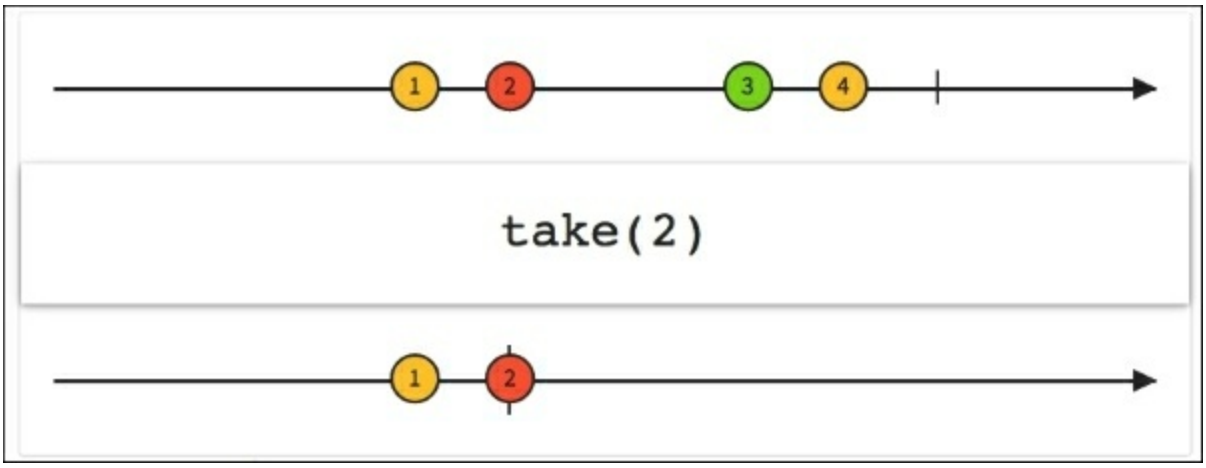
When we don't need the whole sequence but only a few elements at the beginning or end, we can use `take()` or `takeLast()`.

Take

What if we only want the first three elements of an Observable sequence, emit them, and then let the Observable complete? The `take()` function gets an *N* integer as a parameter, emits only the *N* first elements from the original sequence, and then it completes:

```
private void loadList(List<AppInfo> apps) {
    mRecyclerView.setVisibility(View.VISIBLE);

    Observable.from(apps)
            .take(3)
            .subscribe(new Observer<AppInfo>() {
                @Override
                public void onCompleted() {
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something
went  south!", Toast.LENGTH_SHORT).show();
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onNext(AppInfo appInfo) {
                    mAddedApps.add(appInfo);
                    mAdapter.addApplication(mAddedApps.size()
- 1,  appInfo);
                }
            });
}
```

The next figure shows an Observable sequence emitting numbers. We will apply a `take(2)` function to this sequence, and then we can create a new sequence emitting just the first and second item of the Observable source.


1
2
3
4
take(2)
1
2

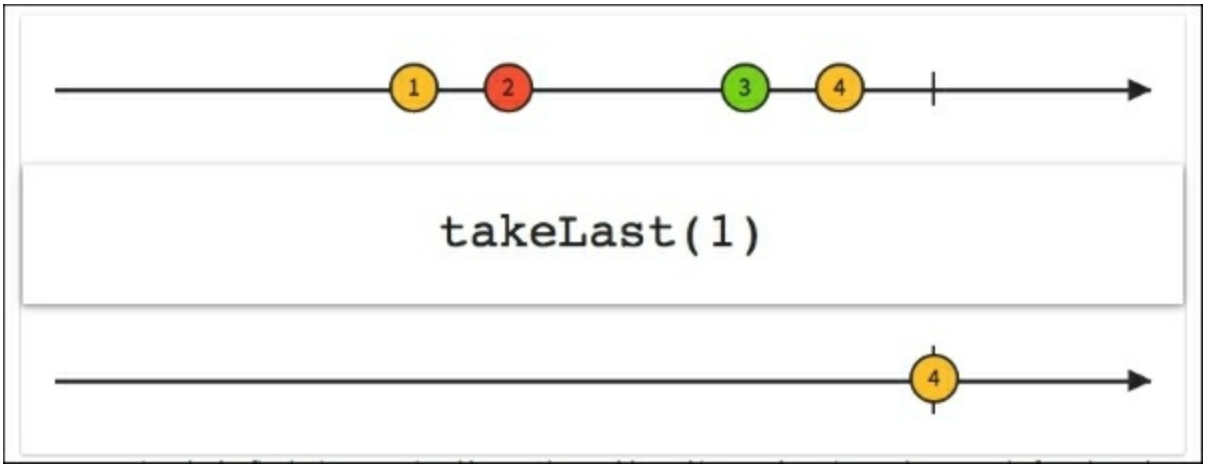
TakeLast

If we want the last *N* elements, we just use the `takeLast()` function:

```
Observable.from(apps)
        .takeLast(3)
        .subscribe(…);
```

As trivial as it sounds, it's important to note that the `takeLast()` function applies only to the sequence that completes due to its nature of working with a finite amount of emitted items.

The next figure shows how we can create a sequence emitting only the last element from a Observable source:



The next figure shows what happens when we apply `take()` and `takeLast()` to our installed applications list example:


15:43
RxJava Essentials
9GAG
AdSense
AirDroid
15:42
RxJava Essentials
Wpa Tester
Yatse
YouTube

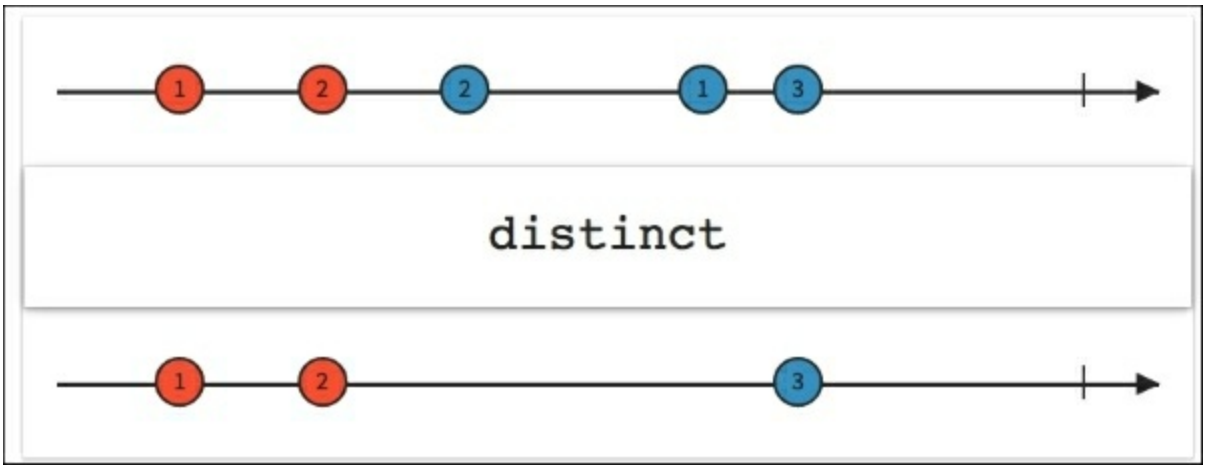
Once and only once

An emitting Observable sequence could emit duplicates by error or by design. The `distinct()` and `distinctUntilChanged()` functions allow us to deal smoothly with duplicates.

Distinct

What if we want to be absolutely sure that we process a specific value only once? We can get rid of the duplicates by applying the `distinct()` function to our Observable sequence. Like `takeLast()`, `distinct()` works with a completed sequence and to achieve this duplication filtering, it needs to keep a track of every emitted item. This is a memory usage concern that you have to keep in mind if you are dealing with a huge sequence or big emitted items.

The next figure shows how we can create a sequence with no duplicate working on an `Observable` source that emits numbers and emits `1` and `2` twice:



To create our example sequence, we are going to use a few methods we have learned so far:

- `take()`: This is to have a small set of recognizable items
- `repeat()`: This is to create a larger sequence containing a few duplicates

Then, we will apply the `distinct()` function to get rid of the duplicates.

Note

We are programmatically creating a sequence full of duplicates and then filtering them out. It sounds crazy, but it's for the sake of the example and is a useful exercise to use what we have learned so far.

```
Observable<AppInfo> fullOfDuplicates = Observable.from(apps)
        .take(3)
        .repeat(3);
```

The `fullOfDuplicates` variable contains the first three elements of our installed apps list repeated three times: nine elements and a lot of duplicates. Then, we apply `distinct()`:

```
fullOfDuplicates.distinct()
        .subscribe(new Observer<AppInfo>() {
            @Override
            public void onCompleted() {
                mSwipeRefreshLayout.setRefreshing(false);
            }

            @Override
            public void onError(Throwable e) {
                Toast.makeText(getActivity(), "Something went
south!", Toast.LENGTH_SHORT).show();
                mSwipeRefreshLayout.setRefreshing(false);
            }

            @Override
            public void onNext(AppInfo appInfo) {
                mAddedApps.add(appInfo);
                mAdapter.addApplication(mAddedApps.size() - 1,
appInfo);
            }
        });
```

As a result, obviously, we get:


12:44
RxJava Essentials
9GAG
AdSense
AirDroid

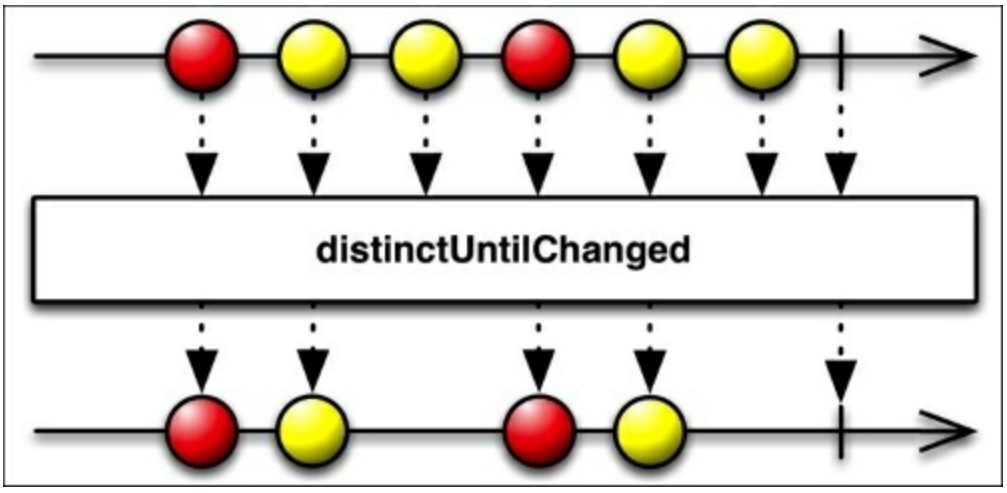
DistinctUntilChanged

What if we want to get notified when an Observable sequence emits a new value that is different from the previous one? Let's suppose we are observing a temperature sensor, emitting the room temperature every second:

`21°… 21°… 21°… 21°… 22°`

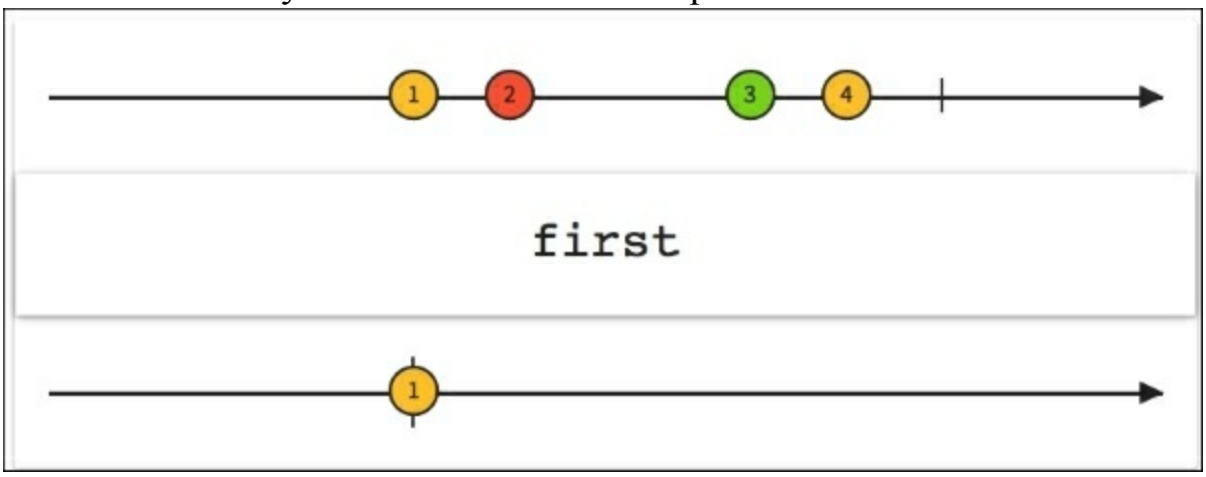
Every time we get a new value, we update a display showing the current temperature. We want to play *resource conservative* and we don't want to update the display if the value is still the same. We want to ignore all repeated values and just get notified when the temperature actually changes. The `distinctUntilChanged()` filtering function will do the trick. It simply ignores all the duplicates and emits only the new value.

The next figure shows, in a graphical way, how we can apply `distinctUntilChanged()` to an existing sequence to create a new sequence that won't emit elements that have already been emitted:



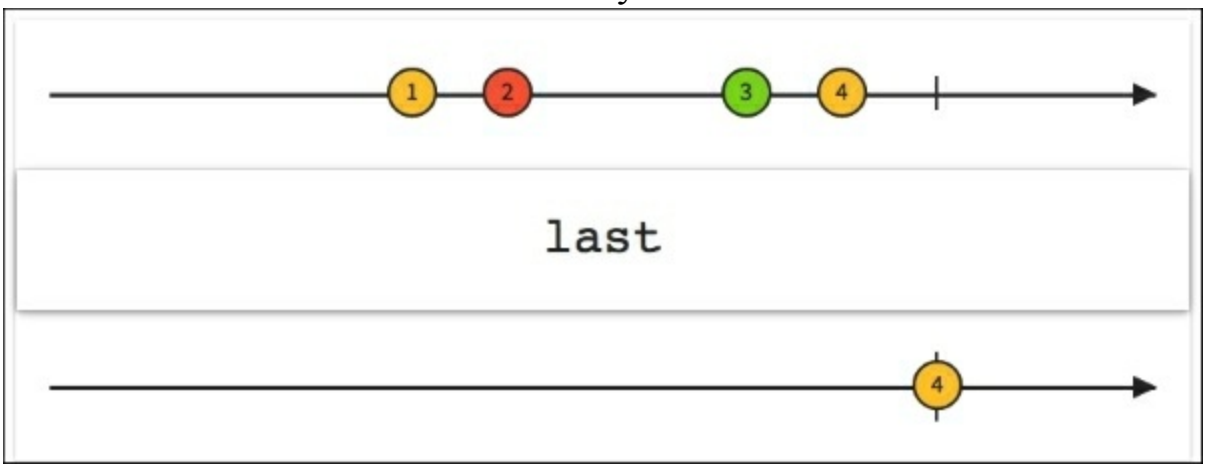
First and last

The next figure shows how to create a sequence by emitting only the first element emitted by a source Observable sequence:



The `first()` and `last()` methods are quite easy to figure out. They emit only the first or last element emitted by Observable. Both of them can even get `Func1` as a parameter: a *predicate* that can be used to decide the actual first or last element we are interested in.

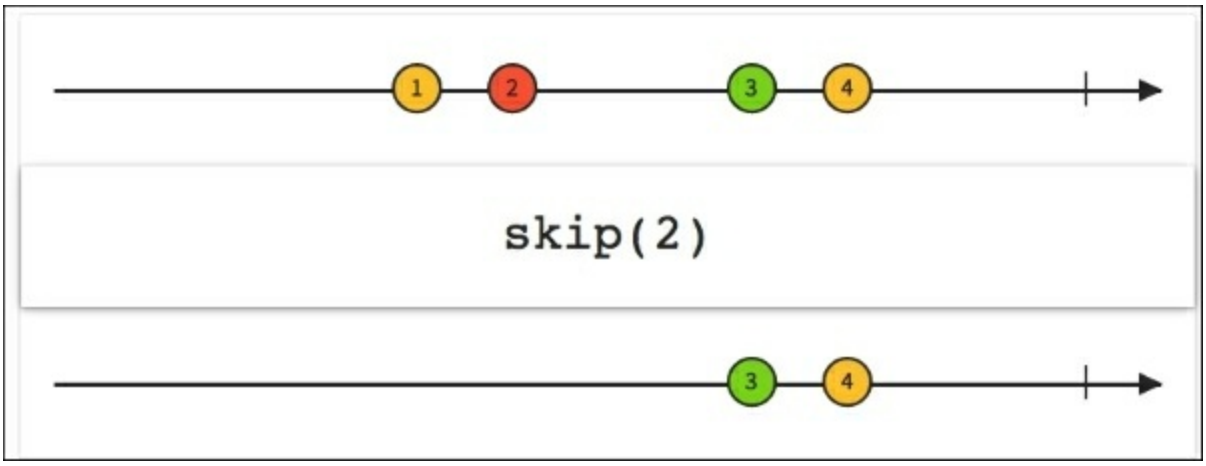
The next figure shows how `last()` can be applied to a completed sequence to create a new Observable that emits only the last emitted element:



Both `first()` and `last()` come with a similar variant: `firstOrDefault()` and `lastOrDefault()`. These two functions come in handy when the Observable sequence completes without emitting any value. In this scenario, we can decide a default value to emit if the Observable didn't emit any.

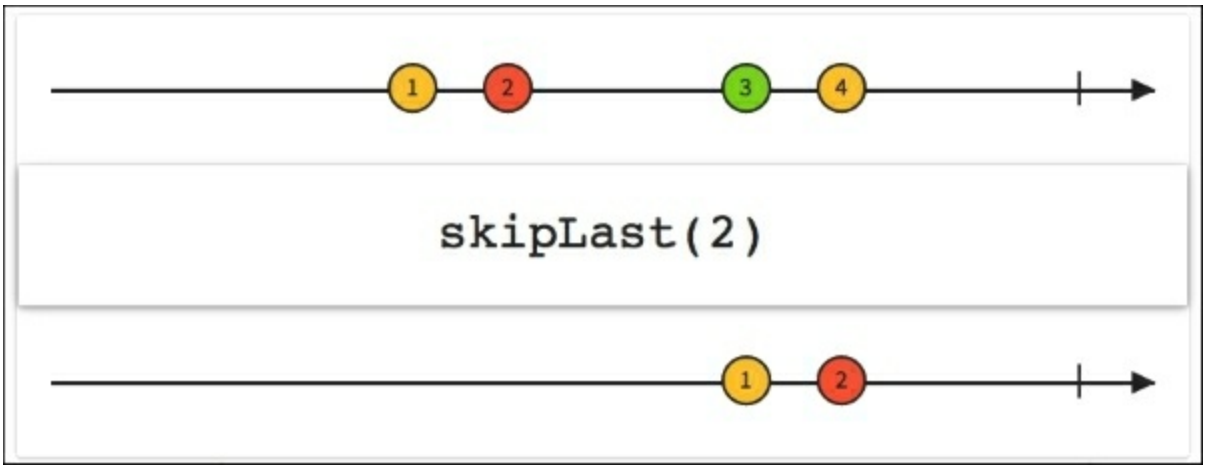
Skip and SkipLast

The next figure shows how we can apply `skip(2)` to create a sequence that doesn't emit the first two elements of an Observable source but emits all the next items:



The `skip()` and `skipLast()` functions are the counterpart of `take()` and `takeLast()`. They get an integer *N* as a parameter and, basically, they suppress the first *N* or last *N* values emitted by an Observable. We can use it if we know that a sequence starts or ends with a *control* element that is not strictly useful for our purpose.

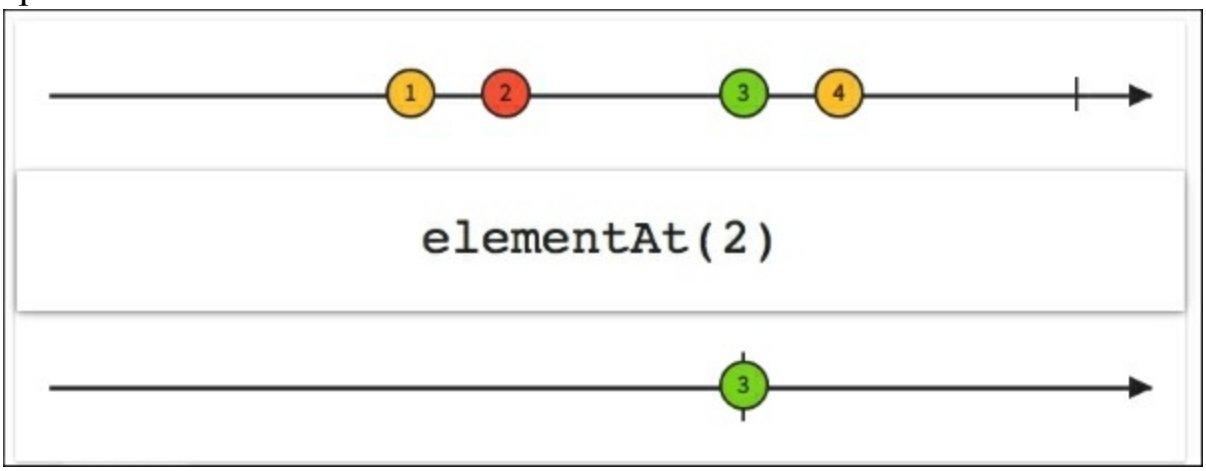
The next figure shows the counterpart scenario of the previous one: we are creating a new sequence that emits every item from the Observable source but skips the last two elements:


1
2
3
4
skipLast(2)
1
2

ElementAt

What if we want only the fifth element emitted by an Observable sequence? `elementAt()` function emits only the *n* element from a sequence, and then it completes.

What if we are looking for the fifth element but the Observable sequence only emitted three elements? We can use `elementAtOrDefault()`, of course. The next figure shows how to cherry-pick the third element from a sequence by applying `elementAt(2)` and creating a new Observable by emitting only this specific element:



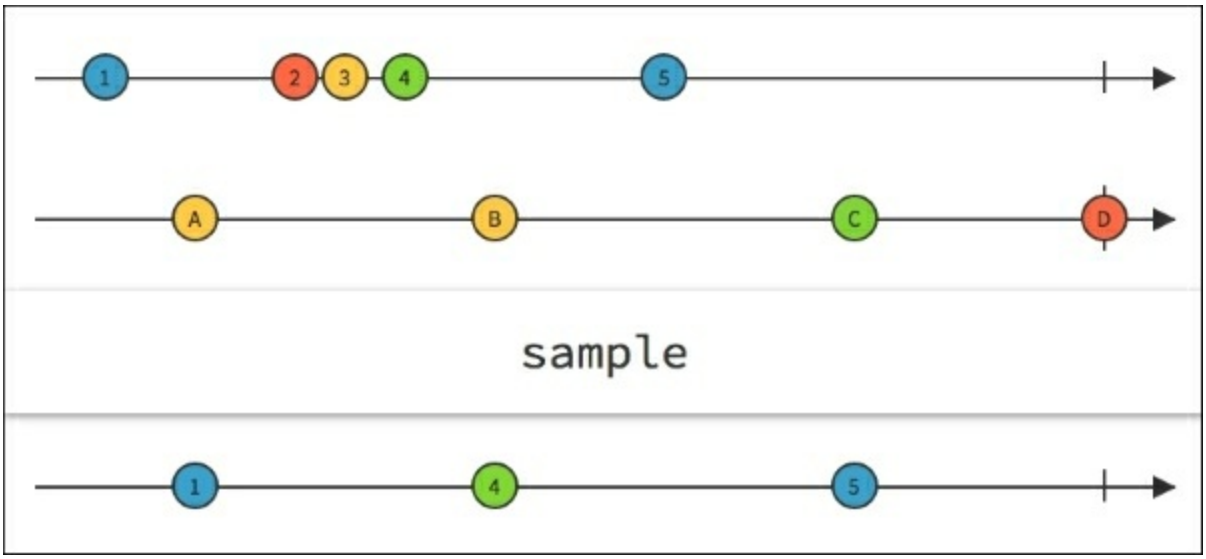
Sampling

Let's go back to our Observable temperature sensor. It's emitting the current room temperature every second. Honestly, we don't think the temperature will change so rapidly, and we could use a less stressful emitting interval. Appending `sample()` to the Observable source, we will create a new Observable sequence that will emit the most recent item emitted by the Observable source in a decided time interval:

```
Observable<Integer> sensor = […]

sensor.sample(30, TimeUnit.SECONDS)
        .subscribe(new Observer<Integer>() {
            @Override
            public void onCompleted() {

            }

            @Override
            public void onError(Throwable e) {

            }

            @Override
            public void onNext(Integer currentTemperature) {
                updateDisplay(currentTemperature);
            }
        });
```

The Observable in the example will observe the Observable source temperature and emit the last emitted temperature value every `30` seconds. Obviously, `sample()` supports a full set of `TimeUnit` values: seconds, milliseconds, day, minutes, and so on.

The next figure shows how an Observable interval emitting *letters* will sample an Observable emitting *numbers*. The Observable result will emit the last number of every emitted letter: 1, 4, 5.



If we want the *first* item emitted in the time interval instead of the last item, we can use `throttleFirst()` instead.

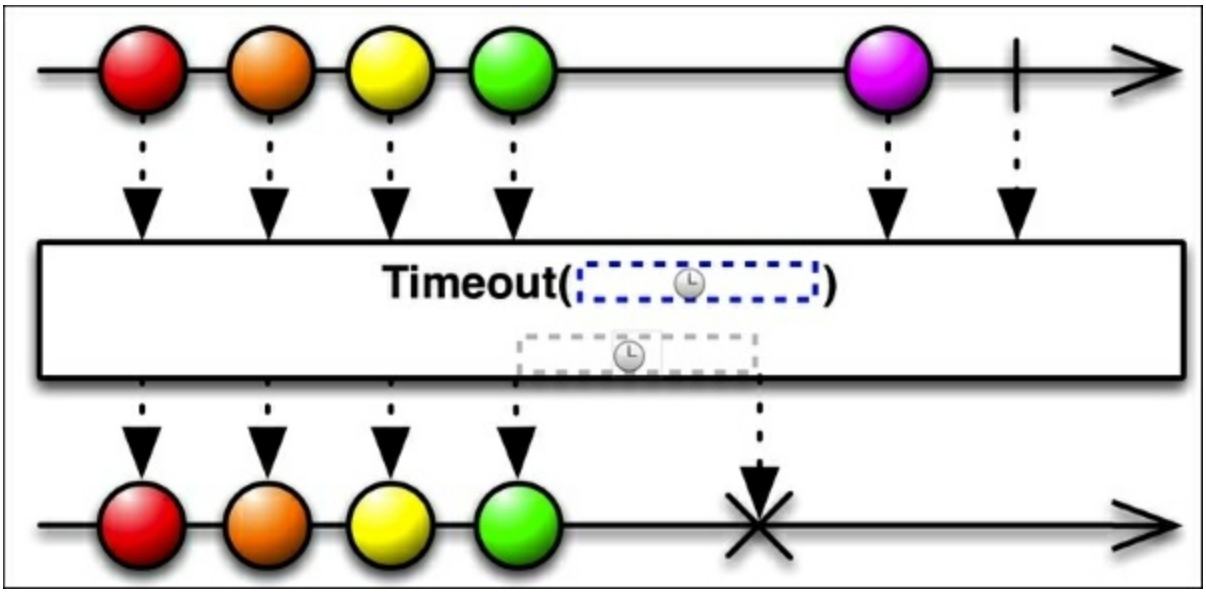
Timeout

Let's suppose we are working in a very time-sensitive environment. Our temperature sensor is emitting a value every second. We want to be absolutely sure we get at least one value every two seconds. We can use the `timeout()` function to mirror the source Observable sequence and emit an error if we don't get a value for the time interval we decided. We can consider `timeout()` as a time-constrained copy of our Observable. It mirrors the original Observable and fires `onError()` if the Observable doesn't emit values in the decided time interval:

```
Subscription subscription = getCurrentTemperature()
        .timeout(2, TimeUnit.SECONDS)
        .subscribe(new Observer<Integer>() {
            @Override
            public void onCompleted() {

            }

            @Override
            public void onError(Throwable e) {
                Log.d("RXJAVA", "You should go check the
sensor,  dude");
            }

            @Override
            public void onNext(Integer temperature) {
                updateDisplay(temperature);
            }
        });
```

Like `sample()`, `timeout()` uses the `TimeUnit` object to specify the time interval.

The next figure shows how the Observable will trigger `onError()` the moment the Observable violates the timeout constraint: the last item won't be emitted because it arrives after the timeout.

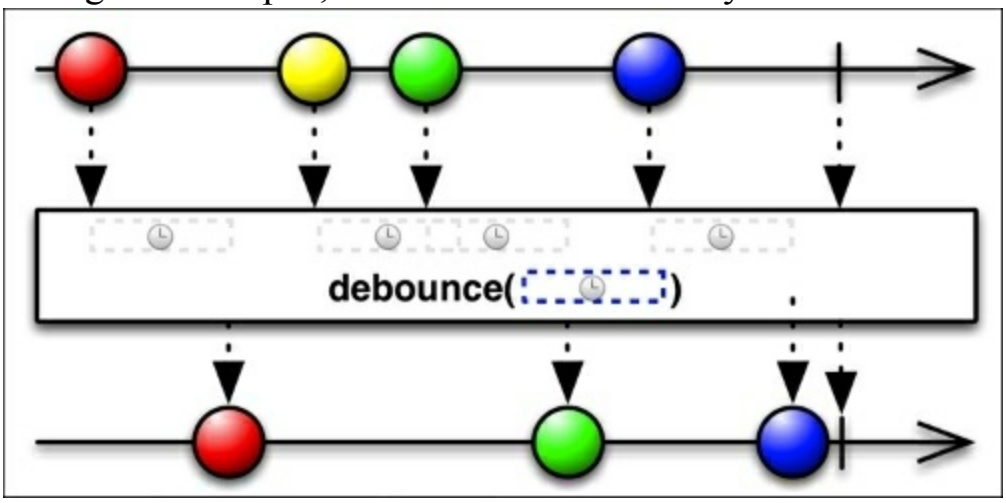

Timeout( )

Debounce

The `debounce()` function filters out items emitted by the Observable that are rapidly followed by another emitted item; it only emits an item from Observable if a particular timespan has passed without it emitting another item.

Like `sample()` and `timeout()`, `debounce()` uses the `TimeUnit` object to specify the time interval.

The next figure shows how every time a new item is emitted from the Observable, `debounce()` starts its internal timer, and if no new item is emitted during this timespan, the last item is emitted by the new Observable:



Summary

In this chapter, we learned how to filter an Observable sequence. We can now use `filter()`, `skip()`, and `sample()` to create the Observable that we want.

In the next chapter, we will learn how to transform a sequence, apply functions to every element, group them, and scan them to create the specific Observable that we need to achieve our goal.

Chapter 5. Transforming Observables

In the previous chapter, we explored the filtering universe of RxJava. We learned how to filter the values we don't need with `filter()`, how to get subsets of the emitted values with `take()`, and how to get rid of duplicates with `distinct()`. We learned how to exploit time with `timeout()`, `sample()`, and `debounce()`.

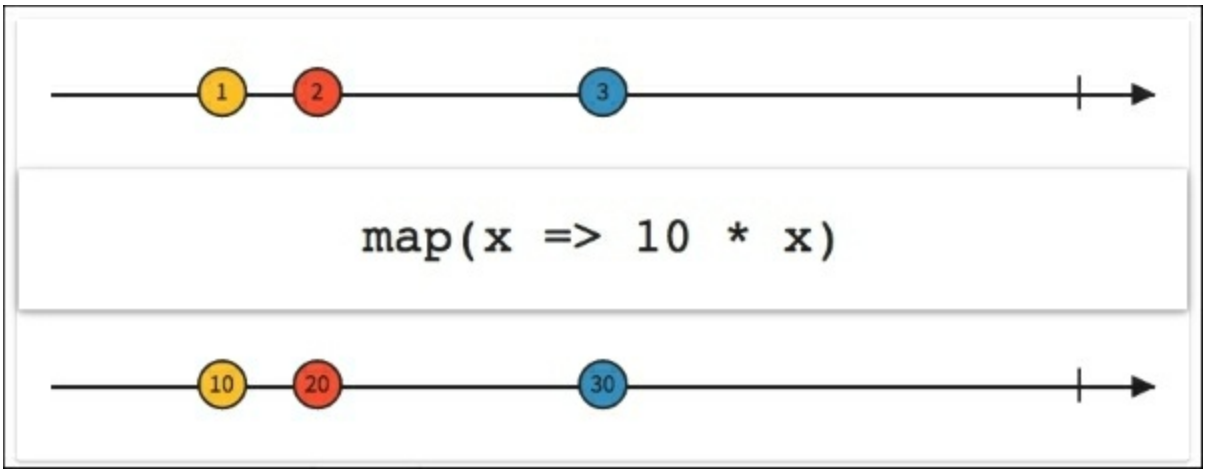
In this chapter, we will learn how to transform Observable sequences to create sequences that better fit our needs.

The *map family

RxJava provides a few mapping functions: `map()`, `flatMap()`, `concatMap()`, `flatMapIterable()`, and `switchMap()`. All these functions apply to an Observable sequence, transform its emitted values, and return them in a new form. Let's look at them one by one with proper real-world examples.

Map

RxJava's `map()` function receives a specific `Func` object and applies it to every value emitted by Observable. The next figure shows how to apply a multiplying function to every emitted item, to create a new Observable, emitting transformed items:



Let's think about our installed applications' list. How could we show the same list, but with all the names in lowercase?

Our `loadList()` function will change to this:

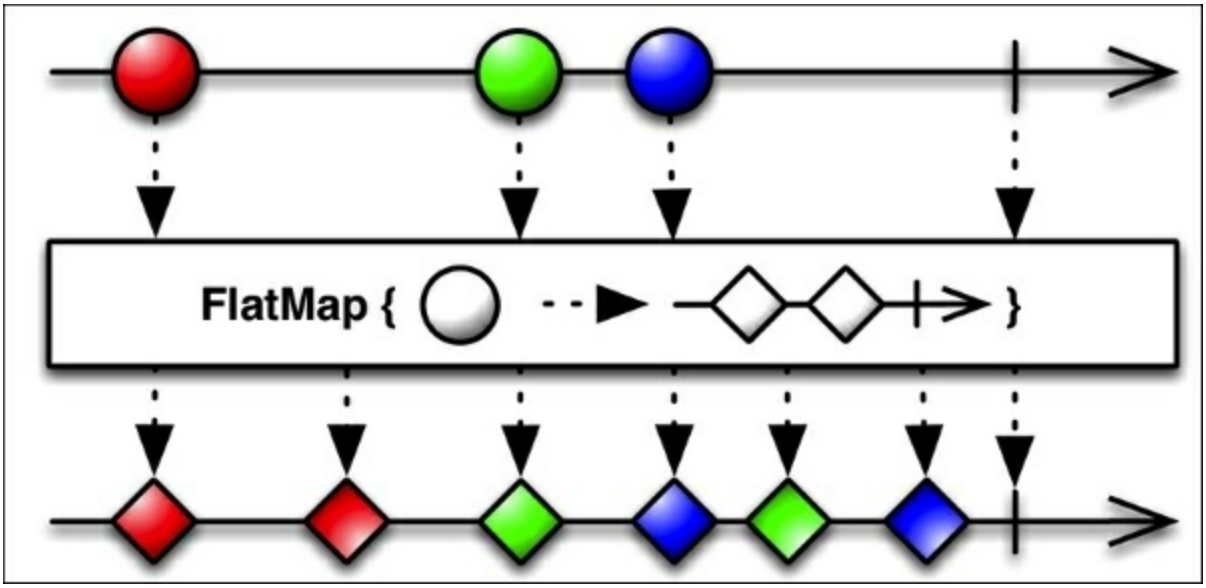
```
private void loadList(List<AppInfo> apps) {
    mRecyclerView.setVisibility(View.VISIBLE);

    Observable.from(apps)
            .map(new Func1<AppInfo, AppInfo>() {
                @Override
                public AppInfo call(AppInfo appInfo) {
                    String currentName = appInfo.getName();
                    String lowerCaseName =
currentName.toLowerCase();
                    appInfo.setName(lowerCaseName);
                    return appInfo;
                }
            })
            .subscribe(new Observer<AppInfo>() {
```

```
                @Override
                public void onCompleted() {
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something
went  south!", Toast.LENGTH_SHORT).show();
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onNext(AppInfo appInfo) {
                    mAddedApps.add(appInfo);
                    mAdapter.addApplication(mAddedApps.size()
- 1,  appInfo);
                }
            });
}
```

As you can see, after creating the emitting Observable as usual, we appended a `map()` call. We created a simple function that updates the `AppInfo` object and provides a new version with the lowercase name to Observer.

FlatMap

In a complex scenario, we could have an Observable that emits a sequence on values, which emit Observables. RxJava's `flatMap()` function provides a way to flatten the sequence, merging all the emitted value into one final Observable.

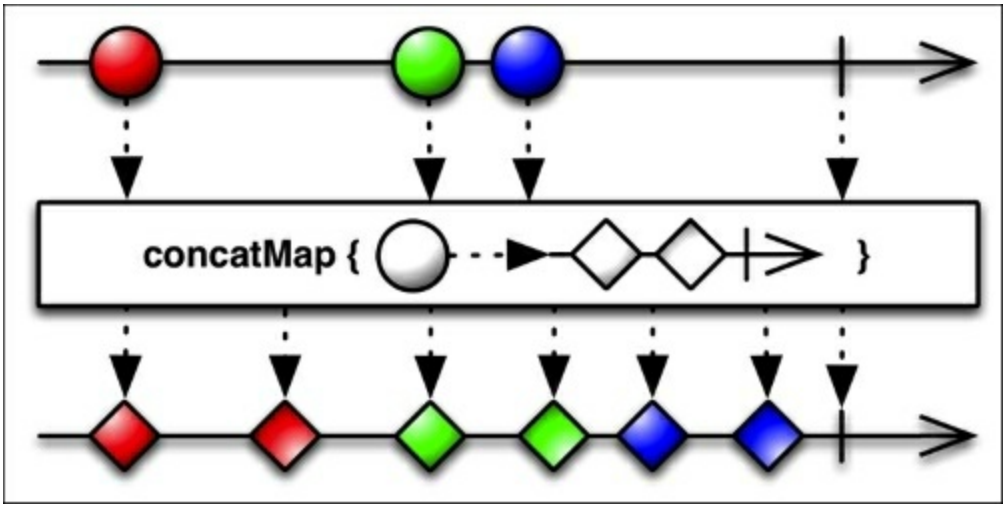


When working with a potentially large number of Observables, it is important to keep in mind that in case of error in any of the Observables, `flatMap()` itself will trigger its `onError()` function and abort the whole chain.

An important note is about the *merging* part: it allows interleaving. This means that `flatMap()` is not able to maintain the exact emitting order of the source Observables in the final Observable, as shown in the previous figure.

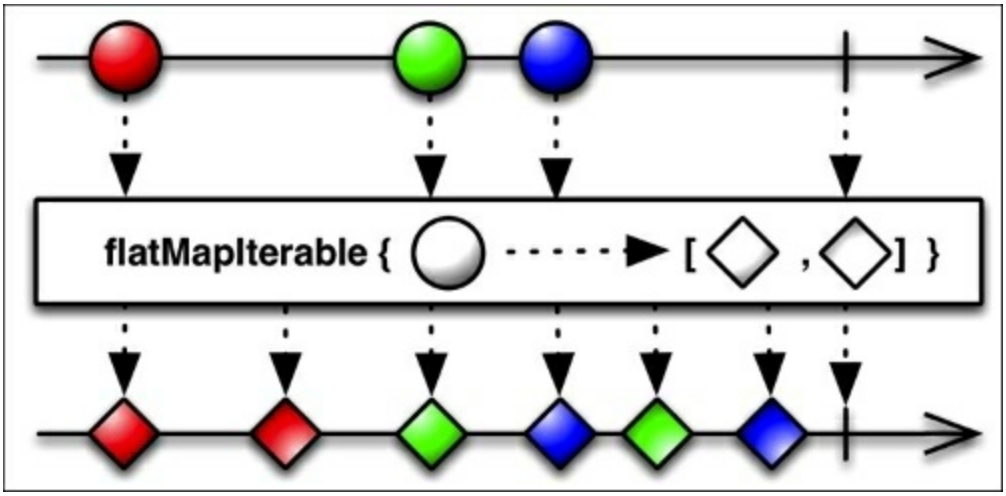
ConcatMap

RxJava's `concatMap()` function solves `flatMap()` interleaving issue, providing a flattening function that is able to concatenate the emitted values, instead of merging them, as shown in the following figure:



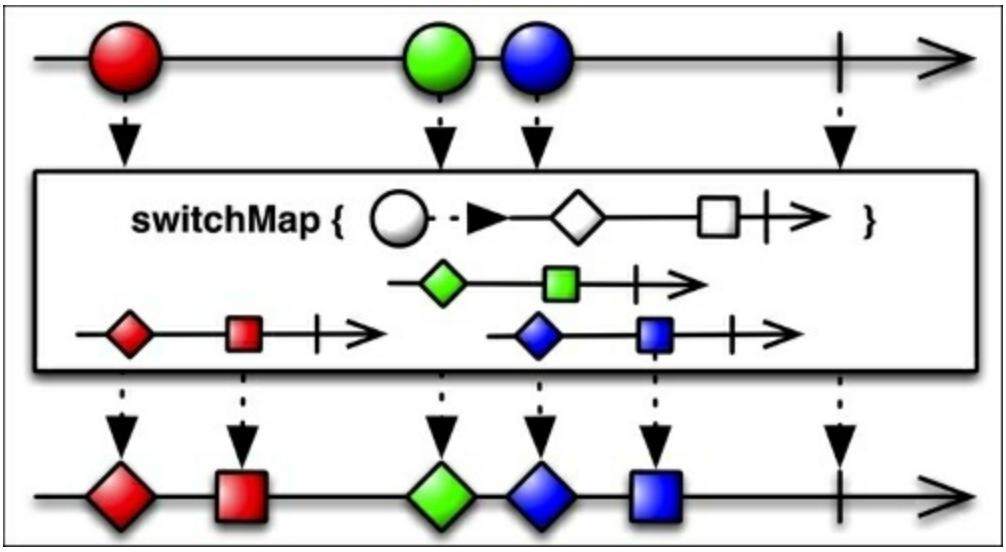
FlatMapIterable

As a member of the `*map` family, `flatMapIterable()` works similarly to `flatMap()`. The only concrete difference is that it pairs up source items and generated `Iterables`, rather than source items and generated Observables:



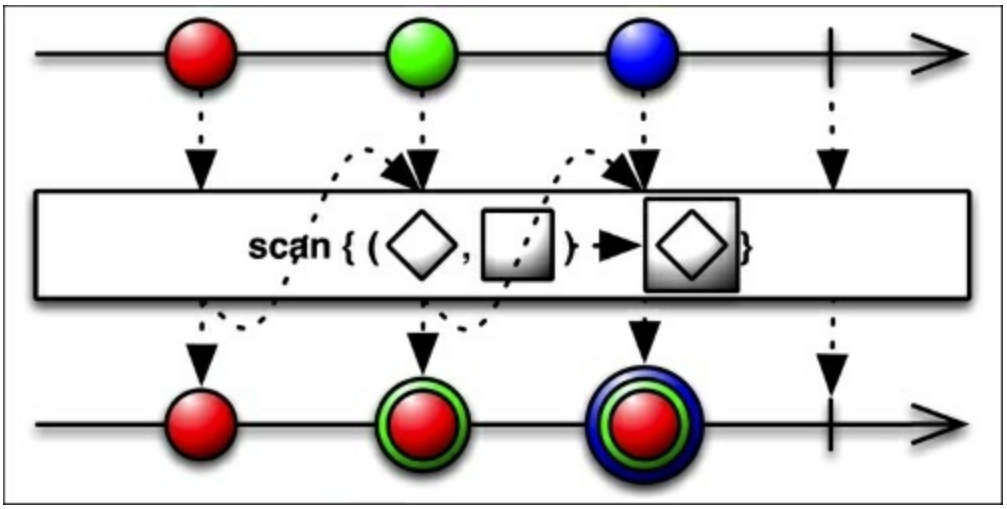
SwitchMap

As shown in the following figure, `switchMap()` acts similarly to `flatMap()`, but it unsubscribes from and stops mirroring the Observable that was generated from the previously emitted item and begins mirroring only the current one whenever a new item is emitted by the source Observable.



Scan

RxJava's `scan()` function can be considered as an *accumulator* function. The `scan()` function applies a function to every item emitted by the Observable, computes the function result, and injects the result back into the Observable sequence, waiting to use it with the next emitted value:



As a generic example, here is an *accumulator*:

```
Observable.just(1, 2, 3, 4, 5)
        .scan((sum, item) -> sum + item)
        .subscribe(new Subscriber<Integer>() {
            @Override
            public void onNext(Integer item) {
                Log.d("RXJAVA", "item is: " + item);
            }

            @Override
            public void onError(Throwable error) {
                Log.e("RXJAVA", "Something went south!");
            }

            @Override
            public void onCompleted() {
                Log.d("RXJAVA", "Sequence completed.");
            }
```

```
        });
```

As a result, we have:

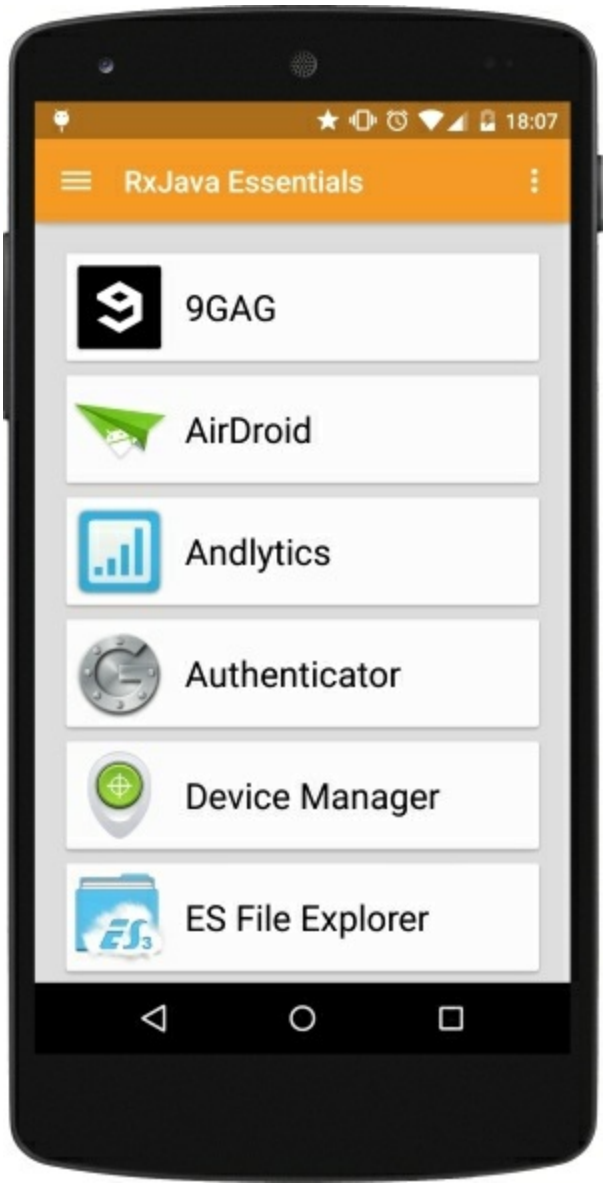
```
RXJAVA: item is: 1
RXJAVA: item is: 3
RXJAVA: item is: 6
RXJAVA: item is: 10
RXJAVA: item is: 15
RXJAVA: Sequence completed.
```

We can even create a new version of our `loadList()` function that compares every installed app name and creates a list of *incrementally* longer names:

```
private void loadList(List<AppInfo> apps) {
    mRecyclerView.setVisibility(View.VISIBLE);

    Observable.from(apps)
            .scan((appInfo, appInfo2) -> {
                if (appInfo.getName().length() >
appInfo2.getName().length()) {
                    return appInfo;
                } else {
                    return appInfo2;
                }
            })
            .distinct()
            .subscribe(new Observer<AppInfo>() {
                @Override
                public void onCompleted() {
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something
went  south!", Toast.LENGTH_SHORT).show();
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onNext(AppInfo appInfo) {
                    mAddedApps.add(appInfo);
                    mAdapter.addApplication(mAddedApps.size()
```

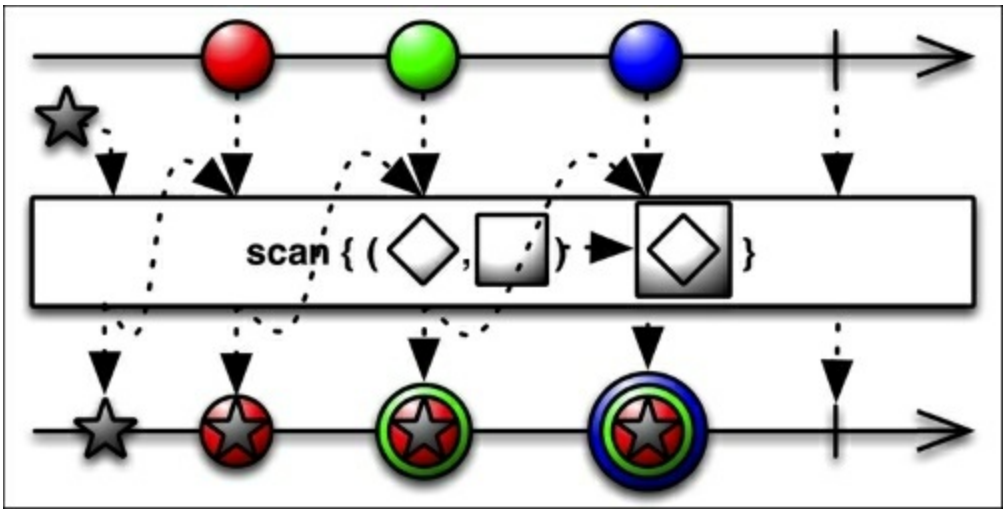
```
- 1,  appInfo);
                    }
                });
}
```

As a result, we have this:



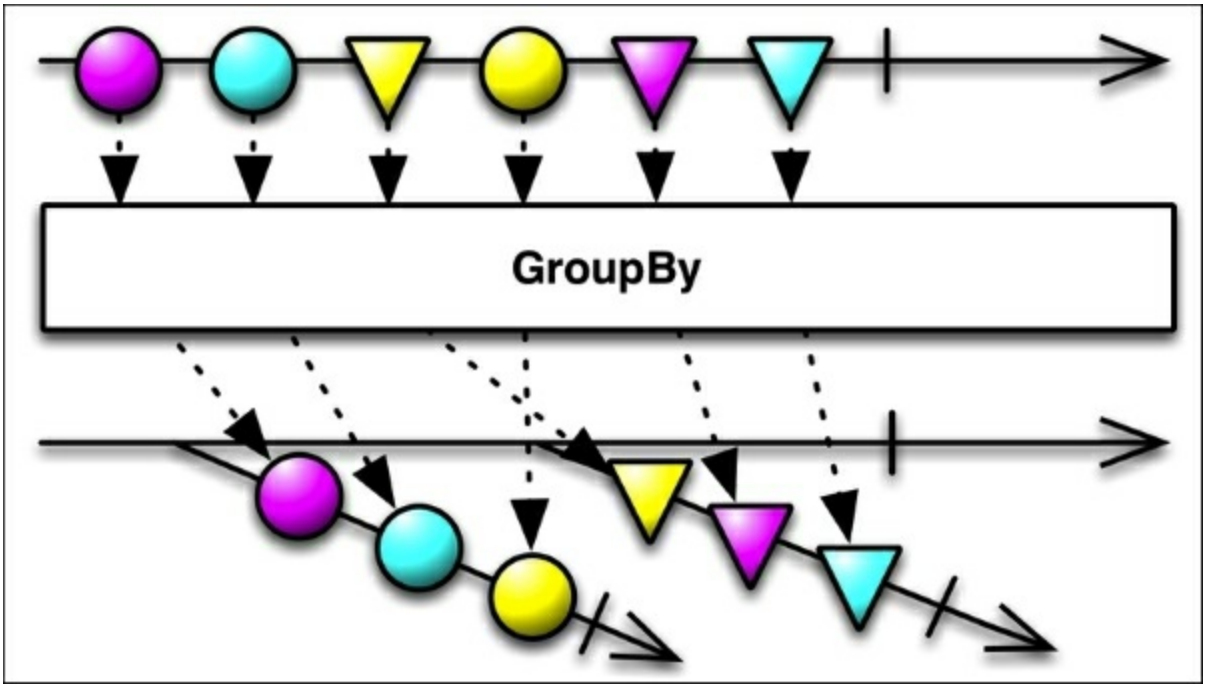
There is also a second variant of `scan()` that takes an initial value to be used with the first emitted value. The method signature looks like this: `scan(R,`

`Func2)`. As a graphical example, it acts like this:



GroupBy

From the outset of the first example, with our installed apps list, the list has been sorted alphabetically. However, what if we want to sort the apps by their last update date? RxJava provides a useful function to group elements from a list according to a specific criteria: `groupBy()`. As a graphical example, the next figure shows how `groupBy()` can group emitted values by their shape:



This function transforms the source Observable into a new Observable, which emits Observables. Each one of these new Observable emits the items of a specific group.

To create a grouped list of out installed apps, we introduce a new element in our `loadList()` function:

```
Observable<GroupedObservable<String, AppInfo>> groupedItems =
Observable.from(apps)
        .groupBy(new Func1<AppInfo, String>() {
            @Override
            public String call(AppInfo appInfo) {
```

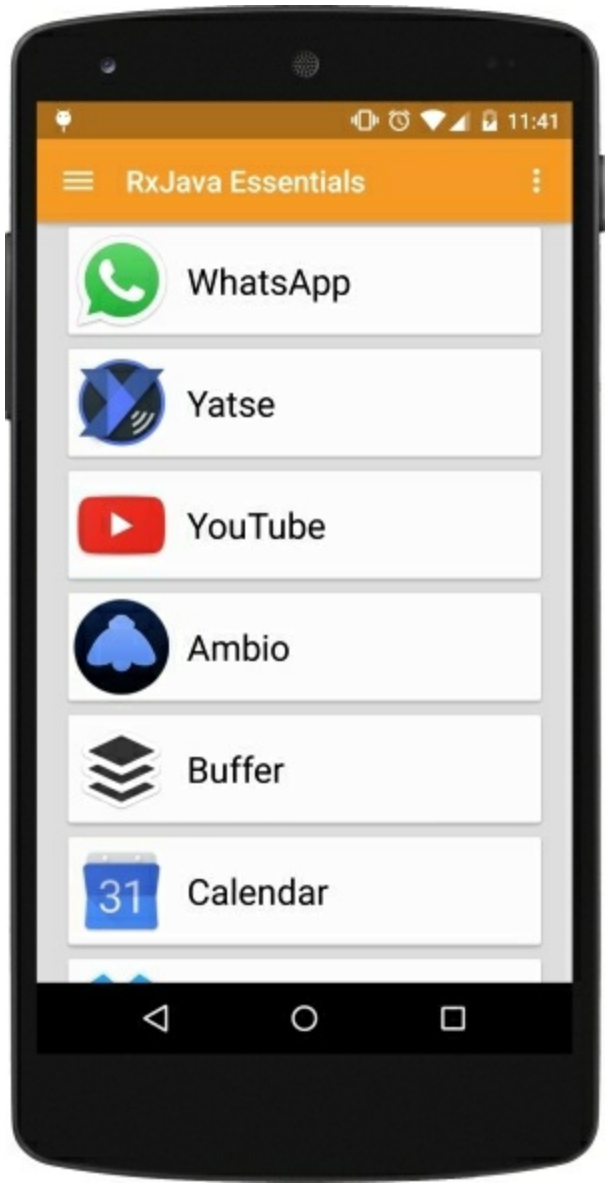
```
                SimpleDateFormat formatter = new
SimpleDateFormat("MM/yyyy");
                return formatter.format(new
Date(appInfo.getLastUpdateTime()));
            }
        });
```

Now we have a new Observable, `groupedItems`, which will emit a sequence of `GroupedObservable` items. `GrouperdObservable` is a particular Observable that comes with a *grouping* key. In our example, the key is `String`, representing the last update date in the *Month/Year* format.

At this point, we have a few Observable emitting `AppInfo` items with which we need to populate our list. We want to preserve the alphabetical sorting order and the grouping order as well. We will create a new Observable that will concatenate all the others, and we will subscribe to it as usual:

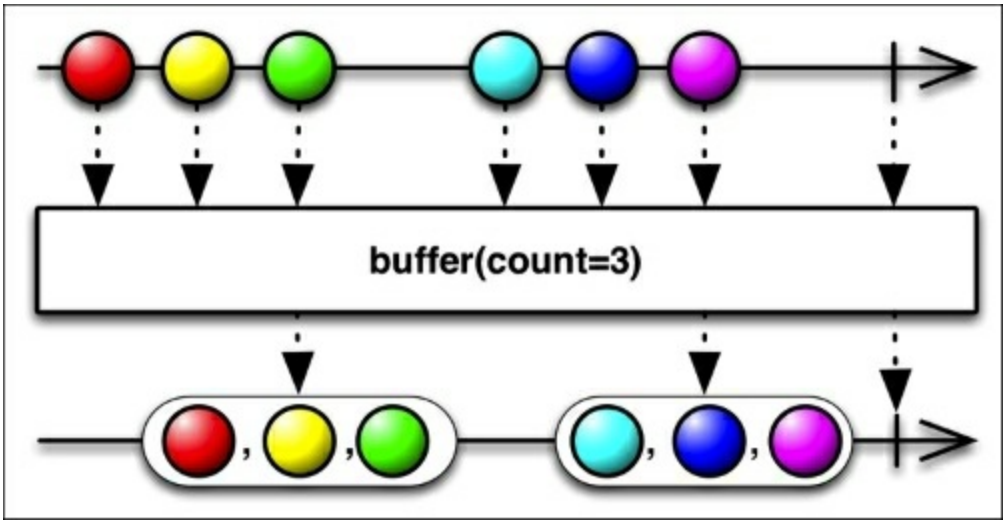
```
Observable
        .concat(groupedItems)
        .subscribe(new Observer<AppInfo>() {
            @Override
            public void onCompleted() {
                mSwipeRefreshLayout.setRefreshing(false);
            }

            @Override
            public void onError(Throwable e) {
                Toast.makeText(getActivity(), "Something went
south!", Toast.LENGTH_SHORT).show();
                mSwipeRefreshLayout.setRefreshing(false);
            }

            @Override
            public void onNext(AppInfo appInfo) {
                mAddedApps.add(appInfo);
                mAdapter.addApplication(mAddedApps.size() - 1,
appInfo);
            }
        });
```

Our `loadList()` function is complete and, as result, we have:

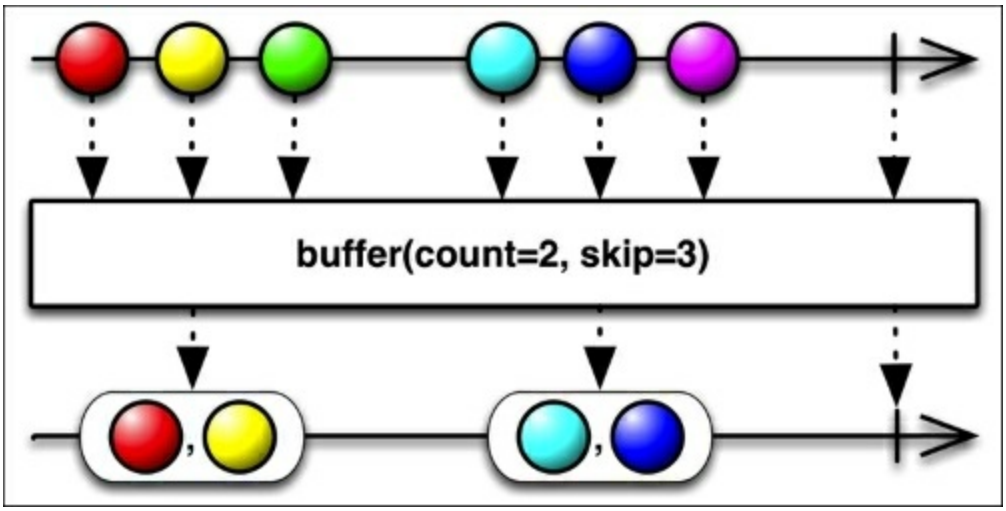

11:41
RxJava Essentials
WhatsApp
Yatse
YouTube
Ambio
Buffer
31
Calendar

Buffer

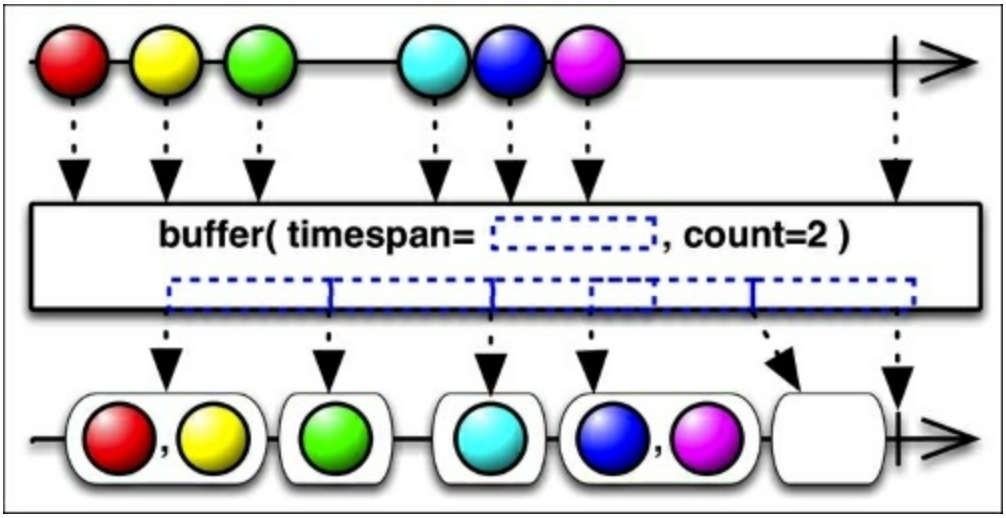
RxJava's `buffer()` function transforms the source Observable into a new Observable, which emits values as a list instead of a single item:



The previous figure shows how `buffer()` takes an integer `count` as a parameter to specify how many items should be included in the emitted list. Indeed, there are a few variations of the `buffer()` function. One of these lets you specify a `skip` value: every `skip` value fills the buffer with `count` elements, as shown in the next figure:

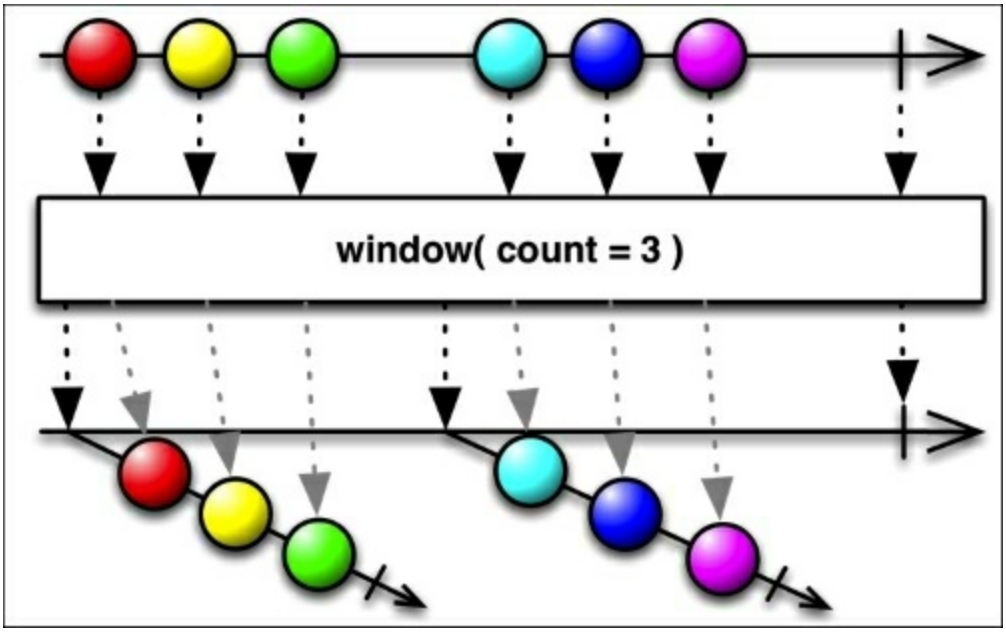


Playing with time, `buffer()` can even take a `timespan` value and create an Observable that emits a list of every elapsed `timespan`:

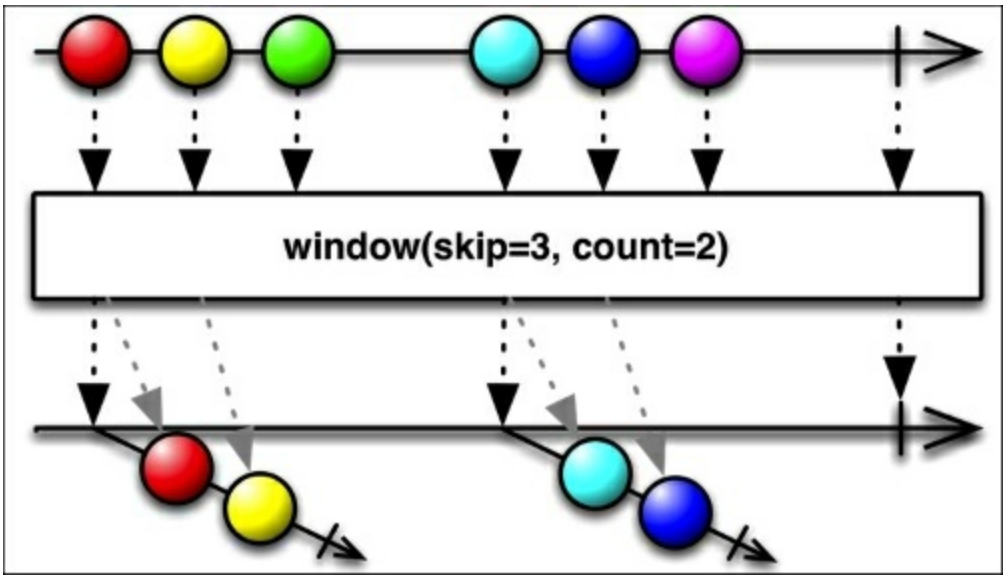


Window

RxJava's `window()` function is similar to `buffer()` but it emits Observables instead of lists. The next figure shows how `window()` will buffer three items and emit them as a new Observable:

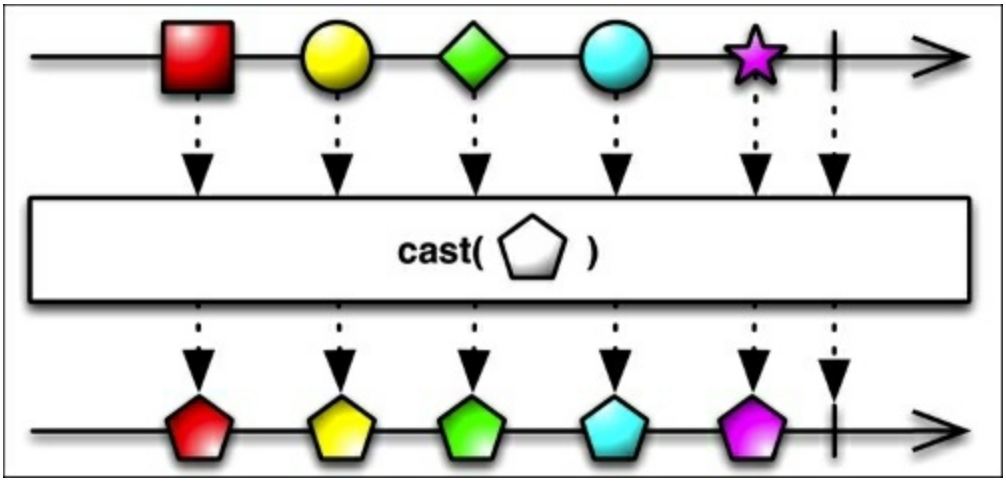


Each Observable emits a subset of the items from the source Observable according to `count`, and then it terminates with a standard `onCompleted()` function. As for `buffer()`, `window()` has a `skip` variation, as shown in the next figure:


window(skip=3, count=2)

Cast

RxJava's `cast()` function is the final operator of this chapter. It is a specialized version of the `map()` operator. It transforms each item from the source Observable into new type, casting it to a different `Class`:



Summary

In this chapter, we learned how RxJava can be used to manipulate and transform Observable sequences. With our current knowledge, we can create, filter and transform any kind of Observable sequence into what we want.

In the next chapter, we will learn how to combine Observables, merging them, joining them, or zipping them.

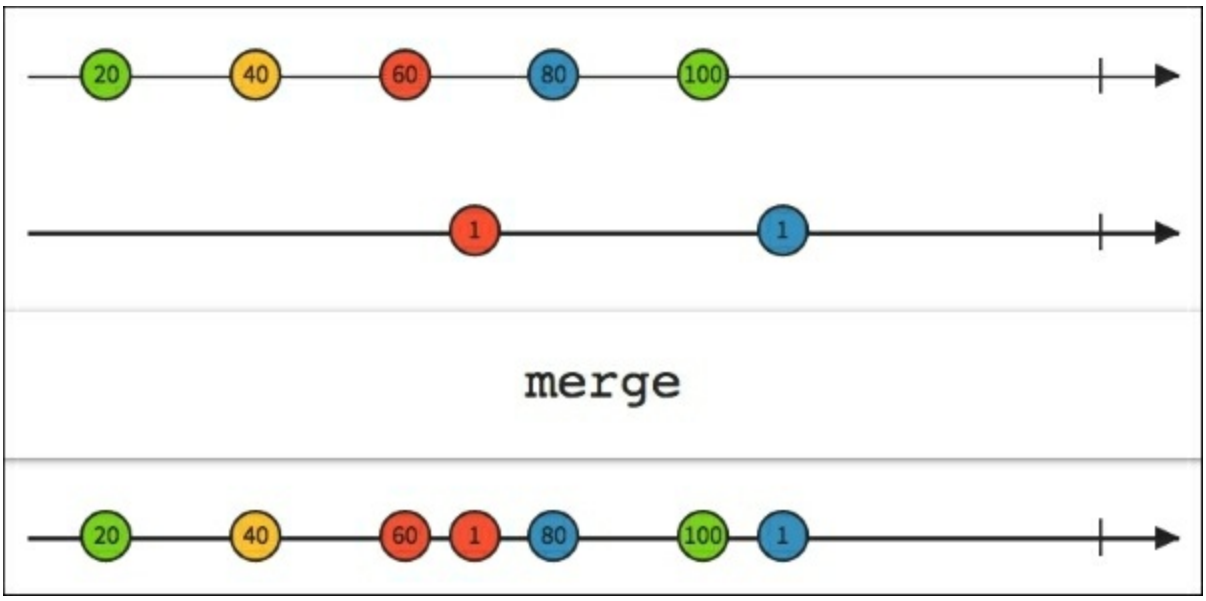
Chapter 6. Combining Observables

In the previous chapter, we learned how to transform `Observable` sequences. We saw a practical example of `map()`, `scan()`, `groupBy()`, and a few more useful functions that will help us manipulate an Observable to create the final Observable we want.

In this chapter, we are going to dig into the combining functions and will learn how to work with multiple Observables at the same time to create the Observable that we want.

Merge

Living in an asynchronous world often creates scenarios in which we have multiple sources but we want to provide only one fruition point: multiple input, single output. RxJava `merge()` helps you to combine two or more Observables, merging their emitted items. The following figure gives you an example of merging two sequences in one final emitting Observable:



As you can see, the emitted items are perfectly merged and interleaved in one final Observable. Note that if you synchronously merge Observables, they are concatenated and not interleaved.

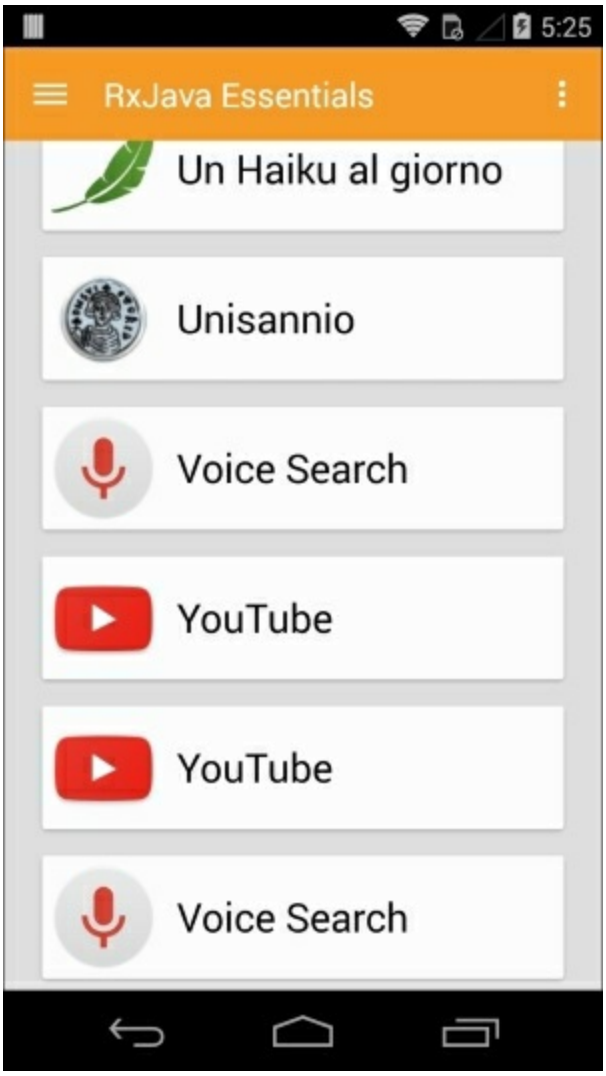
As usual, we are going to create a real-world example using our app and our well-known installed apps list. We are going to need a second Observable for this. We can create a separate installed apps list and we can reverse it. Again, there is no real practical meaning in this, but it's just for the sake of the example. With the second list, our `loadList()` function will be like this:

```
private void loadList(List<AppInfo> apps) {
mRecyclerView.setVisibility(View.VISIBLE);
```

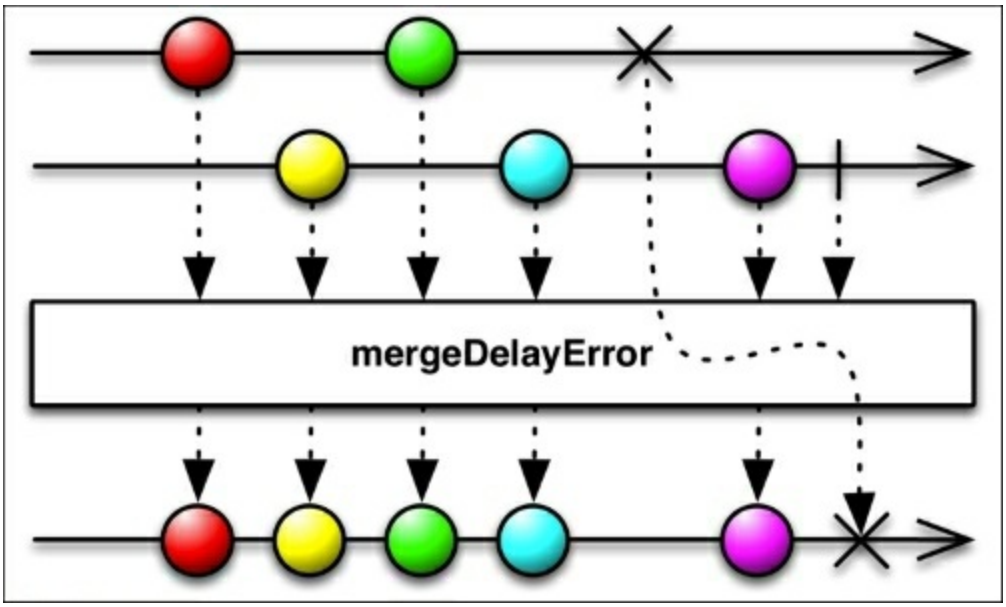
```
    List reversedApps = Lists.reverse(apps);

    Observable<AppInfo> observableApps =
Observable.from(apps);
    Observable<AppInfo> observableReversedApps =
Observable.from(reversedApps);

    Observable<AppInfo> mergedObserbable =
Observable.merge(observableApps, observableReversedApps);

mergedObserbable.subscribe(new Observer<AppInfo>() {
@Override
public void onCompleted() {
mSwipeRefreshLayout.setRefreshing(false);
Toast.makeText(getActivity(), "Here is the list!",
Toast.LENGTH_LONG).show();
        }

@Override
public void onError(Throwable e) {
Toast.makeText(getActivity(), "One of the two Observable threw
an error!",  Toast.LENGTH_SHORT).show();
mSwipeRefreshLayout.setRefreshing(false);
        }

@Override
public void onNext(AppInfoappInfo) {
mAddedApps.add(appInfo);
mAdapter.addApplication(mAddedApps.size() - 1, appInfo);
        }
    });
}
```

We have our usual `Observable` and `observableApps` items and our new `observableReversedApps` reversed list. Using `Observable.merge()`, we can create new `Observable mergedObservable` that will emit all the items a emitted by our source Observables in one single Observable sequence.

As you can see, every method signature stays the same, so our Observer won't notice any difference and we can reuse the code. As result, we have this:

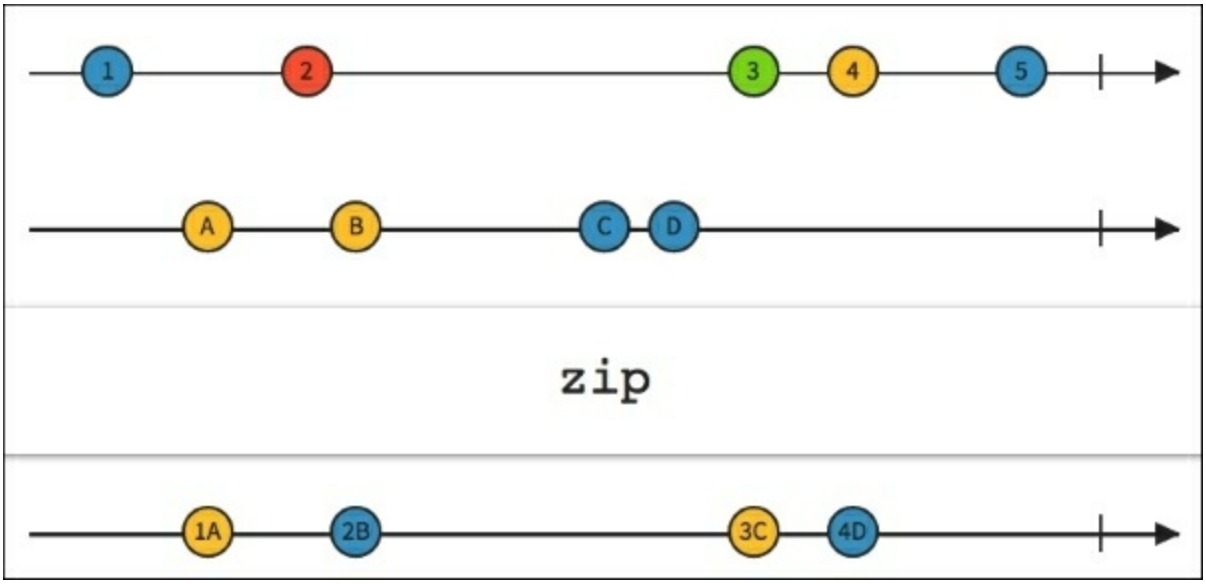


Paying attention to the `Toast` message in the `onError()` message, you can guess that every Observable that throws an error will break the merging. If you need to avoid this scenario, RxJava provides `mergeDelayError()`, which will continue to emit the item from an Observable even if the other one threw an error. When all the still-working Observables have finished, `mergeDelayError()` will emit `onError()`, as shown in the following figure:


mergeDelayError

Zip

Dealing with multiple sources brings in a new possible scenario: receiving data from multiple Observables, processing them, and making them available as a new Observable sequence. RxJava has a specific method to accomplish this: `zip()` combines the value emitted by two or more Observables, transforms them according to a specified function `Func*`, and emits a new value. The following figure shows how the `zip()` method processes the emitted "numbers" and "letters" and makes them available as a new item:



For our real-world example, we are going to use our installed apps list and a new *dynamic* Observable to make the example a bit spicy:

```
Observable<Long> tictoc = Observable.interval(1,
TimeUnit.SECONDS);
```

The `tictoc Observable` variable uses the `interval()` function to generate a `Long` item every second: simple but effective. As stated previously, we are going to need a `Func` object. It will be `Func2` because we are going to pass two parameters to it:

```
private AppInfo updateTitle(AppInfoappInfo, Long time) {
```

```
appInfo.setName(time + " " + appInfo.getName());
return appInfo;
}
```

Now, our `loadList()` function will look like this:

```
private void loadList(List<AppInfo> apps) {
mRecyclerView.setVisibility(View.VISIBLE);

    Observable<AppInfo> observableApp = Observable.from(apps);

    Observable<Long> tictoc = Observable.interval(1,
TimeUnit.SECONDS);

    Observable
            .zip(observableApp, tictoc, (AppInfoappInfo, Long
time) ->updateTitle(appInfo, time))
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Observer<AppInfo>() {
@Override
public void onCompleted() {
Toast.makeText(getActivity(), "Here is the list!",
Toast.LENGTH_LONG).show();
                }

@Override
public void onError(Throwable e) {
mSwipeRefreshLayout.setRefreshing(false);
Toast.makeText(getActivity(), "Something went wrong!",
Toast.LENGTH_SHORT).show();
                }

@Override
public void onNext(AppInfoappInfo) {
if (mSwipeRefreshLayout.isRefreshing()) {
mSwipeRefreshLayout.setRefreshing(false);
                    }
mAddedApps.add(appInfo);
intposition = mAddedApps.size() - 1;
mAdapter.addApplication(position, appInfo);
mRecyclerView.smoothScrollToPosition(position);
                }
            });
}
```

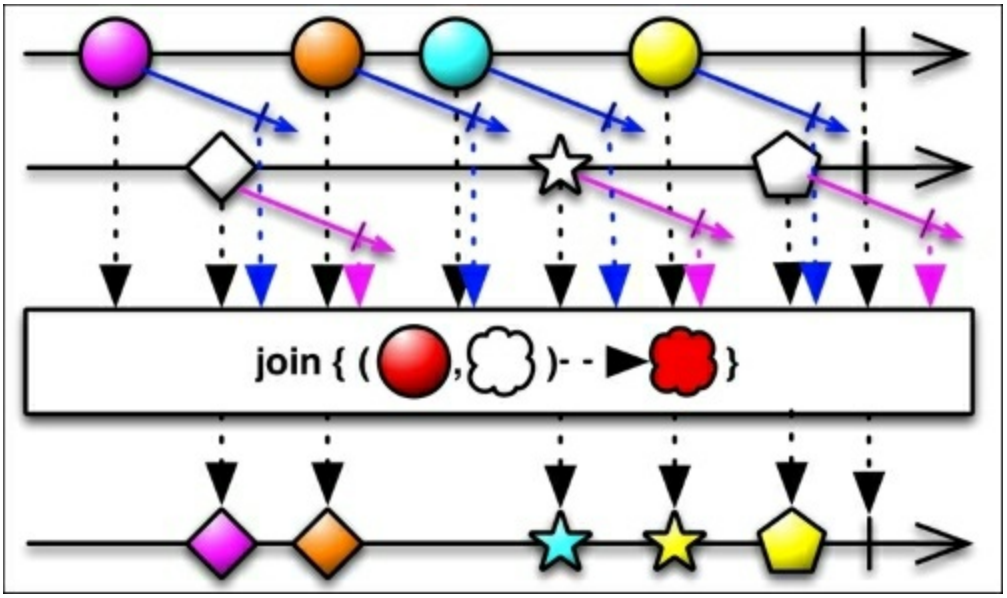
As you can see, `zip()` has three parameters: the two Observables and `Func2`, as expected.

A closer look will reveal the `observeOn()` function. It will be explained in the next chapter: let's say it is a leap of faith for now.

As a result, we have:

Join

The two previous methods, `zip()` and `merge()`, work in the *domain* of the emitted items. There are scenarios in which we have to also consider the time before deciding how to operate on values. RxJava's `join()` function combines items from two Observables, working with time windows.



To properly understand the previous figure, let's explain which parameters `join()` takes:

- The second Observable to combine with the source Observable
- A `Func1` parameter that returns an Observable that specifies a time span defining the time window during which the item emitted by the source Observable will interact with the items from the second Observable
- A `Func1` parameter that returns an Observable that specifies a time span defining the time window during which the item emitted by the second Observable will interact with the items from the source Observable
- A `Func2` parameter that defines how the emitted items will be combined together to emit a new item

As a practical example, we can modify our `loadList()` function like this:

```
private void loadList(List<AppInfo> apps) {
mRecyclerView.setVisibility(View.VISIBLE);

    Observable<AppInfo> appsSequence =
Observable.interval(1000, TimeUnit.MILLISECONDS).map(position
-> {
return apps.get(position.intValue());
            });
    Observable<Long> tictoc = Observable.interval(1000,
TimeUnit.MILLISECONDS);

appsSequence
            .join(
tictoc, appInfo ->Observable.timer(2, TimeUnit.SECONDS),time -
>Observable.timer(0, TimeUnit.SECONDS),this::updateTitle)
            .observeOn(AndroidSchedulers.mainThread())
            .take(10)
            .subscribe(new Observer<AppInfo>() {
@Override
public void onCompleted() {
Toast.makeText(getActivity(), "Here is the list!",
Toast.LENGTH_LONG).show();
                }

@Override
public void onError(Throwable e) {
mSwipeRefreshLayout.setRefreshing(false);
Toast.makeText(getActivity(), "Something went wrong!",
Toast.LENGTH_SHORT).show();
                }

@Override
public void onNext(AppInfoappInfo) {
if (mSwipeRefreshLayout.isRefreshing()) {
mSwipeRefreshLayout.setRefreshing(false);
                    }
mAddedApps.add(appInfo);
intposition = mAddedApps.size() - 1;
mAdapter.addApplication(position, appInfo);
mRecyclerView.smoothScrollToPosition(position);
                }
            });
}
```

We have a new player on the field: `appsSequence`. This is an Observable

sequence emitting apps from our installed apps list every second. The `tictoc Observable` item is just emitting a new `Long` item every second. Joining them, we specify two `Func1` variables:

```
appInfo -> Observable.timer(2, TimeUnit.SECONDS)
```

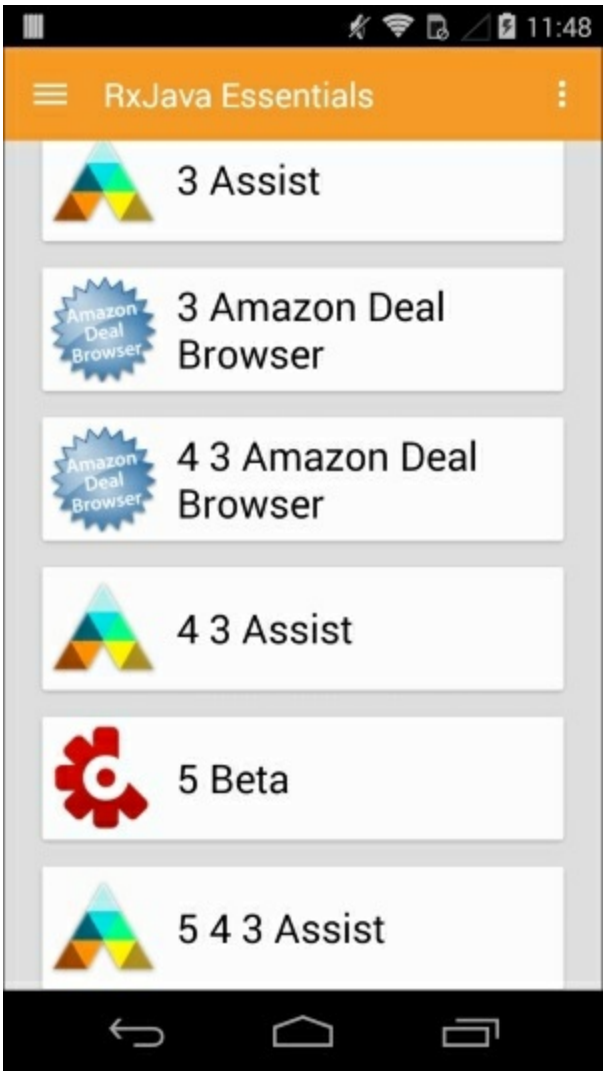
and

```
time -> Observable.timer(0, TimeUnit.SECONDS)
```

These describe the two time windows. The following line describes how we are going to combine the emitted values using a `Func2` variable:

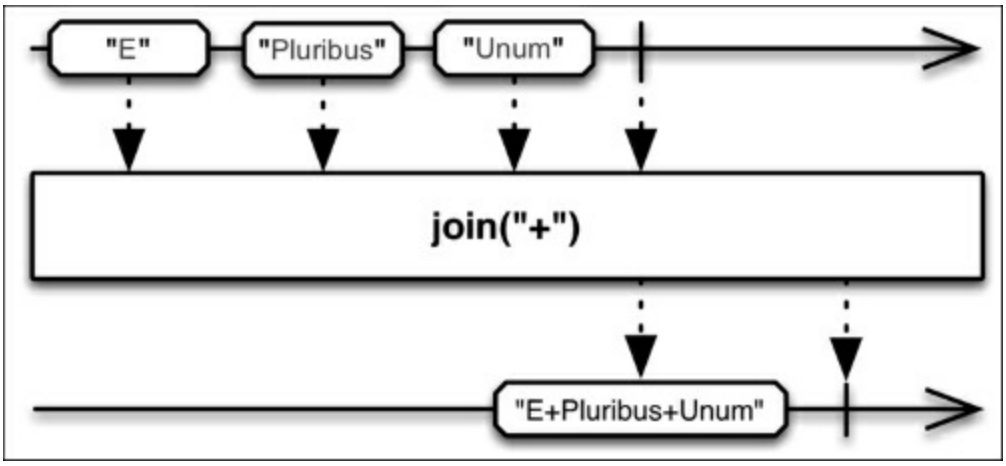
```
this::updateTitle
```

As a result, we have:



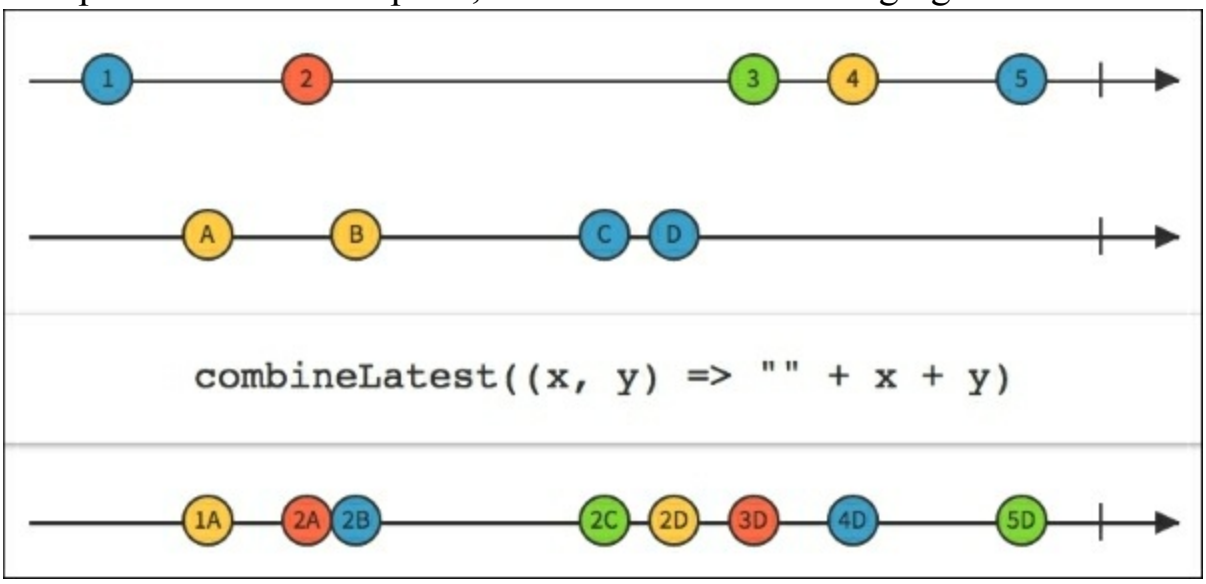
It looks a bit messy, but paying attention to the apps' names and the time windows we specified, we can see what's going on: we are combining the second item with the source item every time the second item is emitted, but we are using the same source item for 2 seconds. This is why the titles repeat themselves and the numbers add up.

It's worth mentioning, to lighten up the situation, that there is also a `join()` operator that works on strings and simply joins the emitted strings into one final string:


"E"
"Pluribus"
"Unum"
join("+")
"E+Pluribus+Unum"

combineLatest

RxJava's `combineLatest()` acts like a particular type of `zip()`. As we already learned, `zip()` works on the latest unzipped item of the two Observables. Instead, `combineLatest()` works on the latest emitted items: if `Observable1` emits *A* and `Observable2` emits *B* and *C*, `combineLatest()` will process *AB* and *AC* pairs, as shown in the following figure:



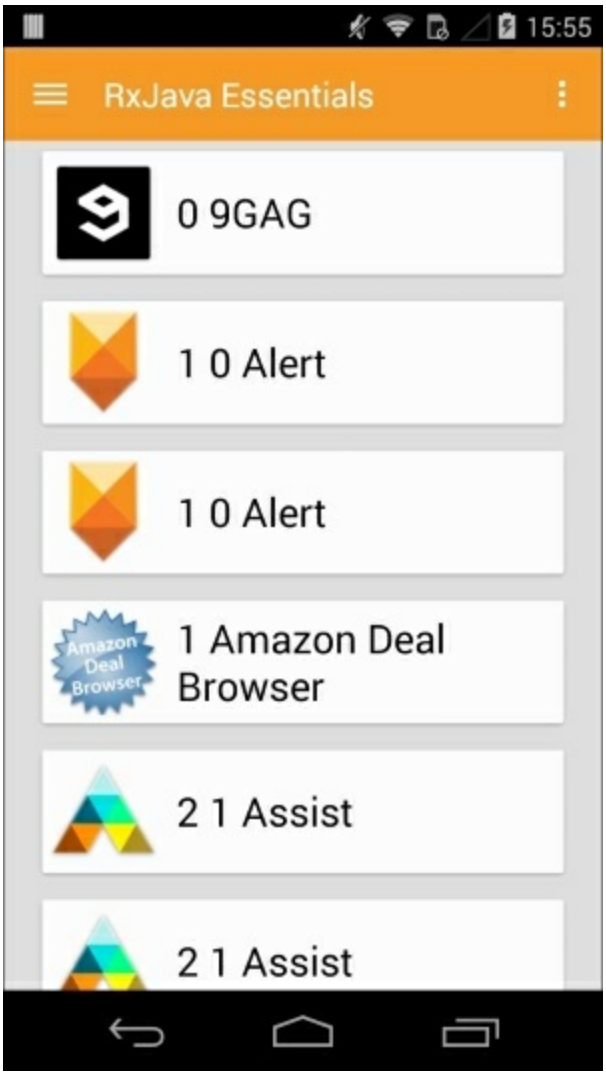
The `combineLatest()` function takes up to nine Observables as parameters, or even a list of Observables, if needed.

Borrowing the `loadList()` function from the previous example, we can modify it to achieve a real-world example for `combineLatest()`:

```
private void loadList(List<AppInfo> apps) {
mRecyclerView.setVisibility(View.VISIBLE);

    Observable<AppInfo> appsSequence =
Observable.interval(1000,  TimeUnit.MILLISECONDS)
            .map(position ->apps.get(position.intValue()));

    Observable<Long> tictoc = Observable.interval(1500,
TimeUnit.MILLISECONDS);
```

```
    Observable
            .combineLatest(appsSequence, tictoc,
this::updateTitle)
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Observer<AppInfo>() {
@Override
public void onCompleted() {
Toast.makeText(getActivity(), "Here is the list!",
Toast.LENGTH_LONG).show();
                }
@Override
public void onError(Throwable e) {
mSwipeRefreshLayout.setRefreshing(false);
Toast.makeText(getActivity(), "Something went wrong!",
Toast.LENGTH_SHORT).show();
                }

@Override
public void onNext(AppInfoappInfo) {
if (mSwipeRefreshLayout.isRefreshing()) {
mSwipeRefreshLayout.setRefreshing(false);
                    }
mAddedApps.add(appInfo);
intposition = mAddedApps.size() - 1;
mAdapter.addApplication(position, appInfo);
mRecyclerView.smoothScrollToPosition(position);
                }
            });
}
```

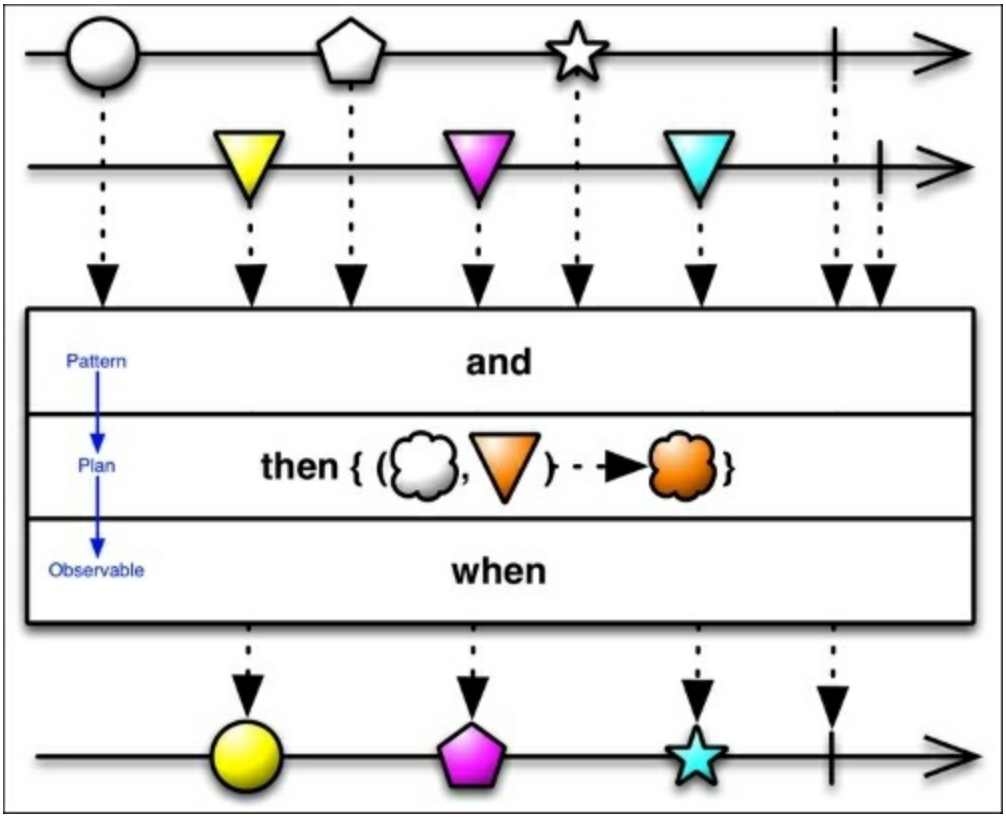
We are using two Observables here: one emits an app from our installed apps list every second, and the second one emits a `Long` item every 1.5 seconds. We combine them and apply the `updateTitle()` function. As result, we obtain:



As you can see, due to the different time intervals, the `AppInfo` object sometimes repeats, as expected.

And, Then, and When

There will be scenarios in your future where `zip()` won't be enough. For complex architecture, or just for personal preference, you could use the And/Then/When solution. This is contained in the *RxJava Joins* package and combines emitted items using structures such as patterns and plans.



Our `loadList()` function will be modified like this:

```
private void loadList(List<AppInfo> apps) {
mRecyclerView.setVisibility(View.VISIBLE);

    Observable<AppInfo> observableApp = Observable.from(apps);

    Observable<Long> tictoc = Observable.interval(1,
TimeUnit.SECONDS);

    Pattern2<AppInfo, Long>pattern =
```

```
JoinObservable.from(observableApp).and(tictoc);
    Plan0<AppInfo> plan = pattern.then(this::updateTitle);
JoinObservable
            .when(plan)
            .toObservable()
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Observer<AppInfo>() {
@Override
public void onCompleted() {
Toast.makeText(getActivity(), "Here is the list!",
Toast.LENGTH_LONG).show();
                }

@Override
public void onError(Throwable e) {
mSwipeRefreshLayout.setRefreshing(false);
Toast.makeText(getActivity(), "Something went wrong!",
Toast.LENGTH_SHORT).show();
                }

@Override
public void onNext(AppInfoappInfo) {
if (mSwipeRefreshLayout.isRefreshing()) {
mSwipeRefreshLayout.setRefreshing(false);
                    }
mAddedApps.add(appInfo);
intposition = mAddedApps.size() - 1;
mAdapter.addApplication(position, appInfo);
mRecyclerView.smoothScrollToPosition(position);
                }
            });
}
```

As usual, we have two emitting sequences, `observableApp`, emitting our installed apps list, and `tictoc` also emitting a `Long` item every second. Now we link the source Observable `and()` the second Observable:

```
JoinObservable.from(observableApp).and(tictoc);
```

This creates a `Pattern` object. Using this `Pattern` object, we can create a `Plan` object: "We have two Observables that are going to emit items, `then()` what?"

```
pattern.then(this::updateTitle);
```

Now, we have a `Plan` object and we can decide what's going to happen when the plan occurs:

```
.when(plan).toObservable()
```

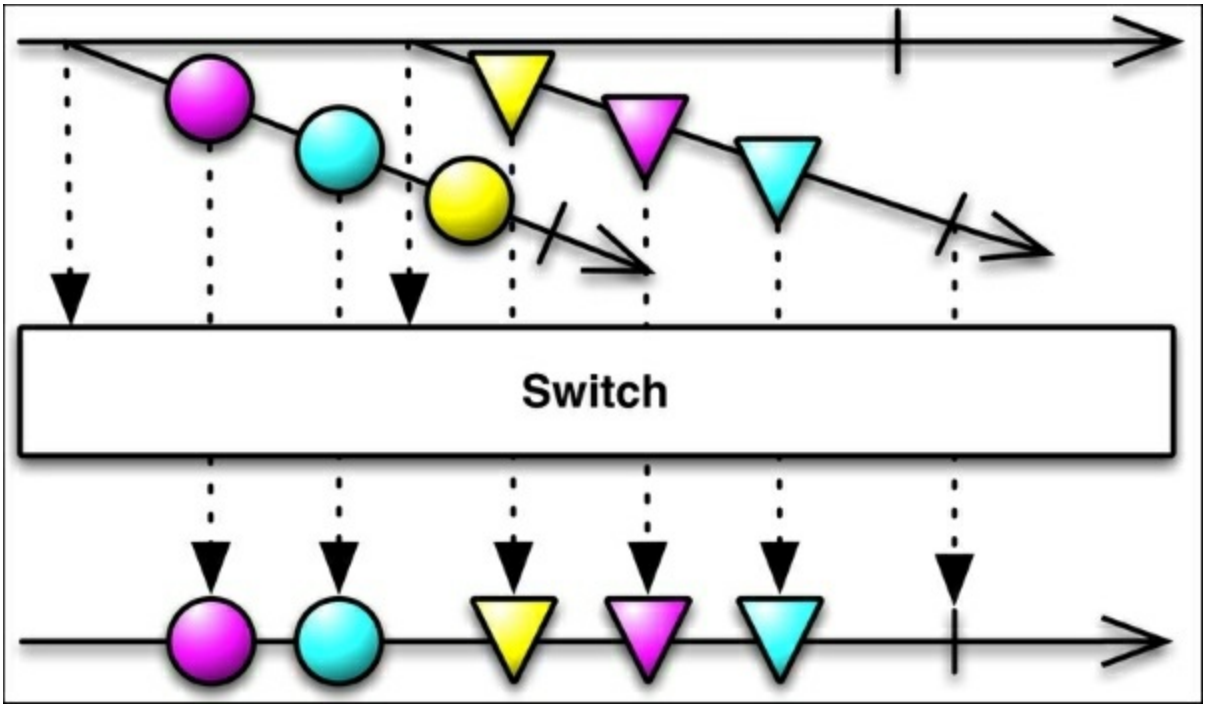
At this point, we can subscribe to the brand new Observable, as we always do.

Switch

There could be complex scenarios wherein we should be able to automatically unsubscribe from an Observable to subscribe to a new one in a continuous *subscribe-unsubscribe* sequence.

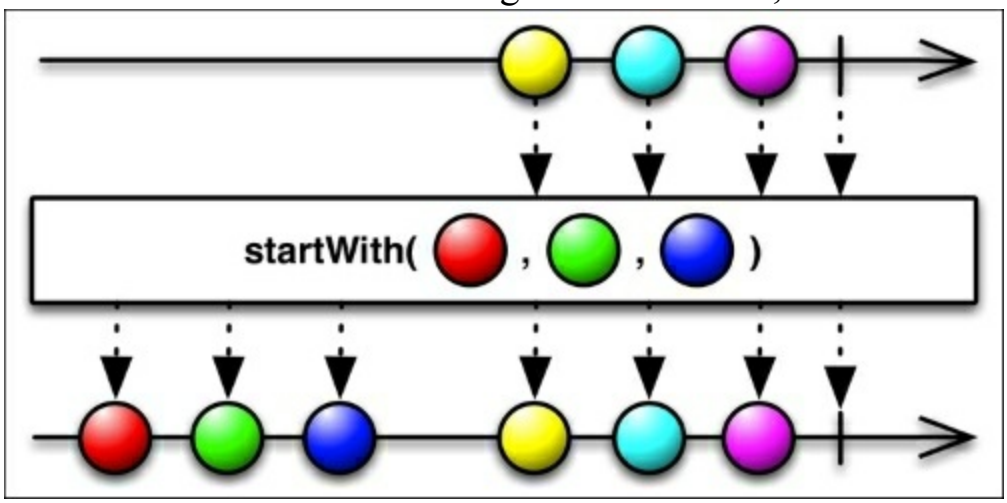
RxJava's `switch()`, as per the definition, transforms an Observable emitting Observables into an Observable emitting the most recent emitted Observable.

Given a source Observable that emits a sequence of Observables, `switch()` subscribes to the source Observable and starts emitting the same items emitted by the first emitted Observable. When the source emits a new Observable, `switch()` immediately unsubscribes from the old Observable (thus interrupting the item flow from it) and subscribes to the new one, starting to emit its items.



StartWith

We have already learned how to concatenate multiple Observables or append specific values to an emitting sequence. RxJava's `startWith()` is the counterpart of `concat()`. As `concat()` appends items to the ones emitted by an Observable, `startWith()` emits a sequence of items, passed as a parameter, before the Observable starts emitting their own items, as shown in the next figure:



Summary

In this chapter, we learned how to combine two or more Observables to create a new Observable sequence. We will now be able to merge Observables, join Observables, zip Observables, and combine them in a few flavors.

In the next chapter, we are going to introduce Schedulers that will help us create multithreading scenarios easily and improve the performance of our apps. We will also learn how to properly perform long tasks or I/O tasks for the best performance.

Chapter 7. Schedulers – Defeating the Android MainThread Issue

The previous chapter was the last chapter about RxJava's Observable creation and manipulation. We learned how to combine two or more Observables, to join them, zip them, merge them, and how to create a new Observable to fit our particular needs.

In this chapter, we will raise the bar and see how to easily deal with multithreading and concurrent programming using RxJava's Schedulers. We will learn how to create network operations, memory accesses, and time-consuming tasks in a reactive way.

StrictMode

To gain information about common issues that could be present in our code base, we are going to enable `StrictMode` in our app.

`StrictMode` helps us detect sensitive activities, such as disk accesses or network calls that we are accidentally performing on the main thread. As you know, performing heavy or long tasks on the main thread is a no-go, because the main thread for Android apps is the UI thread and it should be used only for UI related operations: it's the only way to get smooth animations and a responsive app.

To enable `StrictMode` in our app, we just need to add a few lines to `MainActivity`, and the `onCreate()` methods will look like this:

```
@Override
public void onCreate() {
super.onCreate();


if (BuildConfig.DEBUG) {
        StrictMode.setThreadPolicy(new
StrictMode.ThreadPolicy.Builder()
                .detectAll()
                .penaltyLog()
                                .build());
        StrictMode.setVmPolicy(new
StrictMode.VmPolicy.Builder()
                .detectAll()
                .penaltyLog()
                .build());
    }
}
```

We don't want it enabled all the time, so we limit it only to debug builds. This configuration will report every violation about the main thread usage and every violation concerning possible memory leaks: `Activities`, `BroadcastReceivers`, `Sqlite` objects, and more.

Choosing `penaltyLog()`, `StrictMode` will print a message on logcat when a violation occurs.

Avoiding blocking I/O operations

Blocking operations are those operations that force the app to wait for a result before being able to move on to the next operation. Executing a blocking operation on the UI thread will force the UI to freeze, and this will produce a bad user experience.

After we have enabled `StrictMode`, we start receiving *unpleasant* messages about how our app is doing badly on disk I/O:

```
D/StrictMode: StrictMode policy violation; ~duration=998 ms:
android.os.StrictMode$StrictModeDiskReadViolation: policy=31
violation=2
       at
android.os.StrictMode$AndroidBlockGuardPolicy.onReadFromDisk
(StrictMode.java:1135)
       at libcore.io.BlockGuardOs.open(BlockGuardOs.java:106)
       at libcore.io.IoBridge.open(IoBridge.java:393)
       at java.io.FileOutputStream.<init>
(FileOutputStream.java:88)
       at
android.app.ContextImpl.openFileOutput(ContextImpl.java:918)
       at
android.content.ContextWrapper.openFileOutput(ContextWrapper.
java:185)
       at
com.packtpub.apps.rxjava_essentials.Utils.storeBitmap
(Utils.java:30)
```

The previous message is telling us that our `Utils.storeBitmap()` function is taking `998` ms to complete: that's 1 second of unnecessary work on our UI thread and 1 second of unnecessary slowness of our app. This is happening because we are accessing the disk in a blocking way. Our `storeBitmap()` function contains:

```
FileOutputStream fOut = context.openFileOutput(filename,
Context.MODE_PRIVATE);
```

This is a direct access to the smartphone's solid memory and it's slow. How can we improve this? The `storeBitmap()` function saves the installed app

icon. It returns `void`, so there is no reason to wait until it completes before executing the next operation. We could just launch it and let it execute on a different thread. Thread management in Android has changed during the years and this has led to weird app behaviors. We could use `AsyncTask`, but we saved ourselves from that `onPre… onPost… doInBackGround` hell a few chapters ago. We are going to do it the RxJava way; long live the Schedulers!

Schedulers

Schedulers are the easiest way to bring multi-threading into your apps. They are an important part of RxJava and they play perfectly along with Observables. They provide a handy way to create concurrent routines without having to deal with implementations, synchronizations, threading, platform limitations, and platform changes.

RxJava offers five kinds of Schedulers:

- `.io()`
- `.computation()`
- `.immediate()`
- `.newThread()`
- `.trampoline()`

Let's take a look at them one by one.

Schedulers.io()

This Scheduler is meant for I/O operations. It's based on a thread pool that adjusts its size when necessary, growing or shrinking. This is the Scheduler we are going to use to fix the `StrictMode` violation we saw previously. As it's specific for I/O, it's not the default for any RxJava method; it's up to the developer to use it properly.

It's important to note that with the thread pool unbounded, a large amount of scheduled I/O operations will create a large amount of threads and memory usage. As always, we have to look for the effective balance between performance and simplicity.

Schedulers.computation()

This is the default scheduler for every computational work that is not related to I/O. It's the default for a lot of RxJava methods: `buffer()`, `debounce()`, `delay()`, `interval()`, `sample()`, and `skip()`.

Schedulers.immediate()

This Scheduler allows you to immediately start the specified work on the current thread. It's the default Scheduler for `timeout()`, `timeInterval()`, and `timestamp()`.

Schedulers.newThread()

This Scheduler is what it looks like: it starts a new thread for the specified work.

Schedulers.trampoline()

When we want to execute a task on the current thread, but not immediately, we can queue it with `.trampoline()`. This Scheduler will process its queue and run every queued task sequentially. It's the default Scheduler for `repeat()` and `retry()`.

Nonblocking I/O operations

Now that we know how to schedule a task on an I/O-specific Scheduler, we can modify our `storeBitmap()` function and check the `StrictMode` violation again. For the sake of our example, we can rearrange the code in a new `blockingStoreBitmap()` function:

```
private static void blockingStoreBitmap(Context context,
Bitmap bitmap, String filename) {
    FileOutputStream fOut = null;
try {
        fOut = context.openFileOutput(filename,
Context.MODE_PRIVATE);
        bitmap.compress(Bitmap.CompressFormat.PNG, 100, fOut);
        fOut.flush();
        fOut.close();
    } catch (Exception e) {
        throw new RuntimeException(e);
    } finally {
try {
if (fOut != null) {
    fOut.close();
    }
} catch (IOException e) {
    throw new RuntimeException(e);
  }
 }
}
```

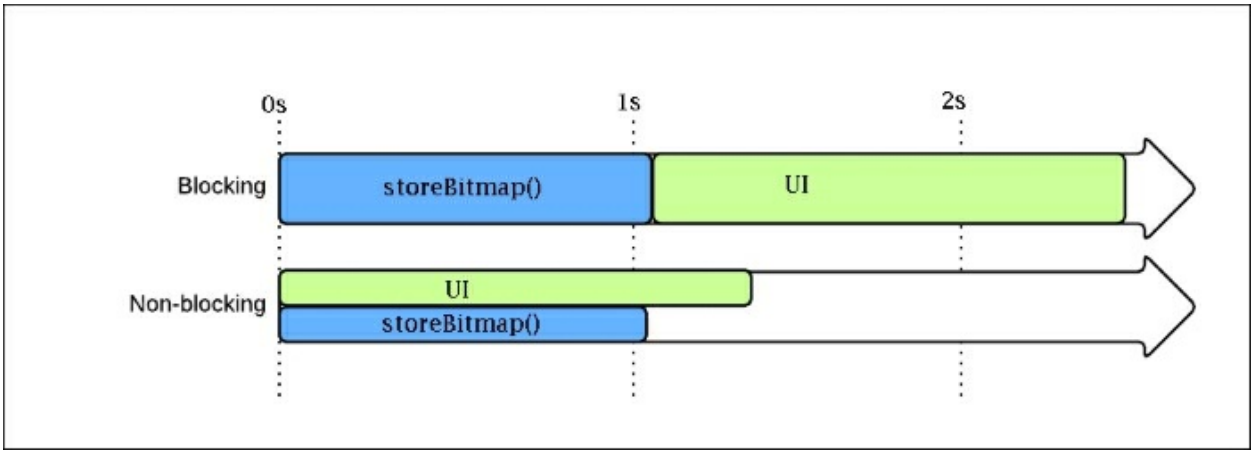
Now, we create our nonblocking version using `Schedulers.io()`:

```
public static void storeBitmap(Context context, Bitmap
bitmap, String filename) {
    Schedulers.io().createWorker().schedule(() -> {
blockingStoreBitmap(context, bitmap, filename);
    });
}
```

Every time we call `storeBitmap()`, RxJava takes care of creating all it needs to execute our task on a specific I/O thread from the I/O thread pool. All the operations are performed away from the UI thread and our app is 1 second

faster than before: no more `StrictMode` violations in the logcat.

The following figure shows the two different approaches we saw in the `storeBitmap()` scenario:



SubscribeOn and ObserveOn

We learned how to run a task on a Scheduler. But how can we use this opportunity to work with Observables? RxJava provides a `subscribeOn()` method that can be used with every `Observable` object. The `subscribeOn()` method takes `Scheduler` as a parameter and takes care of executing the `Observable` call on that Scheduler.

For a real-world example, we will tune up our `loadList()` function. First of all, we need a new `getApps()` method to retrieve our installed apps list:

```
private Observable<AppInfo> getApps() {
return Observable
            .create(subscriber -> {
                List<AppInfo> apps = new ArrayList<>();

                SharedPreferences sharedPref =
getActivity().getPreferences(Context.MODE_PRIVATE);
                Type appInfoType = new
TypeToken<List<AppInfo>>() {
                }.getType();
                String serializedApps =
sharedPref.getString("APPS", "");
if (!"".equals(serializedApps)) {
                    apps = new Gson().fromJson(serializedApps,
appInfoType);
                }

for (AppInfo app : apps) {
                    subscriber.onNext(app);
                }
                subscriber.onCompleted();
            });
}
```

The `getApps()` method returns an Observable of `AppInfo`. It basically reads our installed apps list from Android's `SharedPreferences`, desterilizes it, and starts emitting `AppInfo` items one by one. Using this new method to retrieve our list, `loadList()` needs to be changed like this:

```
private void loadList() {
```

```
mRecyclerView.setVisibility(View.VISIBLE);

    getApps()
            .subscribe(new Observer<AppInfo>() {
@Override
public void onCompleted() {
mSwipeRefreshLayout.setRefreshing(false);
                    Toast.makeText(getActivity(), "Here is the
list!", Toast.LENGTH_LONG).show();
                }

@Override
public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something
went  wrong!", Toast.LENGTH_SHORT).show();
mSwipeRefreshLayout.setRefreshing(false);
                }

@Override
public void onNext(AppInfo appInfo) {
mAddedApps.add(appInfo);
mAdapter.addApplication(mAddedApps.size() - 1, appInfo);
                }
            });
}
```

If we run this code, `StrictMode` will report a violation because `SharedPreferences` are typically slow I/O operations. What we need to do is to specify `getApps()` needs to be executed on which Scheduler:

```
getApps()
        .subscribeOn(Schedulers.io())
        .subscribe(new Observer<AppInfo>() { […]
```

`Schedulers.io()` will get rid of the `StrictMode` violation, but our app is now poorly crashing because of this:

```
at
rx.internal.schedulers.ScheduledAction.run(ScheduledAction.jav
a:58)
            at
java.util.concurrent.Executors$RunnableAdapter.call(Executors.
java:422)
            at
```

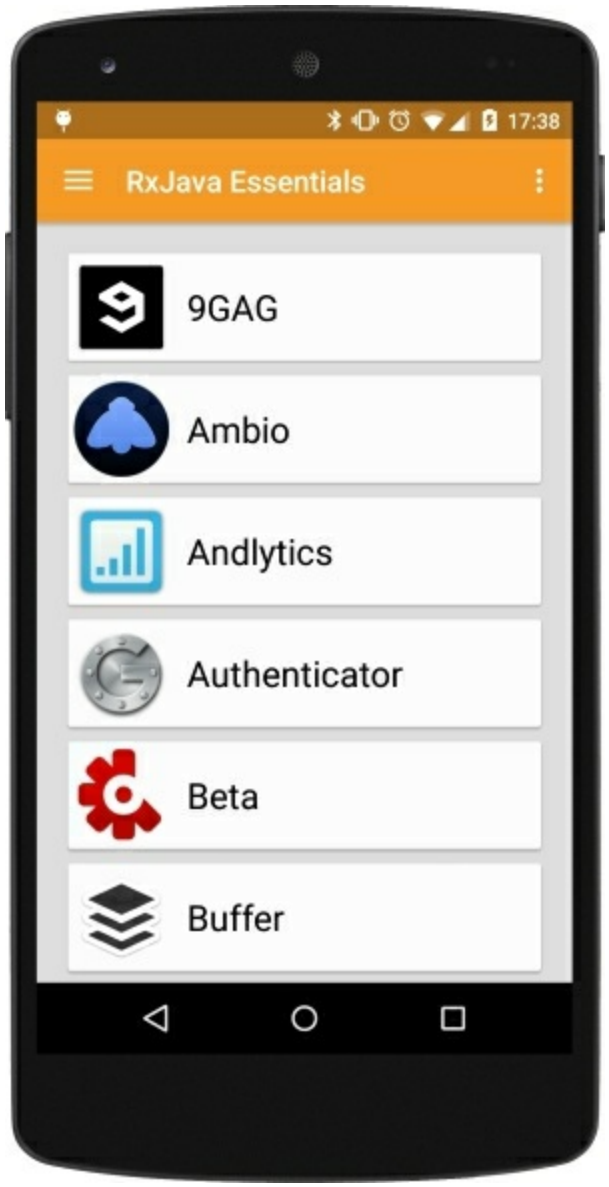
```
java.util.concurrent.FutureTask.run(FutureTask.java:237)
            at
java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutu
reTask.access$201(ScheduledThreadPoolExecutor.java:152)
            at
java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutu
reTask.run(ScheduledThreadPoolExecutor.java:265)
            at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolEx
ecutor.java:1112)
            at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolE
xecutor.java:587)
            at java.lang.Thread.run(Thread.java:841)
     Caused by:
android.view.ViewRootImpl$CalledFromWrongThreadException: Only
the original thread that created a view hierarchy can touch
its views.
```

Only the original thread that created a view hierarchy can touch its views.

We are back to the Android world again. This message is simply telling us that we are trying to modify the UI from a thread that is not the UI thread. It makes sense: we asked to execute the code on the I/O Scheduler. So, basically, we need to execute the code with the I/O Scheduler, but we need to be on the UI thread when the result comes in. RxJava lets you subscribe to a specific Scheduler but also to observe a specific Scheduler. We just need to add a couple of lines to our `loadList()` function, and every piece will be in place:

```
getApps()
.onBackpressureBuffer()
   .subscribeOn(Schedulers.io())
   .observeOn(AndroidSchedulers.mainThread())
   .subscribe(new Observer<AppInfo>() { […]
```

The `observeOn()` method will provide the result on a specific Scheduler: the UI thread in our example. The `onBackpressureBuffer()` method will instruct Observable that if it emits items faster than Observer can consume, it has to store them in a buffer and provide them with the proper timing. After this changes, if we run the app, we have our classic installed apps list:


17:38
RxJava Essentials
9GAG
Ambio
Andlytics
Authenticator
Beta
Buffer

Handling a long task

We already know how to handle slow I/O operations. Let's see an example of a long, slow task that is not I/O-bound. For this example, we will modify our `loadList()` function and create a new `slow` function that will emit our installed apps' items:

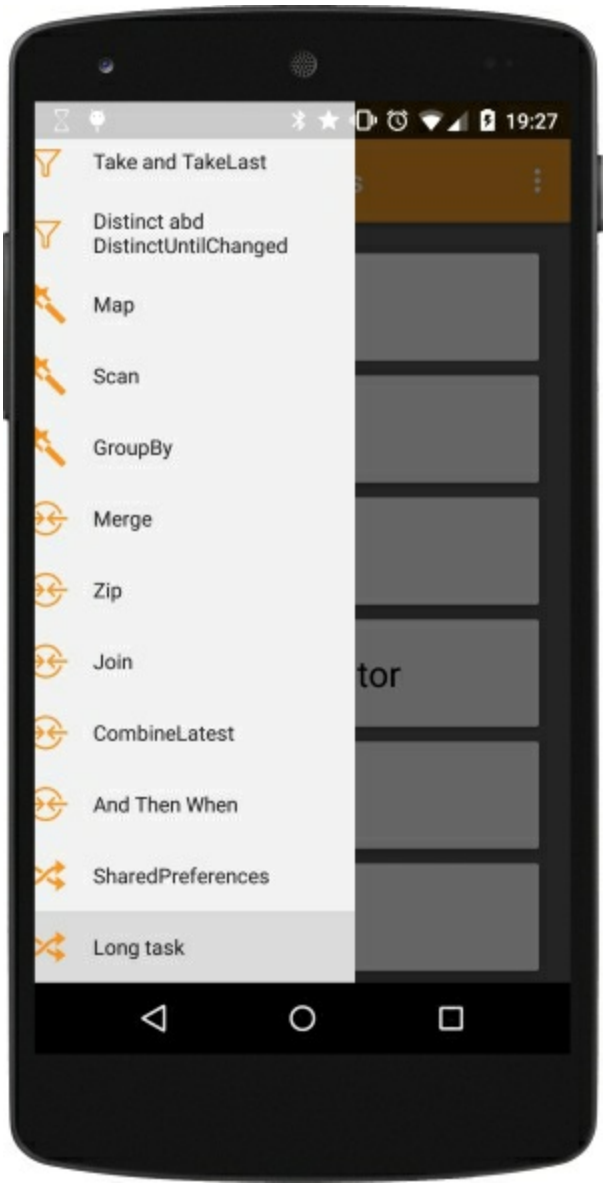
```
private Observable<AppInfo> getObservableApps(List<AppInfo>
apps) {
    return Observable
            .create(subscriber -> {
                for (double i = 0; i < 1000000000; i++) {
                    double y = i * i;
                }

                for (AppInfo app : apps) {
                    subscriber.onNext(app);
                }
                subscriber.onCompleted();
            });
}
```

As you can see, this function performs a few nonsensical computations, only to waste time for the sake of our example, and then it emits our `AppInfo` items from the `List<AppInfo>` object. Now, we can rearrange our `loadList()` function like this:

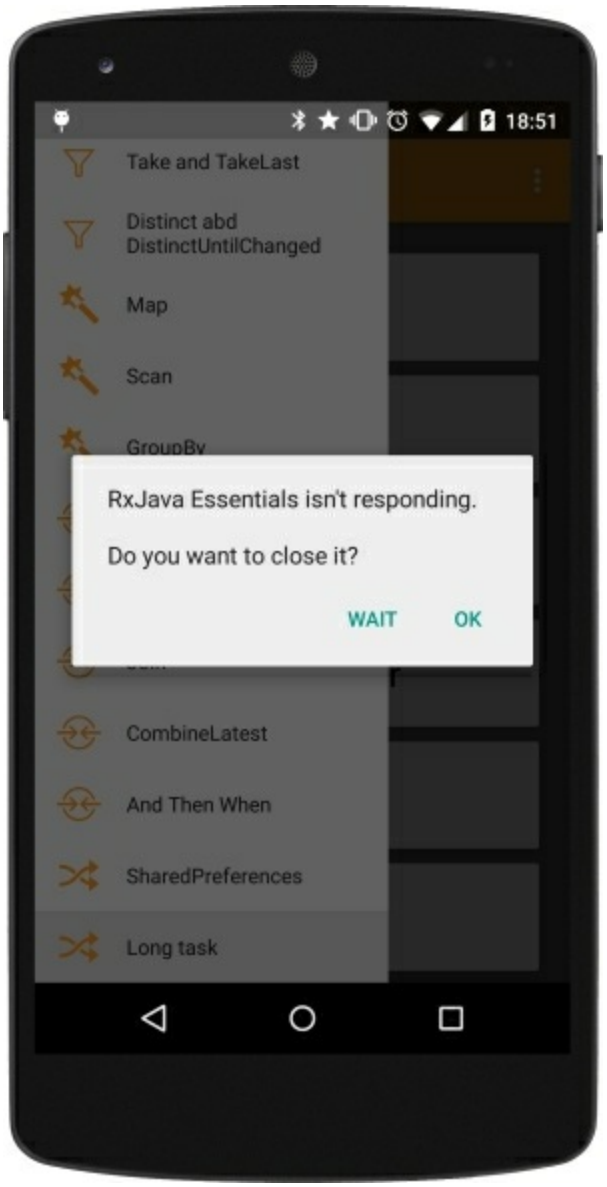
```
private void loadList(List<AppInfo> apps) {
mRecyclerView.setVisibility(View.VISIBLE);

    getObservableApps(apps)
            .subscribe(new Observer<AppInfo>() {
@Override
public void onCompleted() {
mSwipeRefreshLayout.setRefreshing(false);
                    Toast.makeText(getActivity(), "Here is the
list!", Toast.LENGTH_LONG).show();
                }

@Override
public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something
```

```
went  wrong!", Toast.LENGTH_SHORT).show();
mSwipeRefreshLayout.setRefreshing(false);
                }

@Override
public void onNext(AppInfo appInfo) {
mAddedApps.add(appInfo);
mAdapter.addApplication(mAddedApps.size() - 1, appInfo);
                }
            });
}
```

If we run this code, the app will get stuck the moment we click on the `Navigation Drawer` item, as you can perceive from the half-closed menu in the following figure:



If we are unlucky, we can even see the classic *ANR* message shown in the next figure:



For sure, we will see the following unpleasant message in logcat:

```
I/Choreographer: Skipped 598 frames!  The application may be
doing too much work on its main thread.
```

The message is pretty clear: Android is telling us that the user experience will be poor because we are blocking the UI thread with an unnecessary workload. But we already know how to deal with this: we have Schedulers! We just have to add a couple more lines to our Observable chain to get rid of slowness and

the `Choreographer` messages:

```
getObservableApps(apps)
        .onBackpressureBuffer()
        .subscribeOn(Schedulers.computation())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Observer<AppInfo>() { […]
```

With these few lines, we will have a fast closing `Navigation Drawer`, a nice spinning progress bar, and a slow computational task, which is working on a separate thread and will return the results on the UI thread to let us update our installed apps list.

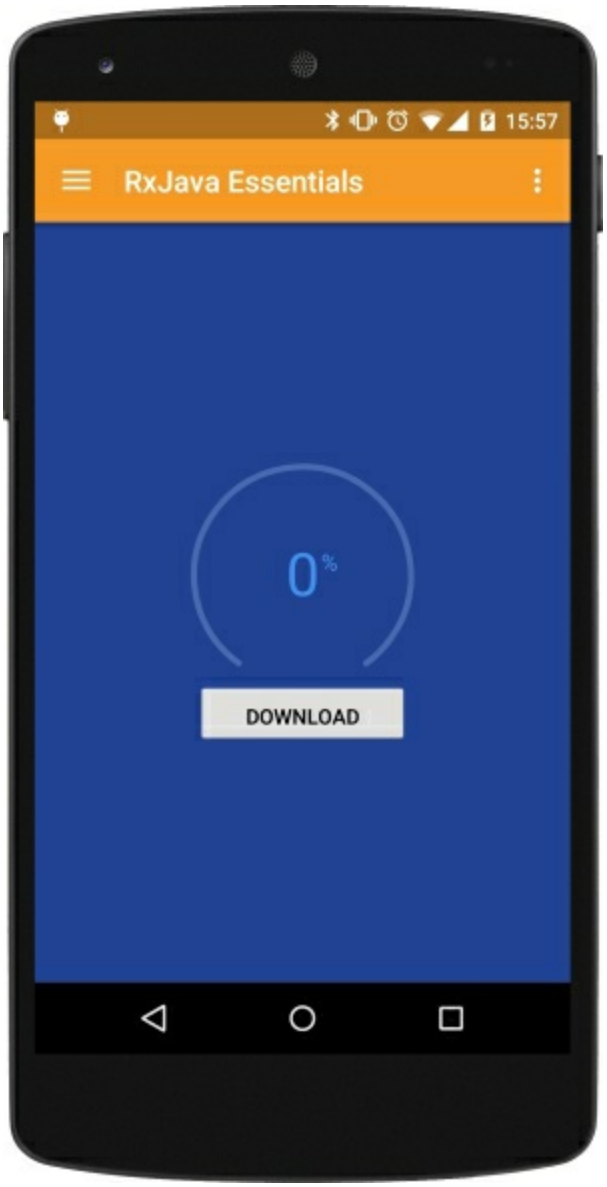
Executing a network task

Networking is part of probably 99 percent of mobile apps nowadays: we always need to connect to a remote server to retrieve the information we need in our app.

As a first approach to networking, we are going to create a new scenario in which we are going to:

- Load a progress bar
- Start a file download using a button
- Update the progress bar during the download
- Start the video player after the download is completed

Our user interface will be very simple. We will just need a fancy progress bar and a **DOWNLOAD** button.



First of all, we will create `mDownloadProgress`:

```
private PublishSubject<Integer>mDownloadProgress =
PublishSubject.create();
```

This is the subject that we are going to use to manage the progress bar updates. This subject works together with the `download` function:

```
private boolean downloadFile(String source, String
destination) {
```

```
boolean result = false;
    InputStream input = null;
    OutputStream output = null;
    HttpURLConnection connection = null;
try {
        URL url = new URL(source);
        connection = (HttpURLConnection) url.openConnection();
        connection.connect();

if (connection.getResponseCode() != HttpURLConnection.HTTP_OK)
{
return false;
        }

int fileLength = connection.getContentLength();

        input = connection.getInputStream();
        output = new FileOutputStream(destination);

byte data[] = new byte[4096];
long total = 0;
int count;
while ((count = input.read(data)) != -1) {
            total += count;

if (fileLength >0) {
int percentage = (int) (total * 100 / fileLength);
mDownloadProgress.onNext(percentage);
            }
            output.write(data, 0, count);
        }
mDownloadProgress.onCompleted();
        result = true;
    } catch (Exception e) {
mDownloadProgress.onError(e);
    } finally {
try {
if (output != null) {
                output.close();
            }
if (input != null) {
                input.close();
            }
        } catch (IOException e) {
mDownloadProgress.onError(e);
```

```
        }

if (connection != null) {
            connection.disconnect();
mDownloadProgress.onCompleted();
        }
    }
return result;
}
```

Using this code as it is will trigger `NetworkOnMainThreadException`. We can easily create a RxJava version of this function and jump into our beloved reactive world to solve this issue:

```
private Observable<Boolean> obserbableDownload(String source,
String destination) {
return Observable.create(subscriber -> {
try {
boolean result = downloadFile(source, destination);
if (result) {
                subscriber.onNext(true);
                subscriber.onCompleted();
            } else {
                subscriber.onError(new Throwable("Download
failed."));
            }
        } catch (Exception e) {
            subscriber.onError(e);
        }
    });
}
```

Now we need to trigger the download, tapping the `download` button:

```
@OnClick(R.id.button_download)
void download() {
mButton.setText(getString(R.string.downloading));
mButton.setClickable(false);

mDownloadProgress
.distinct()
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Observer<Integer>() {
@Override
```

```
public void onCompleted() {
                    App.L.debug("Completed");
                }

@Override
public void onError(Throwable e) {
                    App.L.error(e.toString());
                }

@Override
public void onNext(Integer progress) {
mArcProgress.setProgress(progress);
                }
            });

String destination = "sdcardsoftboy.avi";
obserbableDownload("http://archive.blender.org/fileadmin/movie
s/ softboy.avi", destination)
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(success -> {
resetDownloadButton();
                Intent intent = new
Intent(android.content.Intent.ACTION_VIEW);
                File file = new File(destination);
                intent.setDataAndType(Uri.fromFile(file),
"video/avi");

intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
                startActivity(intent);
            }, error -> {
                Toast.makeText(getActivity(), "Something went
south", Toast.LENGTH_SHORT).show();
resetDownloadButton();
            });
}
```

We are using the Butter Knife annotation's `@OnClick` to bind the button to the method, and we are updating the button message and its clickable status: we don't want the user to trigger multiple downloads with multiple clicks.

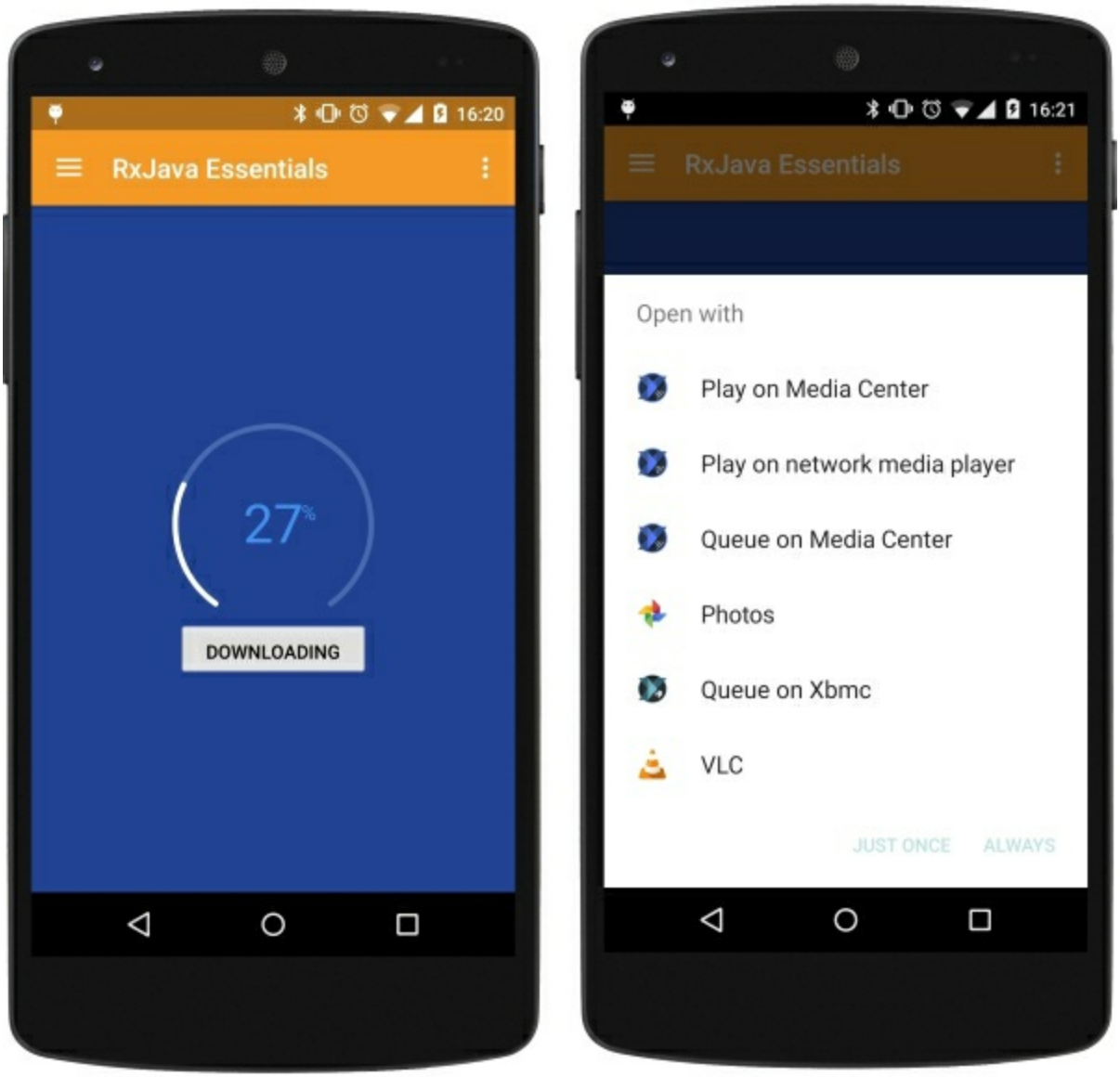
Then, we create a new subscription to observe the download progress and update the progress bar accordingly. Obviously, we are observing on the main

thread because the progress bar is a UI element:

```
obserbableDownload("http://archive.blender.org/fileadmin/movie
s/ softboy.avi", "sdcardsoftboy.avi";)
```

This is the download Observable. A network call is an I/O job and we are using the I/O Scheduler, as expected. When the download is complete, we are in `onNext()` and we can launch the video player, knowing that the downloaded file will be available to the player at the destination URI.

The following figure shows the download progress and the video player dialog:


16:20
RxJava Essentials
27%
DOWNLOADING
16:21
RxJava Essentials
Open with
Play on Media Center
Play on network media player
Queue on Media Center
Photos
Queue on Xbmc
VLC
JUST ONCE
ALWAYS

Summary

In this chapter, we learned how to easily bring multithreading to our apps. RxJava provides an extremely useful tool for this: Schedulers. Schedulers come with a variety of specific optimized scenarios and they free us from `StrictMode` violations and blocking I/O functions. We can now approach memory access and network access in an easy, reactive way and keep a consistent approach all over the app.

In the next chapter, we will raise the stakes and create a real-world app that fetches data from different remote sources using the REST API library by Square, Retrofit, and to create a complex material design UI.

Chapter 8. REST in Peace – RxJava and Retrofit

In the previous chapter, we learned how to operate on threads that are different than the UI thread using Schedulers. We learned how to effectively run I/O tasks without blocking the UI and how to run long, computational tasks without losing out on the app performance. In this final chapter, we are going to create a definitive real-world example, map remote APIs with Retrofit, and query them asynchronously to create a rich UI with zero effort.

The project goal

We are going to create a new `Activity` item in our existing example app. This `Activity` item will retrieve the 10 most popular users from *Stack Overflow* using the `StackExchange` API. Using this information, the app will show a list of users with the picture, name, reputation number, and city of living. For every user, the app will retrieve the current weather forecast using the city of residence and the OpenWeatherMap API, and it will show a tiny weather icon. Based on the information retrieved from *Stack Overflow*, the app will provide an `onClick` event for every user on the list, and it will open their personal website for users that have specified a personal website in their profile, or it will open the *Stack Overflow* user profile.

Retrofit

Retrofit is a type-safe REST client for Android and Java by *Square*. It helps you easily communicate with any REST API. It integrates perfectly with RxJava: all the JSON response objects are mapped to plain old Java objects and all the network calls are based on RxJava Observable of these objects.

Using the API documentation, we can define what JSON response we will receive from the server. To easily map this JSON response to our Java code, we will use **jsonschema2pojo** (http://www.jsonschema2pojo.org). This handy service will generate all the Java classes we need to map the JSON response.

When we have all the Java models in place, we can start setting up Retrofit. Retrofit uses standard Java interfaces to map the API routes. For our example, we are going to use just a route from the API, and our Retrofit `interface` class will be the following:

```
public interface StackExchangeService {
    @GET("2.2users?
order=desc&sort=reputation&site=stackoverflow")Observable<User
sResponse> getMostPopularSOusers(@Query("pagesize") int
howmany);
}
```

This `interface` class contains only one method, for now: `getMostPopularSOusers`. This method takes integer `howmany` as a parameter and returns an Observable of `UsersResponse`.

When we have `interface`, we can create the `RestAdapter` class. We obviously want to organize our code properly, and we will create a `SeApiManager` function to provide a proper way to interact with the StackExchange API:

```
public class SeApiManager {

    private final StackExchangeService mStackExchangeService;

    public SeApiManager() {
        RestAdapter restAdapter = new RestAdapter.Builder()
```

```
                .setEndpoint("https://api.stackexchange.com")
                .setLogLevel(RestAdapter.LogLevel.BASIC)
                .build();

        mStackExchangeService =
restAdapter.create(StackExchangeService.class);
    }

    public Observable<List<User>> getMostPopularSOusers(int
howmany) {
        return mStackExchangeService
              .getMostPopularSOusers(howmany)
                .map(UsersResponse::getUsers)
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread());
    }
}
```

For the simplicity of the example, we will not manage this class as it should be managed: `Singleton`. Using a dependency injection solution, such as **Dagger2**, would bring the code quality even higher.

Creating the `RestAdapter` class, we will set up a few important aspects of our API client. In this example, we are setting the **endpoint** and the **log level**. The endpoint URL is hardcoded only for the sake of the example. It's important to use an external resource to store data like this. Avoiding hardcoded strings in the code is a good practice.

The Retrofit setup ends binding our `RestAdapter` class and our API Interface. This will give us an object that we can use to query the API. We can choose to directly expose this object, or wrap it somehow, to limit access to it. In this example, we are wrapping it and exposing only `getMostPopularSOusers`. This method executes the query, lets Retrofit parse the JSON response, extracts the list of users, and provides it back to the subscriber. As you can see, using Retrofit, RxJava, and Retrolambda, we have almost no boilerplate code: it's very compact and the readability is very high.

Now, we have an API manager that exposes a reactive method, which fetches data from a remote API to an I/O Scheduler, parses it, maps it, and provides a neat users list for our consumers.

The app structure

We are not using any MVC, MVP, or MVVM paradigms because that is not the purpose of this book, so our `Activity` class will contain all the logic we need to create and show our list of users.

Creating the Activity class

We will set up `SwipeRefreshLayout` and `RecyclerView` in our `onCreate()` method; we have a `refreshList()` method to handle the fetching and showing of our list of users and `showRefreshing()` to manage the `ProgressBar` and `RecyclerView` visibility.

Our `refreshList()` function looks like this:

```
private void refreshList() {
    showRefresh(true);
    mSeApiManager.getMostPopularSOusers(10)
            .subscribe(users -> {
                showRefresh(false);
                mAdapter.updateUsers(users);
            }, error -> {
                App.L.error(error.toString());
                showRefresh(false);
            });
}
```

We show `ProgressBar`, and observe the list of users from the StackExchange API manager. The moment the list comes in, we show it and update the `Adapter` content and the `RecyclerView` visibility.

Creating the RecyclerView adapter

After we have obtained the data from the REST API, we need to bind it to the View, populating our layout with an adapter. Our `RecyclerView` adapter is pretty standard. It extends `RecyclerView.Adapter` and specifies its own `ViewHolder`:

```
public static class ViewHolder extends RecyclerView.ViewHolder
{

    @InjectView(R.id.name)
    TextView name;

    @InjectView(R.id.city)
    TextView city;

    @InjectView(R.id.reputation)
    TextView reputation;

    @InjectView(R.id.user_image)
    ImageView user_image;

    public ViewHolder(View view) {
        super(view);
        ButterKnife.inject(this, view);
    }
}
```

The moment we receive the data from the API manager, we can set all the labels on the view: `name`, `city`, and `reputation`.

To display the user image, we are going to use **Universal Image Loader** by Sergey Tarasevich (https://github.com/nostra13/Android-Universal-ImageLoader). UIL is a pretty famous and well-tested image-management library. We could use Picasso by Square, Glide, or Fresco by Facebook as well. It's just a matter of your personal preference. The important thing is not to reinvent the wheel: libraries facilitate developers' lives and allow them to achieve goals faster.

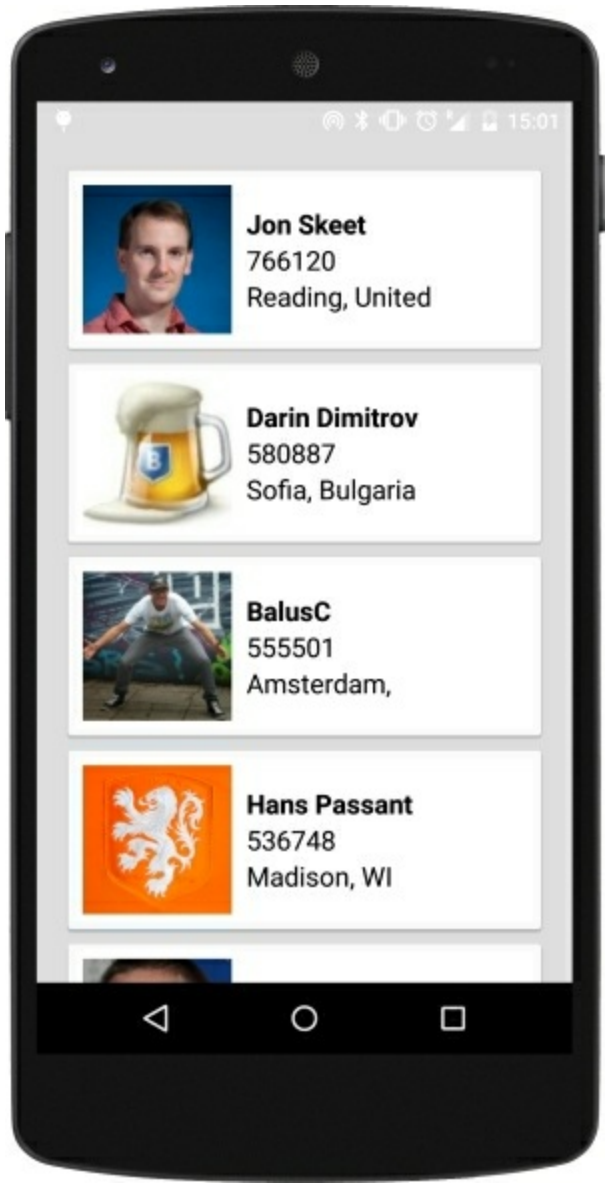
In our adapter, we have:

```
@Override
public void onBindViewHolder(SoAdapter.ViewHolder holder,  int
position) {
    User user = mUsers.get(position);
    holder.setUser(user);
}
```

In `ViewHolder`, we have:

```
public void setUser(User user) {
    name.setText(user.getDisplayName());
    city.setText(user.getLocation());
    reputation.setText(String.valueOf(user.getReputation()));


ImageLoader.getInstance().displayImage(user.getProfileImage(),
  user_image);
}
```

At this point, we can run the code and obtain a list of users, as shown in the following screenshot:



Retrieving the weather forecast

Let's raise the stakes and bring the current city's weather into our list. **OpenWeatherMap** is a handy web service with a public API that we can query to retrieve lots of useful forecast information.

As usual, we will use Retrofit to map the API and access it via RxJava. As for the StackExchange API, we are going to create `interface`, `RestAdapter`, and a handy manager:

```
public interface OpenWeatherMapService {

    @GET("data2.5/weather")
    Observable<WeatherResponse> getForecastByCity(@Query("q")
String city);
}
```

This method will provide the current forecast using the city name as a parameter. We are going to bind this interface to our `RestAdapter` class like this:

```
RestAdapter restAdapter = new RestAdapter.Builder()
        .setEndpoint("http://api.openweathermap.org")
        .setLogLevel(RestAdapter.LogLevel.BASIC)
        .build();
mOpenWeatherMapService =
restAdapter.create(OpenWeatherMapService.class);
```

As before, we are only setting the API endpoint and the log level: the only two things we need right now.

Our `OpenWeatherMapApiManager` class will then provide the method:

```
public Observable<WeatherResponse> getForecastByCity(String
city) {
    return mOpenWeatherMapService
            .getForecastByCity(city)
             .subscribeOn(Schedulers.io())
             .observeOn(AndroidSchedulers.mainThread());
}
```

Right now, we have a list of users, and we can query OpenWeatherMap to receive the weather forecast by the city name. The next step is to modify our `ViewHolder` class to retrieve and use the weather forecast for every user and display a weather icon according to the status.

We first validate the user's profile and get a valid city name using these utility methods:

```
private boolean isCityValid(String location) {
    int separatorPosition = getSeparatorPosition(location);
```

```
    return !"".equals(location) && separatorPosition > -1;
}

private int getSeparatorPosition(String location) {
    int separatorPosition = -1;
    if (location != null) {
        separatorPosition = location.indexOf(",");
    }
    return separatorPosition;
}

private String getCity(String location, int position) {
    if (location != null) {
        return location.substring(0, position);
    } else {
        return "";
    }
}
```

With a valid city name, we can use the following command to obtain all the data we need for the weather:

```
OpenWeatherMapApiManager.getInstance().getForecastByCity(city)
```

Using the weather response, we can obtain the URL of the weather icon with:

```
getWeatherIconUrl(weatherResponse);
```

With the icon URL, we can retrieve the icon bitmap itself:

```
private Observable<Bitmap> loadBitmap(String url) {
    return Observable
            .create(subscriber -> {
                ImageLoader.getInstance().displayImage(url,
city_image, new ImageLoadingListener() {
                    @Override
                    public void onLoadingStarted(String
imageUri,  View view) {

                    }

                    @Override
                    public void onLoadingFailed(String
imageUri,  View view, FailReason failReason) {
```

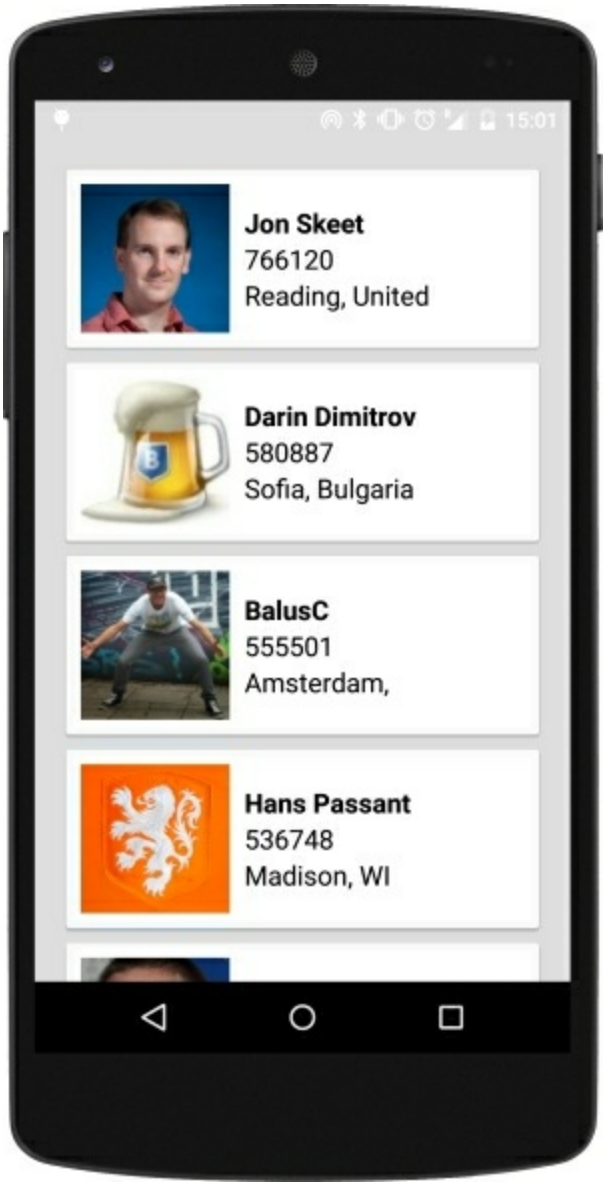
```
subscriber.onError(failReason.getCause());
                    }

                    @Override
                    public void onLoadingComplete(String
imageUri,  View view, Bitmap loadedImage) {
                        subscriber.onNext(loadedImage);
                        subscriber.onCompleted();
                    }

                    @Override
                    public void onLoadingCancelled(String
imageUri, View view) {
                        subscriber.onError(new
Throwable("Image  loading cancelled"));
                    }
                });
            });
}
```

This `loadBitmap()` Observable can be chained to the previous one and we can, in the end, have a one single smooth Observable for the job:

```
if (isCityValid(location)) {
    String city = getCity(location, separatorPosition);
    OpenWeatherMapApiManager.getInstance()
            .getForecastByCity(city)
            .filter(response -> response != null)
            .filter(response -> response.getWeather().size() >
0)
            .flatMap(response -> {
                String url = getWeatherIconUrl(response);
                return loadBitmap(url);
            })
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Observer<Bitmap>() {
                @Override
                public void onCompleted() {

                }

                @Override
                public void onError(Throwable e) {
```

```
                    App.L.error(e.toString());
                }

                @Override
                public void onNext(Bitmap icon) {
                    city_image.setImageBitmap(icon);
                }
            });
}
```

After running the code, we will get the new weather icon for every user in the list.



Opening the website

Using the information contained in the user profile, we will create an `onClick` listener to navigate to the user web page, if present, or to the *Stack Overflow* profile.

To achieve this, we simply implement `Interface` in our `Activity` class, which gets triggered by Android's `onClick` event in our `Adapter`.

Our `Adapter ViewHolder` specifies this `Interface`:

```
public interface OpenProfileListener {

    public void open(String url);
}
```

Our `Activity` implements it:

```
[…] implements SoAdapter.ViewHolder.OpenProfileListener { […]

mAdapter.setOpenProfileListener(this);
[…]

@Override
public void open(String url) {
    Intent i = new Intent(Intent.ACTION_VIEW);
    i.setData(Uri.parse(url));
    startActivity(i);
}
```

`Activity` receives a URL and navigates to it using the external Android web browser. Our `ViewHolder` takes care of creating `OnClickListener` on every card of our list of users and checks whether we are going to open the *Stack Overflow* user profile or the external personal website:

```
mView.setOnClickListener(view -> {
    if (mProfileListener != null) {
        String url = user.getWebsiteUrl();
        if (url != null && !url.equals("") &&
!url.contains("search")) {
```

```
            mProfileListener.open(url);
        } else {
            mProfileListener.open(user.getLink());
        }
    }
});
```

The moment we click, we get redirected to the desired website. Working with Android, we can achieve the same result in a more Rx way by using one of the perks of RxAndroid (`ViewObservable`):

```
ViewObservable.clicks(mView)
        .subscribe(onClickEvent -> {
            if (mProfileListener != null) {
                String url = user.getWebsiteUrl();
                if (url != null && !url.equals("") &&
!url.contains("search")) {
                    mProfileListener.open(url);
                } else {
                    mProfileListener.open(user.getLink());
                }
            }
        });
```

The previous couple of code snippets are absolutely equivalent, and you can pick the one you like the most.

Summary

Our journey ends here. You are ready to bring your Java apps to a new code quality level. You can now enjoy a new coding paradigm and approach your daily programming life with a more fluid mindset. RxJava gives you the opportunity to think about data in a time-oriented way: everything is continuously changing, data is getting updated, events are getting triggered, and you can now create apps that react to these events, are flexible, and run smoothly.

Switching to RxJava can look hard and time consuming at the beginning, but we experienced how effective it can be to approach everyday problems in a reactive way. Now you can start porting your old code to RxJava: give those synchronous *getters* a new reactive life!

RxJava is an evolving and expanding world. There are lots of methods out there that we couldn't explore. There are methods that are not even there yet, because with RxJava, you can create your own operators and push them even further.

Android is a great place to play, but it comes with limitations. As an Android developer, you can overcome many of them with RxJava and RxAndroid. We had just a bite of RxAndroid, with `AndroidScheduler`, but in this last chapter, you learned about `ViewObservable`. RxAndroid gives you a lot more: for instance, `WidgetObservable`, and `LifecycleObservable`. Now it's up to you to push it more and more.

Remember that Observable sequences act like rivers: they flow. You can filter a river, you can transform a river, you can combine two rivers into one, and it will still flow. In the end, it will be the river you want it to be.

"Be water, my friend."

--Bruce Lee

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

    - AsyncSubject / AsyncSubject
- subscribeOn() method
    - about / SubscribeOn and ObserveOn
- switch() function
    - using / Switch
- switchMap() function / SwitchMap

T

U

W

- window() function / Window

Z