RxJava Essentials

Learn reactive programming to create awesome Android and Java apps

Ivan Morgillo
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Hello Reactive World'
- A synopsis of the book’s content
- More information on RxJava Essentials

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.
RxJava Essentials

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 RxAndroidSchedulers 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 a REST client efficiently and effectively.
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:

```gradle
apply plugin: 'com.android.application'
apply plugin: 'io.gitlab.arturbosch.detekt'

android {
    compileSdkVersion 21
    buildToolsVersion "21.1.2"
    defaultConfig {
        applicationId "com.neatapps.neatapp"
        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: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:1.14.0'
        compile 'com.github.scottwatanabe:butterui:0.4.0'
        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() {

}
The AppInfo object looks like this:

```java
@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:

```java
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!
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:

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

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

```java
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:

```java
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:

```java
private void loadApps(AppInfo appOne, AppInfo appTwo, AppInfo appThree) {
    mRecyclerBlurView.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();
            }
        });
}
As you can see, appending \texttt{repeat(3)} after the \texttt{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 \texttt{getInt()} function:

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

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 \texttt{defer()}:

\begin{verbatim}
Observable<Integer> deferred = Observable.defer(this::getInt);
\end{verbatim}
At this time, deferred exists, but the `getInt()` `create()` method hasn't been called yet: there is no "GETINT" in our logcat log:

```java
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`:

```java
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:

```java
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:

```java
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:

```java
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.
Where to buy this book

You can buy RxJava Essentials from the Packt Publishing website.

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

Click here for ordering and shipping details.