Getting Started with RxJava and Android

What is ReactiveX?

ReactiveX is an API that focuses on asynchronous composition and manipulation of observable streams of data or events by using a combination of the Observer pattern, Iterator pattern, and features of Functional Programming. Handling real-time data is a common occurrence, and having an efficient, clean, and extensible approach to handling these scenarios is important. Using Observables and operators to manipulate them, ReactiveX offers a composable and flexible API to create and act on streams of data while simplifying the normal concerns of asynchronous programming like thread creation and concurrency issues.

Intro to RxJava

RxJava is the open-source implementation of ReactiveX in Java. The two main classes are Observable and Subscriber. In RxJava, an Observable is a class that emits a stream of data or events, and a Subscriber is a class that acts upon the emitted items. The standard flow of an Observable is to emit one or more items, and then complete successfully or with an error. An Observable can have multiple Subscribers, and for each item emitted by the Observable, the item will be sent to the Subscriber.onNext() method to be handled. Once an Observable has finished emitting items, it will call the Subscriber.onCompleted() method, or if there is an error the Observable will call the Subscriber.onError() method. Now that we know enough about the Observable and Subscriber class, we can continue on to an introduction on creating and subscribing to Observables.

```
Observable integerObservable = Observable.create(new Observable.OnSubsc
ribe() {
   @Override
   public void call(Subscriber subscriber) {
       subscriber.onNext(1);
       subscriber.onNext(2);
       subscriber.onNext(3);
```

```
        subscriber.onCompleted();
    }
});
```

This Observable emits the integers 1, 2, and 3 and then completes. Now we need to create a Subscriber so we can act upon the stream of emissions.

```
Subscriber integerSubscriber = new Subscriber() {
    @Override
    public void onCompleted() {
        System.out.println("Complete!");
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onNext(Integer value) {
        System.out.println("onNext: " + value);
    }
};
```

Our Subscriber will simply print out any items emitted and notify us upon completion. Once you have an Observable and a Subscriber you can connect them with the Observable.subscribe() method.

```
integerObservable.subscribe(integerSubscriber);
// Outputs:
// onNext: 1
// onNext: 2
// onNext: 3
// Complete!
```

All of this code can be simplified by using the Observable.just() method to create an Observable to emit only the values defined, and changing our Subscriber to an anonymous inner class, we get the following.

```
Observable.just(1, 2 ,3).subscribe(new Subscriber() {
    @Override
    public void onCompleted() {
        System.out.println("Complete!");
```

```
    }

    @Override
    public void onError(Throwable e) {}

    @Override
    public void onNext(Integer value) {
        System.out.println("onNext: " + value);
    }
});
```

Operators

Creating and subscribing to an Observable is simple enough, and may not seem overly useful, but that is just the beginning of what is possible with RxJava. Any Observable can have its output transformed by what is called an Operator, and multiple Operators can be chained onto Observable. For example, in our previous Observable, say we only wanted to emit odd numbers that we receive. To accomplish this, we can use the filter() operator.

```
Observable.just(1, 2, 3, 4, 5, 6) // add more numbers
        .filter(new Func1() {
            @Override
            public Boolean call(Integer value) {
                return value % 2 == 1;
            }
        })
        .subscribe(new Subscriber() {
            @Override
            public void onCompleted() {
                System.out.println("Complete!");
            }

            @Override
            public void onError(Throwable e) {
            }

            @Override
            public void onNext(Integer value) {
                System.out.println("onNext: " + value);
            }
        });
// Outputs:
// onNext: 1
// onNext: 3
// onNext: 5
```

```
// Complete!
```

Our filter() operator defines a function that will take in our emitted Integer values, and return true for all odd numbers, and false for all even numbers. The values that return false from our filter() function are not emitted to the Subscriber, and we will not see them in the output. Notice that the filter() operator returns an Observable that we then subscribe to like we did before. Now say that I want to find the square root of the emitted odd numbers. One way of doing this would be to calculate the square root in each of the onNext() calls of our Subscriber. However, this is desired as it would not be possible to further transform the data stream if we are doing the square root calculations in our Subscriber. To accomplish this we can chain the map() operator with the filter() operator.

```
Observable.just(1, 2, 3, 4, 5, 6) // add more numbers
        .filter(new Func1() {
            @Override
            public Boolean call(Integer value) {
                return value % 2 == 1;
            }
        })
        .map(new Func1() {
            @Override
            public Double call(Integer value) {
                return Math.sqrt(value);
            }
        })
        .subscribe(new Subscriber() { // notice Subscriber type changed
to
            @Override
            public void onCompleted() {
                System.out.println("Complete!");
            }

            @Override
            public void onError(Throwable e) {
            }

            @Override
            public void onNext(Double value) {
                System.out.println("onNext: " + value);
            }
        });
// Outputs:
// onNext: 1.0
// onNext: 1.7320508075688772
// onNext: 2.23606797749979
// Complete!
```

Chaining operators is an integral part to RxJava and gives you the flexibility to accomplish anything you need. With an understanding of how Observables and Operators interact we can move onto the next topic: integrating RxJava with Android.

Simple Threading in Android

A common scenario in Android development is the need to offload some amount of work to a background thread, and once that task has finished, post the results on the main thread in order to display the results. With Android, we are given multiple options to do something like this natively, using AsyncTasks, Loaders, Services, etc. However, these solutions are often not the best. Asynctasks can easily lead to memory leaks, CursorLoaders with a ContentProvider require a large amount of configuration and boilerplate code to setup, and Services are intended for longer running background tasks and not fast-finishing operations such as making a network call or loading content from a database. Let's see how RxJava can help solve these problems. Given the following Layout that has a button to start a long running operation, and an always displayed progress circle so that we can make sure that our operation is being run on a background thread, and not on the main UI thread.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android
"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/root_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    android:orientation="vertical">

    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        app:popupTheme="@style/AppTheme.PopupOverlay"
        app:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" />

    <Button
        android:id="@+id/start_btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:text="@string/start_operation_text" />
```

```
    <ProgressBar
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:indeterminate="true" />

</LinearLayout>
```

Once the button is clicked, it will disable the button and start the operation, and once the operation is completed a Snackbar will be displayed and the button re-enabled. Here is the sample AsyncTask, and our “long running operation”. The onClick for the button simply creates a new SampleAsyncTask and executes it.

```
public String longRunningOperation() {
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        // error
    }
    return "Complete!";
}

private class SampleAsyncTask extends AsyncTask {

    @Override
    protected String doInBackground(Void... params) {
        return longRunningOperation();
    }

    @Override
    protected void onPostExecute(String result) {
        Snackbar.make(rootView, result, Snackbar.LENGTH_LONG).show();
        startAsyncTaskButton.setEnabled(true);
    }
}
```

Now how would we convert this AsyncTask to use RxJava? First, we need to add the following to our app’s gradle build file: compile 'io.reactivex:rxjava:1.0.14'. Then we would need to create an Observable that calls our long running operation. This can be done using the Observable.create() method.

```
final Observable operationObservable = Observable.create(new Observable
.OnSubscribe() {
    @Override
    public void call(Subscriber subscriber) {
```

```
        subscriber.onNext(longRunningOperation());
        subscriber.onCompleted();
    }
});
```

Our created Observable will call the longRunningOperation and post the result to onNext(), and then call onCompleted() to complete the Observable (NOTE: our operation is not called until we subscribe to our Observable). Next, we need to subscribe to our Observable when our button is clicked. I've added a new button that will handle the RxJava version of our task.

```
startRxOperationButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(final View v) {
        v.setEnabled(false);
        operationObservable.subscribe(new Subscriber() {
            @Override
            public void onCompleted() {
                v.setEnabled(true);
            }

            @Override
            public void onError(Throwable e) {}

            @Override
            public void onNext(String value) {
                Snackbar.make(rootView, value, Snackbar.LENGTH_LONG).sho
w();
            }
        });
    }
});
```

Now when we build our application and click the new button, what happens? Our progress indicator freezes, and our UI becomes unresponsive. This is because we have not defined what thread we should observe our Observable on and what thread we should subscribe to it on. This is the function of RxJava's Schedulers. With any Observable you can define two different threads that the Observable will be operated upon. Using Observable.observeOn() can define a thread that is used to monitor and check for newly emitted items from the Observable (the Subscriber's onNext, onCompleted, and onError methods execute on the observeOn() thread), and using Observable.subscribeOn() can define a thread that will run our Observable's code (the long running operation). RxJava is single-threaded by default, and you will need to make

use of the observeOn() and subscribeOn() methods to bring multi-threading to your application. RxJava comes with several out-of-the-box Schedulers to use with Observables, such as Schedulers.io() (for blocking I/O operations), Schedulers.computation() (computational work), and Schedulers.newThread() (creates new thread for the work). However, from an Android perspective, you might be wondering how to schedule code to execute on the main UI Thread. We can achieve this using the RxAndroid library.

RxAndroid is a lightweight extension to RxJava that providers a Scheduler for Android's Main Thread, as well as the ability to create a Scheduler that runs on any given Android Handler class. With the new Schedulers, the Observable created before can be modified to allow us to run our work on a background thread, and post our results to the main UI Thread. To get RxAndroid in the app, add the following line to the gradle build file: compile `'io.reactivex:rxandroid:1.0.1'`.

```
final Observable operationObservable = Observable.create(new Observable
.OnSubscribe() {
   @Override
   public void call(Subscriber subscriber) {
       subscriber.onNext(longRunningOperation());
       subscriber.onCompleted();
   }
})
       .subscribeOn(Schedulers.io()) // subscribeOn the I/O thread
       .observeOn(AndroidSchedulers.mainThread()); // observeOn the UI
Thread
```

Our modified Observable will use the Schedulers.io() to subscribe, and will observe the results on the UI Thread using AndroidSchedulers.mainThread(). Now when we build our app and click our Rx operation button, we can see that we are no longer blocking the UI Thread while the operation is running. All the previous examples made use of the Observable class for emitting our results, but we also have another option for when an operation only needs to emit one result and then complete. Release 1.0.13 of RxJava introduced the Single class. The Single can be used to create the functionality of the example as follows.

```
Subscription subscription = Single.create(new Single.OnSubscribe() {
           @Override
           public void call(SingleSubscriber singleSubscriber) {
               String value = longRunningOperation();
               singleSubscriber.onSuccess(value);
           }
       })
```

```
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Action1() {
            @Override
            public void call(String value) {
                // onSuccess
                Snackbar.make(rootView, value, Snackbar.LENGTH_LONG).sho
w();
            }
        }, new Action1() {
            @Override
            public void call(Throwable throwable) {
                // handle onError
            }
        });
```

When subscribing to a Single, there is only an onSuccess Action and an onError action. The Single class has a different set of operators than Observable, with several operators that allow for a mechanism of converting a Single to an Observable. For example, using the Single.mergeWith() operator, two or more Singles of the same type can be merged together to create an Observable, emitting the results of each Single to one Observable.

Preventing Leaks

One of the downsides previously mentioned with AsyncTasks is the ability to leak references to their enclosing Activity/Fragment if proper care is not taken. Unfortunately, using RxJava will not magically alleviate memory leak woes, but preventing them is fairly simple. If you've been following along in the code, you may have noticed that when you call Observable.subscribe() a Subscription object is returned. The Subscription class only has two methods, unsubscribe() and isUnsubscribed(). To prevent a possible memory leak, in your Activity/Fragment's onDestroy, check Subscription.isUnsubscribed() for if your Subscription is unsubscribed, and if not call Subscription.unsubscribe(). Unsubscribing will stop notifications of items to your Subscriber and will allow the garbage collection of all related objects, preventing any RxJava related memory leaks. If you are dealing with multiple Observables and Subscribers, all Subscription objects can be added to a CompositeSubscription and all unsubscribed at the same time using CompositeSubscription.unsubscribe().

Closing Comments

RxJava provides a great alternative to the multi-threading options in the Android ecosystem. Being able to easily offload an operation to a background thread, and then publish the results on the UI Thread is a much needed feature for any Android application, and being able to apply RxJava's multitude of operators to the results of any operation only creates more added value. While RxJava requires a good understanding of the library to fully take advantage of its features, time spent in working with the library is greatly rewarded. Further topics of RxJava that were not covered in this blog include: hot vs cold Observables, dealing with backpressure, Rx's Subject class. The sample code for converting an AsyncTask using RxJava can be found on github.

Bonus: Retrolambda

Java 8 introduced Lambdas Expressions, unfortunately Android does not support Java 8, so we are not able to take advantage of this with RxJava. Luckily there is a library called Retrolambda which backports lambdas to previous versions of Java. There is also a gradle plugin for Retrolambda that will allow the use of lambdas in an Android application. With lambdas, the example Observable and Subscriber can be reduced to the following.

```
final Observable operationObservable = Observable.create(
       (Subscriber subscriber) -> {
           subscriber.onNext(longRunningOperation());
           subscriber.onCompleted();
       })
       .subscribeOn(Schedulers.io())
       .observeOn(AndroidSchedulers.mainThread());

startRxOperationButton = (Button) findViewById(R.id.start_rxjava_operat
ion_btn);
startRxOperationButton.setOnClickListener(v -> {
   v.setEnabled(false);
   operationObservable.subscribe(
           value -> Snackbar.make(rootView, value, Snackbar.LENGTH_LONG
).show(),
           error -> Log.e("TAG", "Error: " + error.getMessage()),
           () -> v.setEnabled(true));
});
```

Lambdas cut down on the boilerplate code for RxJava a lot, and I would highly recommend you checkout Retrolambda for that purpose. It is usable with more than RxJava as well (notice the setOnClickListener call also makes use of Retrolambda).