Table of Contents

Learning RxJava with JavaFX

With RxJavaFX and RxKotlinFX

Thomas Nield

NOTE: This covers RxJavaFX 2.x

Preface

Over the past year or so, I have discovered so much can be achieved leveraging reactive programming in JavaFX applications. To this day, I still discover amazing patterns that RxJava allows in JavaFX applications. RxJavaFX is merely a layer between these two technologies to make them talk to each other, just like RxAndroid bridges RxJava and the Android UI.

I guess the best way to introduce RxJavaFX is to share how it came about. In 2014, I had already developed some Swing applications that were used internally at my company. These applications were quite involved with lots of interactivity, data processing, and complex user inputs. Like most UI applications, these were a beat-down to build. Naturally, I was drawn to JavaFX and was particularly intrigued by the `ObservableValue` , `ObservableList` , and other data structures that notified the UI automatically of their changes. No more `SwingUtilities.invokeLater()` ! Although I briefly considered HTML5 as my next platform, JavaFX showed more promise in the environment I worked in.

Keeping things synchronized between different components in a UI is difficult, so I liked JavaFX's idea of Bindings and ObservableValues. I became fascinated by this idea of events triggering other events, and a value notifying another value of its change. I started to believe there should be a way to express this functionally much like Java 8 Streams, and I had a vague idea what I was looking for.

But as I studied JavaFX deeper, I became discontent. JavaFX does have `Binding` functionality that can synchronize properties and events of different controls, but the ability to express transformations was limited. One morning, someone in an online community suggested I check out ReactFX, a reactive JavaFX library built by Tomas Mikula. This opened up my world and I discovered reactive programming for the first time. I played with the `EventStream` and was composing events and data together. I knew at that moment, reactive programming was the solution I was looking for.

I originally set out to use reactive programming as a way to handle UI events, and ReactFX was perfect for this. But I began to suspect I was missing the bigger picture. I researched reactive programming further and discovered RxJava, a

reactive API with a rich ecosystem of libraries built around it, including [RxAndroid] (https://github.com/ReactiveX/RxAndroid\) and [RxJava-JDBC] (https://github.com/davidmoten/rxjava-jdbc). RxJava rapidly became a core technology in the Android stack, and I wondered if it had the same potential in JavaFX. As I studied RxJava, I was immediately drawn to the RxJava-JDBC library. Effectively, I could leverage bindings that were bound to database queries. It soon became clear that with reactive programming, events are data, and data are events!

But how do I plug RxJava into ReactFX?

To create a fully effective reactive solution, I needed RxJava to talk to ReactFX. I tried this task and it was wrought with problems. Technically, it was difficult turning a ReactFX `EventStream` into an RxJava `Observable` and vice versa. I also realized ReactFX encourages doing everything on the JavaFX thread, but I wanted to switch between threads easily allowing concurrency. There is nothing wrong with ReactFX. It is awesome library that simply had a different purpose and goal.

During my struggle, I stumbled on the RxJavaFX project. It was a small library that converted `Node` and `ObservableValue` events into RxJava Observables. It also contained a `Scheduler` for the JavaFX thread. I knew immediately this was the alternative to ReactFX I needed, but some folks at Netflix were having some build issues with it. Ben Christensen was eager to give it away to someone who knew JavaFX, as nobody at Netflix used JavaFX. After a period of no activity, I reluctantly volunteered to take ownership of it. After hours of Googling, trawling GitHub projects with similar issues, and making a few tweaks, the build and tests finally succeeded. I was able to get it released on Maven Central and RxJavaFX was now live.

When I took ownership of the RxJavaFX library, I doubted it would progress beyond turning JavaFX `Node` events and `ObservableValue` changes into RxJava Observables. But I quickly learned there was much more to be done. JavaFX was built with event hooks everywhere, including collections like `ObservableList`. This provided all the tools needed to make a fully reactive API for JavaFX, and there was a lot of power yet to be exposed. With random epiphanies as well as some guidance from the community, RxJavaFX has become a robust solution to integrate JavaFX into the RxJava ecosystem.

So let's get started!

1. Getting Started

Reactive programming is about composing events and data together, and treating them identically. This idea of "events are data, and data are events" is powerful, and because UI's often have to coordinate both it is the perfect place to learn and apply reactive programming. For this reason, this book will teach RxJava from a JavaFX perspective and assume no prior RxJava knowledge. If you already have experience with RxJava, you are welcome to skip the next chapter.

I would highly recommend being familiar with JavaFX (or TornadoFX) before starting this book. *[Mastering JavaFX 8 Controls* (Hendrik Ebbers) and *Pro JavaFX 8* (James Weaver and Weiqi Gao) are excellent books to learn JavaFX. If you are interested in leveraging JavaFX with the Kotlin language, check out the TornadoFX Guide written by Edvin Syse and Thomas Nield (yours truly). I will explain why this book shows examples in both the Java and Kotlin languages shortly. For now, let us explore the benefits of using RxJava with JavaFX.

Why Use RxJava with JavaFX?

As stated earlier, reactive programming can equalize events and data by treating them the same way. This is a powerful idea with seemingly endless practical use cases. JavaFX provides many hooks that can easily be made reactive. There are many reactive libraries, from Akka and Sodium to ReactFX. But RxJava really hit on something, especially with its simple handling of concurrency, extensibility, and rich ecosystem of third party libraries. It has taken the Android community by storm and continues to make reactive programming a go-to tool to meet modern user demands.

RxJavaFX is a lightweight but comprehensive library to plug JavaFX components into RxJava, and vice versa. This is what this book will cover. Some folks reading this may ask "Why not use ReactFX? Why do we need a second reactive framework for JavaFX when that one is perfectly fine?" ReactFX is an excellent reactive framework made by Tomas Mikula, and you can read more about my experiences with it in the Preface. But the TL;DR is this: **ReactFX encourages keeping all operations on the JavaFX thread, while RxJava embraces full-**

blown concurrency. On top of that, RxJava also has a rich ecosystem of extensible libraries (e.g. RxJava-JDBC, while ReactFX focuses solely on JavaFX events. ReactFX and RxJavaFX simply have different scopes and goals.

RxJava has a rapidly growing and active community. The creators and maintainers of RxJava do an awesome job of answering questions and being responsive to developers of all skill levels. Reactive programming has enabled an exciting new domain filled with new ideas, practical applications, and constant discovery. RxJava is one of the many ReactiveX API's standardized across many programming languages. Speaking of other languages, let us talk about Kotlin.

Using Kotlin (Optional)

This book will present examples in two languages: Java and Kotlin. If you are not familiar, Kotlin is a new JVM language created by JetBrains, the company behind Intellij IDEA, PyCharm, CLion, and several other IDE's and tools. JetBrains believed they could be more productive by creating a new language that emphasized pragmatism and industry over convention. After 5 years of developing and testing, Kotlin 1.0 was released in February 2016 to fulfill this need. A year later Kotlin 1.1 was released with more practical (but tightly-scoped) features.

If you have never checked out Kotlin, I would higly recommend giving it a look. It is an intuitive language that only takes a few hours for a Java developer to learn. The reason I present Kotlin in this book is because it created a unique opportunity on the JavaFX front. Towards the end of Kotlin's beta, Edvin Syse released TornadoFX, a lightweight Kotlin library that significantly streamlines development of JavaFX applications.

For instance, with TornadoFX you can create an entire `TableView` using just the Kotlin code below:

```kotlin
tableview<Person> {
    column("ID", Person::id)
    column("Name", Person::name)
    column("Birthday", Person::birthday)
    column("Age", Person::age)
}
```

I had the privilege of joining Edvin's project not long after TornadoFX's release, and the core team has created a phenomenal JavaFX suite of features enabled by the Kotlin language. I would highly recommend giving the TornadoFX Guide a look to learn more.

There is a Kotlin extension of the RxJavaFX library called RxKotlinFX. It wraps Kotlin extension functions around RxJavaFX and includes some additional operators. The Kotlin versions of examples will use this library, and will also use TornadoFX. Using this stack may add a few more dependencies to your project, but the amount of value it adds through abstraction and productivity may make it worthwhile!

If you are not interested in Kotlin, no worries! The Java version of code samples will be always be presented first and you can ignore the Kotlin ones.

Setting Up

Currently, RxJavaFX and RxKotlinFX support both RxJava 1.x and RxJava 2.x (and aligned with their own respectiveX 1.x and 2.x versions). RxJava 2.x brings a number of large changes to RxJava, and this is the version that the guide will cover.

You should prefer RxJava 2.x because RxJava 1.x support will discontinue early 2018.

Java

To setup RxJavaFX 2.x for Java, use the Gradle or Maven configuration below where `x.y.z` is the version number you want to specify.

Gradle

```
compile 'io.reactivex.rxjava2:x.y.z'
```

Maven

```
<dependency>
    <groupId>io.reactivex.rxjava2</groupId>
    <artifactId>rxjavafx</artifactId>
    <version>x.y.z</version>
</dependency>
```

Kotlin

If you are using Kotlin, you will want to use RxKotlinFX instead of RxJavaFX. Make sure you have configured Maven or Gradle to use a Kotlin configuration, and include the dependencies below. Note the `x.y.z` is where you put the targeted version number, and I included TornadoFX and RxKotlin as dependencies since the examples will use them.

Gradle

```
compile 'com.github.thomasnield:rxkotlinfx:x.y.z'
compile 'no.tornado:tornadofx:x.y.z`
compile 'io.reactivex.rxjava2:rxkotlin:x.y.z'
```

Maven

```
<dependency>
    <groupId>com.github.thomasnield</groupId>
    <artifactId>rxkotlinfx</artifactId>
    <version>x.y.z</version>
</dependency>
<dependency>
    <groupId>no.tornado</groupId>
    <artifactId>tornadofx</artifactId>
    <version>x.y.z</version>
</dependency>
<dependency>
    <groupId>io.reactivex</groupId>
    <artifactId>rxkotlin</artifactId>
    <version>x.y.z</version>
</dependency>
```

Figure 1.1. shows a Venn diagram showing the stack of technologies typically used to built a reactive JavaFX application with Kotlin. The overlaps indicate that library is used to interoperate between the 3 domains: JavaFX, RxJava and Kotlin

Figure 1.1
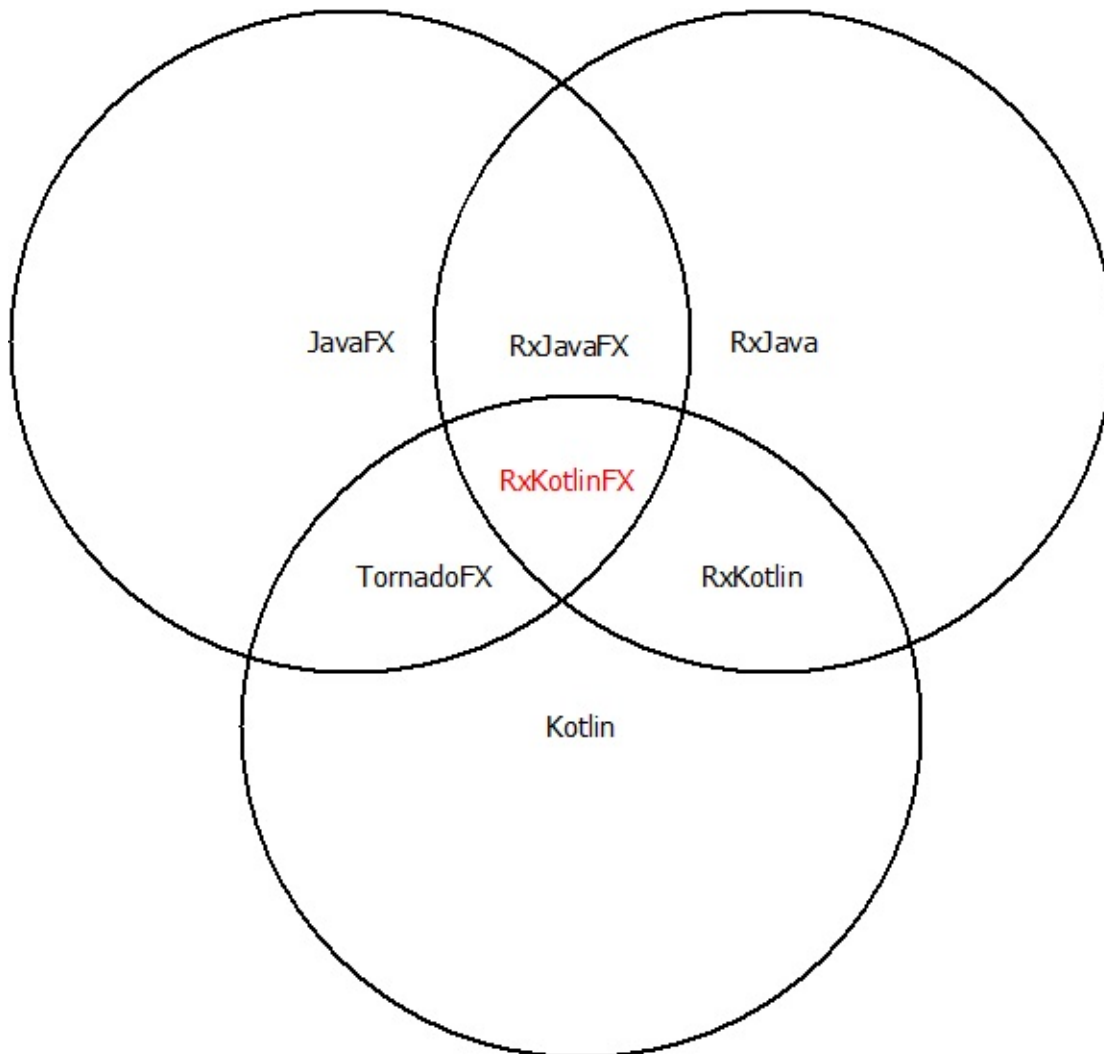
Summary

In this chapter we got a high level overview of reactive programming and the role RxJavaFX plays in connecting JavaFX and RxJava together. There was also an explanation why Kotlin is presented alongside Java in this book, and why both RxKotlinFX and TornadoFX are compelling options when building JavaFX applications. You can go through this book completely ignoring the Kotlin examples if you like.

In the next chapter we will cover the fundamentals of RxJava, and do it from a JavaFX perspective. If you are already experienced with RxJava, you are welcome to skip this chapter. But if you have been looking for a practical domain to apply RxJava, read on!

2. RxJava Fundamentals

RxJava has two core types: the `Observable` and the `Observer`. In the simplest definition, an `Observable` *pushes* things. A given `Observable<T>` will push items of type `T` through a series of operators that form other Observables, and finally the terminal `Observer` is what consumes the items at the end of the chain.

Each pushed `T` item is known as an **emission**. Usually there is a finite number of emissions, but sometimes there can be infinite. An emission can represent either data or an event (or both!). This is where the power of reactive programming differentiates itself from Java 8 Streams and Kotlin Sequences. It has a notion of *emissions over time*, and we will explore this concept in this chapter.

The `Observable` and `Observer`

As stated earlier, an **Observable** pushes things. It pushes things of type `T` through a series of operators forming other `Observables`. Each pushed item is known as an **emission**. Those emissions are pushed all the way to a `Observer` where they are finally consumed.

You will need to create a **source Observable** where emissions originate from, and there are many factories to do this. To create a source `Observable` that pushes items 1 through 5, declare the following:

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {

        Observable<Integer> source = Observable.just(1,2,3,4,5);
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable

fun main(args: Array<String>) {
    val source = Observable.just(1,2,3,4,5)
}
```

This source `Observable<Integer>` is saved to a variable named `source`. However, it has not pushed anything yet. In order to start pushing emissions, you need to create an `Observer`. The quickest way to do this is call `subscribe()` and pass a lambda specifying what to do with each emission.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {

        Observable<Integer> source = Observable.just(1,2,3,4,5);

        source.subscribe(System.out::println);
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable

fun main(args: Array<String>) {

    val source = Observable.just(1,2,3,4,5)
    source.subscribe(::println)
}
```

A **lambda** is a special type of argument specifying an action. This one will take each emission and print it, and this `subscribe()` operation creates a `Observer` for us based on this lambda argument.

> Java 8 and Kotlin have their own ways of expressing lambdas. If you need to learn more about Java 8 lambdas, I would recommend reading at least the first two chapters of *Java 8 Lambdas* by Richard Warburton before proceeding. You can read the Kotlin Reference to learn about lambdas in Kotlin. Lambdas are a critical syntax feature that we will use constantly in this book.

Go ahead and run the code above, and you should get the following:

OUTPUT:

```
1
2
3
4
5
```

This effectively pushed the integers 1 through 5, one-at-a-time, to the `Observer` defined by the lambda in the `subscribe()` method. The `subscribe()` method does not have to print items. It could populate them in a JavaFX control, write them to a database, or post it as a server response.

Understandings Observers

You can specify up to three lambda arguments on the `subscribe()` method to not only handle each emission, but also handle the event of an error as well as an action on completion when there are no more emissions.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {

        Observable<Integer> source = Observable.just(1,2,3,4,5);

        source.subscribe(System.out::println,
                Throwable::printStackTrace,
                () -> System.out.println("Done!")
        );
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import rx.lang.kotlin.subscribeBy

fun main(args: Array<String>) {

    val source = Observable.just(1, 2, 3, 4, 5)

    source.subscribeBy(
        onNext = ::println,
        onError = { it.printStackTrace() },
        onComplete = { println("Done!") }
    )
}
```

OUTPUT:

```
1
2
3
4
5
Done!
```

> Typically, you should always supply an `onError` lambda to your `subscribe()` call so errors do not quietly go unhandled. We will not use `onError` very much in this book for the sake of brevity, but be sure to use it when putting reactive code in production.

Let's briefly break down the `Observer` to understand it better. The lambdas are just a shortcut to allow the `subscribe()` method to quickly create an `Observer` for you. You can create your own `Observer` object explicitly by extending `ResourceObserver` and implementing its three abstract methods: `onNext()` , `onError()` , and `onComplete()` . You can then pass this `Observer` to the `subscribe()` method.

Java

```java
import io.reactivex.Observable;
import io.reactivex.Observer;
import io.reactivex.observers.ResourceObserver;

public class Launcher {

    public static void main(String[] args) {

        Observable<Integer> source = Observable.just(1,2,3,4,5);

        Observer<Integer> subscriber = new ResourceObserver<Integer>() {

            @Override
            public void onComplete() {
                System.out.println("Done!");
            }

            @Override
            public void onError(Throwable e) {
                e.printStackTrace();
            }

            @Override
            public void onNext(Integer integer) {
                System.out.println(integer);
            }
        };

        source.subscribe(subscriber);
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import io.reactivex.observers.ResourceObserver

fun main(args: Array<String>) {
    val source = Observable.just(1, 2, 3, 4, 5)

    val subscriber = object: ResourceObserver<Int>() {
        override fun onComplete() = println("Done!")

        override fun onNext(i: Int) = println(i)

        override fun onError(e: Throwable) = e.printStackTrace()
    }

    source.subscribe(subscriber)

}
```

The `Observer` interface defines these three methods. The `onNext()` is what is called to pass an emission. The `onError()` is called when there is an error, and `onComplete()` is called when there are no more emissions. Logically with infinite Observables, the `onComplete()` is never called.

> We extend a `ResourceObserver` because the `Observer` actually has a fourth abstract method, `onSubscribe()`. This is something you will rarely need to implement yourself, and `ResourceObserver` will take care of it for you. Essentially, the `onSubscribe()` allows the `Observer` implementation to have control of the `Disposable` which is beyond the scope of this book. We will touch on the `Dispsoable` later, but you can learn much more about it in Chapter 2 of my book Learning RxJava by Packt Publishing.

Although this example is helpful for understanding the `Observer`, it also shows implementing `Observer` objects can be pretty verbose. Therefore, it is helpful to use lambdas instead for conciseness. These three methods on the `Observer` are critical for understanding RxJava, and we will revisit them several times in this chapter.

It is critical to note that the `onNext()` can only be called by one thread at a time. There should never be multiple threads calling `onNext()` concurrently, and we will learn more about this later when we cover concurrency. For now just note RxJava has no notion of parallelization, and when you subscribe to a factory like `Observable.just(1,2,3,4,5)`, you will always get those emissions serially, in that exact order, and *on a single thread*.

Source Observable Factories

Going back to the `Observable`, there are other factories to create source Observables. Above we emitted the integers 1 through 5. Since these are consecutive, we can use `Observable.range()` to accomplish the same thing. It will emit the numbers 1 through 5 based on their range, and then call `onComplete()`.

Note these examples have no subscribers, so there will be no output when running them.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {

        Observable<Integer> source = Observable.range(1,5);
    }
}
```

Kotlin

```
import io.reactivex.Observable

fun main(args: Array<String>) {

    val source = Observable.range(1,5)
}
```

You can also turn any `Iterable<T>` into an `Observable<T>` quickly using `Observable.fromIterable()`. It will emit all items in that `Iterable<T>` and then call `onComplete()` when it is done.

Java

```
import io.reactivex.Observable;
import java.util.Arrays;
import java.util.List;

public class Launcher {

    public static void main(String[] args) {

        List<Integer> list = Arrays.asList(1,2,3,4,5);
        Observable<Integer> source = Observable.fromIterable(lis
t);    }
}
```

Kotlin

```
import io.reactivex.rxkotlin.toObservable

fun main(args: Array<String>) {

    val list = listOf(1,2,3,4,5)
    val source = list.toObservable()
}
```

Using Operators

Let us do something a little more useful than just connecting a source `Observable` and an `Observer`. Let's put some operators between them to actually transform emissions and do work.

map()

Say you have an `Observable<String>` that pushes `String` values.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {

        Observable<String> source =
                Observable.just("Alpha", "Beta", "Gamma", "Delta"
, "Epsilon");
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable

fun main(args: Array<String>) {

    val source = Observable.just("Alpha","Beta","Gamma","Delta",
"Epsilon")


}
```

In RxJava, you can use hundreds of operators to transform emissions and create new Observables with those transformations. For instance, you can create an `Observable<Integer>` off an `Observable<String>` by using the `map()`

operator, and use it to emit each String's length.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {

        Observable<String> source =
                Observable.just("Alpha","Beta","Gamma","Delta",
"Epsilon");

        Observable<Integer> lengths = source.map(String::length)
;

        lengths.subscribe(System.out::println);
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable

fun main(args: Array<String>) {

    val source =  Observable.just("Alpha","Beta","Gamma","Delta"
, "Epsilon")

    val lengths = source.map { it.length }

    lengths.subscribe(::println)
}
```

OUTPUT:

```
5
4
5
5
7
```

The `source` Observable pushes each `String` to the `map()` operator where it is mapped to its `length` . That length is then pushed from the `map()` operator to the `Observer` where it is printed.

You can do all of this without any intermediary variables holding each `Observable` , and instead do everything in a single "chain" call. This can be done in one line or broken up into multiple lines.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
                .map(String::length)
                .subscribe(System.out::println);
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable

fun main(args: Array<String>) {

    Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
            .map { it.length }
            .subscribe(::println)
}
```

Operators behave as both an intermediary `Observer` and an `Observable`, receiving emissions from the upstream source, transforming them, and passing them downstream to the final `Observer`.

```
Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon") // ca
lls onNext() on map()
    .map(s -> s.length()) // calls onNext() on Observer
    .subscribe(i -> System.out.println(i));
```

filter()

Another common operator is `filter()`, which suppresses emissions that fail to meet a certain criteria, and pushes the ones that do forward. For instance, you can emit only Strings where the `length()` is at least 5. In this case, the `filter()` will stop "Beta" from proceeding since it is 4 characters.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
                .filter(s -> s.length() >= 5)
                .subscribe(System.out::println);
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable

fun main(args: Array<String>) {

    Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
            .filter { it.length >= 5 }
            .subscribe(::println)
}
```

OUTPUT:

```
Alpha
Gamma
Delta
Epsilon
```

distinct()

There are also operators like `distinct()`, which will suppress emissions that have previously been emitted to prevent duplicate emissions (based on each emission's `hashcode()` / `equals()` implementation).

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon"
)
                .map(String::length)
                .distinct()
                .subscribe(System.out::println);
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable

fun main(args: Array<String>) {

    Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon"
)
            .map { it.length }
            .distinct()
            .subscribe(::println)
}
```

OUTPUT:

```
5
4
7
```

You can also provide a lambda specifying an attribute of each emitted item to distinct on, rather than the item itself.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
                .distinct(String::length)
                .subscribe(System.out::println);
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable

fun main(args: Array<String>) {

    Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
            .distinct { it.length }
            .subscribe(::println)
}
```

OUTPUT:

```
Alpha
Beta
Epsilon
```

take()

The `take()` operator will cut off at a fixed number of emissions and then unsubscribe from the source. Afterwards, it will call `onComplete()` downstream to the final `Observer`.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon"
)
                .take(3)
                .subscribe(System.out::println);
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable

fun main(args: Array<String>) {

    Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
            .take(3)
            .subscribe(::println)
}
```

OUTPUT:

```
Alpha
Beta
Gamma
```

`takeWhile()` will do something similar to `take()`, but specifies a lambda condition to determine when to stop taking emissions rather than using a fixed count.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {
        Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
                .takeUntil((String s) -> s.matches("D.*"))
                .subscribe(System.out::println);
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable

fun main(args: Array<String>) {

    Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
            .takeUntil { it.startsWith("D" )}
            .subscribe(::println)
}
```

OUTPUT:

```
Alpha
Beta
Gamma
Delta
```

count()

Some operators will aggregate the emissions in some form (in a classic MapReduce fashion), and then push that aggregation as a single emission to the `Observer`. Obviously, this requires the `onComplete()` to be called so that the aggregation can be finalized and pushed to the `Observer`.

One of these aggregation operators is `count()`. It will simply count the number of emissions and when its `onComplete()` is called, and push the count up to the `Observer` as a single emission. Then it will call `onComplete()` up to the `Observer`.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {

        Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
                .count()
                .subscribe(System.out::println);
    }
}));
```

Kotlin

```kotlin
import io.reactivex.Observable
import io.reactivex.rxkotlin.subscribeBy

fun main(args: Array<String>) {

    Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
            .count()
            .subscribeBy { println(it) }
}
```

OUTPUT:

```
5
```

The `count()` actually returns a `Single` , which is a specialized `Observable` type that only emits one item. The `Single` does not have an `onNext` or `onComplete` , but rather an `onSuccess` event which passes the single item. If you ever need to turn a `Single` back into an `Observable` (so it works with certain `Observable` operators), just call its `toObservable()` method.

> Another variant of `Observable` we will encounter is the `Maybe` , which emits 0 or 1 values. There is also the `Completable` which ignores emissions but we will not be using it in this book.

toList()

The `toList()` is similar to the `count()` , and it also will yield a `Single` rather than an `Observable` . It will collect the emissions until its `onComplete()` is called. After that it will push an entire `List` containing all the emissions to the `Observer` .

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {

        Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
                .toList()
                .subscribe(System.out::println);
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import io.reactivex.rxkotlin.subscribeBy


fun main(args: Array<String>) {


    Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
            .toList()
            .subscribeBy { println(it) }
}
```

OUTPUT:

```
[Alpha, Beta, Gamma, Delta, Epsilon]
```

Aggregate operators like `toList()` will misbehave on infinite Observables because collections can only be finite, and it needs that `onComplete()` to be called to push the `List` forward. Otherwise it will collect and work infinitely.

> When using Singles in Kotlin, we use RxKotlin's `subscribeBy()` instead of `subscribe()` because there is an inference bug with the Kotlin compiler working with SAM types. Hopefully this will be fixed soon by JetBrains. You can follow the filed issue here to track its status.

reduce()

When you need to do a custom aggregation or reduction, you can use `reduce()` to achieve this in most cases (to aggregate into collections and other mutable structures, you can use its cousin `collect()`). This will return a `Single` (if a "seed" value is provided) or a `Maybe` (if no "seed" value is provided). But say we wanted the sum of all lengths for all emissions. Starting with a seed value of zero, we can use a lambda specifying how to "fold" the emissions into a single value.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {

        Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
                .map(String::length)
                .reduce(0,(current,next) -> current + next)
                .subscribe(System.out::println);
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import io.reactivex.rxkotlin.subscribeBy

fun main(args: Array<String>) {

    Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
            .map { it.length }
            .reduce(0) { current,next -> current + next }
            .subscribeBy { println(it) }
}
```

OUTPUT:

```
26
```

The lambda in `reduce()` will keep adding two `Integer` values, where one of them is the "rolling total" (`current`) or seed `0` value, and the other is the new value (`next`) to be added. As soon as `onComplete()` is called, it will push the result to the `Observer` .

scan()

The `reduce()` will push a single aggregated value derived from all the emissions. If you want to push the "running total" for each emission, you can use `scan()` instead. This can work with infinite Observables since it will push each accumulation for each emission, rather than waiting for all emissions to be accumulated.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {

        Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
                .map(String::length)
                .reduce(0,(current,next) -> current + next)
                .subscribe(System.out::println);
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable

fun main(args: Array<String>) {

    Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
            .map { it.length }
            .reduce(0) { current,next -> current + next }
            .subscribe(::println)
}
```

OUTPUT:

```
0
5
9
14
19
26
```

flatMap()

There are hundreds of operators in RxJava, but we will only cover one more for now. Throughout the book we will learn more as we go, and the most effective way to learn operators is to seek them out of need.

The `flatMap()` is similar to `map()`, but will map the emission to another set of emissions via another `Observable`. This is one of the most powerful operators in RxJava and is full of use cases, but for now we will just stick with a simple example.

Say we have some `String` emissions where each one contains concatenated numbers separated by a slash `/`. We want to break up these numbers into separate emissions (and omit the slashes). You can call `split()` on each `String` and specify splitting on the slashes `/`, and this will return an array of the separated `String` values. Then you can turn that array into an `Observable` inside the `flatMap()`.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {

        Observable.just("123/52/6345","23421/534","758/2341/7493
2")
                .flatMap(s -> Observable.fromArray(s.split("/"))
)
                .subscribe(System.out::println);
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import io.reactivex.rxkotlin.toObservable

fun main(args: Array<String>) {

    Observable.just("123/52/6345","23421/534","758/2341/74932")
            .flatMap { it.split("/").toObservable() }
            .subscribe(::println)
}
```

OUTPUT:

```
123
52
6345
23421
534
758
2341
74932
```

If you observe this closely, hopefully you will find the `flatMap()` is pretty straightforward. You are taking each emission and replacing it with another set of emissions, by providing another `Observable` derived off that emission. There is a lot of very powerful ways to leverage the `flatMap()`, especially when used with infinite, concurrent, and hot Observables which we will cover later.

Also note that `flatMapSingle()` can be used to `flatMap()` to a `Single`, and `flatMapMaybe()` to a `Maybe`. This saves you the step of having to call `toObservable()` on each resulting `Maybe` or `Single`. If we wanted to sum each set of numbers, we would do it like this since this `reduce()` will yield a `Single`.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {

        Observable.just("123/52/6345","23421/534","758/2341/7493
2")
                .flatMapSingle(s ->
                        Observable.fromArray(s.split("/"))
                        .map(Integer::valueOf)
                        .reduce(0, (curr,next) -> curr + next)
                )
                .subscribe(System.out::println);
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import io.reactivex.rxkotlin.toObservable


fun main(args: Array<String>) {


    Observable.just("123/52/6345","23421/534","758/2341/74932")
            .flatMapSingle {
                it.split("/").toObservable()
                        .map { it.toInt() }
                        .reduce(0) { curr, next -> curr + next }
            }
            .subscribe(::println)
}
```

OUTPUT:

```
6520
23955
78031
```

Observables and Timing

If you are a somewhat experienced developer, you might be asking how is the `Observable` any different than a Java 8 Stream or Sequences in Kotlin, C#, or Scala. Up to this point you are correct, they do not seem much different. But recall that Observables *push*, while Java 8 Streams and Sequences *pull*. This enables RxJava to achieve much more and unleashes capabilities that these other functional utilities do not offer.

But the fundamental benefit of *pushing* is it allows a notion of *emissions over time*. Our previous examples do not exactly show this, but now we will dive into some examples that do.

Making Button Click Events an Observable

So far we just pushed data out of Observables. But did you know you can push events too? As stated earlier, data and events are basically the same thing in RxJava. Let's take a simple JavaFX `Application` with a single `Button`.

Java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;



public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox vBox = new VBox();
        Button button = new Button("Press Me");

        vBox.getChildren().add(button);
        stage.setScene(new Scene(vBox));
        stage.show();
    }
}
```
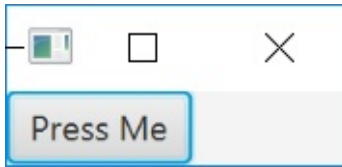
Kotlin

```kotlin
import tornadofx.*

class MyApp: App(MyView::class)

class MyView: View() {
    override val root = vbox {
        button("Press Me")
    }
}
```

Rendered UI:

We can use RxJavaFX or RxKotlinFX to create an `Observable<ActionEvent>` that pushes an `ActionEvent` emission each time the `Button` is pressed.

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox vBox = new VBox();
        Button button = new Button("Press Me");

        JavaFxObservable.actionEventsOf(button)
                .subscribe(System.out::println);

        vBox.getChildren().add(button);
        stage.setScene(new Scene(vBox));
        stage.show();
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.actionEvents
import tornadofx.*

class MyApp: App(MyView::class)

class MyView: View() {
    override val root = vbox {
        button("Press Me")
                .actionEvents()
                .subscribe { println(it) }
    }
}
```

If you click the `Button` a couple times your console should look something like this:

OUTPUT:

```
javafx.event.ActionEvent[source=Button@751b917f[styleClass=butto
n]'Press Me']
javafx.event.ActionEvent[source=Button@751b917f[styleClass=butto
n]'Press Me']
javafx.event.ActionEvent[source=Button@751b917f[styleClass=butto
n]'Press Me']
```

Wait, did we just treat the `ActionEvent` like any other emission and push it through the `Observable`? Yes we did! As said earlier, this is the powerful part of RxJava. It treats events and data the same way, and you can use all the operators we covered earlier. For example, we can use `scan()` to push how many times the `Button` was pressed, and push that into a `Label`. Just `map()` each `ActionEvent` to a `1` to drive increments.

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox vBox = new VBox();
        Button button = new Button("Press Me");
        Label countLabel = new Label("0");

        JavaFxObservable.actionEventsOf(button)
                .map(ae -> 1)
                .scan(0,(x,y) -> x + y)
                .subscribe(clickCount -> countLabel.setText(clic
kCount.toString())));

        vBox.getChildren().add(countLabel);
        vBox.getChildren().add(button);

        stage.setScene(new Scene(vBox));
        stage.show();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.actionEvents
import tornadofx.*

class MyApp: App(MyView::class)

class MyView: View() {

    override val root = vbox {

        val countLabel = label("0")

        button("Press Me")
                .actionEvents()
                .map { 1 }
                .scan(0) {x,y -> x + y }
                .subscribe { countLabel.text = it.toString() }
    }
}
```
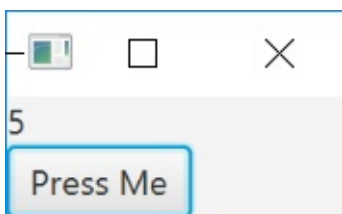
RENERED UI: After I clicked the button 4 times

So how does all this work? The `Observable<ActionEvent>` we created off this `Button` is emitting an `ActionEvent` item every time the `Button` is pressed. Every time that `Button` is clicked, it pushes an `ActionEvent` emission through the `Observable`. There is no notion of completion either as this `Observable` is always alive during the life of the `Button`.

Of course you could use operators that make the operation finite, like `take()`. If you only take 5 `ActionEvent` emissions from the `Button`, it will stop pushing on emission `4`. Then it will unsubscribe from the source and call `onComplete()` down the chain to the `Observer`.

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox vBox = new VBox();
        Button button = new Button("Press Me");
        Label countLabel = new Label("0");
        Label doneLabel = new Label("");

        JavaFxObservable.actionEventsOf(button)
                .map(ae -> 1)
                .scan(0,(x,y) -> x + y)
                .take(5)
                .subscribe(
                        clickCount -> countLabel.setText(clickCount.toString()),

                        Throwable::printStackTrace,
                        () -> doneLabel.setText("Done!")
                );

        vBox.getChildren().addAll(countLabel, doneLabel,button);

        stage.setScene(new Scene(vBox));
        stage.show();
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.actionEvents
import io.reactivex.rxkotlin.subscribeBy
import tornadofx.*

class MyApp: App(MyView::class)

class MyView: View() {

    override val root = vbox {

        val countLabel = label("0")
        val doneLabel = label("")

        button("Press Me")
                .actionEvents()
                .map { 1 }
                .scan(0) {x,y -> x + y }
                .take(5)
                .subscribeBy(
                    onNext =  { countLabel.text = it.toString() },

                    onError = { it.printStackTrace() },
                    onComplete = { doneLabel.text = "Done!" }
                )
    }
}
```
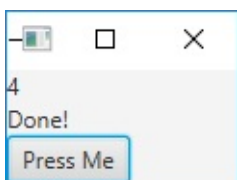
RENDERED UI: After 4 `Button` clicks (emits an initial 0 from `scan()`)

A `Button` emitting an `ActionEvent` item every time it is clicked is an example of a hot `Observable` , as opposed to cold Observables which typically push data. Let's dive into this discussion next.

Cold vs. Hot Observables

The `Observable<ActionEvent>` we created off a `Button` is an example of a hot `Observable` . Earlier in this chapter, all of our examples emitting `Integer` and `String` items are cold Observables. So what is the difference?

Remember this source `Observable` that simply pushes five `String` emissions?

Java

```java
Observable<String> source =
    Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon");
```

Kotlin

```kotlin
val source = Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")
```

What do you think will happen if we `subscribe()` to it twice? Try it out.

Java

```java
Observable<String> source =
        Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon"
);

source.subscribe(s -> System.out.println("Observer 1: " + s));
source.subscribe(s -> System.out.println("Observer 2: " +s));
```

```kotlin
val source = Observable.just("Alpha","Beta","Gamma","Delta", "Epsilon")

source.subscribe { println("Observer 1: $it") }
source.subscribe { println("Observer 2: $it") }
```

You will find the emissions are *replayed* for each `Observer` .

OUTPUT:

```
Observer 1: Alpha
Observer 1: Beta
Observer 1: Gamma
Observer 1: Delta
Observer 1: Epsilon
Observer 2: Alpha
Observer 2: Beta
Observer 2: Gamma
Observer 2: Delta
Observer 2: Epsilon
```

With a **Cold Observable**, every `Observer` independently receives all the emissions regardless of when they `Subscribe`. There is no notion of timing making an impact to which emissions they receive. Cold Observables are often used to "play" data independently to each `Observer`. This is like giving every `Observer` a music CD to play, and they can independently play all the tracks.

Hot Observables, however, will simultaneously push emissions to all Observers at the same time. Logically, an effect of this is Observers that come later and have missed previous emissions will not receive them. They will only get emissions going forward from the time they `subscribe()`. Instead of a music CD, Hot Observables are more like radio stations. They will broadcast a given song (emission) to all listeners (Observers) at the same time. If a listener misses a song, they missed it.

While data and events are the same in RxJava, Hot Observables are often used to represent events, such as an `Observable<ActionEvent>` built off a `Button`.

Let's do an experiment to see if tardy Observers indeed miss previous emissions. `subscribe()` immediately to a `Button`'s clicks to create the first `Observer`. But have another `Button` that when clicked, will `subscribe()` a second `Observer`.

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
```

```java
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox vBox = new VBox();
        Button button = new Button("Press Me");
        Button secondSubButton = new Button("Subscribe Observer
2");

        Observable<ActionEvent> clicks =
            JavaFxObservable.actionEventsOf(button);

        //Observer 1
        clicks.subscribe(ae ->
            System.out.println("Observer 1 Received Click!"));

        //Subscribe Observer 2 when secondSubButton is clicked
        secondSubButton.setOnAction(event -> {
                System.out.println("Observer 2 subscribing!");
                secondSubButton.disableProperty().set(true);
                //Observer 2
                clicks.subscribe(ae ->
                    System.out.println("Observer 2 Received Clic
k!")
                );
            });

        vBox.getChildren().addAll(button,secondSubButton);

        stage.setScene(new Scene(vBox));
        stage.show();
    }
```

```
    }
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.actionEvents
import tornadofx.*

class MyApp: App(MyView::class)

class MyView: View() {

    override val root = vbox {

        val clicks = button("Press Me").actionEvents()

        //Observer 1
        clicks.subscribe { println("Observer 1 Received Click!")
 }

        //Subscribe Observer 2 when this button is clicked
        button("Subscribe Observer 2") {
            setOnAction {
                println("Observer 2 subscribing!")
                isDisable = true
                clicks.subscribe { println("Observer 2 Received
 Click!") }
            }
        }
    }
}
```
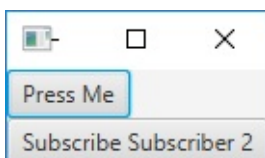
RENDERED UI:

Click the "Press Me" `Button` 3 times, then click the "Subscribe Observer 2" `Button` . Finally click "Press Me" 2 more times, and you should get this output in your console.

```
Observer 1 Received Click!
Observer 1 Received Click!
Observer 1 Received Click!
Observer 2 subscribing!
Observer 1 Received Click!
Observer 2 Received Click!
Observer 1 Received Click!
Observer 2 Received Click!
```

Notice that `Observer 1` received those first three clicks, and then we subscribed `Observer 2` . But notice that `Observer 2` has missed those first three clicks. It will never get them because it subscribed too late to a hot Observable. The only emissions `Observer 2` receives are the ones that happen after it subscribes.

After `Observer 2` is subscribed, you can see the last two emissions were pushed simultaneously to both `Observer 1` and `Observer 2` .

Again, Cold Observables will replay emissions to each `Observer` independently. Hot Observables play emissions all at once to whomever is subscribed, and it will not replay missed emissions to tardy Observers.

ConnectableObservable

We will learn several ways to create hot Observables in this book for different tasks, but one that is worth mentioning now is the `ConnectableObservable` . Among a few other subtle behaviors it creates, it can turn a cold `Observable` into a hot one by forcing its emissions to become hot. To create one, you can take any `Observable` and call its `publish()` method. You can then set up the Observers and then call `connect()` to start firing the emissions.

One reason you may do this is because it might be expensive to replay emissions for each `Observer`, especially if it is emitting items from a slow database query or some other intensive operation. Notice too that each emission interleaves and goes to each `Observer` simultaneously.

Java

```java
import io.reactivex.Observable;
import io.reactivex.observables.ConnectableObservable;

public class Launcher {

    public static void main(String[] args) {

        ConnectableObservable<String> source =
                Observable.just("Alpha","Beta","Gamma","Delta",
"Epsilon").publish();

        source.subscribe(s -> System.out.println("Observer 1: "
+ s));
        source.subscribe(s -> System.out.println("Observer 2: "
+s));

        source.connect();
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable

fun main(args: Array<String>) {
    val source = Observable.just("Alpha","Beta","Gamma","Delta",
"Epsilon").publish()

    source.subscribe { println("Observer 1: $it") }
    source.subscribe { println("Observer 2: $it") }

    source.connect()
}
```

OUTPUT:

```
Observer 1: Alpha
Observer 2: Alpha
Observer 1: Beta
Observer 2: Beta
Observer 1: Gamma
Observer 2: Gamma
Observer 1: Delta
Observer 2: Delta
Observer 1: Epsilon
Observer 2: Epsilon
```

Remember though that the `ConnectableObservable` is a hot `Observable` too, so you got to be careful when pushing data through it. If any `Observer` comes in *after* the `connect()` is called, it will miss data that was emitted previously.

Disposing

There is one last operation we need to cover: unsubscribing. Unsubscription should happen automatically for finite Observables once `onComplete()` is called. But for infinite or long-running Observables, there will be times you want to

stop the emissions and cancel the entire operation. This will also free up resources in the `Observable` chain and clean up any resources it was using.

If you want to disconnect an `Observer` from an `Observable` so it stops receiving emissions, there are a couple ways to do this. The easiest way is to note the `subscribe()` method returns a `Disposable` object. This represents the connection between the `Observable` and the `Observer` , and you can call `dispose()` on it at any time to dispose the connections so no more emissions are pushed.

For instance, let's take our incrementing `Button` example from earlier and add another `Button` that will unsubscribe the emissions. We need to save the `Disposable` returned from the `subscribe()` method, and then we can refer to it later to call `dispose()` and stop emissions.

Java

```java
import io.reactivex.disposables.Disposable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox vBox = new VBox();
        Button button = new Button("Press Me");
        Button unsubscribeButton = new Button("Unsubscribe");

        Label countLabel = new Label("0");

        Disposable disposable = JavaFxObservable.actionEventsOf(
button)
                .map(ae -> 1)
                .scan(0,(x,y) -> x + y)
                .subscribe(clickCount -> countLabel.setText(clic
kCount.toString())));

        unsubscribeButton.setOnAction(e -> disposable.dispose())
;


        vBox.getChildren().addAll(button,unsubscribeButton,count
Label);
        stage.setScene(new Scene(vBox));
        stage.show();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.actionEvents
import tornadofx.*

class MyApp: App(MyView::class)

class MyView: View() {

    override val root = vbox {

        val countLabel = label("0")

        val subscription = button("Press Me")
                .actionEvents()
                .map { 1 }
                .scan(0) {x,y -> x + y }
                .subscribe { countLabel.text = it.toString() }


        button("Unsubscribe").setOnAction {
            subscription.dispose()
        }
    }
}
```

Note that when you press the "Unsubscribe" `Button`, the increments stop because the `Observer` was disposed, and it instructed the `Observable` to stop sending emissions. Disposal automatically happens with finite Observables once `onComplete()` is called. But with infinite or long-running Observables, you need to manage their disposal if you intend to terminate them at some point.

When you have infinite Observables that need to be disposed, it is *very* critical to call `dispose()` on any Disposables when you are done with them. If you do not do this, you will run into memory leak problems and the garbage collector will not be able to free those resources.

When you have a lot of Disposables to manage and you want to dispose them all at once, you can use a `CompositeDisposable` which acts as a collection of Disposables. You can add any number of Disposables to it, and when you want to

unsubscribe all of them just call its `dispose()` method.

Java

```java
Observable<ActionEvent> buttonClicks = ...

CompositeDisposable disposables = new CompositeDisposable();

Disposable disposable1 =
    buttonclicks.subscribe(ae -> System.out.println("Clicked!"))
;

subscriptions.add(subscription1);

Disposable disposable1 =
    buttonclicks.subscribe(ae -> System.out.println("Clicked Her
e Too!"));

disposables.add(disposable1);

//work with UI, then dispose when done
disposables.dispose();
```

Kotlin

```kotlin
val buttonClicks: Observable<ActionEvent> = ...
val disposables = CompositeDisposable()

buttonclicks.subscribe { println("Clicked!") }
    .addto(disposables)

buttonclicks.subscribe { println("Clicked Here Too!") }
    .addto(disposables)

//work with UI, then dispose when done
disposables.dispose()
```

Using doOnXXX() Operators

It might be helpful to create a few "side effects" in the middle of an `Observable` chain. In other words, we want to put Observers in the middle of the chain at certain points. For instance, it might be helpful to change a "Submit" Button's text to "WORKING" when a request is being processed, as well as disable it so no more requests can be sent until the current one completes.

RxJava has `doOnXXX()` operators that allow you to "peek" into an `Observable` at that point in the chain. For instance, you can use `doOnNext()` and pass a lambda to do something with each emission, like print it. `doOnComplete()` will fire a specified action when that point of the chain received a completion notification, and `doOnError()` will do the same for an error event. Here is a complete list of these `doOnXXX()` operators in RxJava.

Operator	Example	Description
doOnNext()	doOnNext(i -> System.out.println(i))	Performs an action for each emission
doOnComplete()	doOnComplete(() -> System.out.println("Done!"))	Performs an action on completion
doOnError()	doOnError(e -> e.printStackTrace())	Performs an action on an error
doOnSubscribe()	doOnSubscribe(() -> System.out.println("Subbing!"))	Performs an action on subscription
doOnDispose()	doOnDispose(() -> System.out.println("Disposing!"))	Performs an action on unsubscription
doOnTerminate()	doOnTerminated(() -> System.out.println("I'm done or had an error"))	Performs an action for completion or an error

Summary

In this chapter we covered some RxJava fundamentals. The `Observable` treats data and events in the same way, and this is a powerful idea that applies really well with JavaFX. Cold Observables replay emissions to each `Observer` independently. Hot Observables will broadcast emissions live to all Observers simultaneously, and not replay missed emissions to tardy Observers.

This book will continue to cover RxJava and apply it in a JavaFX context. There are hundreds of operators and unfortunately we will not be able to cover them all, but we will focus on the ones that are especially helpful for building JavaFX applications. If you want to learn more about RxJava and its operators comprehensively, please check out my Packt book Learning RxJava).

In the next chapter, we are going to dive a little deeper into JavaFX events, and turn `Node` and `ObservableValue` events into Observables.

3. Events and Value Changes

In the previous chapter, we got a brief introduction to handling events reactively. But RxJavaFX is equipped to handle almost any event type for various `Node` controls. JavaFX also utilizes the `ObservableValue`, and its value changes can be turned into Observables as well.

Turning JavaFX Events into Observables

To create an `Observable` for *any* event on *any* `Node`, you can target the `Node`'s events using a `JavaFxObservable.eventsOf()` factory for Java, and the `Node` extension function `events()` for Kotlin. You can pass the `EventType` you are targeting as a parameter, and an `Observable` emitting that `EventType` will be returned.

Here is an example with a `ListView` containing `String` items representing the integers 0 through 9. Whenever a numeric key is pressed on your keyboard, it will select that item in the `ListView` (Figure 3.1).

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.input.KeyEvent;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox vBox = new VBox();

        ListView<String> listView = new ListView<>();

        for (int i = 0; i <= 9; i++) {
            listView.getItems().add(String.valueOf(i));
        }

        JavaFxObservable.eventsOf(listView, KeyEvent.KEY_TYPED)
                .map(KeyEvent::getCharacter)
                .filter(s -> s.matches("[0-9]"))
                .subscribe(s -> listView.getSelectionModel().sel
ect(s));

        vBox.getChildren().add(listView);
        stage.setScene(new Scene(vBox));
        stage.show();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.events
import javafx.scene.input.KeyEvent
import tornadofx.*

class MyApp: App(MyView::class)

class MyView: View() {

    override val root = vbox {

        listview<String> {
            (0..9).asSequence().map { it.toString() }.forEach {
items.add(it) }

            events(KeyEvent.KEY_TYPED)
                    .map { it.character }
                    .filter { it.matches(Regex("[0-9]")) }
                    .subscribe { selectionModel.select(it)}
        }
    }
}
```

Figure 3.1 - A `ListView` that "jumps" to the numeric key input

Notice above we targeted `KeyEvent.KEY_TYPED` and the returned `Observable` will emit a `KeyEvent` item every time a `KEY_TYPED` event occurs. Some events like this one have helpful information on them, such as the character `String` representing the value for that key. We used a regular expression to validate the character `String` was a single numeric character, and filter emissions that are not. Then we selected it in the `ListView` 's `SelectionModel` .

> If you want to combine keystrokes to form entire Strings rather than a series of single characters, you will want to use throttling, buffering, and switching operators to combine them based on timing windows. We will cover these later in Chapter 9.

Here is another example that targets `MouseEvent.MOVE_MOVED` events on a `Rectangle` . As you move your cursor over the `Rectangle` , the `x` and `y` positions of the cursor will be concatenated and pushed into a `Label` .

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox vBox = new VBox();

        Label positionLabel = new Label();
        Rectangle rectangle = new Rectangle(200,200);
        rectangle.setFill(Color.RED);

        JavaFxObservable.eventsOf(rectangle, MouseEvent.MOUSE_MOVED)
                .map(me -> me.getX() + "-" +  me.getY())
                .subscribe(positionLabel::setText);

        vBox.getChildren().addAll(positionLabel,rectangle);
        stage.setScene(new Scene(vBox));
        stage.show();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.events
import javafx.scene.input.MouseEvent
import javafx.scene.paint.Color
import tornadofx.*


class MyApp: App(MyView::class)


class MyView: View() {

    override val root = vbox {

        val positionLabel = label()

        rectangle(height = 200.0, width = 200.0) {

            fill = Color.RED

            events(MouseEvent.MOUSE_MOVED)
                    .map { "${it.x}-${it.y}" }
                    .subscribe { positionLabel.text = it }
        }
    }
}
```
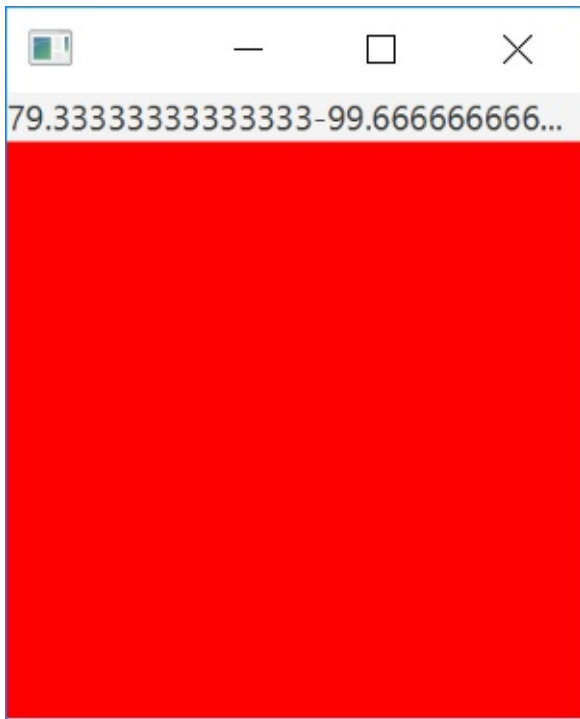
Figure 3.2 - A red rectangle that pushes the cursor coordinates when its hovered over.

JavaFX is packed with events everywhere, and you will need to know which events you are targeting on a given `Node` control. Be sure to look at the JavaDocs for the control you are using to see which event types you want to target.

Currently you can target events on `Node` , `Window` , and `Scene` types and there should be factories to support each one.

ActionEvents

In the previous chapter we were exposed to the simple `ActionEvent` . You can actually target the `ActionEvent` using the events factory and emit them through an `Observable<ActionEvent>` .

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp2 extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox vBox = new VBox();
        Button button = new Button("Press Me!");

        JavaFxObservable.eventsOf(button, ActionEvent.ACTION)
                .subscribe(ae -> System.out.println("Pressed!"))
;

        vBox.getChildren().add(button);
        stage.setScene(new Scene(vBox));
        stage.show();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.events
import javafx.event.ActionEvent
import tornadofx.*

class MyView : View() {
    override val root = hbox {
        button("Press Me")
                .events(ActionEvent.ACTION)
                .subscribe { println("Pressed!") }
    }
}
```

`ActionEvent` is a pretty common event that indicates a simple action was performed, like pressing a `Button` or `MenuItem`. It is so common that it is given its own factory as shown below, which is what we used in the previous chapter.

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox vBox = new VBox();
        Button button = new Button("Press Me!");

        JavaFxObservable.actionEventsOf(button)
                .subscribe(ae -> System.out.println("Pressed!"))
;

        vBox.getChildren().add(button);
        stage.setScene(new Scene(vBox));
        stage.show();
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.actionEvents
import tornadofx.*

class MyApp: App(MyView::class)

class MyView : View() {
    override val root = hbox {
        button("Press Me")
                .actionEvents()
                .subscribe { println("Pressed!")}
    }
}
```

Currently, the `ActionEvent` factory supports `Button` , `MenuItem` , and `ContextMenu` .

ObservableValue Changes

This is where reactive JavaFX starts to get interesting. Up to this point we only have worked with events. There is some metadata on event emissions that can be useful, but we are not quite working with data in the traditional sense.

JavaFX has many implementations of its `ObservableValue<T>` type. This is essentially a wrapper around a mutable value of a type `T` , and it notifies any listeners when the value changes. This provides a perfect opportunity to hook a listener onto it and make a reactive stream of value changes.

Create a simple UI with a `ComboBox<String>` and use the `JavaFxObservable.valuesOf()` factory to emit its value changes in a hot `Observable` .

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        HBox hBox = new HBox();
        ComboBox<String> comboBox = new ComboBox<>();
        comboBox.getItems().setAll("Alpha","Beta","Gamma","Delta"
,"Epsilon");

        JavaFxObservable.valuesOf(comboBox.valueProperty())
                .subscribe(v -> System.out.println(v + " was sel
ected"));

        hBox.getChildren().add(comboBox);
        stage.setScene(new Scene(hBox));
        stage.show();
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.toObservable
import tornadofx.*

class MyApp: App(MyView::class)

class MyView : View() {

    override val root = hbox {
        combobox<String> {

            items.setAll("Alpha","Beta","Gamma","Delta","Epsilon"
)

            valueProperty().toObservable()
                    .subscribe { println("$it was selected") }
        }
    }
}
```

When you select different items in the `ComboBox`, you should get a console output that looks something like this:

```
null was selected
Alpha was selected
Delta was selected
Epsilon was selected
```

For the next few examples, let's just focus on the `Observable` chain. Notice that the `JavaFxObservable.valuesOf()` (or `toObservable()` for Kotlin) does not push the initial null value, because RxJava 2 does not emit null values. However, you can provide a second argument to put a sentinel value for a null.

Java

```java
JavaFxObservable.valuesOf(comboBox.valueProperty(), "N/A")
        .subscribe(v -> System.out.println(v + " was selected")
);
```

Kotlin

```kotlin
valueProperty().toObservable("N/A")
        .subscribe { println("$it was selected") }
```

If you want to emit an `Optional<T>` to handlue nullability, use the `JavaFxObservable.nullableValuesOf()` factory as shown below:

Java

```java
JavaFxObservable.nullableValuesOf(comboBox.valueProperty())
        .subscribe(v -> System.out.println(v + " was selected")
);
```

Kotlin

```kotlin
valueProperty().toNullableObservable()
        .subscribe { println("$it was selected") }
```

Remember that we can use any RxJava operators. We can `map()` each String's `length()` and push that to the `Observer`.

Java

```java
JavaFxObservable.valuesOf(comboBox.valueProperty())
        .map(String::length)
        .subscribe(i ->
            System.out.println("A String with length " + i + " w
as selected")
        );
```

Kotlin

```
valueProperty().toObservable()
        .map { it.length }
        .subscribe { println("A String with length $it was selec
ted") }
```

OUTPUT:

```
A String with length 5 was selected
A String with length 4 was selected
A String with length 7 was selected
A String with length 4 was selected
```

Let's get a little more creative, and use `scan()` to do a rolling sum of the lengths with each emission.

Java

```
JavaFxObservable.valuesOf(comboBox.valueProperty())
        .map(String::length)
        .scan(0,(x,y) -> x + y)
        .subscribe(i -> System.out.println("Rolling length total
: " + i));
```

Kotlin

```
valueProperty().toObservable()
        .map { it.length }
        .scan(0,(x,y) -> x + y)
        .subscribe { println("Rolling length total: $it") }
```

When you make a few selections to the `ComboBox` , your output should look something like this depending on which Strings you selected.

OUTPUT:

```
Rolling length total: 0
Rolling length total: 5
Rolling length total: 10
Rolling length total: 17
Rolling length total: 22
Rolling length total: 26
Rolling length total: 31
```

This example may be a bit contrived, but hopefully you are starting to see some of the possibilities when you have a chain of operators "reacting" to a change in a `ComboBox` . Pushing each value every time it is selected in a `ComboBox` allows you to quickly tell other parts of the UI to update accordingly.

Again, you can use this factory on *any* `ObservableValue` . This means you can hook into any JavaFX component property and track its changes reactively. The possibilities are quite vast. For instance, for every selection event in a `ComboBox` , you can query a database for items of that selection, and populate them into a `TableView` . Then that `TableView` may have Observables built off its events and properties to trigger other controls to update.

> You might be wondering if making lots of `ComboBox` selections resulting in expensive queries could overwhelm the application. If the queries are that expensive, yes that will happen. But in Chapter 9 we will learn how to switch, throttle, and buffer which will resolve this issue effectively.

You also have the option of pushing the old and new value in a `Change` item through the `changesOf()` factory. This can be helpful for validation, and you can restore that old value back into the control if the new value fails to meet a condition.

For example, you can emit the old value and new value together on each typed character in a `TextField` . If at any moment the text is not numeric (or is an empty `String`), the previous value can be restored immediately using the `Observer` .

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        HBox hBox = new HBox();
        TextField textField = new TextField();

        JavaFxObservable.changesOf(textField.textProperty())
                .map(s -> s.getNewVal().matches("[0-9]+") ? s.ge
tNewVal() : s.getOldVal())
                .subscribe(textField::setText);

        hBox.getChildren().add(textField);
        stage.setScene(new Scene(hBox));
        stage.show();
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.toObservableChanges
import tornadofx.*

class MyView : View() {

    override val root = hbox {
        textfield {
            textProperty().toObservableChanges()
                    .filter { !it.newVal.matches(Regex("[0-9]+")
) }
                    .map { it.oldVal }
                    .subscribe {
                        text = it
                    }
        }
    }
}
```

If you study the `Observable` operation above, you can see that each `Change` item is emitted holding the old and new value for each text input. Using a regular expression, we validated for text inputs that are not numeric or are empty. We then `map()` it back to the old value and set it to the `TextField` in the `Observer`.

Error Recovery

When working with Observables built off UI events, sometimes an error can occur which will be communicated up the chain via `onError()`. In production, you should always have the `Observer` handle an `onError()` so the error does not just quietly disappear. But when you are dealing with UI input events, there is likely one other error handling issue to consider.

Say you have this simple JavaFX `Application` with a `Button` that adds a numeric input from a `TextField`, and adds it to a total in a `Label` (Figure 3.3).

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox root = new VBox();

        Label label = new Label("Input Number");
        TextField input = new TextField();
        Label totalLabel = new Label();

        Button button = new Button("Add to Total");

        JavaFxObservable.actionEventsOf(button)
                .map(ae -> Integer.valueOf(input.getText()))
                .scan(0,(x,y) -> x + y)
                .subscribe(i -> {
                    totalLabel.setText(i.toString());
                    input.clear();
                }, e -> new Alert(Alert.AlertType.ERROR, e.getMe
ssage()).show());

        root.getChildren().setAll(label,input, totalLabel, butto
n);
        stage.setScene(new Scene(root));
        stage.show();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.actionEvents
import io.reactivex.rxkotlin.subscribeBy
import javafx.scene.control.Alert
import tornadofx.*

class MyApp: App(MyView::class)

class MyView: View() {

    override val root = vbox {

        label("Input Number")
        val input = textfield()
        val totalLabel = label("")

        button("Add to Total").actionEvents()
                .map { input.text.toInt() }
                .scan(0) {x,y -> x + y }
                .subscribeBy(
                    onNext  = {
                        totalLabel.text = it.toString()
                        input.clear()
                    },
                    onError  = { Alert(Alert.AlertType.ERROR, it
.message).show() }
                )
    }
}
```
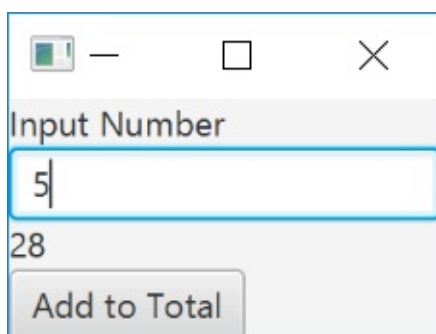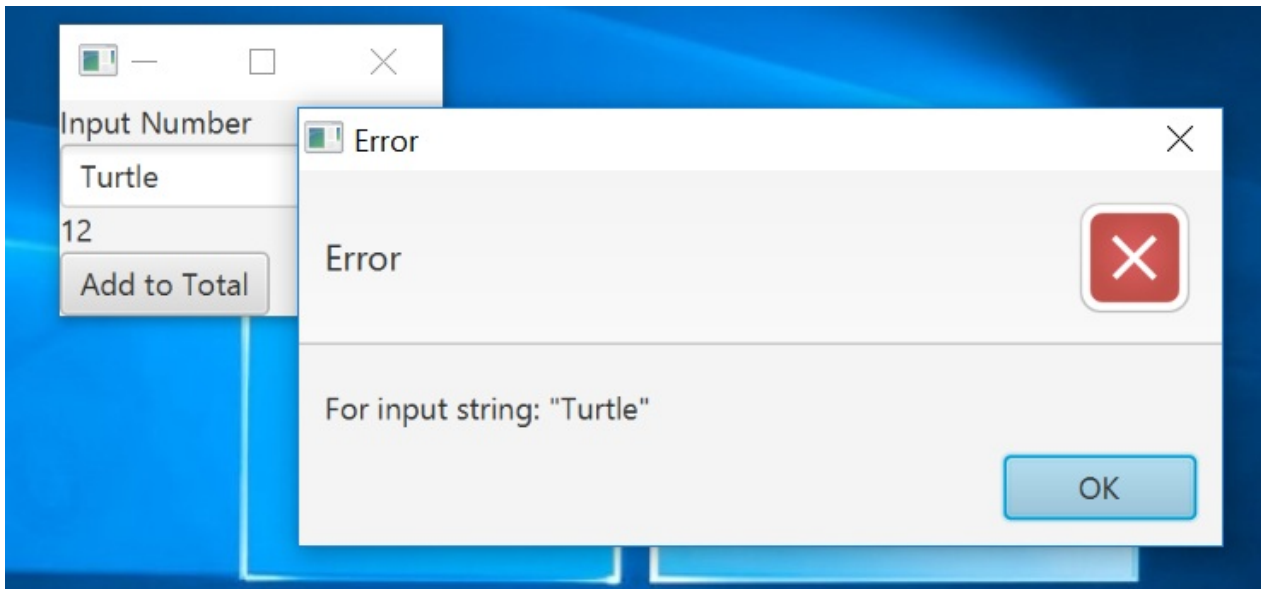
Figure 3.3

Input Number

5

28

Add to Total

That `TextField` should only have numeric inputs, but nothing is stopping non-numeric inputs from being emitted. Therefore, if you type a non-numeric value in that `TextField` and click the `Button` , you will get an `Alert` as specified in the Observers's `onError()` (Figure 3.4).

Figure 3.4

Despite the error being handled, there is one problem here. The `Observable` is now dead. It called `onError()` and closed the stream, assuming nothing could be done to recover from it. You will find the `Button` is no longer sending emissions. You can fix this by adding the `retry()` operator right before the `Observer` . When its `onError()` is called, it will intercept and swallow the error, then resubscribe again.

Java

```java
JavaFxObservable.actionEventsOf(button)
     .map(ae -> Integer.valueOf(input.getText()))
     .scan(0,(x,y) -> x + y)
     .retry()
     .subscribe(i -> {
         totalLabel.setText(i.toString());
         input.clear();
     }, e -> new Alert(Alert.AlertType.ERROR, e.getMessage()).show());
```

Kotlin

```kotlin
button("Add to Total").actionEvents()
        .map { input.text.toInt() }
        .scan(0) {x,y -> x + y }
        .retry()
        .subscribeBy(
            onNext  = {
                totalLabel.text = it.toString()
                input.clear()
            },
            onError  = { Alert(Alert.AlertType.ERROR, it.message
).show() }
        )
```

If you type in a non-numeric input, it will resubscribe and start all over. The `scan()` operator will send another initial emission of `0` and result in everything being reset. You also have the option of moving the `retry()` before the `scan()` operation (so it intercepts the error before the `scan()`), and that would maintain the current rolling total rather than canceling it and starting over at 0.

But notice that the `onError()` in the `Observer` is never called, and we never get an `Alert` . This is because the `retry()` intercepted the `onError()` call and kept it from going to the `Observer` . To get the `Alert` , you may want to move it to a `doOnError()` operator before the `retry()` . The error will flow through it to trigger the `Alert` before the `retry()` intercepts it.

Java

```
JavaFxObservable.actionEventsOf(button)
      .map(ae -> Integer.valueOf(input.getText())))
      .scan(0,(x,y) -> x + y)
      .doOnError( e -> new Alert(Alert.AlertType.ERROR, e.getMes
sage()).show())
      .retry()
      .subscribe(i -> {
          totalLabel.setText(i.toString());
          input.clear();
      });
```

Kotlin

```
button("Add to Total").actionEvents()
        .map { input.text.toInt() }
        .scan(0) {x,y -> x + y }
        .doOnError { Alert(Alert.AlertType.ERROR, it.message).sh
ow() }
        .retry()
        .subscribe {
                totalLabel.text = it.toString()
                input.clear()
        }
```

By default, `retry()` will resubscribe an unlimited number of times for an unlimited number of errors. This means for cold data sources, this can spiral out of control quickly by retrying an infinite number of times! You can pass an `Integer` argument like `retry(3)` so that it will only retry three times and the fourth `onError()` will go to the `Observer`. There is also a `retryWhen()` operator that allows you to conditionally resubscribe based on some attribute of the error (like its type).

There are a couple of error-handling operators in RxJava that are worth being familiar with. But for UI input controls, you will likely want to leverage `retry()` so Observables built off UI controls do not remain dead after an error. This is especially critical if you are kicking off complex reactive processes.

It is also worth noting that the best way to handle errors is to handle them proactively. In this example, it would have been good to forbid numbers from being entered in the `TextField` in the first place (like our previous exercise). Another valid check would be to `filter()` out non-numeric values so they are suppressed before being turned into an `Integer`.

Java

```
JavaFxObservable.actionEventsOf(button)
    .map(ae -> input.getText())
    .filter(s -> s.matches("[0-9]+"))
    .map(Integer::valueOf)
    .scan(0,(x,y) -> x + y)
    .subscribe(i -> {
        totalLabel.setText(i.toString());
        input.clear();
    });
```

Kotlin

```
button("Add to Total").actionEvents()
    .map { input.text }
    .filter { it.matches(Regex("[0-6]+")) }
    .map { it.toInt() }
    .scan(0) {x,y -> x + y }
    .subscribeBy(
        onNext  = {
            totalLabel.text = it.toString()
            input.clear()
        },
        onError  = { Alert(Alert.AlertType.ERROR, it.message).sh
ow() }
    )
```

Summary

In this chapter, we learned the basic RxJavaFX/RxKotlinFX factories to create RxJava Observables off JavaFX Events and ObservableValues. Definitely spend some time experimenting with this small but powerful set of factories that can be applied almost anywhere in the JavaFX API. We also learned how to resubscribe Observables built off UI events in the event an `onError()` occurs.

But there are a few more facilities we need to be truly productive with reative JavaFX, starting next with JavaFX Collections. This is where the line between data and events truly become blurred in surpringly useful ways.

4. Collections and Data

Any sizable application needs to work with data and collections of items. One of the greatest utilities to come out of JavaFX are ObservableCollections such as `ObservableList` , `ObservableSet` , and `ObservableMap` . These implementations of `List` , `Set` , and `Map` are built specifically for JavaFX to notify the UI when it has been modified, and any control built off it will visually update accordingly.

However, these ObservableCollections can have custom listeners added to them. This creates an opportunity to reactively work with data through collections. The idea of emitting a collection every time it changes allows some surprisingly useful reactive transformations, and we will see plenty of examples in this chapter.

> Do not confuse the JavaFX `ObservableValue` , `ObservableList` , `ObservableSet` , and `ObservableMap` to somehow be related to the RxJava `Observable` . This is not the case. Remember that JavaFX's concept of an `Observable` is not the same as an RxJava `Observable` . However, we will turn all of these into an RxJava `Observable` to fully utilize their capabilities.

Emitting an Observable Collection

Let's create a simple application backed by an `ObservableList` of Strings. There will be a `ListView<String>` to display these values, and another `ListView<Integer>` that will hold their distinct lengths. We will use a `TextField` and a `Button` to add Strings to the `ObservableList` , and both ListViews should update accordingly with each addition.

You should get a UI that looks like Figure 4.1 when you run the code below.

Java

```
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
```

```java
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ListView;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox root = new VBox();

        ObservableList<String> values =
                FXCollections.observableArrayList("Alpha","Beta",
"Gamma");

        Label valuesLabel = new Label("VALUES");
        ListView<String> valuesListView = new ListView<>(values)
;

        Label distinctLengthsLabel = new Label("DISTINCT LENGTHS"
);
        ListView<Integer> distinctLengthsListView = new ListView
<>();

        JavaFxObservable.emitOnChanged(values)
                .flatMapSingle(list ->
                        Observable.fromIterable(list).map(String
::length).distinct().toList()
                ).subscribe(lengths -> distinctLengthsListView.g
etItems().setAll(lengths));

        TextField inputField = new TextField();
        Button addButton = new Button("ADD");
```

```java
        JavaFxObservable.actionEventsOf(addButton)
                .map(ae -> inputField.getText())
                .filter(s -> s != null && !s.trim().isEmpty())
                .subscribe(s -> {
                    values.add(s);
                    inputField.clear();
                });

        root.getChildren().addAll(valuesLabel,valuesListView,dis
tinctLengthsLabel,
                distinctLengthsListView,inputField,addButton);

        stage.setScene(new Scene(root));
        stage.show();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.actionEvents
import com.github.thomasnield.rxkotlinfx.onChangedObservable
import io.reactivex.rxkotlin.toObservable
import javafx.collections.FXCollections
import tornadofx.*

class MyApp: App(MyView::class)


class MyView: View() {

    val values = FXCollections.observableArrayList("Alpha", "Bet
a", "Gamma")

    override val root = vbox {

        label("VALUES")
        listview(values)

        label("DISTINCT LENGTHS")
```
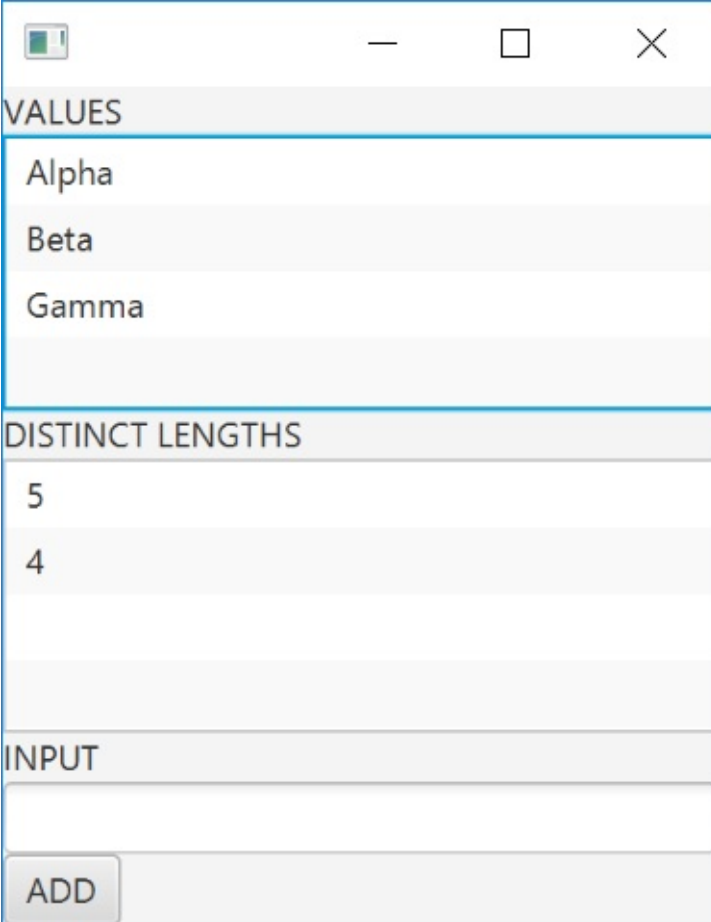
```
        listview<Int> {
            values.onChangedObservable()
                    .flatMapSingle {
                        it.toObservable().map { it.length }.dist
 inct().toList()
                    }.subscribe {
                items.setAll(it)
            }
        }



        label("INPUT")
        val inputField = textfield()

        button("ADD").actionEvents()
                .map { inputField.text }
                .filter { !it.trim().isEmpty() }
                .subscribe {
                    values.add(it)
                    inputField.clear()
                }
    }
 }
```

Figure 4.1
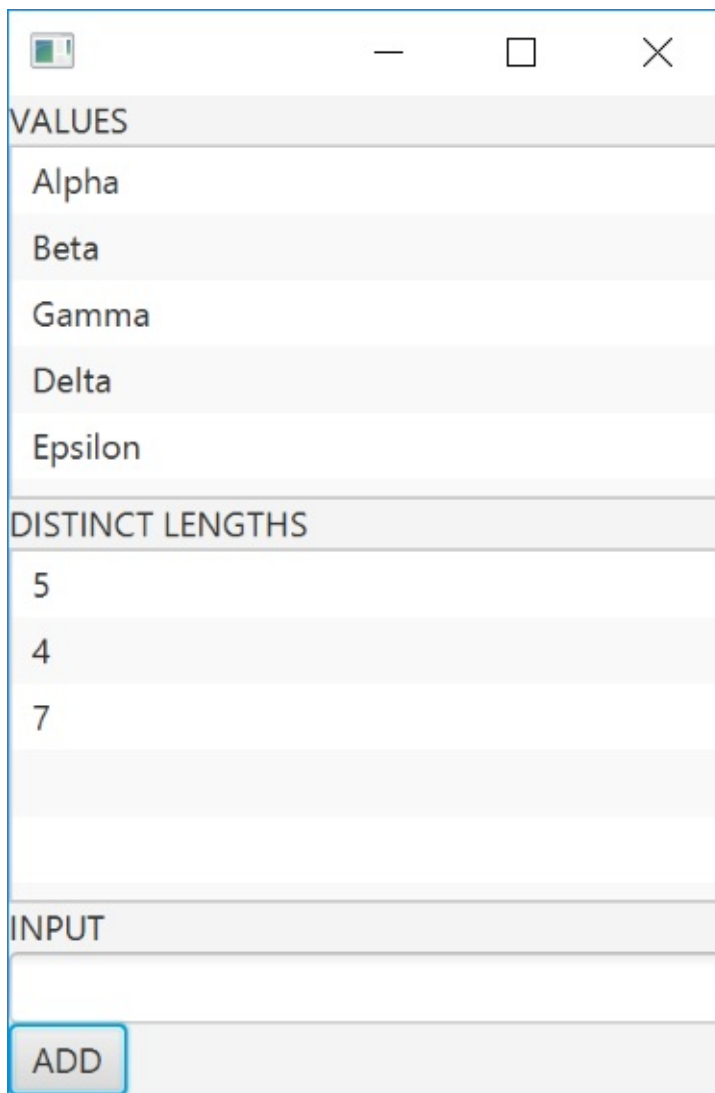
Go ahead and type in "Delta", then click "ADD". Then do the same for "Epsilon". You should now see Figure 4.2.

Figure 4.2

See that? Not only did "Delta" and "Epsilon" get added to the top `ListView` , but the distinct length of 7 was added to the bottom one. So how exactly was this made possible?

Study the code above very closely. We declared an `ObservableList<String>` called `values` . All the magic is built around it. We created an `Observable<ObservableList<String>>` off it using `JavaFxObservable.emitOnChanged()` (or `onChangedObservable()` for Kotlin). While the type looks a little strange the idea is very simple: every time the `ObservableList<String>` changes, it is pushed through the `Observable<ObservableList<String>>` in its entirety as an emission. It will also emit the `ObservableList` on subscription as the initial emission.

This is a useful pattern because as we have just seen, we can transform this `ObservableList` emission inside a `flatMap()` any way we want. In this example, we effectively created a new `ObservableList<Integer>` that

receives the distinct lengths of the `ObservableList<String>`.

> Note the placement of operators is very critical! The `toList()` operator
> occured inside the `flatMapSingle()` where it was working with a finite
> `Observable` derived from an `ObservableList`. Putting that `toList()`
> outside a `flatMap()` will cause it to work against an infinite `Observable`,
> and it will forever collect items and never emit.

Let's leverage this idea in another way. Instead of putting the distinct lengths in
another `ObservableList<Integer>`, let's concatenate them as a `String` and
push it into a `Label`'s text.

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ListView;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox root = new VBox();

        ObservableList<String> values =
                FXCollections.observableArrayList("Alpha","Beta",
"Gamma");

        Label valuesLabel = new Label("VALUES");
```

```
        ListView<String> valuesListView = new ListView<>(values)
;

        Label distinctLengthsLabel = new Label("DISTINCT LENGTHS"
);
        Label distinctLengthsConcatLabel= new Label();
        distinctLengthsConcatLabel.setTextFill(Color.RED);

        JavaFxObservable.emitOnChanged(values)
                .flatMapSingle(list ->
                        Observable.fromIterable(list)
                                .map(String::length)
                                .distinct().reduce("",(x,y) -> x
 + (x.equals("") ? "" : "|") + y)
                ).subscribe(distinctLengthsConcatLabel::setText)
;

        TextField inputField = new TextField();
        Button addButton = new Button("ADD");

        JavaFxObservable.actionEventsOf(addButton)
                .map(ae -> inputField.getText())
                .filter(s -> s != null && !s.trim().isEmpty())
                .subscribe(s -> {
                    values.add(s);
                    inputField.clear();
                });

        root.getChildren().addAll(valuesLabel,valuesListView,dis
tinctLengthsLabel,
                distinctLengthsConcatLabel,inputField,addButton)
;

        stage.setScene(new Scene(root));
        stage.show();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.actionEvents
import com.github.thomasnield.rxkotlinfx.onChangedObservable
import io.reactivex.rxkotlin.toObservable
import javafx.collections.FXCollections
import javafx.scene.paint.Color
import tornadofx.*

class MyApp: App(MyView::class)

class MyView: View() {

    val values = FXCollections.observableArrayList("Alpha","Beta"
,"Gamma")

    override val root = vbox {

        label("VALUES")
        listview(values)

        label("DISTINCT LENGTHS")
        label {
            textFill = Color.RED

            values.onChangedObservable()
                    .flatMapSingle {
                        it.toObservable()
                                    .map { it.length }
                                    .distinct()
                                    .reduce("") { x,y -> x + (if (x
== "") "" else "|") + y }
                    }.subscribe {
                text = it
            }
        }


        label("INPUT")
        val inputField = textfield()
```
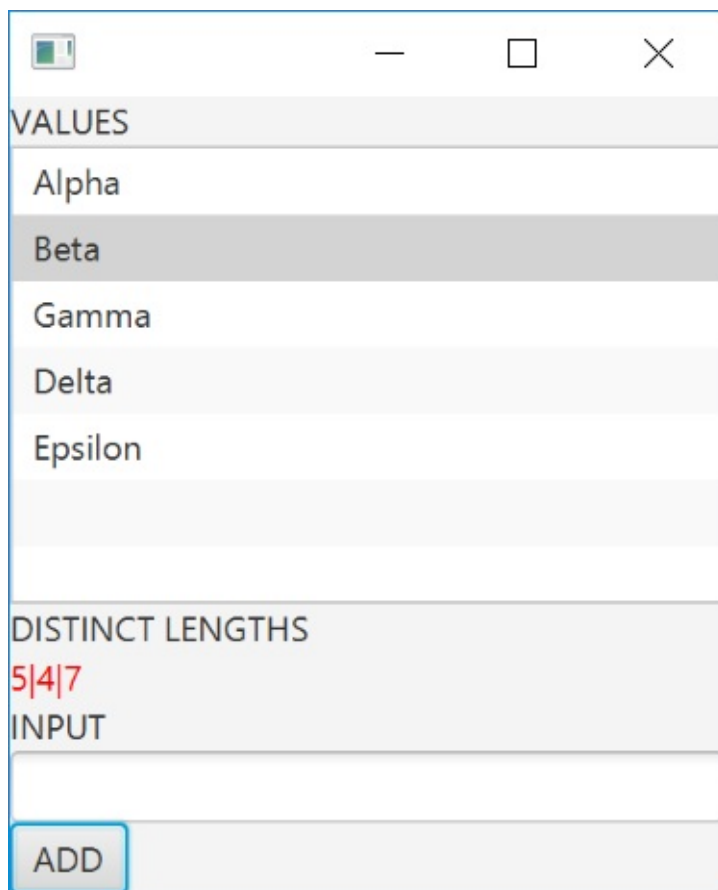
```
        button("ADD").actionEvents()
                .map { inputField.text }
                .filter { !it.trim().isEmpty()}
                .subscribe {
                    values.add(it)
                    inputField.clear()
                }
        }
    }
```

Figure 4.3

Awesome, right? We are pushing a transformation of the `ObservableList` source and driving a `Label` 's text with it. Simply using an `Observable` and `Observer` , we can easily do limitless transformations of data and events that are almost impractical to do in standard JavaFX.

Note also there are factories for `ObservableSet` and `ObservableMap` to accomplish the same behavior. `JavaFxObservable.emitOnChanged()` will emit an `ObservableSet` every time it changes,

Java

```java
public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        ObservableSet<String> values =
                FXCollections.observableSet("Alpha","Beta","Gamm
a");

        JavaFxObservable.emitOnChanged(values)
                .subscribe(System.out::println);

        values.add("Delta");
        values.add("Alpha"); //no effect

        values.remove("Beta");

        System.exit(0); //quit
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.onChangedObservable
import javafx.collections.FXCollections
import tornadofx.*

class MyApp: App(MyView::class)

class MyView: View() {
    override val root = vbox {
        val values = FXCollections.observableSet("Alpha","Beta",
"Gamma")

        values.onChangedObservable()
                .subscribe { println(it) }

        values.add("Delta")
        values.add("Alpha") //no effect

        values.remove("Beta")

        System.exit(0) //quit
    }
}
```

OUTPUT:

```
[Alpha, Gamma, Beta]
[Alpha, Gamma, Delta, Beta]
[Alpha, Gamma, Delta]
```

`JavaFxObservable.emitOnChanged()` will also emit an `ObservableMap` every time it changes.

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableMap;
import javafx.stage.Stage;


public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        ObservableMap<Integer,String> values =
                FXCollections.observableHashMap();

        JavaFxObservable.emitOnChanged(values)
                .subscribe(System.out::println);

        values.put(1,"Alpha");
        values.put(2,"Beta");
        values.put(3,"Gamma");
        values.put(1,"Alpha"); //no effect
        values.put(3,"Delta");
        values.remove(2);

        System.exit(0);
    }
}
```

Kotlin

```
import io.reactivex.rxjavafx.observables.JavaFxObservable
import javafx.collections.FXCollections
import tornadofx.*

class MyApp: App(MyView::class)

class MyView: View() {
    override val root = vbox {
        val values = FXCollections.observableHashMap<Int, String>()

        JavaFxObservable.emitOnChanged(values)
                .subscribe { println(it) }

        values.put(1, "Alpha")
        values.put(2, "Beta")
        values.put(3, "Gamma")
        values.put(1, "Alpha") //no effect
        values.put(3, "Delta")
        values.remove(2)

        System.exit(0);
    }
}
```

OUTPUT:

```
{}
{1=Alpha}
{1=Alpha, 2=Beta}
{1=Alpha, 2=Beta, 3=Gamma}
{1=Alpha, 2=Beta, 3=Delta}
{1=Alpha, 3=Delta}
```

Add, Remove, and Update Events

There are factories for `ObservableList` , `ObservableSet` , and
`ObservableMap` to emit specific change events against those collections. To get
an emission for each modification to an `ObservableList` , you can use
`changesOf()` . It will pair each affected element `T` with a `Flag` in a
`ListChange` emission. The `Flag` is an enum with possible values `ADDED` ,
`REMOVED` , or `UPDATED` .

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        ObservableList<String> values =
                FXCollections.observableArrayList("Alpha","Beta",
"Gamma");

        JavaFxObservable.changesOf(values)
                .subscribe(System.out::println);

        values.add("Delta");
        values.add("Epsilon");
        values.remove("Alpha");
        values.set(2,"Eta");

        System.exit(0);
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.changes
import javafx.collections.FXCollections
import tornadofx.*

class MyApp: App(MyView::class)

class MyView: View() {
    override val root = vbox {
        val values = FXCollections.observableArrayList("Alpha",
"Beta", "Gamma")

        values.changes().subscribe { println(it) }

        values.add("Delta")
        values.add("Epsilon")
        values.remove("Alpha")
        values[2] = "Eta"

        System.exit(0)
    }
}
```

OUTPUT:

```
ADDED Delta
ADDED Epsilon
REMOVED Alpha
ADDED Eta
REMOVED Delta
```

There are equivalent factories for `ObservableMap` and `ObservableSet` as well, which are overloads for `changesOf()`.

Note that this factory has no initial emission. It will only emit changes going forward after subscription. A `ListChange` is emitted with the affected value and whether it was `ADDED`, `REMOVED`, or `UPDATED`. Interestingly, note that calling `set()` on the `ObservableList` will replace an element at a given index, and

result in two emissions: one for the `REMOVED` item, and another for the `ADDED` item. When we set the item at index `2` to "Eta", it replaced "Delta" which was `REMOVED` , and then "Eta" was `ADDED` .

An `UPDATED` emission occurs when an `ObservableValue` property of a `T` item in an `ObservableList<T>` changes. This is a lesser-known feature in JavaFX but can be enormously helpful. Consider a `User` class with an updateable `Property` called `name` .

Java

```java
class User {
    private final int id;
    private final Property<String> name =
        new SimpleStringProperty();

    User(int id, String name) {
        this.id = id;
        this.name.setValue(name);
    }
    public int getId() {
        return id;
    }
    public Property<String> nameProperty() {
        return name;
    }
    @Override
    public String toString() {
        return id + "-" + name.getValue();
    }
}
```

Kotlin

```kotlin
class User(val id: Int, name: String) {
    var name: String by property(name)
    fun nameProperty() = getProperty(User::name)

    override fun toString() = "$id-$name"
}
```

Whenever this `name` property for any `User` changes, this change will be pushed as an emission. It will be categorized in a `ListChange` as `UPDATED` .

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.stage.Stage;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.Property;
import javafx.beans.value.ObservableValue;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        ObservableList<User> values =
                FXCollections.observableArrayList(user -> new Ob
servableValue[]{ user.nameProperty() });

        JavaFxObservable.changesOf(values)
                .subscribe(System.out::println);

        values.add(new User(503,"Tom Nield"));
        values.add(new User(504,"Jason Shwartz"));

        values.get(0).nameProperty().setValue("Thomas Nield");

        System.exit(0);
```

```java
    }

    static final class User {
        private final int id;
        private final Property<String> name = new SimpleStringPr
operty();

        User(int id, String name) {
            this.id = id;
            this.name.setValue(name);
        }
        public int getId() {
            return id;
        }
        public Property<String> nameProperty() {
            return name;
        }
        @Override
        public String toString() {
            return id + "-" + name.getValue();
        }
    }
}
```

Kotlin

```kotlin
import io.reactivex.rxjavafx.observables.JavaFxObservable
import javafx.collections.FXCollections
import tornadofx.*
import javafx.beans.value.ObservableValue


class MyApp: App(MyView::class)


class MyView: View() {
    override val root = vbox {
        val values = FXCollections.observableArrayList<User> { u
ser ->
            arrayOf<ObservableValue<*>>(user.nameProperty())
        }

        JavaFxObservable.changesOf(values)
                .subscribe { println(it) }

        values.add(User(503, "Tom Nield"))
        values.add(User(504, "Jason Shwartz"))

        values[0].nameProperty().value = "Thomas Nield"

        System.exit(0)
    }
}

class User(val id: Int, name: String) {
    var name: String by property(name)
    fun nameProperty() = getProperty(User::name)

    override fun toString() = "$id-$name"
}
```

OUTPUT:

```
ADDED 503-Tom Nield
ADDED 504-Jason Shwartz
UPDATED 503-Thomas Nield
```

We declared a lambda specifying an array of `ObservableValue` properties we are interested in listening to, which in this case is only the `name` property. When the first element containing the `User` named "Tom Nield" had its `name` property changed to `Thomas Nield`, it was emitted as a change. This will also work with the `emitOnChanged()` factory we saw earlier, and the entire `ObservableList<T>` will be pushed every time any specified property changes.

This can be helpful to react not just to items in the list being added or removed, but also when their properties are modified. You can then use this behavior to, for example, to drive updates to concatenations.

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        ObservableList<User> values =
                FXCollections.observableArrayList(user -> new Ob
servableValue[]{user.nameProperty()});

        JavaFxObservable.emitOnChanged(values)
                .flatMapSingle(list ->
                        Observable.fromIterable(list)
                                .map(User::getName)
                                .reduce("",(u1,u2) -> u1 + (u1.e
quals("") ? "" : ", ") + u2)
                )
                .subscribe(System.out::println);

        values.add(new User(503,"Tom Nield"));
```

```java
        values.add(new User(504,"Jason Shwartz"));

        values.get(0).nameProperty().setValue("Thomas Nield");

        System.exit(0);
    }
    static final class User {
        private final int id;
        private final Property<String> name = new SimpleStringPr
operty();

        User(int id, String name) {
            this.id = id;
            this.name.setValue(name);
        }
        public int getId() {
            return id;
        }
        public Property<String> nameProperty() {
            return name;
        }
        @Override
        public String toString() {
            return id + "-" + name.getValue();
        }
    }
}
```

Kotlin

```kotlin
import io.reactivex.rxjavafx.observables.JavaFxObservable
import io.reactivex.rxkotlin.toObservable
import javafx.collections.FXCollections
import tornadofx.*
import javafx.beans.value.ObservableValue

class MyApp: App(MyView::class)

class MyView: View() {
    override val root = vbox {
        val values = FXCollections.observableArrayList<User> { u
ser ->
            arrayOf<ObservableValue<*>>(user.nameProperty())
        }

        JavaFxObservable.emitOnChanged(values)
                .flatMapSingle {
                    it.toObservable().map {it.name }
                            .reduce("") { u1,u2 -> u1 + (if (u1
== "") "" else ", ") + u2 }
                }
                .subscribe { println(it) }

        values.add(User(503, "Tom Nield"))
        values.add(User(504, "Jason Shwartz"))

        values[0].nameProperty().value = "Thomas Nield"

        System.exit(0)
    }
}

class User(val id: Int, name: String) {
    var name: String by property(name)
    fun nameProperty() = getProperty(User::name)

    override fun toString() = "$id-$name"
}
```

OUTPUT:

```
Tom Nield
Tom Nield, Jason Shwartz
Thomas Nield, Jason Shwartz
```

Note also there are factories that target only `ADDED` , `REMOVED` , and `UPDATED` events. These will only emit items corresponding to those event types, and also are available under the `JavaFxObservable` utility class. Here is a complete list of these additional factories as well as the others we covered so far.

Figure 4.4 - JavaFX Collection Factories

Collection Type	Java Factory	Kotlin Extensio
ObservableList<T>	emitOnChanged()	onChangedObserva
ObservableList<T>	additionsOf()	additions()
ObservableList<T>	removalsOf()	removals()
ObservableList<T>	updatesOf()	updates()
ObservableSet<T>	emitOnChanged()	onChangedObserva
ObservableSet<T>	additionsOf()	additions()
ObservableSet<T>	removalsOf()	removals()
ObservableSet<T>	fromObservableSetUpdates()	updates()
ObservableMap<T>	emitOnChanged()	onChangedObserva
ObservableMap<<K,T>	additionsOf()	additions()
ObservableMap<K,T>	removalsOf()	removals()
ObservableMap<K,T>	fromObservableMapUpdates()	updates()

Distinct ObservableList Changes

There may be times you want to emit only *distinct* changes to a JavaFX `ObservableList`. What this means is you want to ignore duplicates added or removed to the collection and not emit them as a change. This can be helpful to synchronize two different ObservableLists, where one has duplicates and the other does not.

Take this application that will hold two `ListView<String>` instances each backed by an `ObservableList<String>`. The top `ListView<String>` will hold duplicate values, but the bottom `ListView<String>` will hold only distinct values from the top `ListView` (Figure 4.5).

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import io.reactivex.rxjavafx.sources.Flag;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ListView;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox root = new VBox();

        Label header1 = new Label("VALUES");
        ListView<String> listView = new ListView<>();

        Label header2 = new Label("DISTINCT VALUES");
        ListView<String> distinctListView = new ListView<>();

        JavaFxObservable.distinctChangesOf(listView.getItems())
                .subscribe(c -> {
                    if (c.getFlag().equals(Flag.ADDED))
```

```
                        distinctListView.getItems().add(c.getVal
ue());
                    else
                        distinctListView.getItems().remove(c.get
Value());
                });

        TextField inputField = new TextField();

        Button addButton = new Button("Add");
        JavaFxObservable.actionEventsOf(addButton)
                .map(ae -> inputField.getText())
                .subscribe(s -> {
                    listView.getItems().add(s);
                    inputField.clear();
                });

        Button removeButton = new Button("Remove");
        JavaFxObservable.actionEventsOf(removeButton)
                .map(ae -> inputField.getText())
                .subscribe(s -> {
                    listView.getItems().remove(s);
                    inputField.clear();
                });

        root.getChildren().addAll(header1,listView,header2,
                distinctListView,inputField,addButton,removeButt
on);

        stage.setScene(new Scene(root));
        stage.show();
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.actionEvents
import com.github.thomasnield.rxkotlinfx.distinctChanges
import io.reactivex.rxjavafx.sources.Flag
```

```
import tornadofx.*

class MyApp: App(MyView::class)

class MyView: View() {
    override val root = vbox {

        label("Values")
        val listView = listview<String>()

        label("Distinct Values")
        val distinctListView = listview<String>()

        label("Input/Remove Value")
        val inputField = textfield()

        listView.items.distinctChanges()
                .subscribe {
                    if (it.flag == Flag.ADDED)
                        distinctListView.items.add(it.value)
                    else
                        distinctListView.items.remove(it.value)
                }

        button("Add").actionEvents()
                .map { inputField.text }
                .subscribe {
                    listView.items.add(it)
                    inputField.clear()
                }

        button("Remove").actionEvents()
                .map { inputField.text }
                .subscribe {
                    listView.items.remove(it)
                    inputField.clear()
                }
    }
}
```
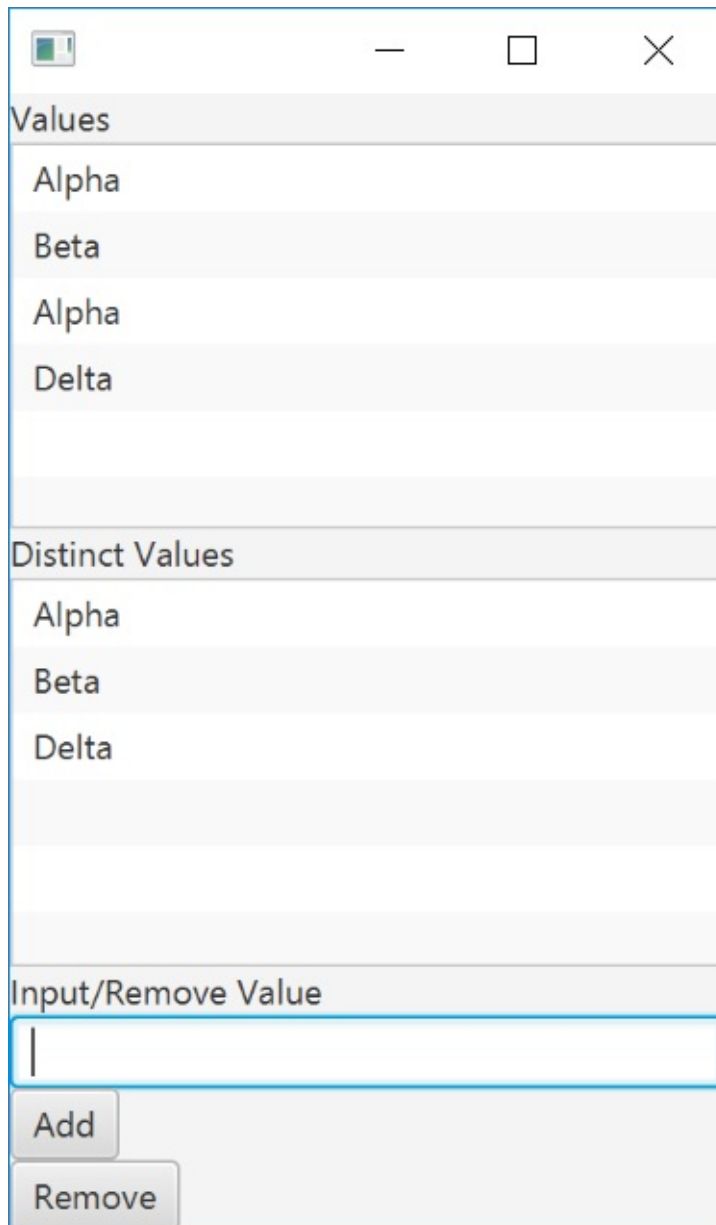
Figure 4.5

The key factory here is the `distinctChangesOf()` (or `distinctChanges()` for Kotlin). It pushes only *distinct* changes from the top `ListView` to the bottom one. If you input "Alpha" twice, the top `ListView` will hold both instances, but the bottom will only receive one. The second `ADDED` emission was suppressed. If you remove one of the "Alpha" values, it will not fire the `REMOVED` emission until you rid the other one too.

You also have the option of choosing an attribute of the item to distinct on rather than the item itself. If you wanted to only emit distinct values based on the first character, you can pass a lambda argument to the factory that substrings out the first character.
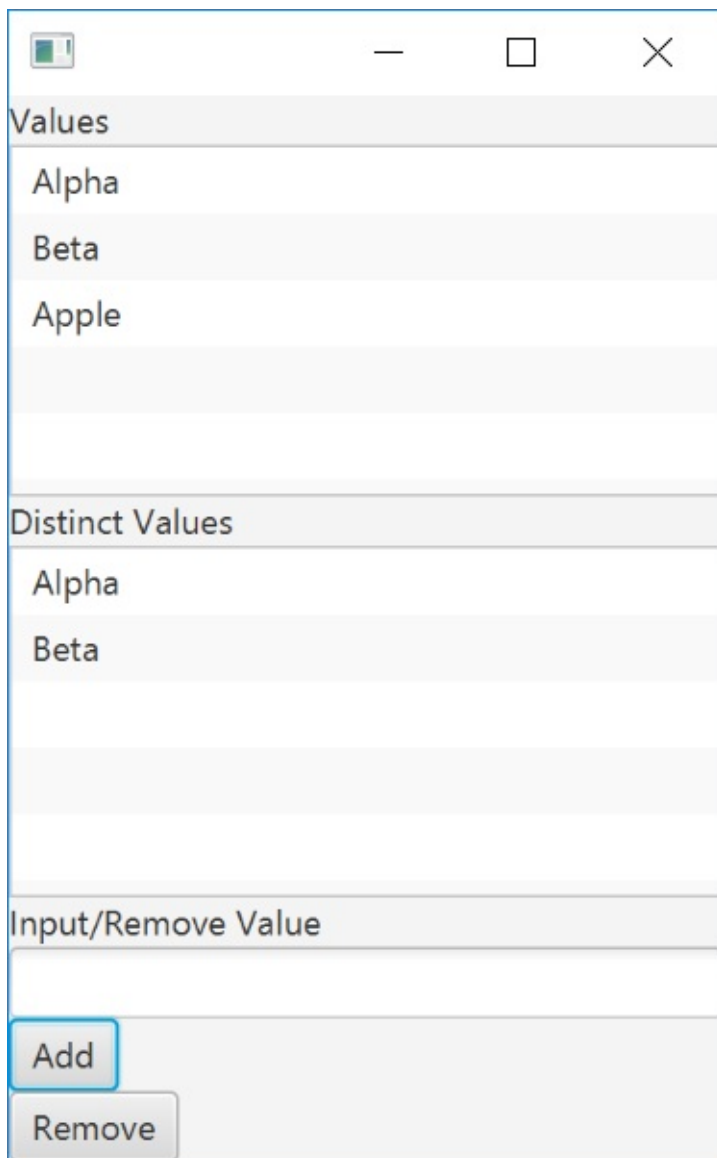
Java

```java
 JavaFxObservable.distinctChangesOf(listView.getItems(), s -> s.
substring(0,1))
        .subscribe(c -> {
            if (c.getFlag().equals(Flag.ADDED))
                distinctListView.getItems().add(c.getValue());
            else
                distinctListView.getItems().remove(c.getValue());
        });
```

Kotlin

```kotlin
listView.items.distinctChanges { it.substring(0,1) }
    .subscribe {
        if (it.flag == Flag.ADDED)
            distinctListView.items.add(it.value)
        else
            distinctListView.items.remove(it.value)
    }
```

Figure 4.6

As you may see, this might be helpful to sample only one item with a distinct property. If you add "Alpha" and then "Apple", only "Alpha" will be emitted to the bottom `ListView` since it was the first to start with "A". The "Alpha" will only be removed from the bottom `ListView` when both "Alpha" and "Apple" are removed, when there are no longer any "A" samples.

If you want to push the mapped value itself rather than the item it was derived from, you can use the `distinctMappingsOf()` factory (or `distinctMappingChanges()` for Kotlin) (Figure 4.7).
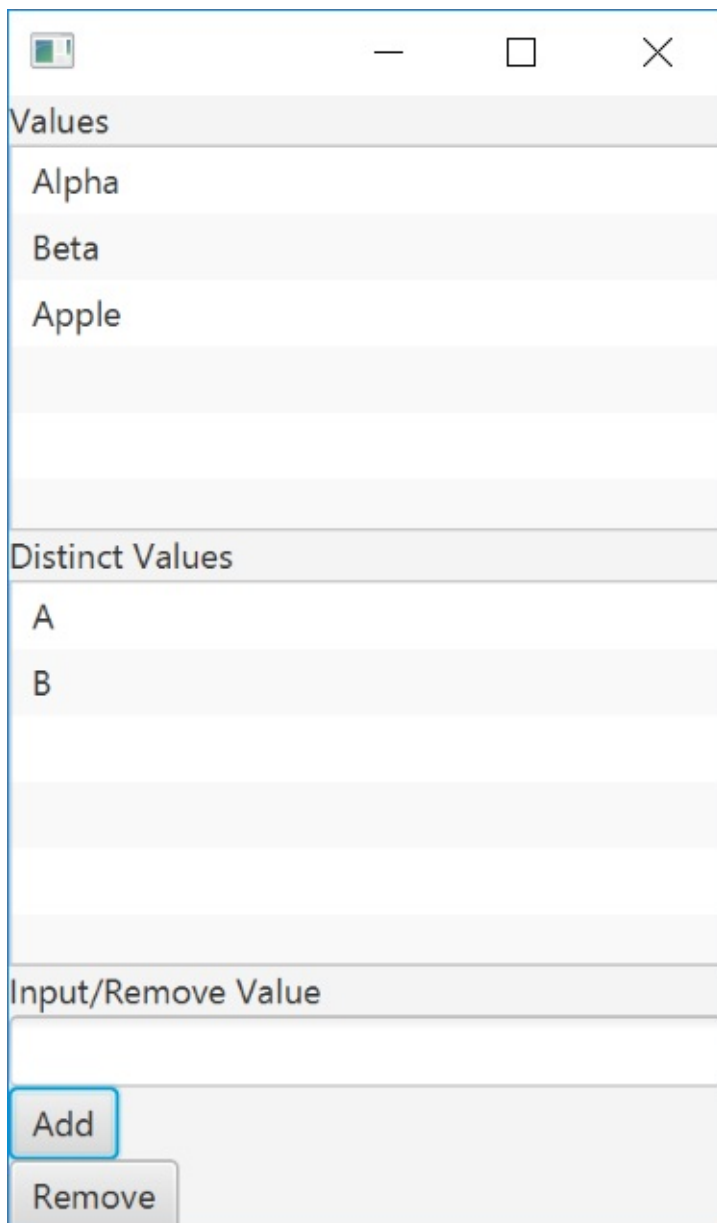
Java

```
JavaFxObservable.distinctMappingsOf(listView.getItems(), s -> s.
substring(0,1))
                .subscribe(c -> {
                    if (c.getFlag().equals(Flag.ADDED))
                        distinctListView.getItems().add(c.getVal
ue());
                    else
                        distinctListView.getItems().remove(c.get
Value());
                });
```

Kotlin

```
listView.items.distinctMappingChanges { it.substring(0,1) }
            .subscribe {
                if (it.flag == Flag.ADDED)
                    distinctListView.items.add(it.value)
                else
                    distinctListView.items.remove(it.value)
            }
```

Figure 4.7

If you input "Alpha", an "A" will show up in the bottom `ListView`. Adding "Apple" will have no effect as "A" (its first character) has already been distincly `ADDED`. When you remove both "Alpha" and "Apple", the "A" will then be `REMOVED` from the bottom.

Summary

In this chapter we covered how to reactivly use JavaFX ObservableCollections. When you emit an entire collection every time it changes, or emit the elements that changed, you can get a lot of functionality that simply does not exist with

JavaFX alone. We also covered distinct additions and removals, which can be helpful to create an `ObservableList` that distincts off of another `ObservableList` .

Hopefully by now, RxJava is slowly starting to look useful. But we have only just gotten started. The power of Rx really starts to unfold when we combine Observables, leverage concurrency, and use other features that traditionally take a lot of effort. Next, we will cover combining Observables.

5. Combining Observables

So far in this book, we have merely set the stage to make Rx useful. We learned how to emit JavaFX Events, ObservableValues, and ObservableCollections through RxJava Observables. But there is only so much you can do when a reactive stream is built off one source. When you have emissions from multiple Observables being joined together in some form, this is truly where things transition from merely being useful to definitely game-changing.

There are several ways to combine emissions from multiple Observables, and we will cover many of these combine operators. What makes these operators especially powerful is they are not only threadsafe, but also non-blocking. They can merge concurrent sources from different threads, and we will see this in action later in Chapter 7.

Concatenation

One of the simplest ways to combine Observables is to use the `concat()` operators. You can specify two or more Observables emitting the same type `T` and it will fire emissions from each one in order.

Java

```java
import io.reactivex.Observable;

public static void main(String[] args) {
    Observable<String> source1 = Observable.just("Alpha","Beta",
"Gamma");
    Observable<String> source2 = Observable.just("Delta","Epsilo
n");

    Observable.concat(source1,source2)
        .map(String::length)
        .toList()
        .subscribe(System.out::println);
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import io.reactivex.rxkotlin.subscribeBy

fun main(args: Array<String>) {

    val source1 = Observable.just("Alpha","Beta","Gamma")
    val source2 = Observable.just("Delta","Epsilon")

    Observable.concat(source1,source2)
            .map { it.length }
            .toList()
            .subscribeBy { println(it) }
}
```

OUTPUT:

```
[5, 4, 5, 5, 7]
```

It is very critical to note that `onComplete()` must be called by each `Observable` so it moves on to the next one. If you have an infinite `Observable` in a concatenated operation, it will hold up the line by infinitely emitting items, forever keeping any Observables after it from getting fired.

Concatentation is also available as an operator and not just a factory, and it should yield the same output.

Java

```java
import io.reactivex.Observable;

public static void main(String[] args) {
    Observable<String> source1 = Observable.just("Alpha","Beta",
"Gamma");
    Observable<String> source2 = Observable.just("Delta","Epsilo
n");

    source1.concatWith(source2)
        .map(String::length)
        .toList()
        .subscribe(System.out::println);
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import io.reactivex.rxkotlin.subscribeBy

fun main(args: Array<String>) {

    val source1 = Observable.just("Alpha","Beta","Gamma")
    val source2 = Observable.just("Delta","Epsilon")

    source1.concatWith(source2)
            .map { it.length }
            .toList()
            .subscribeBy { println(it) }
}
```

OUTPUT:

```
[5, 4, 5, 5, 7]
```

If you want to do a concenation but put another `Observable` in front rather than after, you can use `startWith()` instead.

Java

```java
import io.reactivex.Observable;

public static void main(String[] args) {
    Observable<String> source1 = Observable.just("Alpha","Beta",
"Gamma");
    Observable<String> source2 = Observable.just("Delta","Epsilo
n");

    source1.startWith(source2)
        .subscribe(System.out::println);
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import io.reactivex.rxkotlin.subscribeBy

fun main(args: Array<String>) {

    val source1 = Observable.just("Alpha","Beta","Gamma")
    val source2 = Observable.just("Delta","Epsilon")

    source1.startWith(source2)
            .subscribe { println(it) }
}
```

OUTPUT:

```
Delta
Epsilon
Alpha
Beta
Gamma
```

Again, this operator is likely not one you would use with infinite Observables. You are more likely to use this for data-driven Observables rather than UI events. Technically, you can specify an infinite `Observable` to be the last `Observable`

to concatenate. That way it is not holding up any other Observables.

When you want to simultaneously combine all emissions from all Observables, you might want to consider using `merge()`, which we will cover next.

Merging

Merging is almost like concatenation but with one important difference: it will combine all Observables of a given emission type `T` *simultaneously*. This means all emissions from all Observables are merged together at once into a single stream without any regard for order or completion.

This is pretty helpful to merge multiple UI event sources since they are infinite. For instance, you can consolidate the `ActionEvent` emissions of two buttons into a single `Observable<ActionEvent>` using `Observable.merge()`. (Figure 5.1).

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        VBox root = new VBox();

        Button firstButton = new Button("Press Me");
        Button secondButton = new Button("Press Me Too");

        Observable.merge(
                JavaFxObservable.actionEventsOf(firstButton),
                JavaFxObservable.actionEventsOf(secondButton)
        ).subscribe(i -> System.out.println("You pressed one of
the buttons!"));

        root.getChildren().addAll(firstButton,secondButton);
        stage.setScene(new Scene(root));
        stage.show();
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.actionEvents
import io.reactivex.Observable
import tornadofx.*

class MyApp: App(MyView::class)


class MyView: View() {
    override val root = vbox {
        val buttonUp = button("Press Me")
        val buttonDown = button("Press Me Too")

        Observable.merge(
                buttonUp.actionEvents(),
                buttonDown.actionEvents()
        )
        .subscribe {
            println("You pressed one of the buttons!")
        }
    }
}
```

Figure 5.1
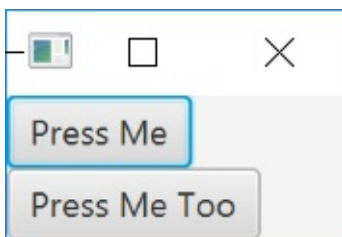
When you press either `Button`, it will consolidate the emissions into a single `Observable<ActionEvent>` which goes to a single `Observer`. But let's make this more interesting. Change these two Buttons so they are labeled "UP" and "DOWN", and map their `ActionEvent` to either a `1` or `-1` respectively. Using a `scan()` we can create a rolling sum of these emissions and push the incrementing/decrementing number to a `Label` (Figure 5.2).

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        VBox root = new VBox();

        Label label = new Label("0");

        Button buttonUp = new Button("UP");
        Button buttonDown = new Button("DOWN");

        Observable.merge(
                JavaFxObservable.actionEventsOf(buttonUp).map(ae
 -> 1),
                JavaFxObservable.actionEventsOf(buttonDown).map(
ae -> -1)
        ).scan(0,(x,y) -> x + y)
                .subscribe(i -> label.setText(i.toString()));

        root.getChildren().addAll(label,buttonUp,buttonDown);
        stage.setScene(new Scene(root));
        stage.show();
    }
}
```

Kotlins

```
import com.github.thomasnield.rxkotlinfx.actionEvents
import io.reactivex.Observable
import tornadofx.*

class MyApp: App(MyView::class)

class MyView: View() {
    override val root = vbox {
        val label = label("0")
        val buttonUp = button("UP")
        val buttonDown = button("DOWN")

        Observable.merge(
                buttonUp.actionEvents().map { 1 },
                buttonDown.actionEvents().map { -1 }
        ).scan(0) { x,y -> x + y }
        .subscribe {
            label.text = it.toString()
        }
    }
}
```

Figure 5.2
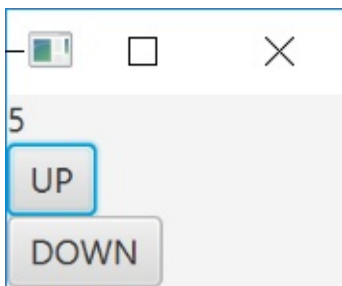
When you press the "UP" `Button` , it will increment the integer in the `Label` . When you press the "DOWN" `Button` , it will decrement it. This was accomplished by merging the two infinite Observables returned from the `map()` operator. The `1` or `-1` is then pushed to the `scan()` operation where it is emitted as a rolling total.

Like concatenation, there is also an operator version you can use instead of the factory to merge an `Observable<T>` with another `Observable<T>`

Java

```java
JavaFxObservable.actionEventsOf(buttonUp).map(ae -> 1)
    .mergeWith(
        JavaFxObservable.actionEventsOf(buttonDown).map(ae -> -1
)
    ).scan(0,(x,y) -> x + y)
    .subscribe(i -> label.setText(i.toString()));
```

Kotlin

```kotlin
buttonUp.actionEvents().map { 1 }
    .mergeWith(
        buttonDown.actionEvents().map { -1 }
    ).scan(0) { x,y -> x + y }
    .subscribe {
        label.text = it.toString()
    }
```

With both concatentation and merging, you can combine as many Observables as you want (up to 9 before you have to pass an `Iterable<Observable>` instead). But these two operators work with Observables emitting the same type `T`. There are ways to combine emissions of different types which we will see next.

Zip

One way you can combine multiple Observables, even if they are different types, is by "zipping" their emissions together. Think of a zipper on a jacket and how the teeth pair up. From a reactive perspective, this means taking one emission from the first `Observable`, and one from a second `Observable`, and combining both emissions together in some way.

Take these two Observables, one emitting Strings and the other emitting Integers. For each `String` that is emitted, you can pair it with an emitted `Integer` and join them together somehow.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {

        Observable<String> letters = Observable.just("A", "B", "C", "D", "E", "F");
        Observable<Integer> numbers = Observable.just(1, 2, 3, 4, 5);

        Observable.zip(letters, numbers, (l, n) -> l + "-" + n)
                .subscribe(System.out::println,
                        Throwable::printStackTrace,
                        () -> System.out.println("Done!")
                );
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import io.reactivex.rxkotlin.Observables
import io.reactivex.rxkotlin.subscribeBy

fun main(args: Array<String>) {

    val letters = Observable.just("A","B","C","D","E","F")
    val numbers = Observable.just(1,2,3,4,5)

    Observables.zip(letters,numbers) { l, n -> "$l-$n"}
            .subscribeBy(
                onNext  = { println(it) },
                onError = { it.printStackTrace() },
                onComplete = { println("Done!") }
            )
}
```

OUTPUT:

```
A-1
B-2
C-3
D-4
E-5
Done!
```

Notice that "A" paired with the "1", and "B" paired with the "2", and so on. Again, you are "zipping" them just like a jacket zipper. But take careful note of something: there are 6 `letters` emissions and 5 `numbers` emissions. What happened to that sixth letter "F" since it had no number to zip with? Since the two zipped sources do not have the same number of emissions, it was ignored the moment `onComplete()` was called by `numbers`. Logically, it will never have anything to pair with so it gave up and proceeded to skip it and call `onComplete()` down to the `Subscriber`.

There is also an operator equivalent called `zipWith()` you can use. This should yield the exact same output.

Java

```java
import io.reactivex.Observable;

public class Launcher {

    public static void main(String[] args) {

        Observable<String> letters = Observable.just("A", "B", "C", "D", "E", "F");
        Observable<Integer> numbers = Observable.just(1, 2, 3, 4, 5);

        letters.zipWith(numbers, (l, n) -> l + "-" + n)
                .subscribe(System.out::println,
                        Throwable::printStackTrace,
                        () -> System.out.println("Done!")
                );
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import io.reactivex.rxkotlin.subscribeBy
import io.reactivex.rxkotlin.zipWith

fun main(args: Array<String>) {

    val letters = Observable.just("A","B","C","D","E","F")
    val numbers = Observable.just(1,2,3,4,5)

    letters.zipWith(numbers) {l,n -> "$l-$n"}
            .subscribeBy(
                onNext  = { println(it) },
                onError = { it.printStackTrace() },
                onComplete = { println("Done!") }
            )
}
```

Zipping can be helpful when you need to sequentially pair things from two or more sources, but from my experience this rarely works well with UI events. Let's adapt this example to see why.

Suppose you create two `ComboBox` controls holding these letters and numbers respectively. You want to create an `Observable` off each one that emits the selected values. Then you want to zip the values together, concatentate them into a single `String`, and print them in an `Observer`. You are looking to combine two different user inputs together (Figure 5.3).

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        ComboBox<String> letterCombo = new ComboBox<>();
        letterCombo.getItems().setAll("A", "B", "C", "D", "E", "F");

        ComboBox<Integer> numberCombo = new ComboBox<>();
        numberCombo.getItems().setAll(1,2,3,4,5);

        Observable<String> letterSelections =
                JavaFxObservable.valuesOf(letterCombo.valuePrope
rty());

        Observable<Integer> numberSelections =
                JavaFxObservable.valuesOf(numberCombo.valuePrope
rty());
```

```java
        Observable.zip(letterSelections, numberSelections, (l, n
) -> l + "-" + n)
                .subscribe(System.out::println,
                        Throwable::printStackTrace,
                        () -> System.out.println("Done!")
                );


        HBox root = new HBox();
        root.getChildren().setAll(letterCombo,numberCombo);


        stage.setScene(new Scene(root));
        stage.show();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.toObservable
import io.reactivex.rxkotlin.Observables
import io.reactivex.rxkotlin.subscribeBy
import tornadofx.*


class MyApp: App(MyView::class)

class MyView : View() {

    override val root = hbox {

        val letterSelections = combobox<String> {
            items.setAll("A","B","C","D","E","F")
        }.valueProperty().toObservable()

        val numberSelections = combobox<Int> {
            items.setAll(1,2,3,4,5)
        }.valueProperty().toObservable()

        Observables.zip(letterSelections,numberSelections) { l,
 n -> "$l-$n"}
                .subscribeBy(
                        onNext  = { println(it) },
                        onError = { it.printStackTrace() },
                        onComplete = { println("Done!") }
                )
    }
}
```
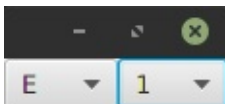
Figure 5.3

This seems like a good idea, right? When I select a letter, and I select a number, the two are zipped together and sent to the `Observer`! But there is something subtle and problematic with this. Select multiple letters without selecting any numbers, *then* select multiple numbers. Notice how the letters are backlogged and

each one is waiting for a number to be paired with? This is problematic and probably not what you want. If you select "A", then "B", then "C" followed by "1", then "2", then "3", you are going to get "A-1", "B-2", and "C-3" printed to the console.

Here is another way of looking at it. The problem with our zipping example is for every selected "letter", you need to select a "number" to evenly pair with it. If you make several selections to one combo box and neglect to make selections on the other, you are going to have a backlog of emissions waiting to be paired. If you select eight different letters (shown below), and only four numbers, the next number you select is going to pair with the "D", not "F" which is currently selected. If you select another letter its only going to worsen the backlog and make it more confusing as to what the next number will pair with.

```
A 1
B 5
A 3
A 6
D
C
A
F
```

If you want to only combine the *latest* values from each `Observable` and ignore previous ones, you might want to use `combineLatest()` which we will cover next.

Note you can make zip more than two Observables. If you have three Observables, it will zip three emissions together before consolidating them into a single emission.

Combine Latest

With our zipping example earlier, it might be more expected if we combine values by chasing after the *latest* values. Using `combineLatest()` instead of `zip()`, we can select a value in either `ComboBox`. Then it will emit with the latest value from the other `ComboBox`.

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        ComboBox<String> letterCombo = new ComboBox<>();
        letterCombo.getItems().setAll("A", "B", "C", "D", "E", "F");

        ComboBox<Integer> numberCombo = new ComboBox<>();
        numberCombo.getItems().setAll(1,2,3,4,5);

        Observable<String> letterSelections =
                JavaFxObservable.valuesOf(letterCombo.valueProperty());

        Observable<Integer> numberSelections =
                JavaFxObservable.valuesOf(numberCombo.valueProperty());

        Observable.combineLatest(letterSelections, numberSelections, (l, n) -> l + "-" + n)
                .subscribe(System.out::println,
                        Throwable::printStackTrace,
                        () -> System.out.println("Done!")
                );

        HBox root = new HBox();
        root.getChildren().setAll(letterCombo,numberCombo);
```

```java
        stage.setScene(new Scene(root));
        stage.show();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.toObservable
import io.reactivex.rxkotlin.Observables
import io.reactivex.rxkotlin.subscribeBy
import tornadofx.*



class MyApp: App(MyView::class)

class MyView : View() {

    override val root = hbox {

        val letterSelections = combobox<String> {
            items.setAll("A","B","C","D","E","F")
        }.valueProperty().toObservable()

        val numberSelections = combobox<Int> {
            items.setAll(1,2,3,4,5)
        }.valueProperty().toObservable()

        Observables.combineLatest(letterSelections,numberSelecti
ons) {l,n -> "$l-$n"}
                .subscribeBy(
                        onNext  = { println(it) },
                        onError = { it.printStackTrace() },
                        onComplete = { println("Done!") }
                )
    }
}
```

If you select "A","4", "E", and then "1", you should get this output.

OUTPUT:

```
A-4
E-4
E-1
```

Selecting "4" with "A" will emit with the latest letter "A". Then selecting "E" will emit with the latest number "4", and finally selecting "1" will emit with "E".

Simply put, a change in value for either `ComboBox` will result in the latest value for both being pushed forward. For combining UI input events, we often are only concerned with the latest user inputs and do not care about previous ones. Therefore, `combineLatest()` is often useful for JavaFX.

Another powerful usage of `combineLatest()` is merging two ObservableLists into one, and always keeping it synchronized when additions or removals happen to the ObservableLists it was derived off of.

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.stage.Stage;
import java.util.ArrayList;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        //declare two ObservableLists
        ObservableList<String> startLocations =
                FXCollections.observableArrayList("Dallas", "Houston", "Chicago", "Boston");

        ObservableList<String> endLocations =
```

```java
                FXCollections.observableArrayList("San Diego", "
Salt Lake City", "Seattle");

        //this ObservableList will hold contents of both
        ObservableList<String> allLocations = FXCollections.obse
rvableArrayList();

        //this will pump both ObservableLists into `allLocations`

        Observable.combineLatest(
                JavaFxObservable.emitOnChanged(startLocations),
                JavaFxObservable.emitOnChanged(endLocations),
                (l1, l2) -> {
                    ArrayList<String> combined = new ArrayList<>
();

                    combined.addAll(l1);
                    combined.addAll(l2);
                    return combined;
                }
        ).subscribe(allLocations::setAll);

        //print `allLocations` every time it changes, to prove i
ts working
        JavaFxObservable.emitOnChanged(allLocations).subscribe(S
ystem.out::println);

        //do modifications to trigger above operations
        startLocations.add("Portland");
        endLocations.add("Dallas");
        endLocations.add("Phoenix");
        startLocations.remove("Boston");

        System.exit(0);
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.onChangedObservable
```

```kotlin
import io.reactivex.rxkotlin.Observables
import javafx.collections.FXCollections
import tornadofx.*



class MyApp: App(MyView::class)


class MyView : View() {

    override val root = hbox {

        //declare two ObservableLists
        val startLocations =
                FXCollections.observableArrayList("Dallas","Houston","Chicago","Boston")

        val endLocations =
                FXCollections.observableArrayList("San Diego", "Salt Lake City", "Seattle")

        //this ObservableList will hold contents of both
        val allLocations = FXCollections.observableArrayList<String>()

        //this will pump both ObservableLists into `allLocations`

        Observables.combineLatest(startLocations.onChangedObservable(),
                endLocations.onChangedObservable()) {l1,l2 ->
            ArrayList<String>().apply {
                addAll(l1)
                addAll(l2)
            }
        }.subscribe {
            allLocations.setAll(it)
        }

        //print `allLocations` every time it changes, to prove its working
        allLocations.onChangedObservable().subscribe { println(i
```

```
   t) }

        //do modifications to trigger above operations
        startLocations.add("Portland")
        endLocations.add("Dallas")
        endLocations.add("Phoenix")
        startLocations.remove("Boston")

        System.exit(0)
    }
}
```

OUTPUT:

```
[Dallas, Houston, Chicago, Boston, San Diego, Salt Lake City, Se
attle]
[Dallas, Houston, Chicago, Boston, Portland, San Diego, Salt Lak
e City, Seattle]
[Dallas, Houston, Chicago, Boston, Portland, San Diego, Salt Lak
e City, Seattle, Dallas]
[Dallas, Houston, Chicago, Boston, Portland, San Diego, Salt Lak
e City, Seattle, Dallas, Phoenix]
[Dallas, Houston, Chicago, Portland, San Diego, Salt Lake City,
Seattle, Dallas, Phoenix]
```

Whenever either `ObservableList` (`startLocations` or `endLocations`) is modified, it will update the combined `ObservableList` (`allLocations`) so it always reflects the contents of both. This is a powerful way to leverage JavaFX ObservableCollections and combine them to drive the content of other ObservableCollections.

If you want to go a step further, you can easily modify this operation so that the combined `ObservableList` only contains *distinct* items from both ObservableLists. Simply add a `flatMapSingle()` before the `Observer` that intercepts the `ArrayList` , turns it into an `Observable` , distincts it, and collects it back into a `List` . Notice when you run it, the duplicate "Dallas" emission is held back.

Java

```java
Observable.combineLatest(
        JavaFxObservable.emitOnChanged(startLocations),
        JavaFxObservable.emitOnChanged(endLocations),
        (l1,l2) -> {
            ArrayList<String> combined = new ArrayList<>();
            combined.addAll(l1);
            combined.addAll(l2);
            return combined;
        }
).flatMapSingle(l -> Observable.fromIterable(l).distinct().toList())
  .subscribe(allLocations::setAll);
```

Kotlin

```kotlin
Observables.combineLatest(startLocations.onChangedObservable(),
        endLocations.onChangedObservable()) {l1,l2 ->
      mutableListOf<String>().apply {
          addAll(l1)
          addAll(l2)
      }
}.flatMapSingle {
    it.toObservable().distinct().toList()
}.subscribe {
    allLocations.setAll(it)
}
```

OUTPUT:

```
[Dallas, Houston, Chicago, Boston, San Diego, Salt Lake City, Se
attle]
[Dallas, Houston, Chicago, Boston, Portland, San Diego, Salt Lak
e City, Seattle]
[Dallas, Houston, Chicago, Boston, Portland, San Diego, Salt Lak
e City, Seattle]
[Dallas, Houston, Chicago, Boston, Portland, San Diego, Salt Lak
e City, Seattle, Phoenix]
[Dallas, Houston, Chicago, Portland, San Diego, Salt Lake City,
Seattle, Phoenix]
```

While this is a pretty procedural example, using `combineLatest()` with
ObservableLists has very powerful applications, especially with data controls.
Combining data from two different data controls (like TableViews), you can merge
the two data sets into some form of aggregation in a third control. All three data
controls will always be synchronized, and you can published the combined
`ObservableList` while it is driven by two or more ObservableCollections
backing it.

A more advanced but elegant way to accomplish either task above is to return an
`Observable<Observable<String>>` from the `combineLatest()` , and then
flatten it with a `flatMap()` afterwards. This avoids creating an intermediary
`ArrayList` and is a bit leaner.

Java

```java
Observable.combineLatest(
        JavaFxObservable.emitOnChanged(startLocations),
        JavaFxObservable.emitOnChanged(endLocations),
        (l1,l2) -> Observable.fromIterable(l1).concatWith(Observ
able.fromIterable(l2))
).flatMapSingle(obs -> obs.distinct().toList())
  .subscribe(allLocations::setAll);
```

Kotlin

```
Observables.combineLatest(startLocations.onChangedObservable(),
        endLocations.onChangedObservable()) { l1,l2 ->
    l1.toObservable().concatWith(l2.toObservable())
}.flatMapSingle {
    it.distinct().toList()
}.subscribe {
    allLocations.setAll(it)
}
```

This is somewhat more advanced, so do not worry if you find the code above challenging to grasp. It is a pattern where an `Observable` is emitting Observables, and you can feel free to move on and study it again later as you get more comfortable with Rx.

Summary

In this chapter, we covered combining Observables and which combine operators are helpful to use with UI events vs simply merging data. Hopefully by now, you are excited that you can achieve tasks beyond what the JavaFX API provides. Tasks like synchronizing an `ObservableList` to the contents of two other ObservableLists become almost trival with reactive programming. Soon we will get to the most anticipated feature of RxJava: concurrency with `observeOn()` and `subscribeOn()`. But first, we will cover a few final topics before we hit that.

6. Bindings

There are situations where JavaFX will want a `Binding` rather than an RxJava `Observable` or `Observer`, and we will cover some streamlined utilities to meet this need. We will also cover JavaFX Dialogs and how to use them reactively.

Bindings and RxJava

In JavaFX, a `Binding` is an implementation of `ObservableValue` that is derived off other ObservableValues in some way. Bindings also allow you to synchronize JavaFX `ObservableValue` items through `bind()` and `bindBidirectional()` methods. You can express transformations of an `ObservableValue` and bind on those transformations, but RxJava expresses this task much more easily. As you probably observed, RxJavaFX provides a robust and expressive way to make controls communicate their changes.

For instance, you can leverage bindings to disable a `Button` when a `TextField` does not contain six characters.

Java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        VBox root = new VBox();

        Label label = new Label("Input a 6-character String");

        TextField input = new TextField();
        Button button = new Button("Proceed");

        button.disableProperty().bind(
                input.textProperty().length().isNotEqualTo(6)
        );
        root.getChildren().addAll(label,input,button);
        stage.setScene(new Scene(root));
        stage.show();
    }
}
```

Kotlin

```
import tornadofx.*


class MyApp: App(MyView::class)

class MyView: View() {

    override val root = vbox {

        label("Input a 6-character String")

        val input = textfield()
        val button = button("Proceed")

        button.disableProperty().bind(
                input.textProperty().length().isNotEqualTo(6)
        )
    }
}
```

Figure 6.1 Using bindings to disable a `Button` unless a `TextField` is six characters

Of course, the need for `Binding` in this case is eliminated thanks to RxJava. Knowing what you know now, RxJava creates a more streamlined and inuitive way to "push" the `input` text values, map them to a boolean expression, and finally sends them to an `Observer` that sets the `disableProperty()` of the `Button` .

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        VBox root = new VBox();

        Label label = new Label("Input a 6-character String");

        TextField input = new TextField();
        Button button = new Button("Proceed");

        JavaFxObservable.valuesOf(input.textProperty())
                .map(s -> s.length() != 6)
                .subscribe(b -> button.disableProperty().setValu
e(b));

        root.getChildren().addAll(label,input,button);
        stage.setScene(new Scene(root));
        stage.show();
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.toObservable
import tornadofx.*


class MyApp: App(MyView::class)

class MyView: View() {

    override val root = vbox {

        label("Input a 6-character String")

        val input = textfield()
        val button = button("Proceed")

        input.textProperty().toObservable()
                .map { it.length  != 6 }
                .subscribe { button.disableProperty().set(it) }
    }
}
```

If you are fluent in Rx, this is more intuitive than native JavaFX Bindings. It is also much more flexible as a given `ObservableValue` remains openly mutable rather than being strictly bound to another `ObservableValue`. But there are times you will need to use Bindings to fully work with the JavaFX API. If you need to create a `Binding` off an RxJava `Observable`, there is a factory/extension function to turn an RxJava `Observable<T>` into a JavaFX `Binding<T>`. Let's take a look at one place where Bindings are needed: TableViews.

Say you have the given domain type `Person`. It has a `birthday` property that holds a `LocalDate`. The `getAge()` is an `Observable<Long>` driven off the `birthday` and is converted to a `Binding<Long>`. When you change the `birthday`, it will push a new `Long` value to the `Binding` (Figure 6.2).

Java

```
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import io.reactivex.rxjavafx.observers.JavaFxObserver;
```

```java
import javafx.beans.binding.Binding;
import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleObjectProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;


import java.time.LocalDate;
import java.time.temporal.ChronoUnit;


public final class Person {

    private final StringProperty name;
    private final ObjectProperty<LocalDate> birthday;
    private final Binding<Long> age;

    Person(String name, LocalDate birthday) {
        this.name = new SimpleStringProperty(name);
        this.birthday = new SimpleObjectProperty<>(birthday);

        this.age = JavaFxObserver.toBinding(
                JavaFxObservable.valuesOf(birthdayProperty())
                        .map(dt -> ChronoUnit.YEARS.between(dt,LocalDate.now())))
        );
    }

    public StringProperty nameProperty() {
        return name;
    }

    public ObjectProperty<LocalDate> birthdayProperty() {
        return birthday;
    }

    public Binding<Long> getAge() {
        return age;
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.toBinding
import com.github.thomasnield.rxkotlinfx.toObservable
import tornadofx.*
import java.time.LocalDate
import java.time.temporal.ChronoUnit


class Person(name: String, birthday: LocalDate) {
    var name by property(name)
    fun nameProperty() = getProperty(Person::name)

    var birthday by property(birthday)
    fun birthdayProperty() = getProperty(Person::birthday)

    val age = birthdayProperty().toObservable()
            .map { ChronoUnit.YEARS.between(it,LocalDate.now())
}
            .toBinding()
}
```

In Java, you can also fluently use the `to()` operator to map the `Observable` to any arbitrary type. We can use it to streamline turning it into a `Binding`.

```java
this.age = JavaFxObservable.valuesOf(birthdayProperty())
                .map(dt -> ChronoUnit.YEARS.between(dt,LocalDate
.now()))
                .to(JavaFxObserver::toBinding);
);
```

Now if you put a few instances of `Person` in a `TableView`, each row will then come to life (Figure 6.2).

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
```

```java
import javafx.scene.control.*;
import javafx.scene.control.cell.TextFieldTableCell;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.util.converter.LocalDateStringConverter;

import java.time.LocalDate;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        TableView<Person> table = new TableView<>();
        table.setEditable(true);

        table.getItems().setAll(
                new Person("Thomas Nield", LocalDate.of(1989,1,18
)),
                new Person("Sam Tulsa",LocalDate.of(1980,5,12)),
                new Person("Ron Johnson",LocalDate.of(1975,3,8))
        );

        TableColumn<Person,String> nameCol = new TableColumn<>("
Name");
        nameCol.setCellValueFactory(v -> v.getValue().nameProper
ty());

        TableColumn<Person,LocalDate> birthdayCol = new TableCol
umn<>("Birthday");
        birthdayCol.setCellValueFactory(v -> v.getValue().birthd
ayProperty());
        birthdayCol.setCellFactory(TextFieldTableCell.forTableCo
lumn(new LocalDateStringConverter()));

        TableColumn<Person,Long> ageCol = new TableColumn<>("Age"
);
        ageCol.setCellValueFactory(v -> v.getValue().getAge());

        table.getColumns().addAll(nameCol,birthdayCol,ageCol);
```

```
        stage.setScene(new Scene(table));
        stage.show();
    }


 }
```

Kotlin

```kotlin
 import javafx.util.converter.LocalDateStringConverter
 import tornadofx.*
 import java.time.LocalDate



 class MyApp: App(MyView::class)


 class MyView: View() {

     override val root = tableview<Person> {
         isEditable = true
         items.setAll(
                 Person("Thomas Nield",LocalDate.of(1989,1,18)),
                 Person("Sam Tulsa",LocalDate.of(1980,5,12)),
                 Person("Ron Johnson",LocalDate.of(1975,3,8))
         )

         column("Name",Person::nameProperty)
         column("Birthday",Person::birthdayProperty).useTextField
 (LocalDateStringConverter())
         column("Age",Person::age)
     }
 }
```
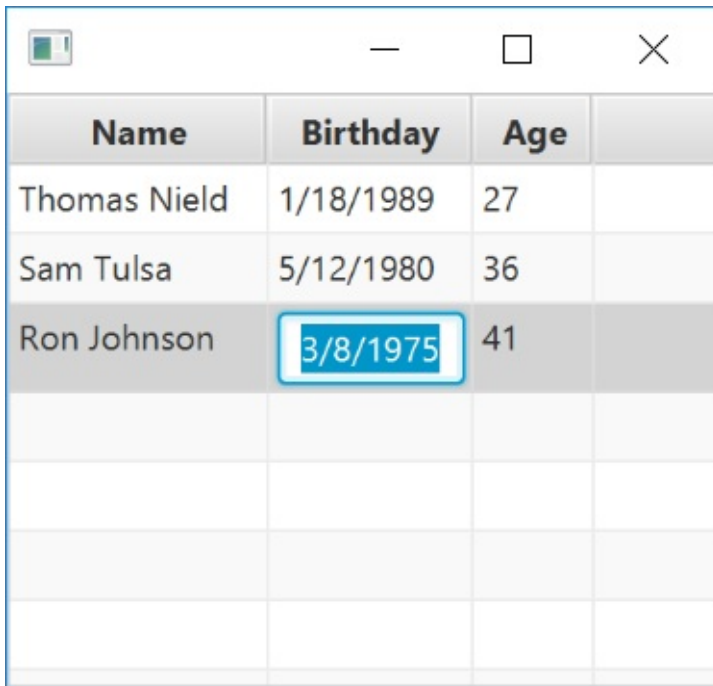
Figure 6.2

When you edit the "Birthday" field for a given row, you will see the "Age" field update automatically. This is because the age `Binding` is subscribed to the RxJava `Observable` derived from the birthday `Property`.

Handling Errors with Reactive Bindings

When you create a JavaFX `Binding<T>` off an `Observable<T>`, it usually is a good idea to pass a lambda to handle the `onError()` event. Otherwise errors may go unnoticed and unhandled. Try to make this part of your best practices, even if we do not do this for the rest of the book (for sake of brevity).

Java

```java
private final Binding<Long> age = JavaFxObservable.valuesOf(birthdayProperty())
        .map(dt -> ChronoUnit.YEARS.between(dt,LocalDate.now()))
        .to(obs -> JavaFxObserver.toBinding(obs, Throwable::printStackTrace));
```

Kotlin

```
val age = birthdayProperty().toObservable()
        .map { ChronoUnit.YEARS.between(it,LocalDate.now()) }
        .toBinding { it.printStackTrace() }
```

Disposing Bindings

If we are going to remove records from a `TableView` . we will need to dispose any Bindings that exist on each item. This will `dispose()` the `Binding` from the RxJava `Observable` to prevent any memory leaks and free resources.

It is good practice to put a method on your domain type that will dispose all Bindings on that item. For our `Person` , we will want to `dispose()` the age `Binding` when that `Person` is no longer needed.

Java

```java
public final class Person {

    // existing code

    public void dispose() {
        age.dispose();
    }
}
```

Kotlin

```kotlin
class Person(name: String, birthday: LocalDate) {

    //existing code

    fun dispose() = age.dispose()
}
```

Whever you remove items from the `TableView` , call `dispose()` on each `Person` so all Observables are unsubscribed. If your domain type has several Bindings, you can add them all to a `CompositeBinding` . This is basically a

collection of Bindings that you can `dispose()` all at once. Say we added another `Binding` to `Person` called `isAdult` (which is conveniently built off `age` by turning it into an `Observable`). It may be convenient to add both Bindings to a `CompositeBinding` in the constructor, so `dispose()` will dispose them both.

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import io.reactivex.rxjavafx.observers.JavaFxObserver;
import io.reactivex.rxjavafx.subscriptions.CompositeBinding;
import javafx.beans.binding.Binding;
import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleObjectProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public final class Person {

    private final StringProperty name;
    private final ObjectProperty<LocalDate> birthday;
    private final Binding<Long> age;
    private final Binding<Boolean> isAdult;

    private final CompositeBinding bindings = new CompositeBinding();

    Person(String name, LocalDate birthday) {
        this.name = new SimpleStringProperty(name);
        this.birthday = new SimpleObjectProperty<>(birthday);

        this.age = JavaFxObservable.valuesOf(birthdayProperty())
                .map(dt -> ChronoUnit.YEARS.between(dt,LocalDate.now()))
                .to(JavaFxObserver::toBinding);

        this.isAdult = JavaFxObservable.valuesOf(age)
```

```java
                .map(age -> age >= 18)
                .to(JavaFxObserver::toBinding);

        bindings.add(age);
        bindings.add(isAdult);
    }


    public StringProperty nameProperty() {
        return name;
    }


    public ObjectProperty<LocalDate> birthdayProperty() {
        return birthday;
    }


    public Binding<Long> getAge() {
        return age;
    }


    public void dispose() {
        bindings.dispose();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.addTo
import com.github.thomasnield.rxkotlinfx.toBinding
import com.github.thomasnield.rxkotlinfx.toObservable
import io.reactivex.rxjavafx.subscriptions.CompositeBinding
import tornadofx.*
import java.time.LocalDate
import java.time.temporal.ChronoUnit


class Person(name: String, birthday: LocalDate) {
    var name by property(name)
    fun nameProperty() = getProperty(Person::name)

    var birthday by property(birthday)
    fun birthdayProperty() = getProperty(Person::birthday)

    private val bindings = CompositeBinding()

    val age = birthdayProperty().toObservable()
            .map { ChronoUnit.YEARS.between(it,LocalDate.now())
}
            .toBinding()
            .addTo(bindings)

    val isAdult = age.toObservable()
            .map { it >= 18 }
            .toBinding()
            .addTo(bindings)

    fun dispose() = bindings.dispose()
}
```

Lazy Bindings

When you create a `Binding<T>` off an `Observable<T>`, it will subscribe
eagerly and request emissions immediately. There may be situtations you would
rather a `Binding<T>` be lazy and not subscribe to the `Observable<T>` until a
value is first needed (specifically, when `getValue()` is called). This is

particularly helpful for data controls like `TableView` where only visible records in view will request values. If you scroll quickly, it will only request values when you slow down on a set of records. This way, the `TableView` does not have to calculate all values for all records, but rather just the ones you see.

If we wanted to make our two reactive Bindings on `Person` lazy, so they only subscribe when that `Person` is in view, call `toLazyBinding()` instead of `toBinding()`.

Java

```java
this.age = JavaFxObservable.valuesOf(birthdayProperty())
              .map(dt -> ChronoUnit.YEARS.between(dt,LocalDate
.now()))
              .to(JavaFxObserver::toLazyBinding);
```

Kotlin

```kotlin
val age = birthdayProperty().toObservable()
        .map { ChronoUnit.YEARS.between(it,LocalDate.now()) }
        .toLazyBinding())
```

In some situations, you may have a `Binding` that is driven off an `Observable` that queries a database (using [RxJava-JDBC](#)) or some other service. Because these requests can be expensive, `toLazyBinding()` can be valuable to initiatlize the `TableView` more quickly. Of course, this lazy loading can sometimes cause laggy scrolling by holding up the JavaFX thread, and we will learn about concurrency later in this book to mitigate this.

Summary

In this chapter we learned about turning Observables into JavaFX Bindings, which helps interop RxJava with JavaFX more thoroughly. Typically you do not need to use Bindings often as RxJava provides a robust means to synchronize properties and events, but some parts of the JavaFX API expect a `Binding` which you now have the means to provide.

Dialogs and Multicasting

In this chapter we will cover using Dialogs as well as multicasting. Dialogs are helpful for getting user inputs, and they can be helpful in an Rx context. Multicasting is a way to force Observables to be hot, and we will learn why it is critical to do this when multiple Subscribers to a UI event `Observable` are present.

Dialogs

JavaFX Dialogs are popups to quickly show a message to the user or solicit an input. They can be helpful in reactive applications, so they also have a factory to turn their response into an `Observable`.

You can pass an `Alert` or `Dialog` to the `fromDialog()` factory, and it will return an `Observable` that emits the response as a single emission. Then it will call `onCompleted()`.

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.control.Alert;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        JavaFxObservable.fromDialog(
                new Alert(Alert.AlertType.CONFIRMATION, "Are you
 sure you want to do this?")
        ).subscribe(response -> System.out.println("You pressed "
 + response.getText()));

        System.exit(0);
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.toObservable
import javafx.scene.control.Alert
import tornadofx.*


class MyApp: App(MyView::class)

class MyView: View() {

    override val root = pane {
        Alert(Alert.AlertType.CONFIRMATION, "Are you sure you wa
nt to do this?")
                .toMaybe()
                .subscribe { println("You pressed " + it.text) }

        System.exit(0)
    }
}
```

Dialogs can become more useful in a `flatMap()` , `flatMapSingle()` , or `flatMapMaybe()` to intercept and manipulate emissions. If you `flatMapMaybe()` a `Button` 's ActionEvents to a `Dialog` response, you can use `filter()` on the response to conditionally allow an emission to go forward or be suppressed.

For example, say you have a "Run Process" `Button` that will kick of a simple process emitting the integers 1 through 10, and then collects them into a `List` . Pretend this process was something more intensive, and you want the user to confirm on pressing the `Button` if they want to run it. You can use a `Dialog` to intercept `ActionEvent` emissions in a `flatMapMaybe()` , map to the `Dialog` 's response, and allow only emissions that are `ButtonType.OK` . Then you can `flatMap()` that emission to kick off the process (Figure 6.3), which actually yields a `Single` so we will use `flatMapSingle()` .

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Button;
import javafx.scene.control.ButtonType;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        Button runButton = new Button("Run Process");

        JavaFxObservable.actionEventsOf(runButton)
                .flatMapMaybe(ae ->
                        JavaFxObservable.fromDialog(new Alert(Alert.AlertType.CONFIRMATION, "Are you sure you want to run the process?"))
                                .filter(response -> response.equals(ButtonType.OK))
                ).flatMapSingle(response -> Observable.range(1,10).toList())
                .subscribe(i -> System.out.println("Processed integer list: " + i));

        VBox root = new VBox();
        root.getChildren().add(runButton);

        stage.setScene(new Scene(root));

        stage.show();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.actionEvents
import com.github.thomasnield.rxkotlinfx.toObservable
import io.reactivex.Observable
import javafx.scene.control.Alert
import javafx.scene.control.ButtonType
import tornadofx.*


class MyApp: App(MyView::class)

class MyView : View() {

    override val root = vbox {
        button("Run Process").actionEvents()
                .flatMapMaybe {
                    Alert(Alert.AlertType.CONFIRMATION, "Are you
 sure you want to run the process?")
                            .toMaybe()
                            .filter { it == ButtonType.OK }
                }.flatMapSingle { Observable.range(1,10).toList(
) }
                .subscribe { println("Processed integer list: $i
t") }
    }
}
```
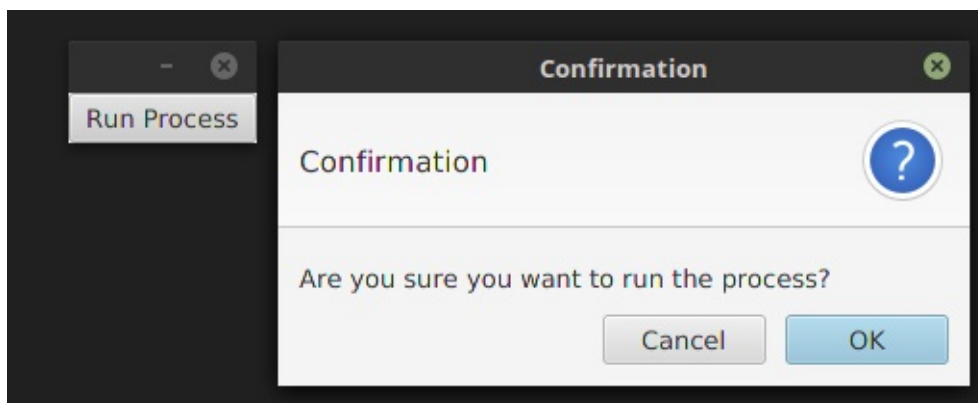
Figure 6.3

That `flatMapMaybe()` to an `Alert` dialog will emit a `ButtonData.OK`, `ButtonData.CANCEL`, or no emission at all depending on what the user chooses. Filtering for only `ButtonData.OK` emissions, only those emissions will result in a kickoff of the `flatMapSingle { Observable.range(1,10).toList() }` process. Otherwise it will be empty and no `List<Integer>` will be emitted. This shows we can use a `Dialog` or `Alert` inputs to intercept and manipulate emissions in an `Observable` chain.

Here is another example. Let's say clicking a `Button` will emit an `ActionEvent`. You will then have integers 0 through 10 emitted inside a `flatMap()` for each `ActionEvent`, and you want the user to decide which integers should proceed to the `Observer`. Using some creative flatmapping, this is not terribly hard. You can use an `Alert` or `Dialog` for each integer emission to control which ones will go forward (Figure 6.4).

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Button;
import javafx.scene.control.ButtonType;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        Button runButton = new Button("Run Process");

        JavaFxObservable.actionEventsOf(runButton)
                .flatMap(ae ->
                        Observable.range(1,10)
                                .flatMapMaybe(i ->
                                        JavaFxObservable.fromDia
```

```
log(
                                        new Alert(Alert.
AlertType.CONFIRMATION,
                                            "Are you
 sure you want to process integer " + i + "?",
                                            ButtonTy
pe.NO, ButtonType.YES)
                                    ).filter(response -> res
ponse.equals(ButtonType.YES))
                                    .map(response -> i)
                            )

                ).subscribe(i -> System.out.println("Processed i
nteger: " + i));

        VBox root = new VBox();
        root.getChildren().add(runButton);

        stage.setScene(new Scene(root));

        stage.show();
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.actionEvents
import com.github.thomasnield.rxkotlinfx.toObservable
import io.reactivex.Observable
import javafx.scene.control.Alert
import javafx.scene.control.ButtonType
import tornadofx.*


class MyApp: App(MyView::class)

class MyView : View() {

    override val root = vbox {
        button("Run Process").actionEvents()
                .flatMapMaybe {
                    Observable.range(1, 10).flatMap { i ->
                        Alert(Alert.AlertType.CONFIRMATION,
                                "Do you want to process integer
$i?",
                                ButtonType.NO, ButtonType.YES
                        ).toMaybe()
                            .filter { it == ButtonType.YES }
                            .map { response -> i }
                    }
                }.subscribe { println("Processed integer: $it")
    }
    }
}
```

Figure 6.4
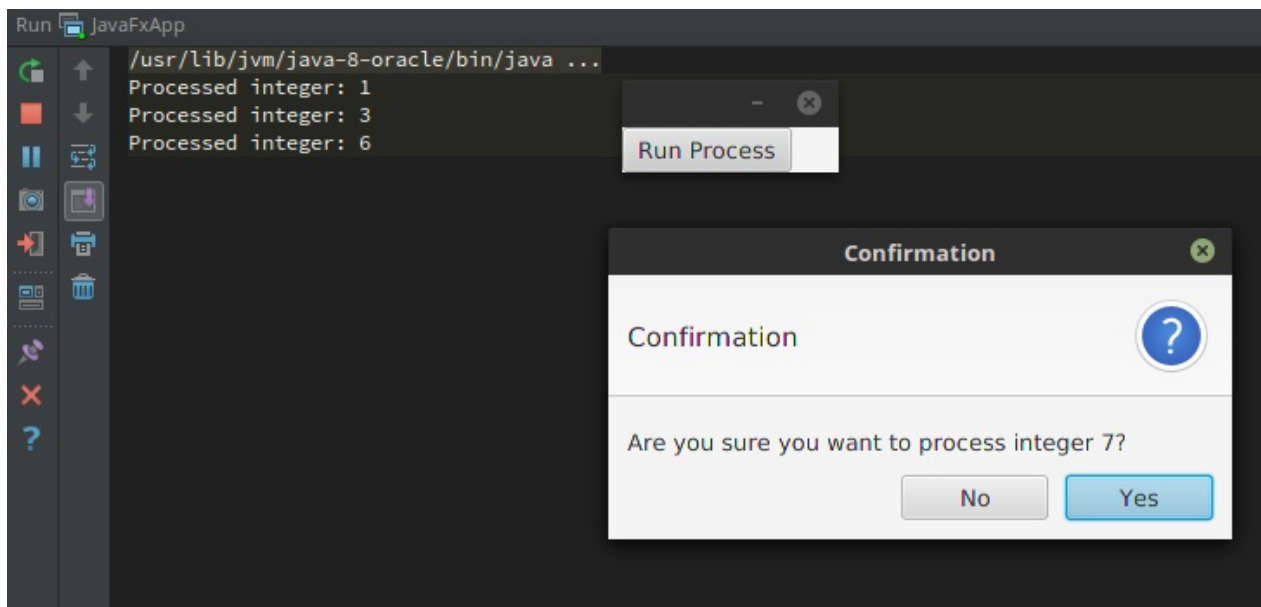
The `map(response -> i)` is a simple trick you can do to take a response after it has been filtered, and map it back to the integer. If you say "YES" to 1, 3, 6 and "NO" to everything else, you should get the output above. 2,4,5,7,9, and 10 never made it to the `Observer` because "NO" was selected and filtered them out.

That is how you can reactively leverage Dialogs and Alerts, and any control that implements `Dialog` to return a single result can be reactively emitted in this way.

Multicasting

For the sake of keeping the previous chapters accessible, I might have mislead you when I said UI events are hot Observables. The truth is by default, they are a gray area between a hot and cold `Observable` (or should I say "warm"?). Remember, a "hot" `Observable` will emit to all Observers at once, while a "cold" `Observable` will replay emissions to each `Observer` individually. This is a pragmatic way to separate the two, but UI event factories in RxJavaFX (as well as RxBindings for Android) awkwardly operate as both hot and cold unless you **multicast**, or force an emission to hotly be emitted to all Observers.

To understand this subtle impact, here is a trick question. Say you have a `Maybe` driven off a `Dialog` or `Alert` , and it has two Observers. Do you think the response is going to go to both Subscribers?

Java

```java
import io.reactivex.Maybe;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.control.Alert;
import javafx.scene.control.ButtonType;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        Maybe<Boolean> response = JavaFxObservable.fromDialog(
                new Alert(Alert.AlertType.CONFIRMATION,"Are you
sure you want to proceed?")
        ).map(r -> r.equals(ButtonType.OK));

        response.subscribe(r -> System.out.println("Subscriber 1
 received: " + r));

        response.subscribe(r -> System.out.println("Subscriber 2
 received: " + r));


        System.exit(0);
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.toMaybe
import javafx.scene.control.Alert
import javafx.scene.control.ButtonType
import tornadofx.*



class MyApp: App(MyView::class)


class MyView : View() {

    override val root = vbox {

        val response = Alert(Alert.AlertType.CONFIRMATION, "Are
you sure you want to proceed?")
                .toMaybe()
                .map { it == ButtonType.OK }

        response.subscribe { println("Subscriber 1 received: $it"
) }


        response.subscribe { println("Subscriber 2 received: $it"
) }


        System.exit(0)
    }
}
```

Try running it and you will see the `Alert` popup twice, once for each `Observer`. This is almost like it's a cold `Observable` and it is "replaying" the `Dialog` procedure for each `Observer`. As a matter of fact, that is exactly what is happening. Both Observer are receiving their own, independent streams. You can actually say `OK` on one `Observer` and `CANCEL` to the other. The two Subscribers are, in fact, not receiving the same emission as you would expect in a hot `Observable`.

This behavior is not a problem when you have one `Observer`. But when you have multiple Observers, you will start to realize this is not a 100% hot `Observable`. It is "hot" in that previous emissions are missed by tardy

Observers, but it is not "hot" in that a single set of emissions are going to all Observers. To force the latter to happen, you can multicast, and that will force this `Observable` to be 100% hot.

One way to multicast is to use the `ConnectableObservable` we used in Chapter 2. We can `publish()` the `Observable` to get a `ConnectableObservable`, set up up the `Observers`, *then* call `connect()` to start firing the same emissions to all Observers.

Java

```java
import io.reactivex.observables.ConnectableObservable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.control.Alert;
import javafx.scene.control.ButtonType;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        ConnectableObservable<Boolean> response = JavaFxObservable.fromDialog(
                new Alert(Alert.AlertType.CONFIRMATION,"Are you sure you want to proceed?")
        ).map(r -> r.equals(ButtonType.OK))
         .toObservable().publish(); //returns ConnectableObservable

        response.subscribe(r -> System.out.println("Subscriber 1 received: " + r));

        response.subscribe(r -> System.out.println("Subscriber 2 received: " + r));

        response.connect();

        System.exit(0);
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.toObservable
import javafx.scene.control.Alert
import javafx.scene.control.ButtonType
import tornadofx.*


class MyApp: App(MyView::class)

class MyView : View() {

    override val root = vbox {

        val response = Alert(Alert.AlertType.CONFIRMATION, "Are
you sure you want to proceed?")
                .toObservable()
                .map { it == ButtonType.OK }
                .publish() //returns ConnectableObservable

        response.subscribe { println("Subscriber 1 received: $it"
) }

        response.subscribe { println("Subscriber 2 received: $it"
) }

        response.connect()

        System.exit(0)
    }
}
```

When you run this program, you will now see the `Alert` only pop up once, and the single response will go to both Observers simultaneously. Every operator *before* the `publish()` will be a single stream of emissions. But take note that everything *after* the `publish()` is subject to be on separate streams from that point.

> The `Maybe` as well as the `Single` are not able to multicast, so just turn them into an `Observable` via `toObservable()` when you need to.

If you want this `ConnectableObservable` to automatically `connect()` for you when the first `Observer` is subscribed, you can call `refCount()` to turn it back into a normal `Observable`.

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.control.Alert;
import javafx.scene.control.ButtonType;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        Observable<Boolean> response = JavaFxObservable.fromDialog(
                new Alert(Alert.AlertType.CONFIRMATION,"Are you sure you want to proceed?")
        ).map(r -> r.equals(ButtonType.OK))
        .toObservable()
        .publish()
        .refCount();

        response.subscribe(r -> System.out.println("Subscriber 1 received: " + r));

        response.subscribe(r -> System.out.println("Subscriber 2 received: " + r));

        System.exit(0);
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.toMaybe
import javafx.scene.control.Alert
import javafx.scene.control.ButtonType
import tornadofx.*


class MyApp: App(MyView::class)

class MyView : View() {

    override val root = vbox {

        val response = Alert(Alert.AlertType.CONFIRMATION, "Are
you sure you want to proceed?")
                .toMaybe()
                .map { it == ButtonType.OK }
                .toObservable()
                .publish()
                .refCount()

        response.subscribe { println("Subscriber 1 received: $it"
) }

        response.subscribe { println("Subscriber 2 received: $it"
) }

        System.exit(0)
    }
}
```

`refCount()` is a convenient way to turn a `ConnectableObservable` back into an automatic `Observable`. It is helpful to force emissions to be hot without manually calling `connect()`. Just be aware it will start emitting on the first subscription, and any following subscriptions may miss the first emissions as they are subscribed *after* the firing starts. But for UI events waiting for a user input, chances are all subscriptions will `connect()` in time before the user inputs anything, so `refCount()` is usually acceptable for UI events. But if your

Observable is going to fire emissions the moment it is subscribed, you may just want to manually set up a `ConnectableObservable`, subscribe the Observers, and call `connect()` yourself.

So when should you multicast with a `ConnectableObservable` (or use its `refCount()`)? The answer is when you have multiple Observers to a single UI event `Observable`. When you broadcast something as simple as a `Button`'s ActionEvents, it is more efficient to multicast it so it does not create a `Listener` for each `Observer`, but rather consolidates to one `Listener`.

Again, use multicasting for UI event Observables when there is more than one Observer. Even though most of the time this makes no functional difference, it is more efficient. It also will prevent subtle misbehaviors like we saw in cases like the `Dialog`, where we want to force a single emission stream to go to all Observers rather than each `Observer` getting its own emissions in a cold-like manner. If you have only one `Observer`, the additional overhead of `ConnectableObservable` is not necessary.

Replaying Observables

Another kind of `ConnectableObservable` is the one returned by the `replay()` operator. This will hold on to the last *x* number of emissions and "replay" them to each new `Observer`.

For instance, an `Observable` can emit the numbers 1 through 5. If I call `replay(1)` on it, it will return a `ConnectableObservable` that will emit the last emission "5" to later Observers. However if I am going to multicast this, I may want to use `autoConnect()` instead of `refCount()`. Here is why: the `refCount()` will "shut down" when it has no more active Observers (particularly Observers that call `onComplete()`). This will reset everything and clear the "5" from its cache. If another Observer comes in, it will be treated as the first Observer and receive all 5 emissions rather than just the "5". The `autoConnect()`, however, will always stay alive whether or not it has Subscribers, and persist the cached value of "5" indefinitely until a new value replaces it.

Java

```java
Observable<Integer> source = Observable.range(1,5)
    .replay(1).autoConnect();


range.subcribe(i -> System.out.prinln(i)); //receives 1,2,3,4,5
Thread.sleep(3000); //sleep 3 seconds, try-catch this
range.subscribe(i -> System.out.println(i)); //receives 5
```

Kotlin

```kotlin
val source = Observable.range(1,5)
    .replay(1).autoConnect()

source.subscribe { println(it) } //receives 1,2,3,4,5
Thread.sleep(3000) //sleep 3 seconds, try-catch this
source.subscribe { println(it) } //receives 5
```

OUTPUT:

```
1
2
3
4
5
5
```

The `replay()` operator can be helpful to replay the last emitted value for a UI input (e.g a `ComboBox`) so new Observers immediately get that value rather than missing it. There are other argument overloads for `replay()` to replay emissions for other scopes, like time windows. But simply holding on to the last emission is a common helpful pattern for reactive UI's.

> You can also use the `cache()` operator to horde and replay *ALL* emissions, but keep in mind this can increase memory usage and cause data to go stale.

Summary

If you got this far, congrats! We have covered a lot. We ran through reactive usage of Dialogs, which you can use to intercept emissions and get a user input for each one, as well as multicasting. The topic of multicasting is critical to understand because UI Observables do not always act hot when multiple Observers are subscribed. Creating a `ConnectableObservable` is an effective way to force an `Observable` to become hot and ensure each emission goes to all Observers at once, and remove redundant listeners

Make sure you are somewhat comfortable with the material we covered so far, because next we are going to cover concurrency. This is the topic that everything leads up to, as RxJava revolutionizes how we multithread safely.

8. Concurrency

Concurrency has become a critical skill in software development. Most computers and smart phones now have multiple core processors, and this means the most effective way to scale performance is to leverage all of them. For this full utilization to happen, code must explicitly be coded for multiple threads that can be worked by multiple cores.

The idea of concurrency is essentially multitasking. Multiple threads execute multiple tasks at the same time. Suppose you had some yard work to do and had three tasks: mow the lawn, trim the trees, and sweep the patio. If you are working alone, there is no way you can do all three of these tasks at the same time. You have to sequentially work on each task one-at-a-time. But if you had two friends to help out, you can get done more quickly as all three of you can execute all three tasks simultaneously. In essence, each person is a thread and each chore is a task.

Even if you have less threads than tasks (such as two threads and three tasks), the two threads can tackle two of the tasks immediately. The first one to get done can then move on to the third task. This is essentially what a thread pool does. It has a fixed number of threads and is given a "queue" of tasks to do. Each thread will take a task, execute it, and then take another. "Reusing" threads and giving them a queue of tasks, rather than creating a thread for each task, is usually more efficient since threads are expensive to create and dispose.

Traditionally, Java concurrency is difficult to master. A lot can go wrong especially with mutable variables being accessed by multiple threads. Thankfully, RxJava makes concurrency easier and safer. When you stay within an `Observable` chain, it does not matter what thread emissions get pushed on (except of course Observers and operators affecting JavaFX UI's, which need to happen on the JavaFX thread). A major selling point of RxJava is its ability to make concurrency trivial to compose, and this is helpful to make JavaFX UI's responsive and resilient.

> It is recommended to study concurrency without RxJava, just so you are aware of the "gotchas" that can happen with multithreading. Benjamin Winterberg has an awesome online tutorial walking through Java 8 concurrency. If you want deep knowlege in Java concurrnecy, [Java Concurrency in Practice](http://jcip.net/) is an excellent book to gain low-level knowledge.

Using `subscribeOn()`

By default, for a given `Observable` chain, the thread that calls the `subscribe()` method is the thread the `Observable` sends emissions on. For instance, a simple subscription to an `Observable` inside a `main()` method will fire the emissions on the `main` daemon thread.

Java

```java
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable.range(1,5)
                .subscribe(i ->
                        System.out.println("Receiving " + i + "
on thread "
                                + Thread.currentThread().getName
())
                );
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable

fun main(args: Array<String>) {
    Observable.range(1,5)
            .subscribe { println("Receiving $it on thread ${Thre
ad.currentThread().name}") }
}
```

OUTPUT:

```
Receiving 1 on thread main
Receiving 2 on thread main
Receiving 3 on thread main
Receiving 4 on thread main
Receiving 5 on thread main
```

However, we can easily switch these emissions to happen on another thread using `subscribeOn()` . We can pass a `Scheduler` as an argument, which specifies where it gets a thread from. In this case we can pass `subscribeOn()` an argument of `Schedulers.newThread()` , so it will execute on a new thread for each `Observer` .

Java

```java
import io.reactivex.Observable;
import io.reactivex.schedulers.Schedulers;
import java.util.concurrent.TimeUnit;

public class Launcher {
    public static void main(String[] args) {
        Observable.range(1,5)
                .subscribeOn(Schedulers.newThread())
                .subscribe(i ->
                        System.out.println("Receiving " + i + "
on thread "
                                + Thread.currentThread().getName
())
                );

        try {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import io.reactivex.schedulers.Schedulers
import java.util.concurrent.TimeUnit

fun main(args: Array<String>) {
    Observable.range(1,5)
            .subscribeOn(Schedulers.newThread())
            .subscribe { println("Receiving $it on thread ${Thre
ad.currentThread().name}") }

    TimeUnit.SECONDS.sleep(3)
}
```

OUTPUT:

```
Receiving 1 on thread RxNewThreadScheduler-1
Receiving 2 on thread RxNewThreadScheduler-1
Receiving 3 on thread RxNewThreadScheduler-1
Receiving 4 on thread RxNewThreadScheduler-1
Receiving 5 on thread RxNewThreadScheduler-1
```

This way we can declare our `Observable` chain and an `Observer` , but then immediately move on without waiting for the emissions to finish. Those are now happening on a new thread named `RxNewThreadScheduler-1` . Notice too we have to call `TimUnit.SECONDS.sleep(3)` afterwards to make the `main` thread sleep for 3 seconds. This gives our `Observable` a chance to fire all emissions before the program exits. You should not have to do this `sleep()` with a JavaFX application since its own non-daemon threads will keep the session alive.

A critical behavior to note here is that *all* emissions are happening *sequentially* on a single `RxNewThreadScheduler-1` thread. Emissions are strictly happening one-at-a-time on a single thread. There is no parallelization or racing to call `onNext()` throughout the chain. If this did occur, it would break the `Observable` contract. It may surprise some folks to hear that RxJava is not parallel, but we will cover some concurrency tricks with `flatMap()` later to get parallelization without breaking the `Observable` contract.

`subscribeOn()` can be declared anywhere in the `Observable` chain, and it will communicate all the way up to the source what thread to fire emissions on. If you pointlessly declare multiple `subscribeOn()` operators in a chain, the left-most one (closest to the source) will win. Later we will cover the `observeOn()` which can switch emissions to a different thread in the middle of the chain.

Pooling Threads: Choosing a Scheduler

In reality, you should be conservative about using `Schedulers.newThread()` as it creates a new thread for each `Observer` . You will notice that if we attach multiple Observers to this `Observable` , we are going to create a new thread for each `Observer` .

Java

```java
import io.reactivex.Observable;
import io.reactivex.schedulers.Schedulers;
import java.util.concurrent.TimeUnit;


public class Launcher {
    public static void main(String[] args) {
        Observable<Integer> source = Observable.range(1,5)
                .subscribeOn(Schedulers.newThread());


        //Observer 1
        source.subscribe(i ->
                System.out.println("Observer 1 receiving " + i +
" on thread "
                        + Thread.currentThread().getName())
        );


        //Observer 2
        source.subscribe(i ->
                System.out.println("Observer 2 receiving " + i +
" on thread "
                        + Thread.currentThread().getName())
        );


        try {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import io.reactivex.schedulers.Schedulers
import java.util.concurrent.TimeUnit

fun main(args: Array<String>) {
    val source = Observable.range(1,5)
            .subscribeOn(Schedulers.newThread())

    //Observer 1
    source.subscribe { println("Observer 1 receiving $it on thread ${Thread.currentThread().name}") }

    //Observer 2
    source.subscribe { println("Observer 2 receiving $it on thread ${Thread.currentThread().name}") }

    TimeUnit.SECONDS.sleep(3)
}
```

OUTPUT:

```
Observer 2 receiving 1 on thread RxNewThreadScheduler-2
Observer 1 receiving 1 on thread RxNewThreadScheduler-1
Observer 2 receiving 2 on thread RxNewThreadScheduler-2
Observer 1 receiving 2 on thread RxNewThreadScheduler-1
Observer 2 receiving 3 on thread RxNewThreadScheduler-2
Observer 1 receiving 3 on thread RxNewThreadScheduler-1
Observer 2 receiving 4 on thread RxNewThreadScheduler-2
Observer 1 receiving 4 on thread RxNewThreadScheduler-1
Observer 2 receiving 5 on thread RxNewThreadScheduler-2
Observer 1 receiving 5 on thread RxNewThreadScheduler-1
```

Now we have two threads, `RxNewThreadScheduler-1` and `RxNewThreadScheduler-2` . What if we had 100, or even 1000 Observers? This can easily happen if you are flatMapping to hundreds or thousands of Observables each with their own `subscribeOn(Schedulers.newThread())` . Threads are very expensive and can tax your machine, so we want to constrain the number of threads that can be used at a time.

The most effective way to keep thread creation under control is to "reuse" threads. You can do this with the different `Schedulers` . A `Scheduler` is RxJava's equivalent to Java's standard `Executor` . You can create your own `Scheduler` by passing an `Executor` to the `Schedulers.from()` factory. But for most cases, it is better to use RxJava's standard `Schedulers` as they are optimized to be conservative and efficient for most cases.

Computation

If you are doing computation-intensive operations, you will likely want to use `Schedulers.computation()` which will maintain a conservative number of threads to keep the CPU from being taxed.

```java
Observable<Integer> source = Observable.range(1,5)
    .subscribeOn(Schedulers.computation());
```

`Observable` operations that are doing calculation and algorithm-heavy work are optimal to use with the computation `Scheduler` . If you are not sure how many threads will be created by a process, you might want to make this one your go-to.

IO

If you are doing a lot of IO-related tasks, like sending web requests or database queries, these are much less taxing on the CPU and threads can be created more liberally. `Schedulers.io()` is suited for this kind of work. It will add and remove threads depending on how much work is being thrown at it at a given time, and reuse threads as much as possible.

```java
Observable<Integer> source = Observable.range(1,5)
    .subscribeOn(Schedulers.io());
```

But be careful as it will not limit how many threads it creates! As a rule-of-thumb, assume it will create a new thread for each task.

Immediate

The `Schedulers.immediate()` is the default `Scheduler`, and it will work execute work on the immediate thread declaring the `Observer`.

```
Observable<Integer> source = Observable.range(1,5)
    .subscribeOn(Schedulers.immediate());
```

You will likely not use this `Scheduler` very often since it is the default. The code above is no different than declaring an `Observable` with no `subscribeOn()`.

```
Observable<Integer> source = Observable.range(1,5)
```

Trampoline

An interesting `Scheduler` is the `Schedulers.trampoline()`. It will schedule the emissions to happen on the immediate thread, but allow the immediate thread to finish its current task first. In other words, this defers execution of the emissions but will fire them the moment the current thread declaring the subscription is no longer busy.

```
Observable<Integer> source = Observable.range(1,5)
    .subscribeOn(Schedulers.trampoline());
```

You will likely not use the Trampoline Scheduler unless you encounter nuanced situations where you have to manage complex operations on a single thread and starvation can occur. The JavaFX Scheduler uses a trampoline mechanism, which we will cover next.

JavaFX Scheduler

Finally, the `JavaFxScheduler` is packaged with the RxJavaFX library. It executes the emissions on the JavaFX thread so they can safely make modifications to a UI. It uses a trampoline policy against the JavaFX thread, making it highly resilient against recursive hangups and thread starvation.

The JavaFX Scheduler is not in the `Schedulers` class, but rather is stored as a singleton in its own class. You can call it like below:

```
Observable<Integer> source = Observable.range(1,5)
    .subscribeOn(JavaFxScheduler.platform());
```

In Kotlin, The RxKotlinFX library can save you some boilerplate by using an extension function instead.

```
val source = Observable.range(1,5)
    .subscribeOnFx()
```

At the time of writing, all RxJavaFX/RxKotlinFX factories already emit on the `JavaFxScheduler` . Therefore, declaring a `subscribeOn()` against these sources will have no affect. You will need to leverage `observeOn()` to switch to another thread later in the chain, which we will cover shortly.

Java

```
Button button = new Button("Press me");

JavaFxObservable.actionEventsOf(button)
        .subscribeOn(Schedulers.io()) // has no effect
        .subscribe(ae -> System.out.println("You clicked me!"));
```

Kotlin

```
val button = Button("Press me")

button.actionEvents()
        .subscribeOn(Schedulers.io()) // has no effect
        .subscribe { println("You clicked me!") }
```

Also note that the JavaFX Scheduler is already used when declaring UI code, and will be the default `subscribeOn()` Scheduler since it is the immediate thread. Therefore, you will rarely call `subscribeOn()` to specify the JavaFxScheduler. You are more likely to use it with `observeOn()` .

Intervals

While we are talking about concurrency, it is worth mentioning there are other factories that already emit on a specific `Scheduler` . For instance, there are factories in both RxJava and RxJavaFX to emit at a specified time interval.

In RxJava, there is an `Observable.interval()` that will emit a consecutive `Long` at every specified time interval. By default, this runs on the `Schedulers.computation()` unless you specify a different one as a third argument.

Here is an application that will increment a `Label` every second (Figure 8.1).

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.schedulers.JavaFxScheduler;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

import java.util.concurrent.TimeUnit;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox root = new VBox();

        Label label = new Label();

        Observable.interval(1, TimeUnit.SECONDS, JavaFxScheduler
.platform())
                .map(Object::toString)
                .subscribe(label::setText);

        root.getChildren().add(label);

        stage.setScene(new Scene(root));

        stage.setMinWidth(60);
        stage.show();
    }
}
```

Kotlin

```
import io.reactivex.Observable
import io.reactivex.rxjavafx.schedulers.JavaFxScheduler
import tornadofx.*
import java.util.concurrent.TimeUnit


class MyApp: App(MyView::class)

class MyView : View("My View") {

    override val root = vbox {

        minWidth = 60.0

        label {
            Observable.interval(1, TimeUnit.SECONDS, JavaFxSched
uler.platform())
                    .map { it.toString() }
                    .subscribe { text = it }
        }
    }
}
```
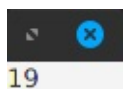
OUTPUT:

```
0
1
2
3
4
```

Figure 8.1

19

> You can also use `JavaFxScheduler.interval()` to pass a `Duration` instead of a `TimeUnit`, and not have to specify the JavaFX Scheduler.

Intervals are helpful to create timer-driven Observables, or perform tasks such as scheduling jobs or periodically driving refreshes. If you want all Observers to not receive separate interval streams, be sure to use `publish().refCount()` or `publish().autoConnect()` to multicast the same interval timer to all Observers downstream.

Using `observeOn()`

A lot of folks get confused by the difference between `subscribeOn()` and `observeOn()`, but the distinction is quite simple. A `subsribeOn()` instructs the source `Observable` what thread to emit items on. However, the `observeOn()` switches emissions to a different thread *at that point* in the chain.

In JavaFX, the most common useage of `observeOn()` is to put items back on the JavaFX thread after a compution or IO operation finishes from another thread. Say you wanted to import some expensive data on `Schedulers.io()` and collect it in a `List`. Once it is ready, you want to move that `List` emission to the JavaFX thread to feed a `ListView`. That is perfectly doable with an `observeOn()` (Figure 8.2).

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.schedulers.JavaFxScheduler;
import io.reactivex.schedulers.Schedulers;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox root = new VBox();

        ListView<String> listView = new ListView<>();

        Observable.just("Alpha","Beta","Gamma","Delta","Epsilon")
                .subscribeOn(Schedulers.io())
                .toList()
                .observeOn(JavaFxScheduler.platform())
                .subscribe(list -> listView.getItems().setAll(list));

        root.getChildren().add(listView);

        stage.setScene(new Scene(root));

        stage.show();
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.observeOnFx
import io.reactivex.Observable
import io.reactivex.rxkotlin.subscribeBy
import io.reactivex.schedulers.Schedulers
import tornadofx.*


class MyApp: App(MyView::class)


class MyView : View("My View") {

    override val root = vbox {

        listview<String> {
            Observable.just("Alpha","Beta","Gamma","Delta","Epsi
lon")
                    .subscribeOn(Schedulers.io())
                    .toList()
                    .observeOnFx()
                    .subscribeBy { items.setAll(it) }
        }
    }
}
```
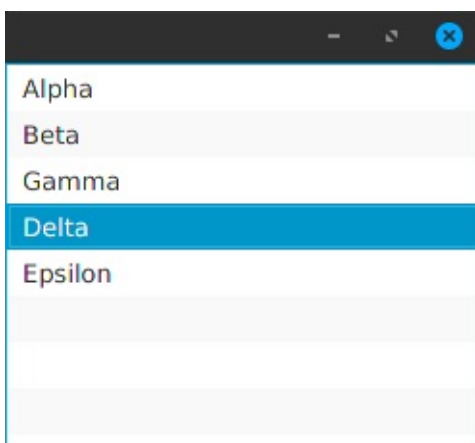
Figure 8.2

The five Strings are emitted and collected into a `List` on a `Schedulers.io()` thread. But immediately after the `toList()` is an `observeOn()` that takes that `List` and emits it on the JavaFX Scheduler. Unlike the `subscribeOn()` where

placement does not matter, the placement of the `observeOn()` does. It switches to a different thread *at that point* in the Observable chain.

This all happens a bit too fast to see this occuring, so let's exaggerate this example and emulate a long-running database query or request. Use the `delay()` operator to delay the emissions by 3 seconds. Note that `delay()` subscribes on the `Schedulers.computation()` by default, so having a `subscribeOn()` no longer has any effect. But we can pass the `Schedulers.io()` as a third argument to make it use an IO thread instead (Figure 8.3).

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.schedulers.JavaFxScheduler;
import io.reactivex.schedulers.Schedulers;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

import java.util.concurrent.TimeUnit;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox root = new VBox();

        ListView<String> listView = new ListView<>();

        Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
                .delay(3, TimeUnit.SECONDS, Schedulers.io())
                .toList()
                .observeOn(JavaFxScheduler.platform())
                .subscribe(list -> listView.getItems().setAll(list));

        root.getChildren().add(listView);

        stage.setScene(new Scene(root));

        stage.show();
    }
}
```

Kotlin

```
import com.github.thomasnield.rxkotlinfx.observeOnFx
import io.reactivex.Observable
import io.reactivex.rxkotlin.subscribeBy
import io.reactivex.schedulers.Schedulers
import tornadofx.*
import java.util.concurrent.TimeUnit


class MyApp: App(MyView::class)


class MyView : View("My View") {

    override val root = vbox {

        listview<String> {
            Observable.just("Alpha","Beta","Gamma","Delta","Epsi
lon")
                    .delay(3, TimeUnit.SECONDS, Schedulers.io())
                    .toList()
                    .observeOnFx()
                    .subscribeBy { items.setAll(it) }
        }
    }
}
```

Figure 8.3

☐

In Figure 8.3, notice that our UI is empty for 3 seconds before it is finally populated. The data importing and collecting into a `List` happens on the IO thread, and then it is safely emitted back on the JavaFX thread where it is populated into the `ListView`. The JavaFX thread does not hold up the UI from displaying due to this operation keeping it busy. If we had more controls we would see the UI is completely interactive as well during this background operation.

Chaining Multiple `observeOn()` Calls

It is also not uncommon to use multiple `observeOn()` calls. Here is a more real-life example: let's say you want to create an application that displays a text response (such as JSON) from a URL. This has the potential to create an unrespsonsive application that freezes while it is fetching the request. But using an `observeOn()` we can switch this work from the FX thread to an IO therad, then call another `observeOn()` afterwards to put it back on the FX thread.

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import io.reactivex.rxjavafx.schedulers.JavaFxScheduler;
import io.reactivex.schedulers.Schedulers;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

import java.net.URL;
import java.util.Scanner;

public class MyApp extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        VBox vBox = new VBox();
        Label label = new Label("Input URL");
        TextField input = new TextField();
        TextArea output = new TextArea();
        Button button = new Button("Submit");

        output.setWrapText(true);

        JavaFxObservable.actionEventsOf(button)
                .map(ae -> input.getText())
                .observeOn(Schedulers.io())
                .map(MyApp::getResponse)
```

```
                .observeOn(JavaFxScheduler.platform())
                .subscribe(output::setText);


        vBox.getChildren().setAll(label,input,output,button);
        stage.setScene(new Scene(vBox));
        stage.show();


    }
    private static String getResponse(String path) {
        try {
            return new Scanner(new URL(path).openStream(), "UTF-
8").useDelimiter("\\A").next();
        } catch (Exception e) {
            return e.getMessage();
        }
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.actionEvents
import com.github.thomasnield.rxkotlinfx.observeOnFx
import io.reactivex.schedulers.Schedulers
import tornadofx.*
import java.net.URL

class MyApp: App(MyView::class)

class MyView : View("My View") {

    override val root =  vbox {

        label("Input URL")
        val input = textfield()
        val output = textarea {
            isWrapText = true
        }

        button("Submit").actionEvents()
                .map { input.text }
                .observeOn(Schedulers.io())
                .map {
                    URL(input.text).readText()
                }.observeOnFx()
                .subscribe {
                    output.text = it
                }
    }
}
```
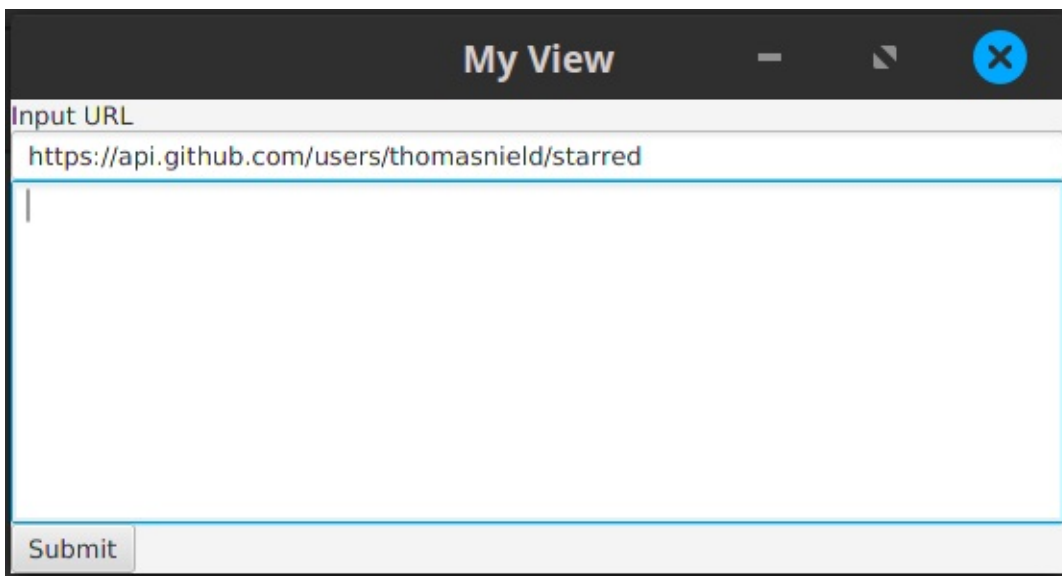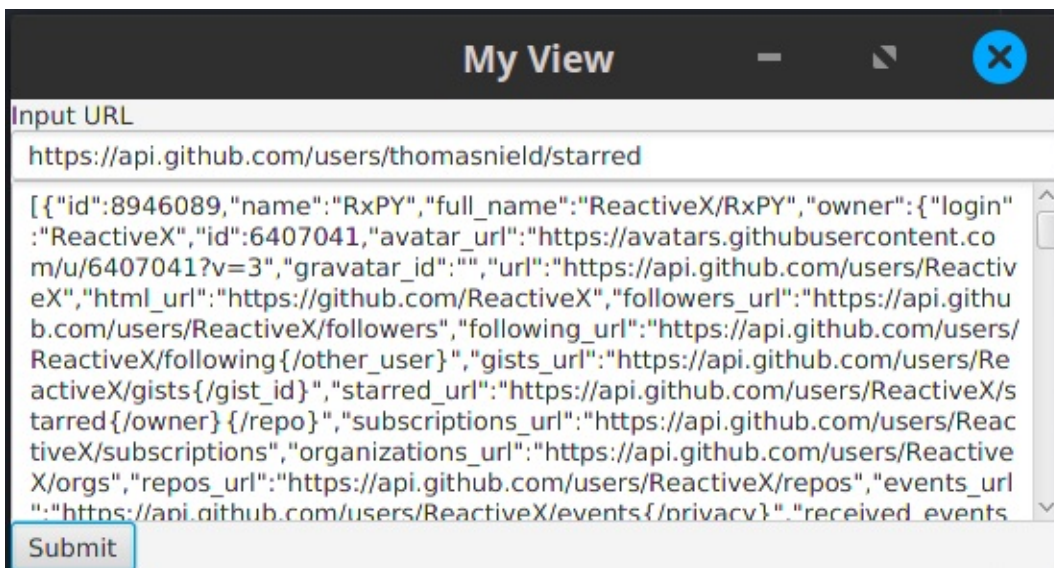
Figure 8.4

You can then put in a URL in the `TextField` (such as "https://api.github.com/users/thomasnield/starred") and then click the "Submit" `Button` to process it. You will notice the UI stays interactive and after a few seconds it will put the response in the `TextArea` (Figure 8.5).

Figure 8.5

Of course, you can click the "Submit" `Button` multiple times and that could queue up the requests in an undesirable way. But at least the work is kept off the UI thread. In the next chapter we will learn about the `switchMap()` to mitigate excessive user inputs and kill previous requests, so only the latest emission is chased after.

But we will take a stateful strategy for now to prevent this from happening.

doOnXXXFx() Operators

Remember the `doOnXXX()` operators like `doOnNext()`, `doOnComplete()`, etc? RxKotlinFX has JavaFX equivalents that will perform on the FX thread, regardless of which `Scheduler` is being used. This can be helpful to modify UI elements in the middle of an `Observable` chain.

For example, you might want to disable the `Button` and change its text during processing. Your first instinct might be to use the `doOnNext()` to achieve this.

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import io.reactivex.rxjavafx.schedulers.JavaFxScheduler;
import io.reactivex.schedulers.Schedulers;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

import java.net.URL;
import java.util.Scanner;

public class MyApp extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        VBox vBox = new VBox();
        Label label = new Label("Input URL");
        TextField input = new TextField();
        TextArea output = new TextArea();
        Button button = new Button("Submit");

        output.setWrapText(true);

        JavaFxObservable.actionEventsOf(button)
                .map(ae -> input.getText())
```

```java
                .observeOn(Schedulers.io())
                .doOnNext(path -> {
                    button.setText("BUSY");
                    button.setDisable(true);
                })
                .map(MyApp::getResponse)
                .observeOn(JavaFxScheduler.platform())
                .subscribe(r -> {
                    output.setText(r);
                    button.setText("Submit");
                    button.setDisable(false);
                });

        vBox.getChildren().setAll(label,input,output,button);
        stage.setScene(new Scene(vBox));
        stage.show();

    }
    private static String getResponse(String path) {
        try {
            return new Scanner(new URL(path).openStream(), "UTF-8").useDelimiter("\\A").next();
        } catch (Exception e) {
            return e.getMessage();
        }
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.actionEvents
import com.github.thomasnield.rxkotlinfx.observeOnFx
import io.reactivex.schedulers.Schedulers
import tornadofx.*
import java.net.URL


class MyApp: App(MyView::class)


class MyView : View("My View") {

    override val root =  vbox {

        label("Input URL")
        val input = textfield()
        val output = textarea {
            isWrapText = true
        }

        val submitButton = button("Submit")

        submitButton.actionEvents()
                .map { input.text }
                .observeOn(Schedulers.io())
                .doOnNext {
                    submitButton.text = "BUSY"
                    submitButton.isDisable = true
                }
                .map {
                    URL(input.text).readText()
                }.observeOnFx()
                .subscribe {
                    output.text = it
                    submitButton.text = "Submit"
                    submitButton.isDisable = false
                }
    }
}
```

But if you try to execute a request this way, you will get an error indicating that the `submitButton` is not being modified on the FX thread. This is occuring because an IO thread (not the FX thread) is trying to modify the `Button`. You could move this `doOnNext()` operator before the `observeOn(Schedulers.io())` so it catches the FX thread, and that would address the issue. But there will be times where you must call a `doOnNext()` deep in an `Observable` chain that is already on another thread (such as updating a `ProgressBar`).

In RxKotlinFX, there are `doOnXXXFx()` operator equivalents that run on the JavaFX thread, regardless of which thread the operator is called on. You can achieve this also with RxJavaFX using a `Transformer`, which is essentially a custom operator you can pass to a `compose()` operator.

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import io.reactivex.rxjavafx.schedulers.JavaFxScheduler;
import io.reactivex.rxjavafx.transformers.FxObservableTransforme
rs;
import io.reactivex.schedulers.Schedulers;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

import java.net.URL;
import java.util.Scanner;

public class MyApp extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        VBox vBox = new VBox();
        Label label = new Label("Input URL");
        TextField input = new TextField();
        TextArea output = new TextArea();
```

```java
        Button button = new Button("Submit");

        output.setWrapText(true);

        JavaFxObservable.actionEventsOf(button)
                .map(ae -> input.getText())
                .observeOn(Schedulers.io())
                .compose(FxObservableTransformers.doOnNextFx(t -
> {

                    button.setText("BUSY");
                    button.setDisable(true);
                }))
                .map(MyApp::getResponse)
                .observeOn(JavaFxScheduler.platform())
                .subscribe(r -> {
                    output.setText(r);
                    button.setText("Submit");
                    button.setDisable(true);
                });
        vBox.getChildren().setAll(label,input,output,button);
        stage.setScene(new Scene(vBox));
        stage.show();

    }
    private static String getResponse(String path) {
        try {
            return new Scanner(new URL(path).openStream(), "UTF-
8").useDelimiter("\\A").next();
        } catch (Exception e) {
            return e.getMessage();
        }
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.actionEvents
import com.github.thomasnield.rxkotlinfx.doOnNextFx
import com.github.thomasnield.rxkotlinfx.observeOnFx
import io.reactivex.schedulers.Schedulers
import tornadofx.*
import java.net.URL

class MyApp: App(MyView::class)

class MyView : View("My View") {

    override val root =  vbox {

        label("Input URL")
        val input = textfield()
        val output = textarea {
            isWrapText = true
        }

        val submitButton = button("Submit")

        submitButton.actionEvents()
                .map { input.text }
                .observeOn(Schedulers.io())
                .doOnNextFx {
                    submitButton.text = "BUSY"
                    submitButton.isDisable = true
                }
                .map {
                    URL(input.text).readText()
                }.observeOnFx()
                .subscribe {
                    output.text = it
                    submitButton.text = "Submit"
                    submitButton.isDisable = true
                }
    }
}
```

> Java does not have extension functions like Kotlin. But RxJava does have a `compose()` operator that you can pass a `Transformer` to, as well as a `lift()` operator to accept custom operators. Between these two methods, it is possible to create your own operators for RxJava. However, these are beyond the scope of this book. You can read about creating your own operators in my book _Learning RxJava.

Here are all the `doOnXXXFx()` operators availabe in RxKotlin. These behave exactly the same way as the `doOnXXX()` operators introduced in Chapter 2, but the action specified in the lambda will execute on the FX thread.

- `doOnNextFx()`
- `doOnErrorFx()`
- `doOnCompletedFx()`
- `doOnSubscribeFx()`
- `doOnUnsubscribeFx()`
- `doOnTerminateFx()`
- `doOnNextCountFx()`
- `doOnCompletedCountFx()`
- `doOnErrorCountFx()`

The `doOnXXXCountFx()` operators will provide a count of emissions that occurred before each of those events. They can be helpful for updating a `ProgressBar`, an incrementing `StatusBar`, or other controls that track progress.

Parallelization

Did you know the `flatMap()` (as well as `flatMapSingle()` and `flatMapMaybe()`) is actually a concurrency tool? RxJava by default does not do parallelization, so effectively there is no way to parallelize an `Observable` . As we have seen, `subscribeOn()` and `observeOn()` merely move emissions from one thread to another thread, not one thread to many threads. However, you can leverage `flatMap()` to create several Observables parallelizing emissions on different threads.

For instance we can parallelize a (simulated) long-running process for 10 consecutive integers.

Java

```java
import io.reactivex.Observable;
import io.reactivex.schedulers.Schedulers;

public class Launcher {
    public static void main(String[] args) {
        Observable.range(1,10)
                .flatMap(i -> Observable.just(i)
                        .subscribeOn(Schedulers.computation())
                        .map(Launcher::runLongProcess)
                ).subscribe(i -> System.out.println("Received "
+
                i + " on " + Thread.currentThread().getName())
        );

        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public static int runLongProcess(int i) {
        try {
            Thread.sleep((long) (Math.random() * 1000));
        } catch (Exception e) {
            e.printStackTrace();
        }
        return i;
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import io.reactivex.schedulers.Schedulers

fun main(args: Array<String>)  {
    Observable.range(1,10)
            .flatMap {
                Observable.just(it)
                        .subscribeOn(Schedulers.computation())
                        .map { runLongProcess(it) }
            }.subscribe {
                println("Received $it on ${Thread.currentThread().name}")
            }


    Thread.sleep(15000)
}

fun runLongProcess(i: Int): Int {
    Thread.sleep(Math.random().toLong() * 1000)
    return i
}
```

OUTPUT:

```
Received 1 on RxComputationScheduler-1
Received 3 on RxComputationScheduler-3
Received 5 on RxComputationScheduler-1
Received 9 on RxComputationScheduler-1
Received 4 on RxComputationScheduler-4
Received 8 on RxComputationScheduler-4
Received 2 on RxComputationScheduler-3
Received 6 on RxComputationScheduler-3
Received 7 on RxComputationScheduler-3
Received 10 on RxComputationScheduler-3
```

Your output may look different from what is above, and that is okay since nothing is deterministic when we do this sort of parallelized concurrency. But notice we have processing happening on at least three threads (RxComputationScheduler-

1, 3, and 4). Threads will be assigned at random. Since each `Observable` created by an emission inside a `flatMap()` will take its own thread from the given `Scheduler`, each resulting `Observable` will indepedently process emisssions on a separate thread *within the* `flatMap()`.

It is critical to note that the `flatMap()` can fire emissions from multiple Observables inside it, all of which may be running on different threads. But to respect the `Observable` contract, it must make sure that emissions leaving the `flatMap()` towards the `Observer` are serialized in a single `Observable`. If one thread is busy pushing items out of the `flatMap()`, the other threads will leave their emissions for that occupying thread to take ownership of in a queue. This allows the benefit of concurrency without any blocking or synchronization of threads.

> You can learn more about achieving RxJava parallelization in two articles written by yours truly: Acheiving Parallelization and [Maximizing Parallelization](http://tomstechnicalblog.blogspot.com/2016/02/rxjava-maximizing-parallelization.html).

Summary

In ths chapter we have learned one of the main selling points of Rxjava: concise, flexible, and composable concurrency. You can compose Observables to change concurrency policies at any time with the `subscribeOn()` and `observeOn()`. This makes applications adaptable, scalable, and evolvable over long periods of time. You do not have to mess with synchronizers, semaphores, or any other low-level concurrency tools as RxJava takes care of these complexities for you.

But we are not quite done yet. As we will see in the next chapter, we can leverage concurrency to create features you might have thought impractical to put in your applications.

9. Switching, Throttling, and Buffering

In the previous chapter, we learned that RxJava makes concurrency accessible and fairly trivial to accomplish. But being able to compose concurrency easily enables us to do much more with RxJava.

In UI development, users will inevitably click things that kick off long-running processes. Even if you have concurrency in place, users that rapidly select UI inputs can kick of expensive processes, and those processes will start to queue up undesirably. Other times, we may want to group up rapid emissions to make them a single unit, such as typing keystrokes. There are tools to effectively overcome all these problems, and we will cover them in this chapter.

A Quick Note About Flowables and Backpressure

One topic that I've decided is beyond the scope of this book is backpressure, which involves using a `Flowable` instead of an `Observable`. A `Flowable` is just like an `Observable` but it has a notion of "pushing back" on the source and telling it to slow down. This way, highly concurrent chains of operations do not create bottlenecks between consumers and producers, potentially causing the JVM to run out of memory.

Flowables are highly critical if you are working with large amounts of data concurrently. However, you cannot effectively use Flowables against user interface events because a user cannot programmatically be told to "slow dow' and respect a backpressure request. You can, however, use reactive operators to "knock down" emissions from a rapidly-firing source which this chapter will cover. You can also compose Flowables and Observables together using conversion operators like `Observable#toFlowable()` and `Flowable#toObservable()`, as well as `Observable#flatMapPublisher()` and `Flowable#flatMapObservable()`.

But backpressure and Flowables are beyond the scope of this book. Please read Chapter 8 of my Packt book *Learning RxJava* to get thorough examples, explanations, and use cases for Flowables and backpressure.

Switching with `switchMap()`

Let's emulate a situation where rapid user inputs could overwhelm your application with requests. Say you have two `ListView<T>` controls. The top one has `String` values, and the bottom one will always display the individual characters for each selected `String`. When you select "Alpha" on the top one, the bottom one will contain items "A","l","p","h", and "a" (Figure 9.1).

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox root = new VBox();

        ListView<String> listView = new ListView<>();

        listView.getItems().setAll("Alpha","Beta","Gamma",
                "Delta","Epsilon","Zeta","Eta");

        ListView<String> itemsView = new ListView<>();

        JavaFxObservable.emitOnChanged(listView.getSelectionMode
    l().getSelectedItems())
```

```
                .flatMapSingle ( list -> Observable.fromIterable
(list)
                        .flatMap (s -> Observable.fromArray(s.sp
lit("(?!^)")))
                        .toList()
                ).subscribe(l -> itemsView.getItems().setAll(l))
;

        root.getChildren().addAll(listView, itemsView);

        stage.setScene(new Scene(root));

        stage.show();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.onChangedObservable
import io.reactivex.rxkotlin.toObservable
import tornadofx.*

class MyApp: App(MyView::class)

class MyView : View() {

    val items = listOf("Alpha","Beta","Gamma",
            "Delta","Epsilon","Zeta","Eta").observable()

    override val root = vbox {

        val listView = listview(items)

        listview<String> {
            listView.selectionModel.selectedItems.onChangedObser
vable()
                    .flatMapSingle { it.toObservable()
                            .flatMap { it.toCharArray().map(Char
::toString).toObservable() }
                            .toList()
                    }.subscribe { items.setAll(it) }
        }
    }
}
```
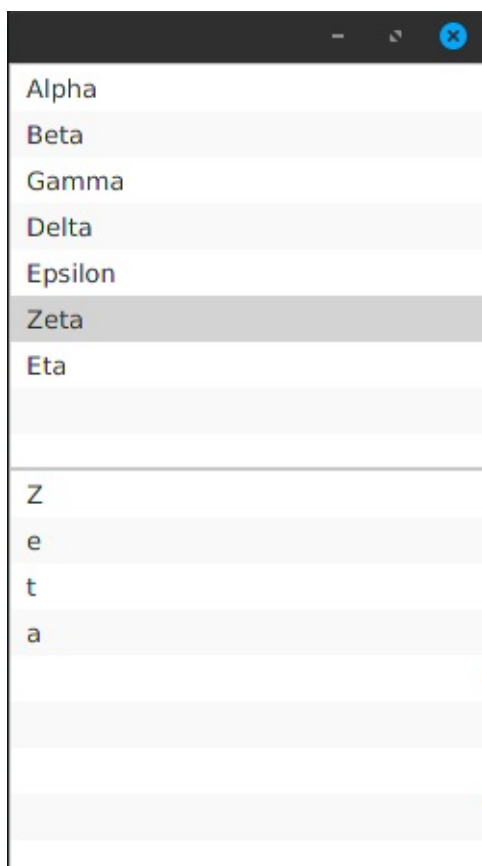
Figure 9.1

This is a pretty quick computation which hardly keeps the JavaFX thread busy. But in the real world, running database queries or HTTP requests can take awhile. The last thing we want is for these rapid inputs to create a queue of requests that will quickly make the application unusable as it works through the queue. Let's emulate this by using the `delay()` operator. Remember that the `delay()` operator already specifies a `subscribeOn()` internally, but we can specify an argument which `Scheduler` it uses. Let's put it in the IO Scheduler. The `Observer` must receive each emission on the JavaFX thread, so be sure to `observeOn()` the JavaFX Scheduler before the emission goes to the `Observer`.

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import io.reactivex.rxjavafx.schedulers.JavaFxScheduler;
import io.reactivex.schedulers.Schedulers;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.layout.VBox;
```

```java
import javafx.stage.Stage;

import java.util.concurrent.TimeUnit;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox root = new VBox();

        ListView<String> listView = new ListView<>();

        listView.getItems().setAll("Alpha","Beta","Gamma",
                "Delta","Epsilon","Zeta","Eta");

        ListView<String> itemsView = new ListView<>();

        JavaFxObservable.emitOnChanged(listView.getSelectionMode
l().getSelectedItems())
                .flatMapSingle( list -> Observable.fromIterable(
list)
                        .delay(3, TimeUnit.SECONDS, Schedulers.i
o())
                        .flatMap (s -> Observable.fromArray(s.sp
lit("(?!^)")))
                        .toList()
                ).observeOn(JavaFxScheduler.platform())
                .subscribe(l -> itemsView.getItems().setAll(l));

        root.getChildren().addAll(listView, itemsView);

        stage.setScene(new Scene(root));

        stage.show();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.observeOnFx
import com.github.thomasnield.rxkotlinfx.onChangedObservable
import io.reactivex.rxkotlin.toObservable
import io.reactivex.schedulers.Schedulers
import tornadofx.*
import java.util.concurrent.TimeUnit

class MyApp: App(MyView::class)

class MyView : View() {

    val items = listOf("Alpha","Beta","Gamma",
            "Delta","Epsilon","Zeta","Eta").observable()

    override val root = vbox {

        val listView = listview(items)

        listview<String> {
            listView.selectionModel.selectedItems.onChangedObservable()
                    .flatMapSingle { it.toObservable()
                            .delay(3,
                                    TimeUnit.SECONDS, Schedulers.io())
                            .flatMap { it.toCharArray().map(Char::toString).toObservable() }
                            .toList()
                    }.observeOnFx().subscribe { items.setAll(it) }
        }
    }
}
```

Now if we click several items on the top `ListView`, you will notice a 3-second lag before the letters show up on the bottom `ListView`. This emulates long-running requests for each click, and now we have these requests queuing up and causing the bottom `ListView` to go berserk, trying to display each previous

request before it gets to the current one. Obviously, this is undesirable. We likely want to kill previous requests when a new one comes in, and this is simple to do. Just change the `flatMapSingle()` that emits the `List<String>` of characters to a `switchMap()`. Since there is no `switchMapSingle()`, just convert that resulting `Single` from `toList()` to an `Observable`.

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import io.reactivex.rxjavafx.schedulers.JavaFxScheduler;
import io.reactivex.schedulers.Schedulers;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

import java.util.concurrent.TimeUnit;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox root = new VBox();

        ListView<String> listView = new ListView<>();

        listView.getItems().setAll("Alpha","Beta","Gamma",
                "Delta","Epsilon","Zeta","Eta");

        ListView<String> itemsView = new ListView<>();

        JavaFxObservable.emitOnChanged(listView.getSelectionModel().getSelectedItems())
                .switchMap ( list -> Observable.fromIterable(list)
                        .delay(3, TimeUnit.SECONDS, Schedulers.io())
```

```
                        .flatMap (s -> Observable.fromArray(s.sp
lit("(?!^)")))
                        .toList()
                        .toObservable()
                ).observeOn(JavaFxScheduler.platform())
                .subscribe(l -> itemsView.getItems().setAll(l));

        root.getChildren().addAll(listView, itemsView);

        stage.setScene(new Scene(root));

        stage.show();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.observeOnFx
import com.github.thomasnield.rxkotlinfx.onChangedObservable
import io.reactivex.rxkotlin.toObservable
import io.reactivex.schedulers.Schedulers
import tornadofx.*
import java.util.concurrent.TimeUnit

class MyApp: App(MyView::class)


class MyView : View() {

    val items = listOf("Alpha","Beta","Gamma",
            "Delta","Epsilon","Zeta","Eta").observable()

    override val root = vbox {

        val listView = listview(items)

        listview<String> {
            listView.selectionModel.selectedItems.onChangedObser
vable()
                    .switchMap { it.toObservable()
                            .delay(3, TimeUnit.SECONDS, Schedule
rs.io())

                            .flatMap { it.toCharArray().map(Char
::toString).toObservable() }
                            .toList()
                            .toObservable()
                    }.observeOnFx().subscribe { items.setAll(it)
 }
        }
    }
}
```

This makes the application much more responsive. The `switchMap()` works identically to any variant of `flatMap()` , but it will only chase after the latest user input and kill any previous requests. In other words, it is only chasing after the

latest `Observable` derived from the latest emission, and unsubscribing any previous requests. The `switchMap()` is a powerful utility to create responsive and resilient UI's, and is the perfect way to handle click-happy users!

You can also use the `switchMap()` to cancel long-running or infinite processes using a neat little trick with `Observable.empty()`. For instance, a `ToggleButton` has a true/false state depending on whether it is selected. When you emit its `false` state, you can return an empty `Observable` to kill the previous processing `Observable`, as shown below. When the `ToggleButton` is selected, it will kick off an `Observable.interval()` that emits a consecutive integer every 10 milliseconds. But unselecting the `ToggleButton` will cause the `flatMap()` to switch to an `Observable.empty()`, killing and unsubscribing from the `Observable.interval()` (Figure 9.2).

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import io.reactivex.rxjavafx.schedulers.JavaFxScheduler;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ToggleButton;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

import java.util.concurrent.TimeUnit;

public class MyApp extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        VBox vBox = new VBox();

        ToggleButton toggleButton = new ToggleButton("START");
        Label timerLabel = new Label("0");

        JavaFxObservable.valuesOf(toggleButton.selectedProperty(
))
                .switchMap(selected -> {
```

```java
                    if (selected) {
                        toggleButton.setText("STOP");
                        return Observable.interval(10, TimeUnit.
MILLISECONDS);
                    } else {
                        toggleButton.setText("START");
                        return Observable.empty();
                    }
                })
                .observeOn(JavaFxScheduler.platform())
                .subscribe(i -> timerLabel.setText(i.toString())
);

        vBox.getChildren().setAll(toggleButton,timerLabel);
        stage.setScene(new Scene(vBox));
        stage.show();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.observeOnFx
import com.github.thomasnield.rxkotlinfx.toObservable
import io.reactivex.Observable
import tornadofx.*
import java.util.concurrent.TimeUnit

class MyApp: App(MyView::class)

class MyView : View() {

    override val root = vbox {

        val toggleButton = togglebutton("START")
        val timerLabel = label("0")

        toggleButton.selectedProperty().toObservable()
                .switchMap { selected ->
                    if (selected) {
                        toggleButton.text = "STOP"
                        Observable.interval(10, TimeUnit.MILLISE
CONDS)
                    } else {
                        toggleButton.text = "START"
                        Observable.empty()
                    }
                }.observeOnFx()
                .subscribe {
                    timerLabel.text = it.toString()
                }
    }
}
```
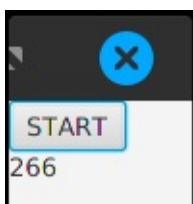
Figure 9.2

The `switchMap()` can come in handy for any situation where you want to switch from one `Observable` source to another.

Buffering

We may want to collect emissions into a `List`, but doing so on a batching condition so several lists are emitted. The `buffer()` operators help accomplish this, and they have several overload flavors.

The simplest `buffer()` specifies the number of emissions that will be collected into a `List` before that `List` is pushed forward, and then it will start a new one. In this example, emissions will be grouped up in batches of `10`.

Java

```java
import io.reactivex.Observable;

public class Launcher {
    public static void main(String[] args) {
        Observable.just(1,100)
                .buffer(10)
                .subscribe(System.out::print);
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable

fun main(args: Array<String>) {
    Observable.range(1,100)
            .buffer(10)
            .subscribe { println(it) }
}
```

OUTPUT:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
[21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
[31, 32, 33, 34, 35, 36, 37, 38, 39, 40]
[41, 42, 43, 44, 45, 46, 47, 48, 49, 50]
[51, 52, 53, 54, 55, 56, 57, 58, 59, 60]
[61, 62, 63, 64, 65, 66, 67, 68, 69, 70]
[71, 72, 73, 74, 75, 76, 77, 78, 79, 80]
[81, 82, 83, 84, 85, 86, 87, 88, 89, 90]
[91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
```

There are other flavors of `buffer()`. Another will collect emissions based on a specified time cutoff. If you have an `Observable` emitting at an interval of 300 milliseconds, you can buffer them into a `List` at every second. This is what the output would look like:

Java

```java
import io.reactivex.Observable;

import java.util.concurrent.TimeUnit;

public class Launcher {
    public static void main(String[] args) {

        Observable.interval(300, TimeUnit.MILLISECONDS)
                .buffer(1, TimeUnit.SECONDS)
                .subscribe(System.out::println);

        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import java.util.concurrent.TimeUnit

fun main(args: Array<String>) {
    Observable.interval(300, TimeUnit.MILLISECONDS)
            .buffer(1, TimeUnit.SECONDS)
            .subscribe { println(it) }


    Thread.sleep(10000)
}
```

OUTPUT:

```
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
[9, 10, 11, 12]
[13, 14, 15]
[16, 17, 18]
[19, 20, 21, 22]
[23, 24, 25]
[26, 27, 28]
[29, 30, 31, 32]
```

Another way to accomplish this is to pass another `Observable` to `buffer()` as an argument, whose each emission (regardless of type) will "cut" and emit the `List` at that moment.

Java

```java
import io.reactivex.Observable;

import java.util.concurrent.TimeUnit;

public class Launcher {
    public static void main(String[] args) {

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

        Observable.interval(300, TimeUnit.MILLISECONDS)
                .buffer(oneSecondInterval)
                .subscribe(System.out::println);

        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import java.util.concurrent.TimeUnit

fun main(args: Array<String>) {

    val oneSecondInterval = Observable.interval(1, TimeUnit.SECONDS)

    Observable.interval(300, TimeUnit.MILLISECONDS)
            .buffer(oneSecondInterval)
            .subscribe { println(it) }

    Thread.sleep(10000)
}
```

OUTPUT:

```
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
[9, 10, 11, 12]
[13, 14, 15]
[16, 17, 18]
[19, 20, 21, 22]
[23, 24, 25]
[26, 27, 28]
[29, 30, 31, 32]
```

This is a helpful way to `buffer()` lists because you can use another `Observable` to control when the Lists are emitted. We will see an example of this at the end of this chapter when we group up keystrokes.

> RxJava-Extras has some additional buffer-like operators, such as `toListWhile()` which will group emissions into a `List` while a condition is true, then it will emit the `List\` and move on to the next one.

Note that there are also `window()` operators that are similar to `buffer()` , but they will return an `Observable<Observable<T>>` instead of an `Observable<List<T>>` . In other words, they will return an `Observable` emitting Observables rather than Lists. These might be more desirable in some situations where you do not want to collect Lists and want to efficiently do further operations on the groupings.

You can read more about `buffer()` and `window()` on the RxJava Wiki.

Throttling

When you have a rapidly firing `Observable` , you may just want to emit the first or last emission within a specified scope. For example, you can use `throttleLast()` (which is also aliased as `sample()`) to emit the last emission for each fixed time interval.

Java

```java
import io.reactivex.Observable;

import java.util.concurrent.TimeUnit;

public class Launcher {
    public static void main(String[] args) {

        Observable.interval(300, TimeUnit.MILLISECONDS)
                .throttleLast(1, TimeUnit.SECONDS)
                .subscribe(System.out::println);

        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import java.util.concurrent.TimeUnit

fun main(args: Array<String>) {

    Observable.interval(300, TimeUnit.MILLISECONDS)
            .throttleLast(1, TimeUnit.SECONDS)
            .subscribe { println(it) }

    Thread.sleep(10000)
}
```

OUTPUT:

```
2
5
8
12
15
18
22
25
28
32
```

`throttleFirst()` will do the opposite and emit the first emission within each time interval. It will not emit again until the next time interval starts and another emission occurs in it.

Java

```java
import io.reactivex.Observable;

import java.util.concurrent.TimeUnit;

public class Launcher {
    public static void main(String[] args) {

        Observable.interval(300, TimeUnit.MILLISECONDS)
                .throttleFirst(1, TimeUnit.SECONDS)
                .subscribe(System.out::println);

        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import java.util.concurrent.TimeUnit

fun main(args: Array<String>) {

    Observable.interval(300, TimeUnit.MILLISECONDS)
            .throttleFirst(1, TimeUnit.SECONDS)
            .subscribe { println(it) }

    Thread.sleep(10000)
}
```

OUTPUT:

```
0
4
8
12
16
20
24
28
32
```

The `debounce()` operator (also aliased as `throttleWithTimeOut()`) will hold off emitting the latest emission until a specified amount of time has passed with no emissions. Below, we have a `debounce()` operator that will push the latest emission after 1 second of no activity. If we send 10 rapid emissions at 100 millisecond intervals, 3 emissions separated by 2 second intervals, and 4 emissions at 500 millisecond intervals, we will likely get this output below:

Java

```java
import io.reactivex.Observable;

import java.util.concurrent.TimeUnit;

public class Launcher {
    public static void main(String[] args) {

        Observable<String> source = Observable.concat(
                Observable.interval(100,TimeUnit.MILLISECONDS).take(10).map(i -> "A" + i),
                Observable.interval(2, TimeUnit.SECONDS).take(3).map(i -> "B" + i),
                Observable.interval(500, TimeUnit.MILLISECONDS).take(4).map(i -> "C" + i)
        );

        source.debounce(1, TimeUnit.SECONDS)
                .subscribe(System.out::println);

        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Kotlin

```kotlin
import io.reactivex.Observable
import java.util.concurrent.TimeUnit

fun main(args: Array<String>) {

    val source = Observable.concat(
            Observable.interval(100,TimeUnit.MILLISECONDS).take(
10).map { "A-$it"},
            Observable.interval(2, TimeUnit.SECONDS).take(3).map
 { "B-$it"},
            Observable.interval(500, TimeUnit.MILLISECONDS).take(
4).map { "C-$it"}
    )

    source.debounce(1, TimeUnit.SECONDS)
            .subscribe { println(it) }

    Thread.sleep(10000)
}
```

OUTPUT:

```
A9
B0
B1
C3
```

I labeled each source as "A", "B", or "C" and concatenated that with the index of the emission that was throttled. You will notice that the 10 rapid emissions resulted in the last emission "A9" getting fired after the 1-second interval of "B" resulted in that inactivity. Then "B0" and "B1" had 1 second breaks between them resulting in them being emitted. But "B3" did not go forward because "C" started firing at 500 millisecond intervals and gave no inactivity interval for it to fire. Then "C3" was the last emission to fire at the final respite.

If you want to see more examples and marble diagrams of these operators, check out the RxJava Wiki article.

Grouping Up Keystrokes

Now we will move on to a real-world example that puts everything in this chapter in action. Say we have a `ListView<String>` containing all 50 states of the United States (I saved them to a [plain text file on GitHub Gist](. When we have the `ListView` selected, we want users to be able to start typing a state and it will *immediately* jump to the first state that starts with that inputted `String`.

Achieving this can be a bit tricky. As a user is typing rapidly, we want to collect those emissions into a single `String` to turn individual characters into words. When the user stops typing, we want to stop collecting characters and push that `String` forward so it is selected in a `ListView`. Here is how we can do that:

Java

```java
import io.reactivex.Observable;
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import io.reactivex.rxjavafx.schedulers.JavaFxScheduler;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.input.KeyEvent;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import java.net.URL;
import java.util.Arrays;
import java.util.List;
import java.util.Scanner;
import java.util.concurrent.TimeUnit;

public final class MyApp extends Application {

    @Override
    public void start(Stage stage) throws Exception {

        VBox root = new VBox();

        //Declare a ListView with all U.S. states
        ListView<String> listView = new ListView<>();
```

```java
        List<String> states = Arrays.asList(getResponse("https:/
/goo.gl/S0xuOi").split("\\r?\\n"));
        listView.getItems().setAll(states);


        //broadcast typed keys
        Observable<String> typedKeys = JavaFxObservable.eventsOf
(listView, KeyEvent.KEY_TYPED)
                .map(KeyEvent::getCharacter)
                .publish().refCount();


        //immediately jump to state being typed
        typedKeys.debounce(200, TimeUnit.MILLISECONDS).startWith(
"")
                .switchMap(s ->
                        typedKeys.scan((x,y) -> x + y)
                                .switchMap(input ->
                                        Observable.fromIterable(
states)
                                                .filter(st -> st
.toUpperCase().startsWith(input.toUpperCase()))
                                                .take(1)
                                )
                ).observeOn(JavaFxScheduler.platform())
                .subscribe(st ->
                        listView.getSelectionModel().select(st)
                );


        root.getChildren().add(listView);


        stage.setScene(new Scene(root));


        stage.show();
    }


    private static String getResponse(String path) {
        try {
            return new Scanner(new URL(path).openStream(), "UTF-
8").useDelimiter("\\A").next();
        } catch (Exception e) {
            return e.getMessage();
```

```
            }
        }
    }
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.events
import com.github.thomasnield.rxkotlinfx.observeOnFx
import io.reactivex.rxkotlin.toObservable
import javafx.collections.FXCollections
import javafx.scene.input.KeyEvent
import tornadofx.App
import tornadofx.View
import tornadofx.listview
import tornadofx.vbox
import java.net.URL
import java.util.concurrent.TimeUnit


class MyApp: App(MyView::class)


class MyView : View("My View") {

    val states = FXCollections.observableList(
            URL("https://goo.gl/S0xuOi").readText().split(Regex(
"\\r?\\n"))
    )

    override val root = vbox {

        val listView = listview<String> {
            items = states
        }

        val typedKeys = listView.events(KeyEvent.KEY_TYPED)
                .map { it.character }
                .publish().refCount()

        typedKeys.debounce(200, TimeUnit.MILLISECONDS).startWith(
"")
```

```
                    .switchMap {
                        typedKeys.scan { x,y -> x + y }
                                .switchMap { input ->
                                    states.toObservable()
                                        .filter { it.toUpperCase
  ().startsWith(input.toUpperCase()) }
                                        .take(1)
                        }
                }.observeOnFx()
                .subscribe {
                    listView.selectionModel.select(it)
                }
        }
    }
```
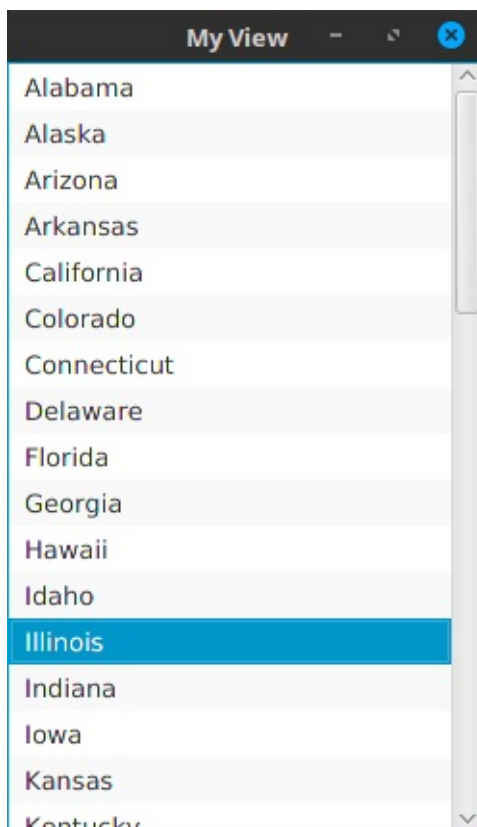
Figure 9.3 - A `ListView` that will select states that are being typed

There is a lot happening here, so let's break it down.

Obviously we set up our `ObservableList<String>` containing all the U.S. states, and set that to back the `ListView` . Then we multicast the keystrokes through the `typedKeys` Observable. We use this `typedKeys` Observable for

two separate tasks: 1) Signal the user has stopped typing after 200ms of inactivity via `debounce()` 2) Receive that signal emission within a `switchMap()` , where `typedKeys` is used again to infinitely `scan()` typed characters and concatentate them together as the user types. Then each concatenation of characters is compared to all the states and finds the first one that matches. That state is then put back on the FX thread and to the `Observer` to be selected.

This is probably the most complex task I have found in using RxJava with JavaFX, but it is achieving an incredible amount of complex concurrent work with little code. Take some time to study the code above. Although it may take a few moments (or perhaps days) to sink in, try to look at what each part is doing in isolation. An infinite `Observable` is doing a rolling concatenation of user keystrokes to form Strings (and using a `switchMap()` to kill off previous searches). That inifinite `Observable` is killed after 200 ms of inactivity and replaced with a new inifinte `Observable` , effectively "resetting" it.

Once you get a hang of this, you will be unstoppable in creating high-quality JavaFX applications that can not only cope, but also leverage rapid user inputs.

Summary

In this chapter, we learned how to handle a high volume of emissions effectively through various strategies. When Observers cannot keep up with a hot `Observable` , you can use switching, throttling, and buffering to make the volume manageable. We also learned some powerful patterns to group up emissions based on timing mechanisms, and make tasks like processing keystrokes fairly trivial.

We are almost done with our RxJava journey. In the final chapter, we will cover a question probably on many readers' minds: decoupling UI's when using RxJava.

10. Decoupling Reactive Streams

In this book, we kept our examples fairly coupled and did not bring any UI code separation patterns. This was to keep the focus on Rx topics and not distract away from them. But in this chapter, we will introduce how you can separate Observables and Observers cleanly so they are not coupled with each other, even if they are in different parts of the UI. This aids goals to create effective code separation patterns and increase maintainability of complex applications.

RxJava has a special reactive type called a `Subject`, which comes in a number of implementations. A `Subject` operates as both an `Observable` and an `Observer`. However, you need to be selective when to use Subjects as they can introduce antipatterns. They can come in handy to decouple reactive streams by having multiple "sources" subscribe a `Subject` to their emissions, and then that `Subject` will pass those emissions downstream to any receiving Observers. Subjects automatically multicast as well.

Using the PublishSubject

A `Subject` cam act as a proxy between one or more source Observables and one or more Observers. The most vanilla type of `Subject` is the `PublishSubject`, which can be called using `PublishSubject.create()`. It will simply relay emissions from one or more upstream sources to one or more downstream Observers, without any additional behaviors.

Below, we do a simple separation between the source of text input values and an `Observer` that consumes them by putting them in a `Label`. The `Subject` will act as a proxy between them.

Java

```java
import io.reactivex.rxjavafx.observables.JavaFxObservable;
import io.reactivex.subjects.PublishSubject;
import io.reactivex.subjects.Subject;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class JavaFxApp extends Application {

    private final Subject<String> textInputs = PublishSubject.create();

    @Override
    public void start(Stage stage) throws Exception {

        TextField textField = new TextField();
        Label label = new Label();

        //pass emissions to the Subject
        JavaFxObservable.valuesOf(textField.textProperty()).subscribe(textInputs);

        //receive emissions from the Subject
        textInputs.map(s -> new StringBuilder(s).reverse().toString())
                .subscribe(label::setText);

        VBox vBox = new VBox(textField, label);

        stage.setScene(new Scene(vBox));
        stage.show();
    }
}
```

Kotlin

```kotlin
import com.github.thomasnield.rxkotlinfx.toObservable
import io.reactivex.subjects.PublishSubject
import tornadofx.*

class MyApp: App(MyView::class)

class MyView : View("My View") {

    val textInputs = PublishSubject.create<String>()

    override val root = vbox {
        textfield {
            textProperty()
                    .toObservable()
                    .subscribe(textInputs)
        }

        label {
            textInputs.map(String::reversed)
                    .subscribe { text = it }
        }
    }
}
```

Using a Subject in a Model

Typically, you will hold the `Subject` in a separate model class of some sort to support your JavaFX applications, and relay emissions from one component to another in an EventBus-like fashion. This is helpful to not only broadcast universal events throughout your application, but also provide several sources to drive a single event.

Java

```java
import io.reactivex.Observable;
import io.reactivex.subjects.PublishSubject;
import io.reactivex.subjects.Subject;
import javafx.event.ActionEvent;



public class MyEventModel {

    private MyEventModel() {}

    private static final MyEventModel instance = new MyEventMode
l();

    public static MyEventModel getInstance() {
        return instance;
    }

    private final Subject<ActionEvent> refreshRequests = Publish
Subject.create();

    public Observable<ActionEvent> getRefreshRequests() {
        return refreshRequests;
    }
}
```

Kotlin

```kotlin
import javafx.event.ActionEvent
import io.reactivex.subjects.PublishSubject

object MyEventModel {
    val refreshRequests = PublishSubject.create<ActionEvent>()
}
```

In this `MyEventModel` we have a `Subject<ActionEvent>` that handles `refreshRequests`, Let's say we wanted three events to drive a refresh: a `Button`, a `MenuItem`, and a key combination "CTRL + R" on a `TableView`.

If you declare these Observables in three separate places throughtout your UI code, you can add each of them to this `CompositeObservable`.

Java

```java
//make refresh Button
Button button = new Button("Refresh");
JavaFxObservable.actionEventsOf(button)
  .subscribe(MyEventModel.getInstance().getRefreshRequests());



//make refresh MenuItem
MenuItem menuItem = new MenuItem("Refresh");
JavaFxObservable.actionEventsOf(menuItemClicks)
  .subscribe(MyEventModel.getInstance().getRefreshRequests());

//CTRL + R hotkeys on a TableView
TableView<MyType> tableView = new TableView<>();


    JavaFxObservable.eventsOf(tableView, KeyEvent.KEY_PRESSED)
        .filter(ke -> ke.isControlDown() && ke.getCode().equals(
KeyCode.R))
        .map(ke -> new ActionEvent());
        .subscribe(MyEventModel.getInstance().getRefreshRequests
());
```

Kotlin

```kotlin
//make refresh button
val button = Button("Refresh")
button.actionEvents().subscribe(MyEventModel.refreshRequests)

//make refresh MenuItem
val menuItem = MenuItem("Refresh")
menuItem.actionEvents().subscribe(MyEventModel.refreshRequests)

//CTRL + R hotkeys on a TableView
val tableView = TableView<MyType>();
tableView.events(KeyEvent.KEY_PRESSED)
    .filter { it.isControlDown && it.code == KeyCode.R }
    .map { ActionEvent() }
    .subscribe(MyEventModel.refreshRequests)
```

These three event sources are now proxied through one `Subject`. You can then have one or more Observers `subscribe()` to this `Subject`, and they will respond to any of these three sources requesting a refresh.

Java

```java
//subscribe to refresh events
MyEventModel.getInstance()
    .getRefreshRequests()
    .subscribe(ae -> refresh());
```

Kotlin

```kotlin
MyEventModel.refreshRequests
    .subscribe { refresh() }
```

You can set up as many models as you like with as many Subjects as you like to pass different data and events back-and-forth throughout your application.

Other Subject Types

There are a couple of other `Subject` implementations to be aware of. `BehaviorSubject` will cache the last emission that will be replayed to every new `Observer`, which can be helpful to always broadcast the latest value selected in a control. `ReplaySubject` will replay all values and indefinitely cache them. `AsyncSubject` will broadcast only the last value after the source calls `onComplete()`, and `UnicastSubject` will cache emissions until it gets the first downstream `Observer`, which it will emit all the items to and then flush its cache.

You can learn more about these subjects in Rx documentation as well as the *Learning RxJava* Packt Book.

Summary

In this chapter we covered how to separate reactive streams between UI components with the `Subject`, which can serve as a proxy between Observable sources and downstream Observers. You can put `Subject` instances in a backing class to serve as an Rx-flavored event bus to relay data and events. Use the `Subject` to consolidate mutliple event sources that drive the same action, or to cleanly separate your Observable sources and terminal Observers.

Closing

You have reached the end of this book. Congrats! Keep researching RxJava and learn what it can do inside and outside of JavaFX. You will find it is used on Android via the RxAndroid and RxBindings libraries, as well as on backend development with RxNetty and other frameworks. I encourage you to keep learning the various operators and check out books and online resources to grow your proficiency.

I highly encourage reading my Packt book *Learning RxJava* to get more thorough knowledge of RxJava beyond JavaFX applications. I wrote it in the same style and spirit as this book, with the intent of helping the largest number of people possible. The book can be purchased on Packt, Amazon, and other book retailers. It is also available via subscription to O'Reilly Safari and Packt Mapt.

https://www.packtpub.com/application-development/learning-rxjava

Please follow me on Twitter @thomasnield9727 for updates on all things Rx. If you have any issues, questions, or concerns please feel free to file an issue or email me at thomasnield@live.com.

Until next time!

Thomas Nield