Reactive Java Programming

Andrea Maglie

Apress®

Reactive Java Programming

Andrea Maglie

Apress®

Reactive Java Programming

Andrea Maglie
Venice, Italy

ISBN-13 (pbk): 978-1-4842-1429-9 ISBN-13 (electronic): 978-1-4842-1428-2
DOI 10.1007/978-1-4842-1428-2

Library of Congress Control Number: 2016957883

Managing Director: Welmoed Spahr
Lead Editor: Steve Anglin
Technical Reviewer: Manuel Jordan Elera
Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Mark Powers
Copy Editor: Mary Behr
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

Dedicated to Alessandra

Contents at a Glance

Contents

About the Author

Andrea Maglie (Venice, Italy, 1981) is an IT Engineer. He graduated from the University of Padua and is a Senior Java/Android Developer.

He has been working on RxJava since 2014, concentrating on Android development.

Currently, he has three apps published in the Play Store as a contributor (MiSiedo, Texa CARe, Musement), plus two apps as an indie developer (Setlist and Loopo). Between 2013 and 2015, he ran Sono Digitale, an Italian podcast about technology and development. In 2015, he founded the Google Developer Group (GDG) of Venice.

In his free time, he plays guitar and writes on his tech blog at `www.andreamaglie.com`.

About the Technical Reviewer

Manuel Jordan Elera is an autodidactic developer and researcher who enjoys learning new technologies for his own experiments and creating new integrations.

Manuel won the 2010 Springy Award – Community Champion and the Spring Champion 2013. Manuel is known as dr_pompeii. He has tech reviewed numerous books for Apress, including *Pro Spring, 4th Edition* (2014), *Practical Spring LDAP* (2013), *Pro JPA 2, Second Edition* (2013), and *Pro Spring Security* (2013).
Read his 13 detailed tutorials about many Spring technologies and contact him through his blog at `www.manueljordanelera.blogspot.com` and follow him on his Twitter account, `@dr_pompeii.`

In his little free time, he reads the Bible and composes music on his guitar.

Acknowledgments

I would like to thank Mark Powers, Steve Anglin, and Apress for enabling me to publish this book.

Above all, I want to thank my love, Alessandra, and the rest of my family, who supported and encouraged me in spite of all the time it took me away from them.

Introduction

Welcome to *Reactive Java Programming*. With this book you'll learn how to transform the way you develop your Java (and Android) applications in a reactive way, moving from synchronous state management with variables to working with asynchronous streams of data. This means that you'll learn how to apply elements of functional programming to Java programs and how to write code that "reacts" to events; you'll also be able to produce shorter, more readable, more maintainable, and less error-prone code. To do this, you'll study the RxJava library, the Java implementation of the reactive extension (Rx) library originally developed by Erik Meijer for .NET.

You'll start by learning what reactive functional programming is and why it's different from imperative programming.

In Chapter 2, you'll see how to include the RxJava library in your projects, and you'll explore the main classes and methods provided by this library.

Chapters 3 and 4 cover more advanced concepts of working with asynchronous streams of data, like error handling and threading.

In Chapter 5, you will apply what you learned in the previous chapters to a specific area: networking.

Finally, in Chapter 6, you will take a look at some libraries created to extend RxJava to Android development.

CHAPTER 1

ReactiveX and RxJava

Introduction

Java is an object-oriented programming language that has been around for many years (it was officially introduced in 1995). Today, it is one of the most appreciated and used languages, thanks to its maturity, stability, and great community support.

Java is anchored to the concepts upon which it was built: Java is an imperative, object-oriented language.

In recent years, new programming paradigms have become popular, such as functional programming and reactive programming. Many new languages have been created.

Java was left behind for some time. However, functional programming was introduced in Java 8 with the support of lambda expressions and streams, but the RxJava library provides the classes and methods we need to implement functional and reactive programming in all Java versions starting from Java 5.

In this chapter, I will introduce the concepts of functional programming, reactive programming, and the Observer pattern. Then I will show you what ReactiveX and RxJava are and how RxJava can help you to write more readable, shorter, more maintainable, and error-free code.

In the following chapters, you will dig into RxJava, learning about the common classes and methods via concrete examples of where you can use them.

I am assuming that you have basic knowledge of Java pogramming, although deep skills are not required.

Imperative and Functional Programming

As you may already know, Java is an imperative programming language. Typically, a Java program consists of a sequence of instructions. Each of these instructions is executed in the same order in which you write them, and the execution leads to changes in the state of the program.

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-1428-2_1) contains supplementary material, which is available to authorized users.

A. Maglie, *Reactive Java Programming*, DOI 10.1007/978-1-4842-1428-2_1

For example, the following code creates a collection of even numbers:

```
List<Integer> input = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> output = new ArrayList<>();

for (Integer x : input) {
  if (x % 2 == 0) {
    output.add(x);
  }
}
```

In order to produce the desired output, you define every step that the program has to take to build the result list, and each step is defined sequentially.

1. Define and create an input list.
2. Define and create an empty output list.
3. Take each item of the input list.
4. If the item is even, add it to the output list.
5. Continue with the following item until the end of the input list is reached.

One alternative to imperative programming is *functional programming*.

In functional programming, the result of the program derives from the evaluation of mathematical functions, without changing the internal program state. In fact, for every function `f(x)`, the result of the function depends on the arguments passed to the function. Each time `f(x)` is called, passing the same parameter `x`, you always get the same result. This is similar to an Object's static method that does not depend on any of the Object's members.

In simpler terms, in functional programming the blocks with which you build the program are not objects but functions and procedures.

So, using functional programming, the example above can be rewritten with the following pseudocode:

```
var output = input.where( x -> x % 2 == 0);
```

Here, you don't have a sequence of steps but just a function (`x % 2 == 0`) passed as a parameter to another function (`where()`) that is applied to an object (`input`). The arrow (`->`) annotation means "apply function `f(x)` (right side of the expression) to the variable `x` (left side of expression)."

The features of a functional language are the following:

- **Higher-order functions**: Higher-order functions are functions that take other functions as arguments.
- **Immutable data**: Data is immutable by default; instead of modifying existing values, functional languages often operate on a copy of original values to preserve them (in Java, primitive types are already immutable but an object is not, so its implementation must not allow the object's state to be changed after creation).

- **Concurrency**: Concurrency is supported and is safer to implement, thanks also to the immutability by default.
- **Referential transparency**: This term defines the fact that computations can be performed at any time, always producing the same result (similar to static methods in Java).
- **Lazy evaluation**: Values can be computed only when needed (lazily) because functions can be evaluated at any time, always giving the same result (these functions do not depend on the program's internal state).

There are programming languages that are defined as *purely functional programming languages,* like Haskell, Hope, and Mercury. Java is not one of these languages, but we can get the advantages of functional programming also in Java.

With the release of Java 8, some constructs of functional programming have been added, like lambda functions and streams. But with the RxJava library we can use concepts of functional programming with Java 1.7 and Java 1.6.

Lambda Expressions

Lambda expressions are anonymous functions; the lambda operator is indicated using an arrow symbol pointing to the right (->). Inputs are placed at the left of the operator, and the function body is placed at the right.

In Java, lambda expressions can be used to replace anonymous inner classes that implement an interface with just one method. For example, consider the following `Button` object:

```
class Button {
  ...
  setOnClickListener(OnButtonClickListener listener) {
    ...
  }
}

interface OnButtonClickListener {
  void onButtonClicked();
}
```

To attach a click listener to the `Button`, you can use an anonymous function:

```
Button button = ...
button.setOnClickListener(
  new OnButtonClickListener() {
    void onButtonClicked() {
        // do something on button clicked
      }
  }
)
```

Using lambda expressions, the code above becomes the following:

```
Button button = ...
button.setOnClickListener( () -> // do something on button clicked )
```

On the right side of the lambda operator you include all of the code to be executed when the button is clicked; this code has no input, as indicated by the two brackets on the left side of the lambda operator.

If the method `onButtonClicked` accepts some parameters, the example above becomes

```
interface OnButtonClickListener {
  void onButtonClicked(Object param);
}

Button button = ...
button.setOnClickListener( param ->
  // do something on button clicked
  // param can be used in this code block
)
```

Java support for lambda expressions was introduced in Java 8, but you can use them in previous Java versions using the retrolambda library (https://github.com/evant/gradle-retrolambda).

Imperative or Functional?

So, why should you choose functional programming over imperative programming?

Functional code is often shorter and easier to understand than the corresponding imperative code. You can do the same work by writing less code, and every programmer knows that less code leads to less bugs.

In imperative programming, implementing abstraction requires you to define interfaces and split code into components that implement those interfaces; functional languages make it easier to create abstractions (just think about how lambda expressions avoid the necessity of creating interfaces with implementations).

Reactive Programming

Reactive programming takes functional programming a little bit further, by adding the concept of data flows (see the next section) and propagation of data changes.

In imperative programming, a value can be assigned to a variable in the following way:

```
x = y + z
```

Here, the sum of `y` and `z` will be assigned to variable `x` at the same time that the function is called; later, variables `y` and `z` can change, but these changes will not automatically influence the value of `x`.

In reactive programming, the value of `x` *should updated whenever the values of* `y` *or* `z` *change.*

So, if the initial values are `y = 1` *and* `z = 1`*, you'll have*

```
x = y + z = 2.
```

If `y` *(or* `z`*) changes its value, this does not mean that* `x` *changes automatically, but you must implement a mechanism to update the values of* `x` *when values of* `y` *and* `z` *are changed.*

Functional reactive programming is a new programming paradigm; it was made popular by Erik Meijer (who created the Rx library for .NET when working at Microsoft) and it's based on two concepts:

- Code "reacts" to events.
- Code handles values as they vary in time, propagating changes to every part of the code that uses those values.

Streams of Data

The key to understand reactive programming is to think about it as operating on a stream of data.

But what do I mean by "stream of data?" I mean a sequence of events, where an event could be user input (like a tap on a button), a response from an API request (like a Facebook feed), data contained in a collection, or even a single variable.

In reactive programming, there's often a component that acts as the source, emitting a sequence of items (or a stream of data), and some other components that observe this flow of items and react to each emitted item (they "react" to item emission).

The Observer Pattern

The Observer pattern is a design pattern in which there are two kinds of objects: observers and subjects. An observer is an object that observes the changes of one or more subjects; a subject is an object that keeps a list of its observers and automatically notifies them when it changes its state.

The definition of the Observer pattern from the "Gang of Four" book (*Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, ISBN 0-201-63361-2) is to

> *"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."*

This pattern is the core of reactive programming. It fits perfectly the concept of reactive programming by providing the structures to implement the produce/react mechanism.

Java SDK implements the Observer pattern with the class `java.util.Observable` and the interface `java.util.Observer`.

```
class Subject extends java.util.Observable {

  public void doWorkAndNotify() {
    Object result = doWork();
    notifyObservers(result);
  }

}

class MyObserver implements Observer {

  @Override
  public void update(Observable obs, Object item) {
    doSomethingWith(item)
  }

}
```

The `Subject` class extends `java.utils.Observable` and is responsible for producing an object and notifying the observers as soon as the item has been produced.

`MyObserver` implements `Observer` and is responsible for observing `Subject` and consuming every item that `Subject` produces.

Putting `Subject` and `MyObserver` together,

```
MyObserver myObserver = new MyObserver();
Subject subject = new Subject();
subject.addObserver(myObserver);
subject.doWorkAndNotify();
```

Unfortunately, this implementation reveals itself to be too simple when you start to write more complex logic.

You'll never use this implementation; instead you'll use the built-in RxJava implementation.

What's ReactiveX?

From http://reactivex.io/ comes the following defintion: *ReactiveX is a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming.*

It's a library that implements functional reactive programming in many languages. It uses "observables" to represent asynchronous data streams, and it abstracts all details related to threading, concurrency, and synchronization. Thanks to ReactiveX, writing concurrent programs becomes a lot easier because

- You don't have to deal with multithreading problems.
- You can easily transform a data stream into another data stream (where the data type can differ from the source stream's data type).
- You can easily combine different data streams (like merging two or more data streams into one stream or concatenating streams).

What's RxJava?

ReactiveX has been implemented as a library for the most used programming languages: Java, JavaScript, C#, Scala, Clojure, C++, Ruby, Python, Groovy, JRuby, Kotlin, Swift, and more. (See the full list at http://reactivex.io/languages.html.)

RxJava is a library that implements the concepts of ReactiveX in Java. As you will see in following chapters, you can rewrite the imperative code that filters even numbers using RxJava:

```
List<Integer> input = Arrays.asList(1, 2, 3, 4, 5);

Observable.from(input)
  .filter(new Func1() {
      @Override
      public Boolean call(Integer x) {
              return x % 2 == 0;
      }
  })
```

Or, using a lambda expression:

```
Observable.from(input)
                .filter(x -> x % 2 == 0);
```

The resulting object (the instance of `rx.Observable`) will generate a sequence of the even numbers contained in the input sequence: 2 and 4.

In RxJava, `rx.Observable` adds two semantics to the Gang of Four's Observer pattern (the default semantic is to emit created items, like a list with items 2,4 in the example above):

- The producer can notify the consumer that there is no more data available.
- The producer can notify the consumer that an error has occurred.

■ **Note** The RxJava library provides a programming model where we can work with events generated from UI or asynchronous calls in the same way in which we operate with collections and streams in Java 8.

The RxJava library was created at Netflix as a smarter alternative to Java Futures and callbacks. Both Futures and callbacks are straightforward to use when there's just one level of asynchronous execution, but they are hard to manage when they're nested.

The following example shows how the nested callbacks problem is handled in RxJava.

EXAMPLE: NESTED API CALLS

Suppose that you need to call a remote API to authenticate a user, then another one to get the user's data, and another API to get a user's contacts. Typically, you would have to write nested API calls like this:

```
User user = null;

serviceEndpoint.login(username, password, new  Callback<AccessToken>() {

  @Override
  public void success(User user, Response response) {

    // store accessToken somewhere

  serviceEndpoint.getUser(new Callback<User>() {
    @Override
    public void success(User userResponse, Response response) {

      user = userResponse;

        serviceEndpoint.getUserContact(user.getId(), new
Callback<Contact>() {
            @Override
            public Contact success(Contact contact, Response response) {
              user.setContact(contact);
            }

            @Override
            public void failure(RetrofitError error) {
              // handle error here...
            }
        });
    }

    @Override
    public void failure(RetrofitError error) {
        // handle error here...
    }
  });
```

```
  }

  @Override
  public void failure(RetrofitError error) {
      // handle error here...
  }
});
```

With RxJava, the nested callbacks are replaced with more efficient, readable, and maintainable composed functions:

```
serviceEndpoint.login()
            .doOnNext(accessToken -> storeCredentials(accessToken))
            .flatMap(accessToken -> serviceEndpoint.getUser())
            .flatMap(user -> serviceEndpoint.getUserContact(user.getId()))
```

As you can see, this piece of code has all the properties of functional programming, with the addition of the reactive component (functions are executed as a reaction to the response received from the `login()` method).

CHAPTER 2

Observables and Observers

Introduction

In this chapter, you'll dive into the RxJava library. First, you'll learn how to include RxJava in your Java project. Then, you will learn

- About the building blocks of RxJava (`rx.Observable<T>`, `rx.Observer<T>`, `rx.Subscriber<T>`)
- The type of events that can be emitted by `rx.Observable<T>` and received by `rx.Observer<T>`
- The operators that can be applied to observables

Adding RxJava to Your Project

The RxJava library (https://github.com/ReactiveX/RxJava) can be included by simply adding the corresponding dependency to your project (no other dependencies are required). At the time of writing, the latest version is 1.1.10.

You include it in a Maven project like so:

```
<dependency>
<groupId>io.reactivex</groupId>
<artifactId>rxjava</artifactId>
<version>1.1.10</version>
</dependency>
```

If you're working with a gradle project, it looks like this:

```
compile 'io.reactivex:rxjava: 1.1.10'
```

RxJava supports all versions of JDK starting from Java 6. If you're working with Java 8, you can take advantage of Java native support for lambda expressions; otherwise, you can use lambda expression by adding retrolambda as a dependency (https://github.com/evant/gradle-retrolambda).

A. Maglie, *Reactive Java Programming*, DOI 10.1007/978-1-4842-1428-2_2

Definition of Observable

An Observable is an object that emits a sequence (or stream) of events. It represents a push-based collection, which is a collection in which events are pushed when they are created.

An observable emits a sequence that can be empty, finite, or infinite. When the sequence is finite, a *complete event* is emitted after the end of the sequence. At any time during the emission (but not after the end of it) an *error event* can be emitted, stopping the emission and cancelling the emission of the complete event.

When the sequence is empty, only the complete event is emitted, without emitting any item. With an infinite sequence, the complete event is never emitted.

As you'll see later, the emission can be tranformed, filtered, or combined with other emissions.

Definition of Observer

An Observer is an object that subscribes to an Observable. It listens and reacts to whatever sequence of items is emitted by the Observable.

The Observer is not blocked while waiting for new emitted items, so in concurrent operations, no blocking occurs. It just wakes up when a new item is emitted.

This is one of the core principles of reactive programming: instead of executing instructions one at a time (always waiting for the previous instruction to be completed), the observable provides a mechanism to retrieve and transform data, and the Observer activates this mechanism, all in a concurrent way.

The following pseudocode is an example of the method that the Observer implements that reacts to the Observable's items:

```
onNext = { it -> doSomething }
```

Here, the method is defined, but nothing is invoked. To start reacting, you need to subscribe to the Observable:

```
observable.subscribe(onNext)
```

Now the observer is listening for items and will react to every new item that will be emitted.

Let's rewrite this example in Java code using RxJava APIs:

```
public void subscribeToObservable(Observable<T> observable) {
  observable.subscribe(nextItem -> {
    // invoked when Observable emits an item
    // usually you will consume the nextItem here
  });
}
```

Now it's clear that in order to connect an observable with an observer, you must use the subscribe method.

onNext, onCompleted, onError

The `rx.Observer<T>` interface does not define only the `onNext(T)` method, but also the following methods:

- `onCompleted()` notifies the Observer when the Observable stops emitting items because the sequence is completed normally.
- `onError(Throwable)` notifies the Observer when the Observable raises an error and stops emitting items, even if the sequence is not completed.

```
public void subscribeToObservable(Observable<T> observable) {
  observable.subscribe(new Subscriber<>() {

    @Override
    public void onCompleted() {
      // invoked when Observable stops emitting items
    }

    @Override
    public void onError(Throwable e) {
      // invoked when Observable throws an exception
      // while emitting items
    }

    @Override
    public void onNext(T nextItem) {
      // invoked when Observable emits an item
      // usually you will consume the nextItem here
    }
  });
}
```

But wait, why are you using an instance of `rx.Subscriber<T>` here? If you take a look at the RxJava documentation, you will see that `Subscriber<T>` is an object that implements the `rx.Observer<T>` interface, so it's legal to use it as an Observer. The reason why you use `Subscriber` instead of any other implementation of the Observer interface is that `Subscriber` also implements the Subscription interface, which allow you to check if the subscriber is unsubscribed (with the `isUnsubscribed()` method) and to unsubscribe it (with the `unsubscribe()` method).

For simplicity, I will omit the generic syntax for `rx.Subscriber<T>`, `Observer<T>`, and `Observable<T>`, unless it affects the readability and comprehension of text and examples.

From the previous example, notice that an Observer reacts to three types of events:

- *Item emission by the Observable*: It occurs zero, one, or more times. If the sequence completes correctly, the `onNext` method will be invoked as many times as the number of items in the sequence. If an error occurs at a certain point, the `onNext` method won't be invoked any further.

- *The completion of items emission*: Only when all items in the sequence are emitted correctly will the `onCompleted` method be invoked. It's invoked only once, and after the last item has been emitted. It also can never happen if you're working with an infinite sequence.
- *An error*: Error can occur in every moment of the sequence, and the sequence will stop immediately. In this case, the method `onError` will be invoked, passing the error as a Throwable object. The other two methods, `onNext` and `onCompleted`, won't be invoked.

An observable cannot notify both `onCompleted` and `onError` methods, only one of them. It will always be the last method invoked.

■ **Note** If you use the shortest notation

```
observable.subscribe(nextItem -> {
    // do something with nextItem

  });
```

you won't get notified when the sequence completes. More importantly, if an error occurs, an exception will be thrown by RxJava because no implementation of `onError` can be found, and your app will crash!

With the `Observable.subscribe()` method (an operation-called subscription), you can connect an Observable to an Observer, but what if you want to disconnect them? This operation is called *unsubscription* and it looks like this:

```
public void subscribeToObservable(Observable<T> observable) {
  Subscription subscription =
    observable.subscribe(new Subscriber() {
          @Override
          public void onCompleted() {
            // invoked when Observable stops emitting items
          }

          @Override
          public void onError(Throwable e) {
            // invoked when Observable throws an exception
            // while emitting items
          }

          @Override
          public void onNext(T nextItem) {
            // invoked when Observable emits an item
```

```
            // usually you will consume the nextItem here
          }
      })

   // disconnect observable and observer
   subscription.unsubscribe()
}
```

You can check if the subscription has been unsubscribed (Observer and Observable are no longer connected) with the following method:

```
subscription.isUnsubscribed()
```

The unsubscribe method can be called at any time during items emission. After the call to unsubscribe, `onNext` won't receive any other item, and the other two methods, `onCompleted` and `onError`, won't be notified. After unsubscription, the observable can stop or continue with item emission, but the observer will not be notified about it.

Hot and Cold Observables

In the examples so far we assumed that an Observable begins emitting a sequence of items when the Observer subscribes to it: they are called *cold observables*. Cold observables always wait to have at least one observer subscribed to start emitting items.

On the other hand, an Observable that begins emitting items before being connected to an observer is called a *hot observable*.

With hot observables, an observer can subscribe and start receiving items at any time during the emission. With hot observables, the observer may receive the complete sequence of items starting from the beginning or not.

■ **Note** There's another kind of observable called a *connectable observable*. This kind of observable begins emitting items when its "connect" method is called, whether or not any observers have subscribed to it.

Let's go with a more concrete, yet simple, example. Let's create an Observable that emits all integers from 1 to 5 and subscribe to it:

```
Observable<Integer> observable =
  Observable.from(new Integer[]{1, 2, 3, 4, 5});

observable.subscribe(new Subscriber<Integer>() {
  @Override
  public void onCompleted() {
    System.out.println("Sequence completed!");
  }
```

```
    @Override
    public void onError(Throwable e) {
      System.err.println("Exception: " + e.getMessage());
    }

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

The expected output is

```
next item is: 1
next item is: 2
next item is: 3
next item is: 4
next item is: 5
Sequence completed!
```

Let's go over the details. This is a cold observable because it will begin emitting items only when the observer subscribes. The observable will generate a sequence of five items, each one representing an integer object (from 1 to 5), so the `onNext` method of the observer will be invoked five times. At the end of the sequence, the method `onCompleted` will be notified. The method `onError` will never be notified because this sequence does not generate any kind of error or exception.

An example of a hot observable could be an observable that emits an event every time a UI button is clicked. It does not start to emit events when the observer subscribes; it emits events even if no Subscriber is subscribed. I will talk about hot observables in the section dedicated to Subjects.

As you may noticed, in this example, you created an observable using the method `Observable.from()`, a static factory method that can create an Observable out of an array, an iterable, or a Future.

This is not the only way to create observables.

Creating Observables

The easiest way to create an observable is to use the factory methods that are implemented in the RxJava library. You've already seen how to create an Observable using the `Observable.from()` method, so let's take a look at the other available methods.

Observable.just()

`Observable.just()` creates an Observable that emits the object or the objects that are passed in as parameters:

```
Observable.just("an item")
```

```
  Observable.just("first item", "second item")
  Observable.just(1, 2, 3)
```

With this operator you can rewrite your previous example as

```
Observable<Integer> observable =
    Observable.just(1, 2, 3, 4, 5);
observable.subscribe(new Subscriber<Integer>() {
  @Override
  public void onCompleted() {
    System.out.println("Sequence completed!");
  }

  @Override
  public void onError(Throwable e) {
    System.err.println("Exception: " + e.getMessage());
  }

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

The output will be the same:

```
next item is: 1
next item is: 2
next item is: 3
next item is: 4
next item is: 5
Sequence completed!
```

Observable.range()

`Observable.range(a, n)` creates an Observable that emits a range of n consecutive integers starting from a.

`Observable.just(1, 2, 3, 4, 5)` and `Observable.range(1, 5)` will emit the same sequence.

Observable.interval()

The previous methods created observables that emit items in sequence, one after another, with no delay between items.

But what if you want your items to be emitted with some time interval?

`Observable.interval(long, TimeUnit)` does exactly this: it creates an Observable that emits a sequence of integers starting from 0 that are spaced by a given time interval. The first argument is the amount of time, and the second argument defines the time unit.

The following observable emits an item every 1 second:

```
Observable.interval(1, TimeUnit.SECONDS)
```

The sequence is an infinite sequence, with no natural end, so `onCompleted` will never be notified. The sequence stops only when no more observers are connected (subscribed) to the observable.

Observable.timer()

`Observable.timer(long, TimeUnit)` creates an Observable that emits just one item after a given delay. It can be useful when combined with other observables to introduce a delay before the beginning of another observable's sequence, as you will see later.

Observable.create()

`Observable.create()` is the method that lets you create an Observable from scratch. For example, if you want to create an observable that emits only one string, "Hello!", you can write

```
Observable.create(
  new Observable.OnSubscribe<String>() {
    @Override
    public void call(Subscriber<? super String> observer) {
      observer.onNext("Hello!");

      observer.onCompleted();
    }
  }
);
```

Suppose now that you want to create an observable that emits a JSON string resulting from a networking operation. If the response is successful, the observable will emit the result and terminate. Otherwise, it will raise an error.

```
Observable.create(
  new Observable.OnSubscribe<String>() {
    @Override
    public void call(Subscriber<? super String> observer) {
      Response response = executeNextworkCall();
      if (observer.isUnsubscribed()) {
        // do not emit the item,
        // observer is not subscribed anymore
        return;
      }
```

```
      if (response != null && response.isSuccessful()) {
        observer.onNext(convertToJson(response));
        observer.onCompleted();
      } else {
        observer
          .onError(new Exception("network call error"));
      }
    }
  }
);
```

Observable.empty()

`Observable.empty()` creates an Observable that emits an empty sequence (zero items) and then completes. So only `onCompleted()` will be notified.

It can be useful if you want to emit an empty sequence insted of emitting null items or throwing errors, like so:

```
Object data = ...;

public Observable<Object> getData() {
  if (data == null) {
    return Observable.empty();
  } else {
    return Observable.just(data);
  }
}
```

Observable.error()

`Observable.error(throwable)` creates an Observable that emits an empty sequence (zero items) and then notifies an error. So only `onError()` will be called.

```
Object data = ...;

public Observable<Object> getData() {
  if (data == null) {
    return Observable.error(new Exception("no data!"));    } else {
    return Observable.just(data);
  }
}
```

Observable.never()

`Observable.never()` creates an Observable that emits an empty sequence (zero items) and never completes. No method of the observer will be invoked.

Observable.defer()

`Observable.defer()` creates an Observable only when a Subscriber subscribes.

The best way to explain what `defer()` does is with the following example. Let's start from the class `Person`, which has two fields: `name` and `age`.

```
class Person {
    private String name;
    private int age;

    public void setAge(int age) {
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public String getName() {
        return name;
    }
}
```

Now create an instance of `Person`, two Observables to be notified with `age` and `name` values, and set the values for `age` and `name`:

```
// create a new instance of Person
final Person person = new Person();

Observable<String> nameObservable =
  Observable.just(person.getName());

Observable<Integer> ageObservable =
  Observable.just(person.getAge());

// set age and name
person.setName("Bob");
person.setAge(35);

ageObservable.subscribe(new Subscriber<Integer>() {
    @Override
    public void onCompleted() {

    }
```

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

    }

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

nameObservable.subscribe(new Subscriber<String>() {
    @Override
    public void onCompleted() {

    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onNext(String name) {
        System.out.println("name is: " + name);
    }
});
```

What happens when you call methods `observeName()` and `observeAge()` on an instance of `Person`? What will be the sequence emitted by the observables? Unfortunately, the output will be

```
age is: 0
name is: null
```

This is not what you wanted. The problem here is that `Observable.just()` is evaluated as soon as it's invoked, so it will create a sequence using the exact value that `name` and `age` reference when the observable is created. In the example, when the observable is created, `age` is `0` and `name` is null.

You'd like to have a little different behavior here: you want the values to be evaluted when you subscribe to the observables, and you can do this using `Observable.defer()`.

`Observable.defer()` accepts an instance of `Func0<Observable<T>>` as parameter. `Func0<R>` is an interface that exposes just one method that accepts zero arguments and returns a value of type `R`. There is also a `Func1<T,R>` interface that exposes just one method that accepts one argument of type `T` and returns a value of type `R`. RxJava provides similar interfaces for up to nine arguments (`Func9<T,R>`) and a varagrs version (`FuncN<R>`).

```
Observable<String> nameObservable =
  Observable.defer(new
    Func0<Observable<String>>() {
      @Override
      public Observable<String> call() {
        return Observable.just(person.getName());
      }
    });

Observable<Integer> ageObservable =
  Observable.defer(new Func0<Observable<Integer>>() {
    @Override
    public Observable<Integer> call() {
      return Observable.just(person.getAge());
    }
});
```

By using these two observables, the output of the previous examples becomes

```
age is: 35
name is: Bob
```

And that's what you expected.

Composing and Transforming Observables

Observables are especially good at being composed and transformed. With the usage of some operators defined in the library, you can compose and transform sequences of data in a easy way that requires little coding, so it's less prone to error.

In the following sections I'll use special diagrams called marble diagrams that efficiently explain what operators do.

map

The `map` operator is one of the most common operators in reactive programming. It lets you transform every item of the emitted sequence with a specified function.

In the example shown in Figure 2-1, the input sequence is a series of integers (1, 2, and 3), and every item of the sequence is multiplied by 10. Applying the `map` operator to an observable, a new observable will be created; this new observable will emit x*10 every time the first observable emits x.

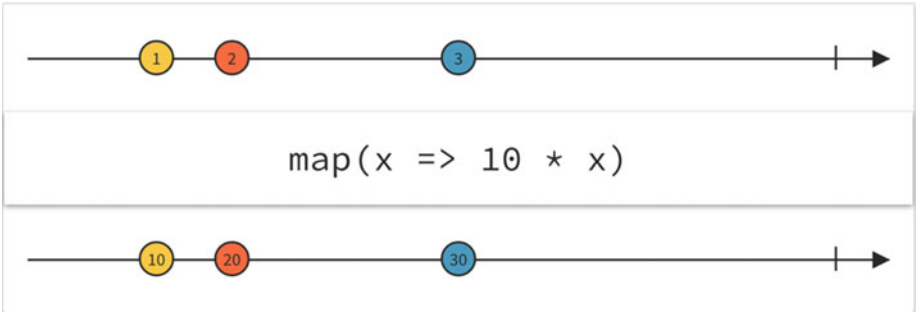

Figure 2-1. *A marble diagram of the map operator*

```
Observable.just(1, 2, 3)
                .map(new Func1<Integer, Integer>() {
                    @Override
                    public Integer call(Integer x) {
                        return x * 10;
                    }
                })
                .subscribe(new Subscriber<Integer>() {
                    @Override
                    public void onCompleted() {

                    }

                    @Override
                    public void onError(Throwable e) {

                    }

                    @Override
                    public void onNext(Integer integer) {
                        System.out.println("next item is: " + integer);
                    }
                });

The output of this operation is

next item is: 10
next item is: 20
next item is: 30
```

Let's try to understand the marble diagram: the line above represents the original sequence of items that are emitted by the original observable. The line below represents the items transformed by the operator and emitted by the second observable (remember that the second observable is the first one plus the operator). The box in the middle contains the function that the operator will perform on each item.

The two lines must be considered as synchronized timelines: the marbles are placed in a way that reflects what happens in time.

Figure 2-1 shows that when the marble with item 1 is emitted, the marble with item 10 is also emitted, so the transformation occurs as soon as the item 1 is emitted, before item 2 is emitted. When item 2 is emitted, the operator is applied and item 20 is emitted, and so on.

flatMap

The `flatMap` operator (Figure 2-2) performs two types of actions: the "map" action that transforms the emitted items into observables and a "flatten" action that converts those observables into one observable. So, you can use `flatMap` to convert the input observable into any other observable (or, in other words, to convert the input stream into a different stream).

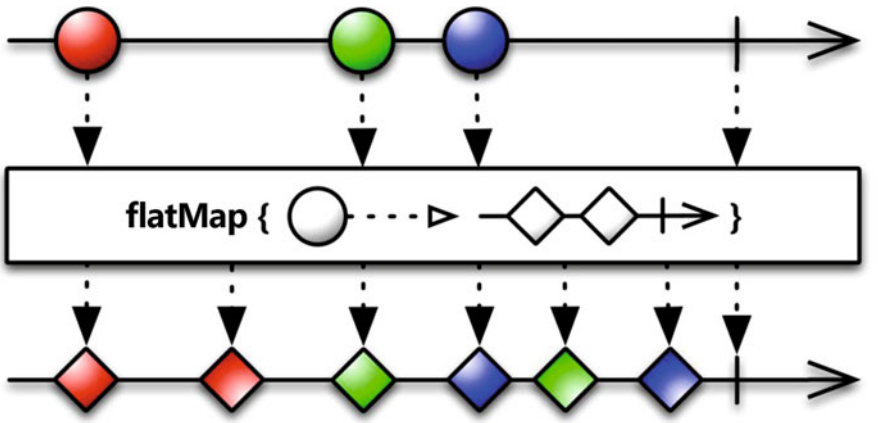

Figure 2-2. *A marble diagram of the flatMap operator*

So, like the `map` operator, `flatMap` applies a function to each emitted item, but this function must return an observable. Then those items are merged into one observable, so the observables may be interleaved. If you need your observables not to be interleaved, you must use the operator `concatMap`.

For example, suppose that you have a list of integers as input and you want to convert it in a sequence of strings in the format "Number x" where x is the next integer in the input list. You can combine `flatMap` and `map` as follows:

```
List<Integer> input = Arrays.asList(1, 2, 3, 4, 5);

  Observable.just(input)
```

```
    .flatMap(new Func1<List<Integer>, Observable<Integer>>() {
      @Override
      public Observable<Integer> call(List<Integer> item) {
        return Observable.from(item);
      }
    })
    .map(new Func1<Integer, String>() {
      @Override
      public String call(Integer t) {
        return "Number " + t;
      }
    })
    .subscribe(new Subscriber<String>() {
      @Override
      public void onCompleted() {
        System.out.println("sequence completed!");
      }

      @Override
      public void onError(Throwable e) {
      }

      @Override
      public void onNext(String item) {
        System.out.println("next item is: " + item);
      }
  });
```

The results are

```
next item is: Number 1
next item is: Number 2
next item is: Number 3
next item is: Number 4
next item is: Number 5
sequence completed!
```

concatMap

The `concatMap` operator (Figure 2-3) behaves like `flatMap`, except that it ensures that the observables are not interleaved but concatenated, keeping their order.

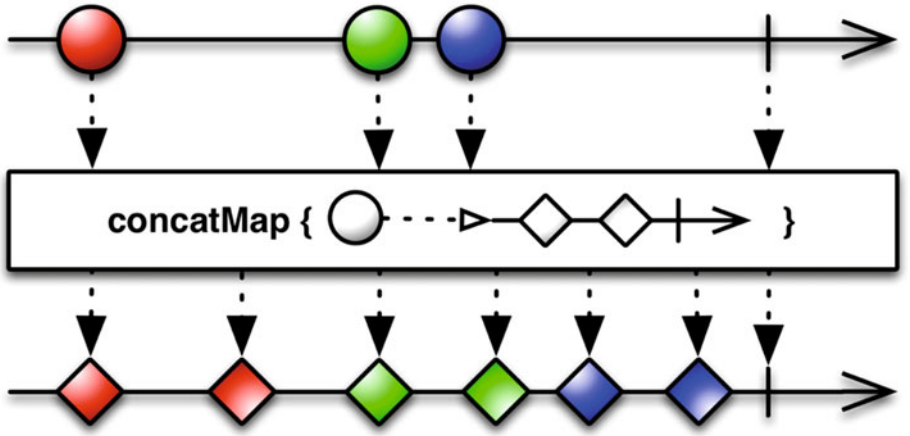

Figure 2-3. *A marble diagram of the concatMap operator*

zip

The `zip` operator (Figure 2-4) takes multiple observables as inputs and combines each emission via a specified function and emits the results of this function as a new sequence.

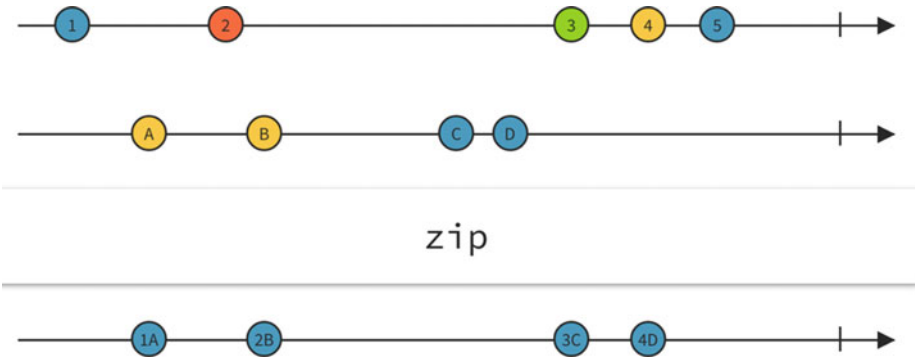

Figure 2-4. *A marble diagram of the zip operator*

The function is applied in strict sequence. If there are two sequences as input, `zip` waits for the first item emitted by the first sequence, then the first item from the second sequence, applies the function to them, and emits the result of the function. It then waits for the second item from the first sequence, the second item from the second sequence, applies the function to these two items, and emits the result as a function. And so on. It will stop when the shortest sequence stops.

Here's an example:

```
Observable<Integer> rangeMajor = Observable.range(1, 3);
```

```
Observable<Integer> rangeMinor = Observable.range(5, 10);

Observable.zip(rangeMajor, rangeMinor,
    new Func2<Integer, Integer, String>() {
          @Override
          // order of parameters here is the same order
          // of zip parameters
          public String call(Integer major, Integer minor) {
              return major + "." + minor;
          }
      }).subscribe(new Subscriber<String>() {
          @Override
          public void onCompleted() {
              System.out.println("sequence completed!");
          }

          @Override
          public void onError(Throwable e) {

          }

          @Override
          public void onNext(String s) {
              System.out.println("next item is: " + s);
          }
      });
```

In this example, the `zip` operator takes two sequences as inputs and a function that simply takes two integers and builds a string.

The output of this code is

```
next item is: 1.5
next item is: 2.6
next item is: 3.7
sequence completed!
```

As you can see, the emitted sequence stops when the shortest sequence stops, and in this case the shortest sequence is the first one (1,2,3).

concat

The `concat` operator (Figure 2-5) concatenates two or more emissions, generating one emission where all the items from the first source emission appear before the items of the second source emission. Also, the `concat` operator waits for each sequence to be completed before subscribing to the next observable.

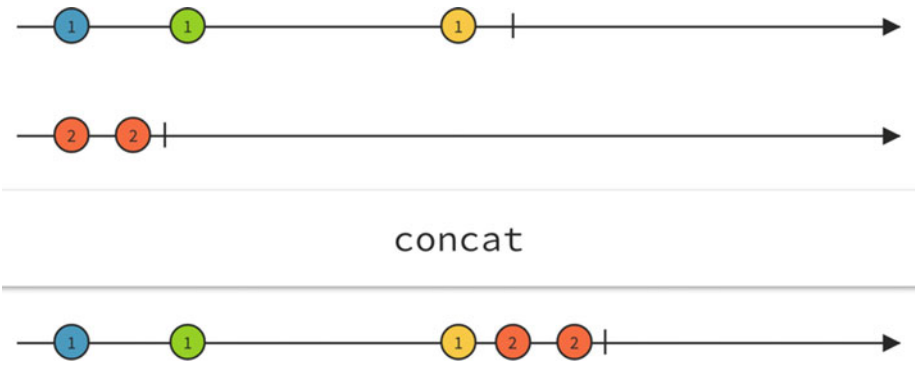

Figure 2-5. *A marble diagram of the concat operator*

In the following example, you concatenate two sequences of strings:

```
Observable<String> first =
  Observable.just("one", "two");

Observable<String> second =
  Observable.just("three", "four", "five");

Observable.concat(first, second)
  .subscribe(new Subscriber<String>() {
      @Override
      public void onCompleted() {
          System.out.println("sequence completed!");
      }

      @Override
      public void onError(Throwable e) {

      }

      @Override
      public void onNext(String s) {
          System.out.println("next item is: " + s);
      }
  });
```

The output is

```
next item is: one
next item is: two
next item is: three
next item is: four
next item is: five
sequence completed!
```

Note that you can only concatenate sequences of objects of the same type (i.e., you cannot concatenate an observable that emits strings with one that emits integers).

Do you remember the definition of hot and cold observables?

What would happen if you concatenate a cold observable (first) with a hot one (second)? The `concat` operator will emit all items from the first observable, and then it will subscribe to the second (the hot observable) and will start to emit all items from the hot observable, losing all of the items that the hot observable emitted before the subscription.

filter

The `filter` operator uses a specified function to allow only some items of the source sequence to be emitted.

Here is the translation of the diagram shown in Figure 2-6 into Java code:

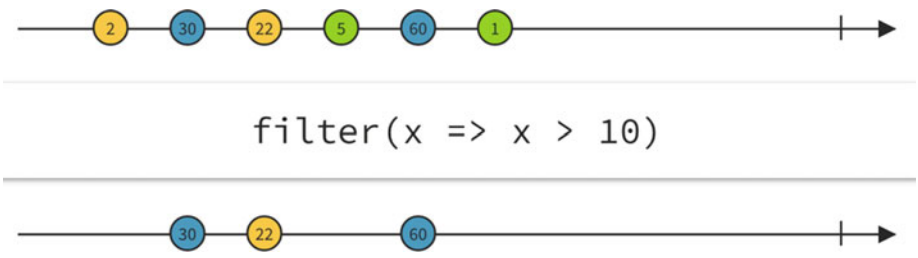

Figure 2-6. *A marble diagram of the filter operator*

```
Observable.from(new Integer[]{2, 30, 22, 5, 60, 1})
                .filter(new Func1<Integer, Boolean>() {
                    @Override
                    public Boolean call(Integer x) {
                        return x > 10;
                    }
                }).subscribe(new Subscriber<Integer>() {
            @Override
            public void onCompleted() {
                System.out.println("sequence completed!");
            }

            @Override
            public void onError(Throwable e) {

            }

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

And the output of this code is

```
next item is: 30
next item is: 22
next item is: 60
sequence completed!
```

distinct

The `distinct` operator (Figure 2-7) applies a filter to the source sequence. If an item is emitted more than once, only the first occurrence will be emitted.

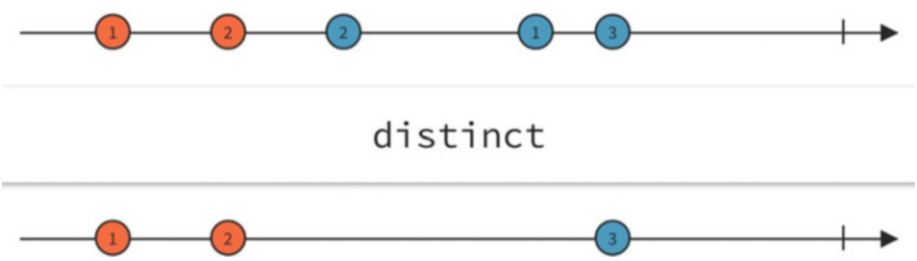

Figure 2-7. *A marble diagram of the distinct operator*

first

The `first` operator (Figure 2-8) emits only the first item of a sequence. If a function is specified, it will be used to filter the items, so only the first item of the sequence that meets the conditions will be emitted.

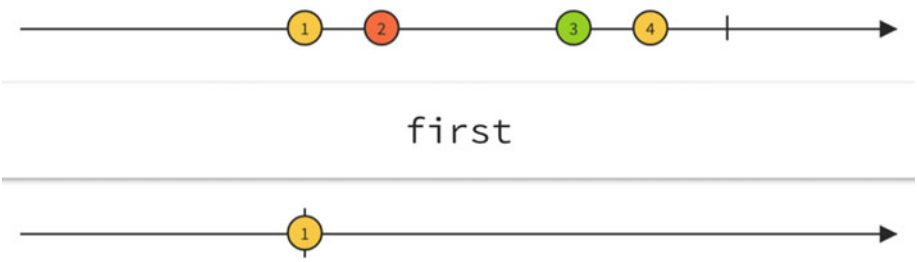

Figure 2-8. *A marble diagram of the first operator*

You can take the example written for the `filter` operator and change it by applying the operator `first` instead of `filter`:

```
Observable.from(new Integer[]{2, 30, 22, 5, 60, 1})
```

```
            .first(new Func1<Integer, Boolean>() {
                @Override
                public Boolean call(Integer x) {
                    return x > 10;
                }
            }).subscribe(new Subscriber<Integer>() {
        @Override
        public void onCompleted() {
            System.out.println("sequence completed!");
        }

        @Override
        public void onError(Throwable e) {

        }

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

As you may expect, the output will be similar to the output of the `filter` operator, but limited to one item:

```
next item is: 30
sequence completed!
```

last

If you can apply a filter to the beginning of the sequence with the operator `first`, you can also filter the end of the sequence with the operator `last` (Figure 2-9).

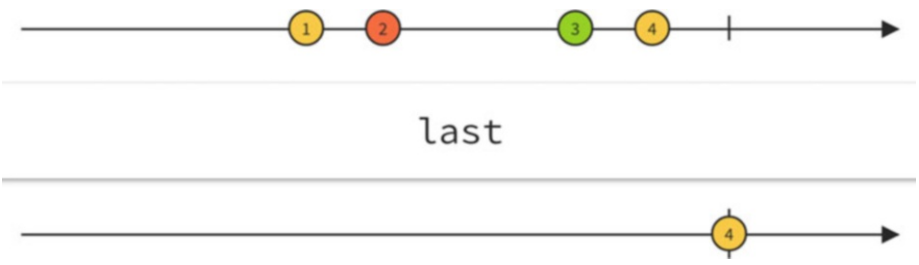

Figure 2-9. *A marble diagram of the last operator*

```
Observable.just("first", "second", "third")
  .last()
```

```
  .subscribe(new Subscriber<Integer>() {
    @Override
    public void onCompleted() {
      System.out.println("sequence completed!");
    }

    @Override
    public void onError(Throwable error) {

    }

    @Override
    public void onNext(String item) {
      System.out.println("next item is: " + item);
    }
  });
```

The output of this code is

```
next item is: third
sequence completed!
```

The last operator can take a predicate as a parameter, and only the last item from the source sequence that evaluates that predicate to true is emitted.

```
  Observable.just("first", "second", "third")
  .last(new Func1<String, Boolean>() {
    @Override
    public Boolean call(String t) {
        return t.startsWith("s");
    }
  })
  .subscribe(new Subscriber<String>() {
    @Override
    public void onCompleted() {
      System.out.println("sequence completed!");
    }

    @Override
    public void onError(Throwable error) {

    }

    @Override
    public void onNext(String item) {
      System.out.println("next item is: " + item);
    }
  });
```

The output of this code is

```
next item is: second
sequence completed!
```

take

The operator `first` is good for filtering the beginning of a sequence, but what if you are interested not only in the first item but you want the first n items? Here's where the `take` operator (Figure 2-10) comes in handy. It takes an integer `n` as a parameter, allowing only the first n items to be emitted.

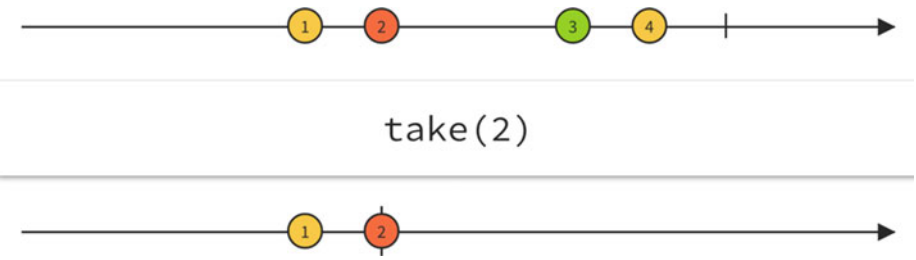

Figure 2-10. *A marble diagram of the take operator*

```
Observable.just("first", "second", "third")
  .take(2)
  .subscribe(new Subscriber<String>() {
    @Override
    public void onCompleted() {
      System.out.println("sequence completed!");
    }

    @Override
    public void onError(Throwable error) {

    }

    @Override
    public void onNext(String item) {
      System.out.println("next item is: " + item);
    }
});
```

The output of this code is

```
next item is: first
next item is: second
sequence completed!
```

Another version of this operator accepts a timeout as input (`take(long, TimeUnit)`), emitting only the items that are emitted by the source sequence before this timeout.

■ **Note** One may think that the `first()` and `take(1)` operators should have the same behavior. They emit the same output sequence if applied to the same input sequence, but there's a difference: `take(1)` emits once or nothing at all, and `first()` emits once or crashes if the source sequence is empty.

---

startWith

The operator `startWith` (Figure 2-11) takes the input sequence and adds a given item to it. It can be useful if you want to force your sequence to begin with a default value, or with a cached one.

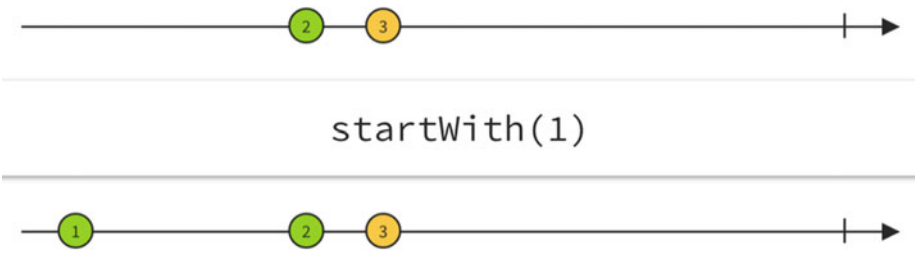

Figure 2-11. *A marble diagram of the startWith operator*

```
Observable.just("first", "second", "third")
  .startWith("zero")
  .subscribe(new Subscriber<String>() {
    @Override
    public void onCompleted() {
      System.out.println("sequence completed!");
    }

    @Override
    public void onError(Throwable error) {

    }

    @Override
    public void onNext(String item) {
```

```
      System.out.println("next item is: " + item);
    }
  });
```

The output of this code is

```
next item is: zero
next item is: first
next item is: second
next item is: third
sequence completed!
```

scan

The operator scan takes one sequence and applies a function to each pair of sequentially emitted items.

```
Observable<Integer> sourceObservable = Observable.range(1, 5);
Observable<Integer> scanObservable = sourceObservable
                .scan(new Func2<Integer, Integer, Integer>() {
                    @Override
                    public Integer call(Integer i1, Integer i2) {
                        return i1 + i2;
                    }
                });

        scanObservable.subscribe(new Subscriber<Integer>() {
            @Override
            public void onCompleted() {
                System.out.println("sequence completed!");
            }

            @Override
            public void onError(Throwable e) {

            }

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

This code mimics the behavior of the diagram in Figure 2-12.

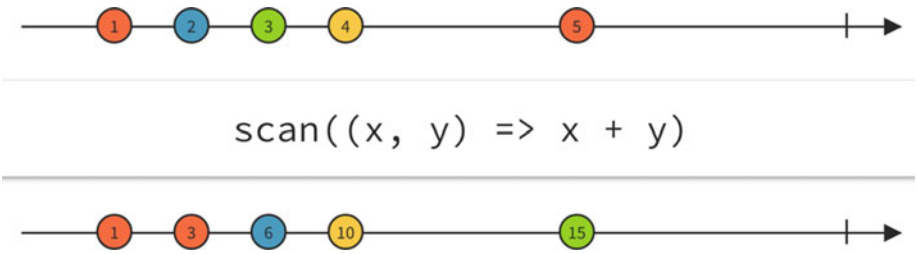

Figure 2-12. *A marble diagram of the scan operator*

After the emission of the first item from `sourceObservable`, `scanObservable` will emit the same item without any transformation. When the second item is emitted from the source, `scanObservable` will apply the function to the first and the second item, emitting the result.

The output of the code is

```
next item is: 1
next item is: 3
next item is: 6
next item is: 10
next item is: 15
sequence completed!
```

Other Operators

There are many other operators available to transform, filter, or combine observables.

Operators for Transforming Observables

- `buffer`: Periodically gathers items from an Observable into bundles and emits these bundles rather than emitting the items one at a time.
- `groupBy`: Divides an Observable into a set of observables that each emit a different group of items from the original Observable, organized by key.
- `window`: Periodically subdivides items from an Observable into Observable windows and emits these windows rather than emitting the items one at a time.

Operators for Filtering Observables

- `debounce`: Only emits an item from an Observable if a particular timespan has passed without it emitting another item.

- `elementAt`: Emits only item n emitted by an Observable.
- `ignoreElements`: Does not emit any items from an Observable but mirrors its termination notification.
- `sample`: Emits the most recent item emitted by an Observable within periodic time intervals.
- `skip`: Suppresses the first n items emitted by an Observable.
- `skipLast`: Suppresses the last n items emitted by an Observable.
- `takeLast`: Emits only the last n items emitted by an Observable.

Operators for Combining Observables

- `And/Then/When`: Combines sets of items emitted by two or more observables by means of pattern and plan intermediaries; these operators are not part of the RxJava library but can be found in the RxJavaJoins library. (`https://github.com/ReactiveX/RxJavaJoins`)
- `combineLatest`: When an item is emitted by either of two Observables, it combines the latest item emitted by each Observable via a specified function and emits items based on the results of this function.
- `join`: Combines items emitted by two Observables whenever an item from one Observable is emitted during a time window defined according to an item emitted by the other Observable.
- `merge`: Combines multiple Observables into one by merging their emissions.
- `switchOnNext`: Converts an Observable that emits Observables into a single Observable that emits the items emitted by the most-recently-emitted of those Observables.

EXAMPLE: GENERATE A SEQUENCE OF ODD INTEGERS

To create a sequence of odd numbers, you need to generate a sequence of integers and then filter the emitted items, removing even integers.

You start with a finite sequence of integers, and this sequence can be created using `just` or `range`.

Then you apply the `filter` operator, passing a function that returns true if the emitted integer is odd.

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

         @Override
         public void onError(Throwable e) {
         }

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

EXAMPLE: FIBONACCI SEQUENCE

You want to create a method that takes an integer `n` and outputs the Fibonacci sequence `F(n)`.

First, you create an observable that emits a sequence of `n` integers: `Observable.range(n)` is the perfect candidate for this role. Second, you apply a function on each of these items, and you use the `map` operator to do so.

Here's the code:

```
public static void rxFibonacci(int n) {
  final int[] tmp = {0, 0};

  Observable.range(1, n)
              .map(new Func1<Integer, Integer>() {
                  @Override
                  public Integer call(Integer x) {
                      if (x < 3) {
                          tmp[0] = 1;
                          tmp[1] = 1;
                          return 1;
```

```
                    } else {
                        int item = tmp[0] + tmp[1];
                        tmp[0] = tmp[1];
                        tmp[1] = item;
                        return item;
                    }
                }
            })
            .subscribe(new Subscriber<Integer>() {
                @Override
                public void onCompleted() {
                    System.out.println("sequence completed!");
                }

                @Override
                public void onError(Throwable e) {

                }

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

`Observable.range(1,n)` will emit a sequence of `n` integers starting from 1. Then, for each emitted item, a function is applied through the `map` operator. The function uses a temporary array to store the results of `F(n-1)` and `F(n-2)` and uses this values to evaluate `F(n)`.

For example, calling `rxFibonacci(10)` produces the following output:

```
next item: 1
next item: 1
next item: 2
next item: 3
next item: 5
next item: 8
next item: 13
next item: 21
next item: 34
next item: 55
```

sequence completed!

CHAPTER 3

Subscription Lifecycle

Introduction

Working with observables is not only about subscribing and receiving items.

In previous chapters, you saw that sequences can also terminate with an error event, but what can you do to handle an error event? And what about threads? Can an Observable operate and notify on different threads?

In this chapter, you'll learn

- The tools that RxJava provides to manage error events
- How to use schedulers to operate on different threads
- How to deal with backpressure

Error Handling

As mentioned, an observable's emission can end with either a completed event or error event, but not both.

If an error occurs at any time during the emission, the sequence is stopped and the `onError()` method is called.

But what do we mean by "error?"

- An error is any exception than is thrown during the emission of the sequence.
- It can be thrown by the source observable or by any operator applied to the source observable.
- The first exception that is thrown causes the sequence to be stopped, so no other exception can be thrown.
- The exception is propagated from the observable that throws it to the subscribers, so the other operators are skipped.
- Operators don’t have to handle the exception; it's all left up to the subscriber (as opposed to the callback mechanism, where you have to handle errors in each callback).

A. Maglie, *Reactive Java Programming*, DOI 10.1007/978-1-4842-1428-2_3

The observable itself does not throw the exception; it only notifies the subscriber calling the `onError()` method. If the observable fails calling the `onError()` method (i.e. an exception is thrown when calling `onError()`), the `onError()` method won't be called anymore and another exception will be thrown.

There are some exceptions that are not handled as error events but they are thrown, causing JVM to stop:

- `rx.exceptions.OnErrorNotImplementedException`
- `rx.exceptions.OnErrorFailedException`
- `rx.exceptions.OnCompletedFailedException`
- `java.lang.StackOverflowError`
- `java.lang.VirtualMachineError`
- `java.lang.ThreadDeath`
- `java.lang.LinkageError`

Typically, in Java code, you catch exceptions using the try/catch block. With observables, there's no try/catch block, but you can choose between different techniques to recover from errors.

Handling Errors in the onError() Method

The most common scenario is to handle errors in the `onError()` method. The subscriber's `onError()` method is responsible for handling the error state.

In the following example, you generate a sequence of strings in which every string should represent an integer, and then you deliberately introduce an invalid string in the middle of the sequence.

```
Observable.just("1", "2", "a", "3", "4")
        .map(new Func1<String, Integer>() {
            @Override
            public Integer call(String s) {
                return Integer.parseInt(s);
            }
        }).subscribe(new Subscriber<Integer>() {
    @Override
    public void onCompleted() {
        System.out.println("sequence completed!");
    }

    @Override
    public void onError(Throwable e) {
        System.err.println("error! " + e.toString());
    }

    @Override
```

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

The first two emitted items (string "1" and "2") are correctly parsed as integers. When item "a" is emitted, the function inside the map operator fails, throwing a `java.lang.NumberFormatException`, and this exception is sent to the subscriber in the `onError()` method. The `NumberFormatException` is passed to the `onError()` method as a parameter.

The output is

```
next item is: 1
next item is: 2
error! java.lang.NumberFormatException: For input string: "a"
```

Ignoring the Exception and Continuing with Item Emission

Continuing the previous example, suppose you want the sequence to terminate normally when an invalid string is emitted. You can achieve this by applying the operator `onErrorResumeNext()` to the observable.

```
Observable.just("1", "2", "a", "3", "4")
        .map(new Func1<String, Integer>() {
            @Override
            public Integer call(String s) {
                return Integer.parseInt(s);
            }
        })
        .onErrorResumeNext(Observable.<Integer>empty())
        .subscribe(new Subscriber<Integer>() {
    @Override
    public void onCompleted() {
        System.out.println("sequence completed!");
    }

    @Override
    public void onError(Throwable e) {
        System.err.println("error! " + e.toString());
    }

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

The output is

```
next item is: 1
next item is: 2
sequence completed!
```

The `onErrorResumeNext()` operator tells the observable to avoid notifying the `onError` method when exception is thrown and to continue by emitting the sequence that is specified as a parameter of the operator. In this case, when an error event occurs, the original sequence is stopped and the emission continues with the sequence emitted by `Observable.empty()`, which is an empty sequence. In other words, in the code above, if an error occurs, the sequence is stopped and no error event is notified.

Two other similar operators are

- `onErrorReturn()`, which makes the observable emit a specified item when an error occurs; the item emitted is the result of the function passed as parameter.
- `onExceptionResumeNext()`, which makes the observable continuing emitting items from a specified observable when an exception (and not any other throwable) is thrown.

Continuing the example above, suppose you want to emit the integer -1 when the error event occurs. For this, you can use `onErrorReturn()`:

```
  Observable.just("1", "2", "a", "3", "4")
    .map(new Func1<String, Integer>() {
        @Override
        public Integer call(String s) {
                return Integer.parseInt(s);
        }
    })
    .onErrorReturn(new Func1<Throwable, Integer>() {
        @Override
        public Integer call(Throwable t) {
                return -1;
        }
    })
    .subscribe(new Subscriber<Integer>() {
        @Override
        public void onCompleted() {
          System.out.println("sequence completed!");
        }

        @Override
        public void onError(Throwable e) {
          System.err.println("error! " + e.toString());
        }
```

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

The output is

```
next item is: 1
next item is: 2
next item is: -1
sequence completed!
```

You can get the same effect using onExceptionResumeNext:

```
  Observable.just("1", "2", "a", "3", "4")
    .map(new Func1<String, Integer>() {
        @Override
        public Integer call(String s) {
                return Integer.parseInt(s);
        }
    })
    .onExceptionResumeNext(Observable.just(-1))
    .subscribe(new Subscriber<Integer>() {
        @Override
        public void onCompleted() {
          System.out.println("sequence completed!");
        }

        @Override
        public void onError(Throwable e) {
          System.err.println("error! " + e.toString());
        }

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

The output is the same:

```
next item is: 1
next item is: 2
next item is: -1
sequence completed!
```

Retry

The RxJava library also contains two operators that let you recover from error implementing with a retry mechanism.

- The `Observable.retry(...)` operator does not notify the error and resubscribes to the source observable.
- The `Observable.retryWhen(...)` operator passes the error event to another observable that becomes responsible for determining whether to resubscribe to the source observable.

Let's apply the `retry()` operator to the previous examples instead of `onErrorResumeNext()`:

```
Observable.just("1", "2", "a", "3", "4")
        .map(new Func1<String, Integer>() {
            @Override
            public Integer call(String s) {
                return Integer.parseInt(s);
            }
        })
        .retry()
        .subscribe(new Subscriber<Integer>() {
    @Override
    public void onCompleted() {
        System.out.println("sequence completed!");
    }

    @Override
    public void onError(Throwable e) {
        System.err.println("error! " + e.toString());
    }

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

Whenever an error occurs, `retry()` will resubscribe to the source, restarting with the emission and encountering the error again and again. As you may have guessed, this code will run forever, emitting the following output:

```
...
next item is: 1
next item is: 2
next item is: 1
next item is: 2
next item is: 1
...
```

You can use `retry(1)` instead of `retry()`, so the retry mechanism will be executed only one time (the argument passed in determines the number of retries):

```
Observable.just("1", "2", "a", "3", "4")
        .map(new Func1<String, Integer>() {
            @Override
            public Integer call(String s) {
                return Integer.parseInt(s);
            }
        })
        .retry(1)
        .subscribe(new Subscriber<Integer>() {
    @Override
    public void onCompleted() {
        System.out.println("sequence completed!");
    }

    @Override
    public void onError(Throwable e) {
        System.err.println("error! " + e.toString());
    }

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

In this case, the emission will be retried just once; after that, if the error condition persists, the error event will be propagated. The output is

```
next item is: 1
next item is: 2
next item is: 1
next item is: 2
error! java.lang.NumberFormatException: For input string: "a"
```

If you want to retry one time after 5 seconds, you can use `retryWhen()`:

```
.retryWhen(new Func1<Observable<? extends Throwable>, Observable<?>>() {
    @Override
    public Observable<?> call(Observable<? extends Throwable> observable) {
        return Observable.timer(5, TimeUnit.SECONDS);
    }
})
```

After the specified timeout (5 seconds in this example), the sequence will be retried. If an error occurs during the retry, the error event will not be notified. Instead the complete event will be notified. The output is

```
next item is: 1
next item is: 2
next item is: 1
next item is: 2
sequence completed!
```

In the following more complex example, you combine different operators to get three retries, one every 5 seconds:

```
.retryWhen(new Func1<Observable<? extends Throwable>, Observable<?>>() {
    @Override
    public Observable<?> call(Observable<? extends Throwable> observable) {
        return observable.zipWith(Observable.range(1, 3),
          new Func2<Throwable, Integer, Integer>() {
            @Override
            public Integer call(Throwable throwable,
              Integer retryCount) {
                System.out.println("retry #" + retryCount);
                return retryCount;
            }
          }).flatMap(new Func1<Integer, Observable<?>>() {
            @Override
            public Observable<?> call(Integer integer) {
                return Observable.timer(5, TimeUnit.SECONDS);
            }
        });
    }
})
```

The idea here is to combine a sequence of three items (because you want three retries) with a delay of 5 seconds for each retry. The sequence of three items is generated by `Observable.range(1,3)` (also `Observable.just(1,2,3)` would work). For the delay, you still use `Observable.timer(5, TimeUnit.SECONDS)`. For every integer emitted, a timer is launched, and this timer triggers the retry mechanism. In addition, you print the current retry count.

The `Observable.zipWith()` operator is applied to the source observable and takes two parameters:

- An observable (`Observable.range(1, 3)` in this example) that is the observable that you want to zip with the source observable (as with the `Observable.zip()` operator).
- An instance of `Func2<Throwable, Integer, Integer>`. A `Func2` is required because you have two inputs: the Throwable emitted by the `retryWhen` operator and an Integer emitted by the `Observable.range(1,3)`.

Then you use `flatMap` to transform the items emitted by the `zipWith` operator into a 5-second delay. The resulting observable is used by `retryWhen` to apply the retry logic to the source observable.

If you apply this retry mechanism to the previous example, you get the following output:

```
next item is: 1
next item is: 2
retry #1
next item is: 1
next item is: 2
retry #2
next item is: 1
next item is: 2
retry #3
next item is: 1
next item is: 2
sequence completed!
```

Schedulers

By default the chain of operations of an observable is executed in the same thread on which the subscribe method is called. You can change this behavior using two operators:

- `Observable.subscribeOn(...)` lets you specify a scheduler on which the observable operates.
- `Observable.observeOn(...)` lets you specify a scheduler on which the observers will be notified.

This means you can use a thread for the executions of all the chains of operations and a different thread to receive the `onNext()`, `onCompleted()`, and `onError()` notifications. This is often used when your observable executes I/O operations (like network requests or reading/writing on disk):

```
myNetworkObservable()
.subscribeOn(<background thread>)
.observeOn(<ui thread>)
.subscribe(...)
```

The two operators can be called from any point in the chain of operators, but their behavior is different:

- `subscribeOn()` changes the thread on which the observable operates, regardless of where in the chain it is applied.
- `observeOn()` changes the thread that Observable will use, starting from the point at which it is applied.

Usually you don't have to create or specify a thread directly; instead, you can choose among the schedulers that are built in the RxJava library:

- `Schedulers.immediate()` schedules work to begin immediately in the current thread.
- `Schedulers.computation()` can be used for computational work, and by default it allocates as many threads as the number of processors. It's not meant for I/O operations.
- `Schedulers.io()` is meant for I/O operations.
- `Schedulers.newThread()` creates a new thread for each operation.
- `Schedulers.trampoline()` keeps a queue of operations, and every operation begins on the current thread after all previous operations have finished.
- `Schedulers.from(java.util.concurrent.Executor)` uses the specified `Executor` as Scheduler.

You have implicitly used schedulers before: `Observable.timer()` operates with the `Scheduler.computation()` scheduler, which will operate on a different thread. There are some other operators that have a particular default scheduler: `buffer`, `debounce`, `delay`, `delaySubscription`, `interval`, `repeat`, `replay`, `retry`, `sample`, `skip`, `skipLast`, `take`, `takeLast`, `takeLastBuffer`, `throttleFirst`, `throttleLast`, `throttleWithTimeout`, `timeInterval`, `timeout`, `timer`, `timestamp`, `window`.

EXAMPLE: MULTITHREADING FIBONACCI

Take a look at the Fibonacci Sequence example from Chapter 2. How can you modify it to add multithreading? You just need one line of code!

```
final int[] tmp = {0, 0};


Observable.range(1, n)
  .map(new Func1<Integer, Integer>() {
    @Override
    public Integer call(Integer x) {
      if (x < 3) {
      tmp[0] = 1;
      tmp[1] = 1;
      return 1;
    } else {
      int item = tmp[0] + tmp[1];
      tmp[0] = tmp[1];
      tmp[1] = item;
      return item;
    }
  }
```

```
})
.subscribeOn(Schedulers.computation())
.subscribe(new Subscriber<Integer>() {
  @Override
  public void onCompleted() {
    System.out.println("sequence completed!");
  }

  @Override
  public void onError(Throwable e) {

  }

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

You only applied the `subscribeOn()` operator, passing the scheduler `Schedulers.computation()` as a parameter, and your code is now multithreading! Remember that the `subscribeOn()` operator can be applied at any point in the chain, but the preferred place is right before the call to the `subscribe()` method. It makes the code more readable; if you have a long chain of operators, you don't want your `subscribeOn()` call to be lost in the chain.

EXAMPLE: NETWORK CALLS

A typical use case in which you have to specify both the `subscribeOn()` and `observeOn()` operators is when you're developing an Android app and you have to update the UI after getting data from a remote server.

```
Observable<MyResponseObject> networkCall = ...

networkCall.subscribeOn(Schedulers.io())
  .observeOn(AndroidSchedulers.mainThread())
  .subscribe(new Subscriber<MyResponseObject>() {
    @Override
    public void onCompleted() {

    }

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

```
        // here we can update the UI showing an error message
      }

      @Override
      public void onNext(MyResponseObject item) {
        // here we can update the UI reading item data
      }
    });
```

In this example, you applied

- The `subscribeOn()` operator because otherwise the UI (the main thread in Android) will hang until the operation is complete (and an ANR - Application Not Responding error will be shown by the operating system). The `Schedulers.io()` scheduler is a good choice because you're dealing with a network operation, which is an I/O operation.
- The `observeOn()` operator because you cannot update the UI from a thread that is different from the UI thread; the `AndroidSchedulers.mainThread()` that you've used here is a special scheduler that is not included in the RxJava library but is implemented in the RxAndroid library (a sort of add-on to RxJava for Android development; see https://github.com/ReactiveX/RxAndroid) which will return the UI thread. If you don't apply the `observeOn()` operator, your app will crash when trying to change the UI items (like updating a `TextView`).

EXAMPLE: INSERTING DATA INTO A DATABASE

Suppose that you want to store some data in a database as soon as a user changes the value of an input form, without having to wait for a click of a button and without blocking the UI thread.

Given an observable that emits an item (the content of the form field) whenever the input changes, you can use schedulers to execute the insertion in a separate thread.

```
Observable<String> inputObservable = ...


inputObservable.flatMap(x -> validate(x))
  .observeOn(Schedulers.io())
  .subscribe(new Subscriber<String>() {
      @Override
      public void onCompleted() {
```

```
        }

        @Override
        public void onError(Throwable e) {

        }

        @Override
        public void onNext(String item) {
          db.insert(item);
        }
    });
```

Here you apply the `observeOn()` operator so that the `onNext` method will be notified on the io scheduler, which is the right one for I/O operations such as writing on a db.

Transformers

As mentioned, the `subscribeOn()`/`observeOn()` pair can be used many times, forcing you to copy/paste the two lines of code necessary to apply these two operators. Copy/paste leads to duplicated code, and it's is not a good practice. So, how can you create an operator that automatically applies these two operators?

You can do it using `Observable.Transformer<T,R>` (`T` is the type of input Observable and `R` is the type of output Observable). Instances of `Transformer` are functions that are applied to an observable using `Observable.compose()`, an operator that gives you the ability to modify the source Observable, not only the items of the sequence.

```
public static <T> Observable.Transformer<T, T> mySchedulers() {
  return new Observable.Transformer<T, T>() {
    @Override
    public Observable<T> call(Observable<T> observable) {
      return observable
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread());
    }
  };
}
```

When you implement `Transformer<T,R>`, you receive in input an `Observable<T>` and you must return an `Observable<R>`. In the example above, `T` and `R` are the same object because your `Transformer` does not modify the items of the emitted sequence.

Now you can rewrite the previous example as

```
  Observable<MyResponseObject> networkCall = ...
```

```
networkCall
  .compose(mySchedulers())
  .subscribe(new Subscriber<MyResponseObject>() {
    @Override
    public void onCompleted() {

    }

    @Override
    public void onError(Throwable e) {
      // here we can update the UI showing an error message
    }

    @Override
    public void onNext(MyResponseObject item) {
    // here we can update the UI reading item data
  }
});
```

Advanced Use of Schedulers

You can also use schedulers without applying operators in a chain of observables. For example, if you want to execute some operations in a dedicated thread, you can use the rx.Scheduler.Worker class, like so:

```
Worker worker = Schedulers.newThread().createWorker();
worker.schedule(new Action0() {
  @Override
  public void call() {
    // do your work here
  }
});
```

The Worker class implements the Subscription interface, so you can call methods unsubscribe() and isUnsubscribed() on it.

Action0 is an interface that defines just one method, call(), which accepts no parameters and returns void.

Worker also implements two methods to execute actions after a certain amount of time or periodically:

- Worker.schedule(Action0 action, long delayTime, TimeUnit unit) executes an action after the specified delay.
- Worker.schedulePeriodically(Action0 action, long initialDelay, long period, TimeUnit unit) executes an action periodically, optionally starting after a delay.

Backpressure

Until now we have considered only situations in which the emission of items is slower than the operations executed on each of the items. So, for every emission, there's enough time for the subscriber to consume the emitted item before the next one is emitted.

There are also situations in which the observable emits items too fast for the subscriber, and the subscriber cannot continue the single item before receiving the next one.

For example, take two observables, A and B, where A emits items twice as fast as B and creates an `Observable.zip` of A and B. The resulting observable combines the n-th item from A and n-th item from B, but meanwhile B has also emitted items n+1 to n+m, so it should keep an always increasing buffer of the emitted items.

Another typical case is when inside the `onNext` method the UI is updated with data from the emitted item, but items are emitted at a frequency higher than the frequency with which the UI is updated.

Backpressure happens every time we face such situations. We can handle backpressure in the following ways:

- By controlling the emission, applying some specific operator to the observable
- At consumption time in the subscriber

Handling Backpressure During Emission: Throttling

You can use some RxJava operators to adjust the rate at which the Observable emits items.

sample

With `Observable.sample()` you can periodically take samples of the sequence and emit the most recent item of each sample. Figure 3-1 shows this process.

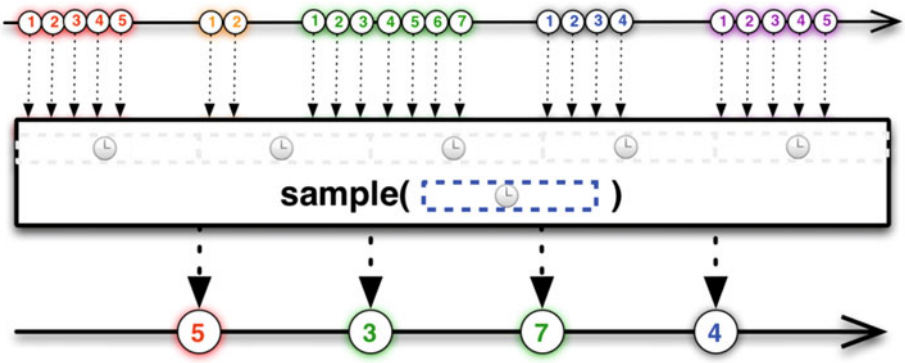

Figure 3-1. *The marble diagram of Observable.sample()*

Looking at the diagram in Figure 3-1, you can see that at the end of each period only the last emitted item (from the source observable) is emitted by the resulting observable, and the other items are discarded.

throttleFirst

`Observable.throttleFirst()` behaves like `sample`, but it emits the first item of each sample (Figure 3-2).

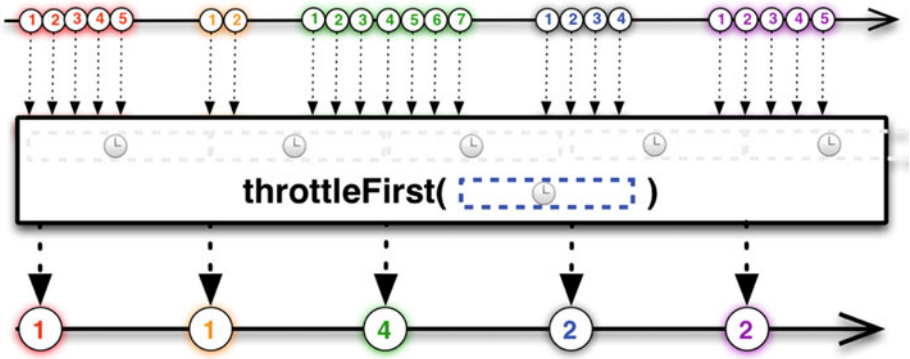

Figure 3-2. *The marble diagram of Observable.throttleFirst()*

debounce

`Observable.debounce()` emits only items that are not followed by any other item within a specified interval, as shown in Figure 3-3.

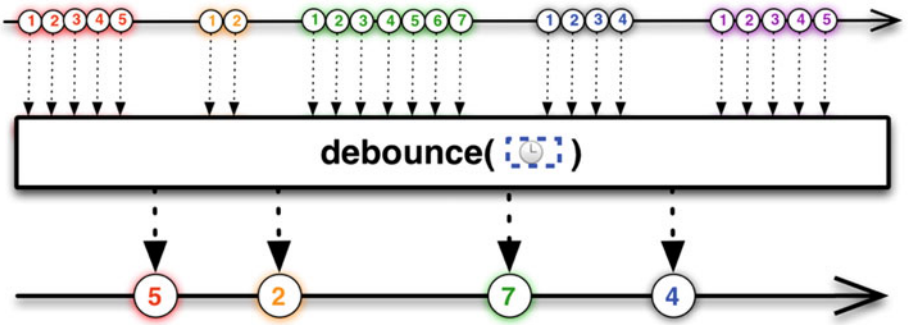

Figure 3-3. *The marble diagram of Observable.debounce()*

Handling Backpressure During Emission: Buffering

Another way to control backpressure is to collect items into buffers before notifying the subscriber. The subscriber's `onNext()` method will not receive the single emitted item but a collection that contains the items collected in the buffer.

To apply this technique, you can use the operators `buffer()` and `window()`.

Buffer

`Observable.buffer()` lets you collect items periodically at a specified interval of time.

You can choose to create buffers based on the number of items or on a time interval by passing a function that computes the necessary conditions to close each buffer (Figure 3-4). You can also specify an Observable that triggers the beginning of a buffer.

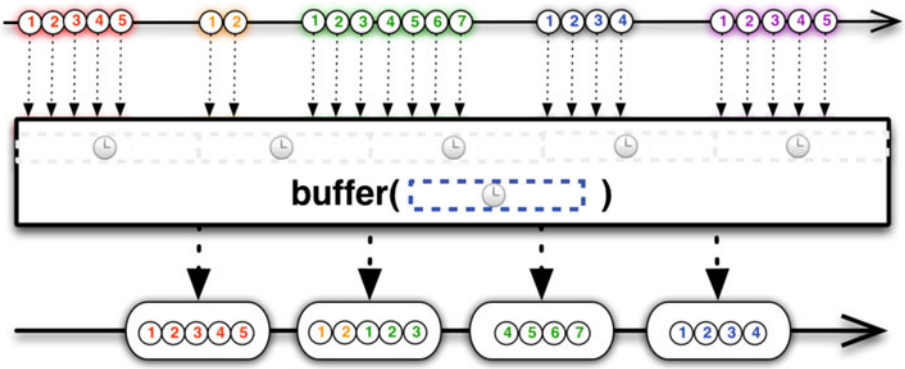

Figure 3-4. *The marble diagram of Observable.buffer()*

Window

`Observable.window()` is similar to `buffer`, but rather than emitting collections of items from the source Observable, it emits observables, and each one of these observables emits a subset of items from the source Observable and then terminates with an `onCompleted` notification.

You can choose to create "windows" based on a time interval (Figure 3-5), the number of items (Figure 3-6), or by passing a function that computes the necessary conditions to close each window.

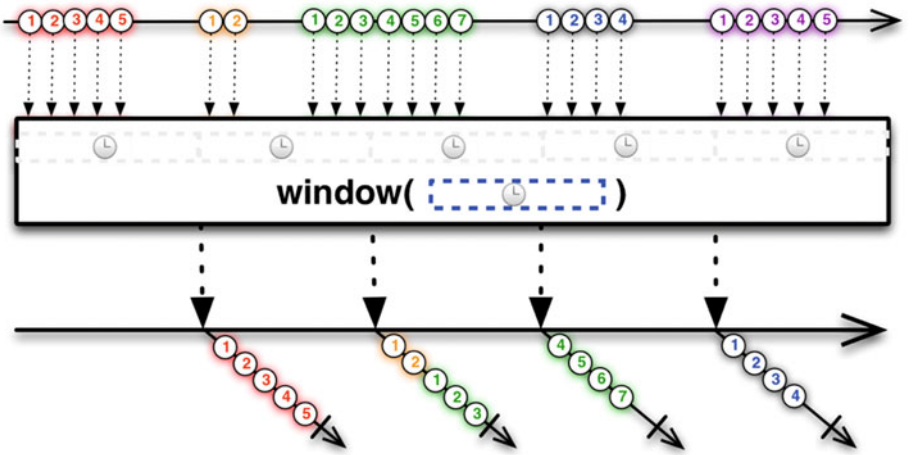

Figure 3-5. *The marble diagram of Observable. window() based on a time interval*

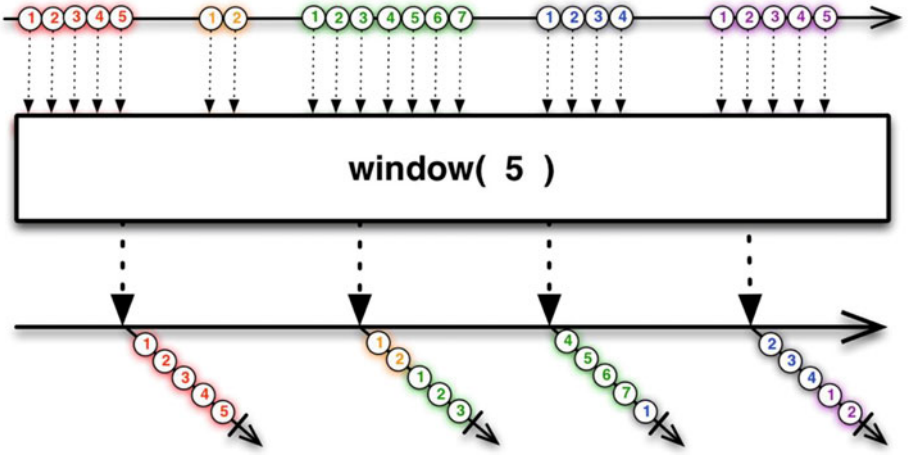

Figure 3-6. *The marble diagram of Observable. window() based on number of items*

■ **Note** If an error occurs, both operators will notify the `onError` method and stop building the current buffer or window. The items contained in the unfinished buffer/window will not be forwarded to the subscriber.

Handling Backpressure Inside the Subscriber

Inside the subscriber you can ask the observable to slow down the emission. So, in the zip example above, you can ask B to emit items slowly instead of applying a throttling operator.

Inside the subscriber's `onStart()` method you must call the `Subscriber.request(n)` method, where n is an integer representing the maximum of items you want the Observable to emit before the next `request()` call. Then, after consuming the item in the `onNext()` method, you have to call `request()` again to tell the observable to continue with the emission of the following n items.

In the following example, you set n = 1, so you can consume each item before the next one is emitted:

```
myObservable.subscribe(new Subscriber<T>() {
  @Override
  public void onStart() {
    request(1);
  }

  @Override
  public void onCompleted() {
  }

  @Override
  public void onError(Throwable e) {
  }

  @Override
  public void onNext(T item) {
    consume(item)
    request(1);
  }
});
```

Here you called the `request()` method in `onStart()` and `onNext()` with the same parameter of 1, because you supposed that you can consume only one item before the next one is emitted, but these values are independent and can be tuned based on the use case.

If you omit the `request()` call, the observable emits items as it would do normally, without slowing down the emission; the same effect can be achieved calling `request(0)`.

CHAPTER 4

Subjects

In previous chapters, you learned about `Observable<T>`, `Observer<T>`, and `Subscriber<T>` and their role in reactive programming with RxJava: an Observable object emits a sequence of events, and a Subscriber object acts as an observer, reacting to each event emitted by the Observable.

The RxJava library provides a proxy that acts both as Observable and Observer: it's called `Subject<T,R>`, where `T` is the type of the input value and `R` is the type of the output value.

A Subject can

- Subscribe to one or more Observables (as a Subscriber)
- Pass through the items it observes by reemitting them
- Emit new items

A subject does not take a scheduler but rather assumes that all serialization and grammatical correctness are handled by the caller of the subject.

Let's see how a Subject can act as Observable (`PublishSubject` will be introduced later):

```
Subject<String, String> subject = PublishSubject.create();
subject.subscribe(new Subscriber<String>() {

      @Override
      public void onCompleted() {
              System.out.println("sequence completed");
      }

      @Override
      public void onError(Throwable e) {

      }

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

});
```


A. Maglie, *Reactive Java Programming*, DOI 10.1007/978-1-4842-1428-2_4

```
subject.onNext("first item");
subject.onNext("second item");
subject.onNext("third item");
subject.onCompleted();
```

As you may expect, if you run this code you get the following output:

```
first item
second item
third item
sequence completed
```

Now let's create a Subject that acts both as Observable and as Subscriber:

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

Subject<Long, Long> subject = PublishSubject.create();
interval.subscribe(subject);

subject.subscribe(new Subscriber<Long>() {

  @Override
  public void onCompleted() {
    System.out.println("first sequence completed");
  }

  @Override
  public void onError(Throwable e) {
  }

  @Override
  public void onNext(Long item) {
    System.out.println("first sequence, item: " + item);
  }
});

subject.subscribe(new Subscriber<Long>() {

  @Override
  public void onCompleted() {
    System.out.println("second sequence completed");
  }
```

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

    @Override
    public void onNext(Long item) {
      System.out.println("second sequence, item: " + item);
    }
  });
```

The Subject here first subscribes to the interval observable (which emits an incremental long value every second), then reemits a `Long` object every time it receive an item from the interval observable. The output of this code is

```
first sequence, item: 0
second sequence, item: 0
first sequence, item: 1
second sequence, item: 1
first sequence, item: 2
second sequence, item: 2
first sequence, item: 3
second sequence, item: 3
first sequence, item: 4
second sequence, item: 4
first sequence, item: 5
...
```

In RxJava, the class `Subject` is an abstract class. In these examples, you didn't create a concrete implementation of `Subject`; you used one of the built-in implementations available, returned by the static factory method `PublishSubject.create()`.

RxJava provides four different implementations of `Subject`.

PublishSubject

`PublishSubject<T>` (Figure 4-1) is a Subject that

- Emits all the items emitted by the source observable, starting from the moment of the subscription
- Notifies of an error event and does not emit any other item if the source Observable terminates with an error

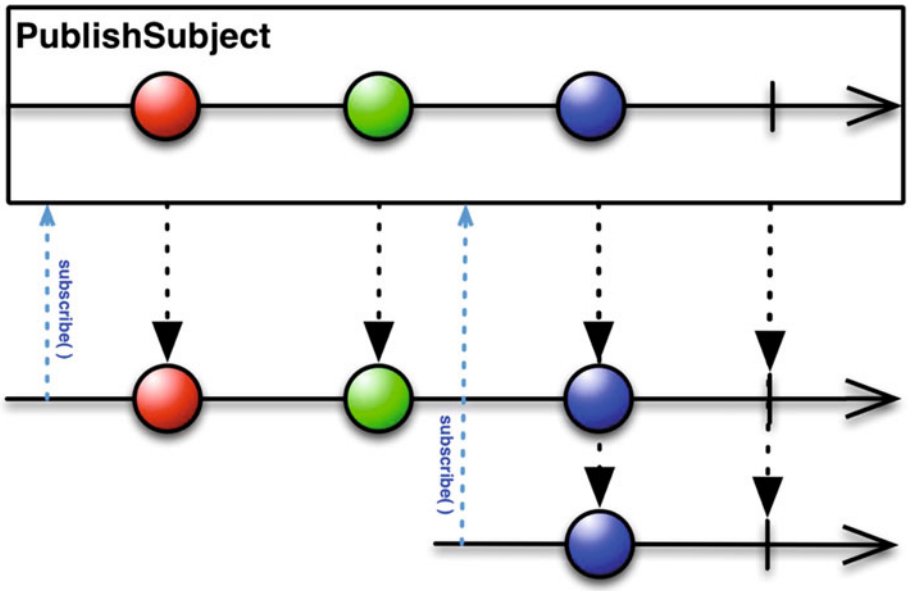

Figure 4-1. *The marble diagram for PublishSubject<T>*

To create an instance of PublishSubject (Figure 4-1), you can use the static factory method PublishSubject.create().

Now, create an instance of PublishSubject that emits a sequence of integers, and then subscribe to two different subscribers:

```
PublishSubject<Integer> subject = PublishSubject.create();
Observable<Integer> subjectAsObservable =
  subject.asObservable();

// subscribe the first Subscriber
subjectAsObservable.subscribe(new Subscriber<Integer>() {

  @Override
  public void onCompleted() {
    System.out.println("first: sequence completed");
  }

  @Override
  public void onError(Throwable e) {

  }

  @Override
  public void onNext(Integer item) {
    System.out.println("first: next item is " + item);
```

```
    }
  });

  subject.onNext(1);
  subject.onNext(2);

  // subscribe the second Subscriber
  subjectAsObservable.subscribe(new Subscriber<Integer>() {

    @Override
    public void onCompleted() {
      System.out.println("second: sequence completed");
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onNext(Integer item) {
      System.out.println("second: next item is " + item);

    }
  });

  subject.onNext(3);
  subject.onNext(4);
  subject.onNext(5);
  subject.onCompleted();
```

The output of this code is

```
first: next item is 1
first: next item is 2
first: next item is 3
second: next item is 3
first: next item is 4
second: next item is 4
first: next item is 5
second: next item is 5
first: sequence completed
second: sequence completed
```

The first subscriber subscribes before `PublishSubject` starts to emit items, so it will receive all five items and then complete. The second subscriber subscribes in the middle of the sequence, so it will receive only the subsequent items.

Are Subjects hot or cold observables? This example makes it easy to answer the question: Subjects are hot observables because they can produce items when no Observer is subscribed.

Another interesting point is the usage of the `Subject.asObservable()` method. Subscribing to *subject* instead of *subjectAsObservable* will produce the same output, but the method `Subject.asObservable()` helps you by wrapping your Subject instance in an Observable instance. This means you can expose only the Observable interface to the subscribers and it also means that no one else can use the Subject instance to emit items or notify completed/error events.

BehaviorSubject

`BehaviorSubject<T>` (Figure 4-2) is similar to `PublishSubject`, except that the subscriber will also receive the last item emitted before its subscription.

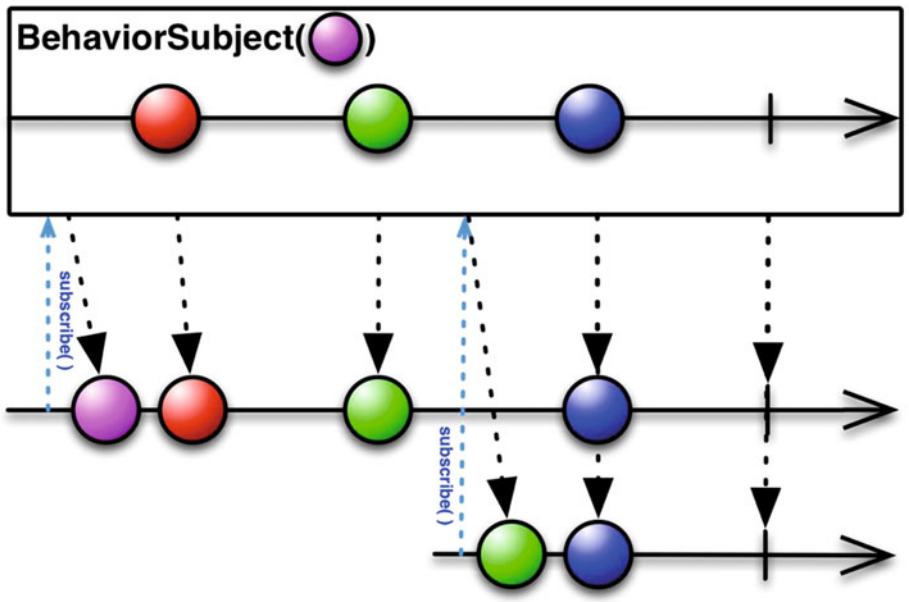

Figure 4-2. *The marble diagram for BehaviorSubject<T>*

Let's take the previous example and change the `PublishSubject` instance to a `BehaviorSubject` instance:

```
BehaviorSubject <Integer> subject = BehaviorSubject.create();
Observable<Integer> subjectAsObservable =
  subject.asObservable();
```

```
// subscribe the first Subscriber
subjectAsObservable.subscribe(new Subscriber<Integer>() {

  @Override
  public void onCompleted() {
    System.out.println("first: sequence completed");
  }

  @Override
  public void onError(Throwable e) {

  }

  @Override
  public void onNext(Integer item) {
    System.out.println("first: next item is " + item);

  }
});

subject.onNext(1);
subject.onNext(2);

// subscribe the second Subscriber
subjectAsObservable.subscribe(new Subscriber<Integer>() {

  @Override
  public void onCompleted() {
    System.out.println("second: sequence completed");
  }

  @Override
  public void onError(Throwable e) {

  }

  @Override
  public void onNext(Integer item) {
    System.out.println("second: next item is " + item);

  }
});

subject.onNext(3);
subject.onNext(4);
subject.onNext(5);
subject.onCompleted();
```

This code produces the following output:

```
first: next item is 1
first: next item is 2
second: next item is 2
first: next item is 3
second: next item is 3
first: next item is 4
second: next item is 4
first: next item is 5
second: next item is 5
first: sequence completed
second: sequence completed
```

Here the second subscriber receives the sequence starting from item 2, and it subscribes after the emission of item 2.

RxJava also provides the static factory method `BehaviorSubject.create(T defaultValue)`; the instance returned by this method will emit the provided default item as long as no item has been received from the source observable.

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

Observable<Integer> subjectAsObservable =
  subject.asObservable();

// subscribe the first Subscriber
subjectAsObservable.subscribe(new Subscriber<Integer>() {

  @Override
  public void onCompleted() {
    System.out.println("first: sequence completed");
  }

  @Override
  public void onError(Throwable e) {

  }

  @Override
  public void onNext(Integer item) {
    System.out.println("first: next item is " + item);

  }
});

subject.onNext(1);
subject.onNext(2);
```

```
// subscribe the second Subscriber
subjectAsObservable.subscribe(new Subscriber<Integer>() {

  @Override
  public void onCompleted() {
    System.out.println("second: sequence completed");
  }

  @Override
  public void onError(Throwable e) {

  }

  @Override
  public void onNext(Integer item) {
    System.out.println("second: next item is " + item);

  }
});

subject.onNext(3);
subject.onNext(4);
subject.onNext(5);
subject.onCompleted();
```

The output is

```
first: next item is -1
first: next item is 1
first: next item is 2
second: next item is 2
first: next item is 3
second: next item is 3
first: next item is 4
second: next item is 4
first: next item is 5
second: next item is 5
first: sequence completed
second: sequence completed
```

What if you want the subscriber to receive all items emitted prior to its subscription instead of only the last one? Just use a `ReplaySubject`.

ReplaySubject

`ReplaySubject<T>` (Figure 4-3) emits all items emitted by the source Observable, regardless of when the observer subscribes, by keeping a buffer of the emitted items.

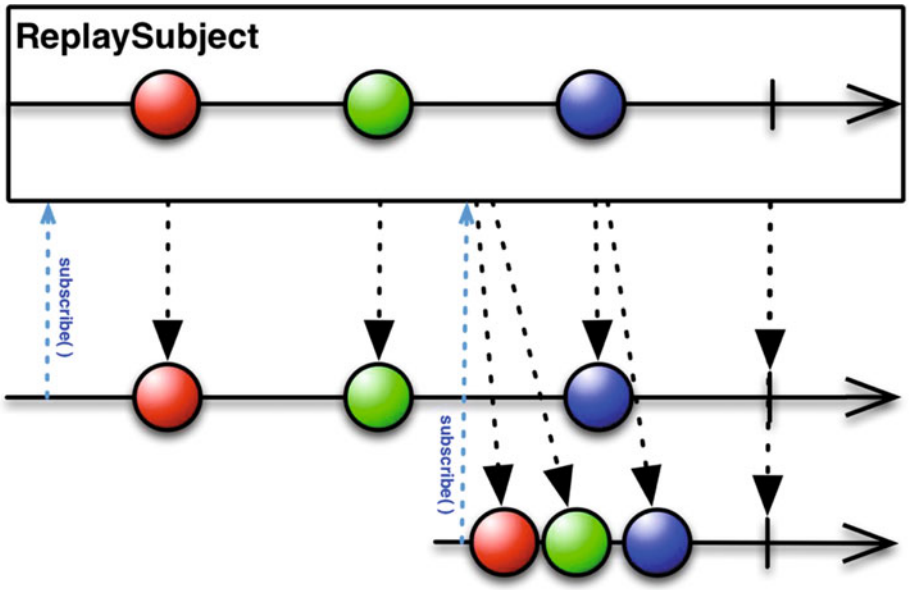

Figure 4-3. *The marble diagram for ReplaySubject<T>*

This buffer is backed up by an ArrayList, and by default it has an initial capacity of 16 and no upper bounds.

RxJava provides other static factory methods to create an instance of `ReplaySubject` with different behaviors:

- `ReplaySubject.create(int capacity)` creates an instance of `ReplaySubject` with the specified initial capacity.
- `ReplaySubject.createWithSize(int size)` creates an instance that will keep a buffer with the specified size; when the buffer is full, older items are discarded.
- `ReplaySubject.createWithTime(long time, TimeUnit unit, Scheduler scheduler)` creates a time-bounded `ReplaySubject`.
- `ReplaySubject.createWithTimeAndSize(long time, TimeUnit unit, int size, Scheduler scheduler)` lets you create a `ReplaySubject` that is both time-bounded and size-bounded.

AsyncSubject

AsyncSubject<T> (Figure 4-4) is an implementation of Subject that

- Emits only the last emitted item from the source observable after that observable completes.

- Emits no item if the source observable does not emit any item.
- Notifies an error event (without emitting any item) if the source Observable terminates with an error.

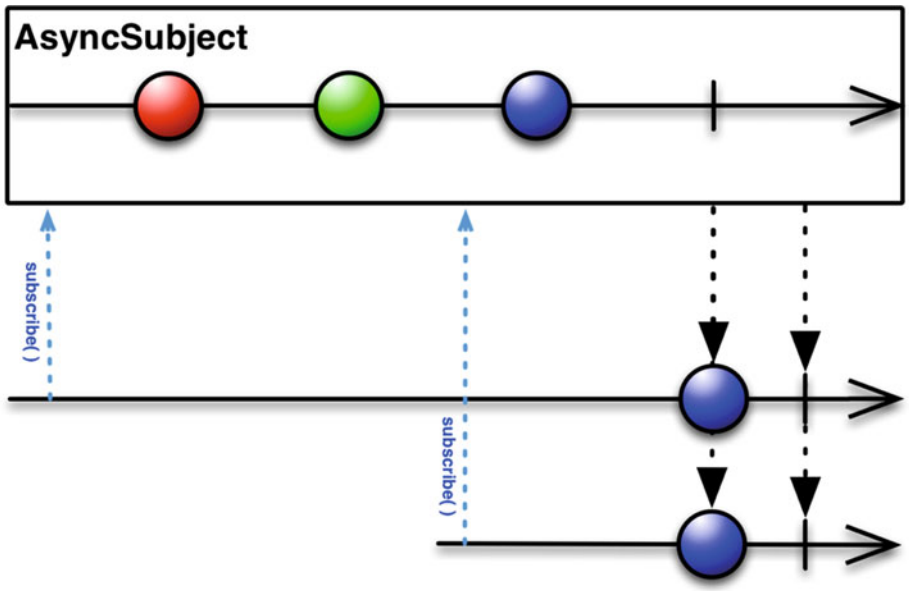

Figure 4-4. *The marble diagram for AsyncSubject<T>*

Let's change the previous example to use `AsyncSubject` instead of `PublishSubject`:

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


Observable<Integer> subjectAsObservable =
  subject.asObservable();

// subscribe the first Subscriber
subjectAsObservable.subscribe(new Subscriber<Integer>() {

  @Override
  public void onCompleted() {
    System.out.println("first: sequence completed");
  }

  @Override
  public void onError(Throwable e) {

  }
```

```
    @Override
    public void onNext(Integer item) {
      System.out.println("first: next item is " + item);

    }
  });

  subject.onNext(1);
  subject.onNext(2);

  // subscribe the second Subscriber
  subjectAsObservable.subscribe(new Subscriber<Integer>() {

    @Override
    public void onCompleted() {
      System.out.println("second: sequence completed");
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onNext(Integer item) {
      System.out.println("second: next item is " + item);

    }
  });

  subject.onNext(3);
  subject.onNext(4);
  subject.onNext(5);
  subject.onCompleted();
```

This code produces the following output:

```
first: next item is 5
first: sequence completed
second: next item is 5
second: sequence completed
```

When Should You Use Subjects?

There's an ongoing debate about the usage of Subjects. Erik Meijer, one of the RxJava's fathers, says that

> *They are the mutable variables of the Rx world and in most cases you do not need them. Typically a solution with Create or the other operators allows you to just wire up continuations without adding extra state. Stated slightly differently, it is good practice to minimize the number of objects that hold on to subscribers, you just want to pass them through.*
>
> [https://social.msdn.microsoft.com/Forums/en-US/bbf87eea-6a17-4920-96d7-2131e397a234/why-does-emeijer-not-like-subjects]

As a general rule, before using a Subject, ask yourself if the same purpose can be achieved with `Observable.create()` (or any other operator that creates observables). And, if you need to use a Subject, it's a good idea to expose it wrapped inside an Observable with the method `Subject.asObservable()`, because by doing so you will not expose the mutable component of the Subject.

EXAMPLE: A REACTIVE VERSION OF ARRAYLIST

The following example shows how to create a reactive version of ArrayList. This implementation of ArrayList exposes two additional methods to get notified when an item is added or removed:

```
class ReactiveArrayList<T> extends ArrayList<T> {


  private PublishSubject<T> addSubject =
        PublishSubject.create();
  private PublishSubject<Object> removeSubject =
        PublishSubject.create();

  @Override
  public boolean add(T item) {
    boolean result = super.add(item);
    if (result) {
      addSubject.onNext(item);
    }
    return result;
  }

  @Override
  public void add(int index, T item) {
    super.add(index, item);
    addSubject.onNext(item);
  }

  @Override
```

```
    public T remove(int index) {
      T removedItem = super.remove(index);
      removeSubject.onNext(removedItem);
      return removedItem;
    }

    @Override
    public boolean remove(Object object) {
      boolean result = super.remove(object);
      if (result) {
        removeSubject.onNext(object);
      }
      return result;
    }

    @Override
    public boolean addAll(Collection<? extends T> c) {
      boolean result = super.addAll(c);
      if (result) {
        for (T t : c) {
          addSubject.onNext(t);
        }
      }
      return result;
    }

    @Override
    public boolean addAll(int index, Collection<? extends T> c) {
      boolean result = super.addAll(index, c);
      if (result) {
        for (T t : c) {
          addSubject.onNext(t);
        }
      }
      return result;
    }

    public Observable<T> observeItemsAdded() {
      return addSubject.asObservable();
    }

    public Observable<Object> observeItemsRemoved() {
      return removeSubject.asObservable();
    }
  }
```

This is a usage example:

```
ReactiveArrayList<String> reactiveList =
  new ReactiveArrayList<String>();
```

```
  reactiveList.observeItemsAdded()
    .subscribe(new Subscriber<String>() {

      @Override
      public void onCompleted() {
      }

      @Override
      public void onError(Throwable e) {
      }

      @Override
      public void onNext(String item) {
        System.out.println("item added: " + item);
      }

    });

  reactiveList.observeItemsRemoved()
    .subscribe(new Subscriber<Object>() {

      @Override
      public void onCompleted() {
      }

      @Override
      public void onError(Throwable e) {
      }

      @Override
      public void onNext(Object item) {
        System.out.println("item removed: " + item);
      }

    });

  reactiveList.add("1");
  reactiveList.add("2");
  reactiveList.remove("1");
  reactiveList.addAll(Arrays.asList("4", "5", "6"));
  reactiveList.remove("5");
```

The output of this code is

```
item added: 1
item added: 2
item removed: 1
```

```
item added: 4
item added: 5
item added: 6
item removed: 5
```

■ **Note** Subjects are not thread-safe by default. They do not perform any synchronization across threads, so you must not call the `onNext( )`/`onCompleted()`/`onError()` methods from multiple threads, as this could lead to non-serialized calls, which violates the Observable contract and creates an ambiguity in the resulting Subject.

To make the Subject thread-safe, convert it into a `SerializedSubject<T,R>` (a subclass of `Subject<T,R>`) with code like the following:

```
new SerializedSubject(myUnsafeSubject);
```

Connectable Observables

Connectable Observables are another alternative to the usage of Subjects. A `ConnectableObservable<T>` behaves like an Observable, but it begins emitting items only when its `connect()` method is called.

```
ConnectableObservable<String> observable =
        Observable.range(0, 5)
          .map(new Func1<Integer, String>() {

            @Override
            public String call(Integer t) {
              return String.valueOf(t);
            }

          }).publish();

    observable.subscribe(new Subscriber<String>() {

      @Override
      public void onCompleted() {
        System.out.println("first: sequence completed");
      }

      @Override
      public void onError(Throwable e) {

      }

      @Override
```

```
      public void onNext(String item) {
        System.out.println("first: next item is " + item);

      }
    });

    observable.subscribe(new Subscriber<String>() {

      @Override
      public void onCompleted() {
        System.out.println("second: sequence completed");
      }

      @Override
      public void onError(Throwable e) {

      }

      @Override
      public void onNext(String item) {
        System.out.println("second: next item is " + item);

      }
    });
```

If you run this code, the output is… nothing! This is because you are missing the call to the `connect()` method:

```
observable.connect();
```

By adding this line, the observable starts emitting items and the output becomes

```
first: next item is 0
second: next item is 0
first: next item is 1
second: next item is 1
first: next item is 2
second: next item is 2
first: next item is 3
second: next item is 3
first: next item is 4
second: next item is 4
first: sequence completed
second: sequence completed
```

As you can see, every event generated by the observable is propagated to both subscribers, preserving the order of subscription (the first subscriber is notified first for each event).

CHAPTER 5

Networking with RxJava and Retrofit

Network operations like calling the RESTful API are an example of a scenario where you can apply RxJava. In fact,

- You need to implement a callback mechanism to react to the response of the network call, which can terminate with a succesful state or a failure.
- Sometimes you need to chain different network calls sequentially.
- Often these operations need to be executed in a separated thread.

It's trivial now to map the idea of a response with a successful or error state to the Subcriber's `onNext` and `onError` methods, chaining operations to an `Observable` concatenation (or transformation), and the idea of using a separate thread for the application of a Scheduler.

Java provides a basic support for network operations (look at packages `java.net` and `javax.net`), but there are many other libraries that really simplify working with a network, like

- Netty (`http://netty.io/`)
- Async Http Client (`https://github.com/AsyncHttpClient/async-http-client`)
- OkHttp (`http://square.github.io/okhttp/`)
- Retrofit (`http://square.github.io/retrofit/`)

This chapter covers Retrofit because it has an interesting feature: built-in support for RxJava that lets you choose if you want the network response to be wrapped inside an `Observable` object.

I'm assuming that you know what RESTful APIs (`https://en.wikipedia.org/wiki/Representational_state_transfer`) are and that you have a basic knowledge of JSON (Google Gson will be used for JSON parsing - `https://github.com/google/gson`).

A. Maglie, *Reactive Java Programming*, DOI 10.1007/978-1-4842-1428-2_5

Retrofit's Built-in Support for RxJava

Retrofit is a library that provides a type-safe HTTP client for Java (and Android). The great point about Retrofit is that you just have to define an interface that will act as a proxy for the HTTP API, and the library will automatically generate the implementation of this interface for you.

All examples are based on version 2.1.0 of Retrofit. It supports Java 7 and 8, but not Java 6.

Setting Up Retrofit in Your Java Project

The setup is straightforward. If you're using maven, just add the following dependency declaration:

```
<dependency>
<groupId>com.squareup.retrofit2</groupId>
<artifactId>retrofit</artifactId>
<version>2.1.0</version>
</dependency>
```

If you're using gradle:

```
compile 'com.squareup.retrofit2:retrofit:2.1.0'
```

Alternatively, you can download the jar from the web site.

Creating a Retrofit Service

Let's consider the APIs that are provided by GitHub (https://developer.github.com/v3/). GitHub provides many APIs to access user's information, repositories, and gists, but for now let's just consider the API to retrieve a user's list of public repositories.

From the GitHub API documentation, the URL of this API is

```
https://api.github.com/users/{user}/repos
```

where {user} is the name of the GitHub user.

A Retrofit interface that maps such API will look like this:

```
import java.util.List;
import retrofit2.http.GET;
import retrofit2.http.Path;
import rx.Observable;

public interface GitHubService {
  @GET("users/{user}/repos")
  Observable<List<Repo>> listRepos(@Path("user") String user);
}
```

The Repo object is used to map the JSON returned by the API to a POJO (plain old Java object); this is a simple implementation that maps only the fields that you need in the examples:

```
public class Repo {

  @SerializedName("id")
  private int id;

  @SerializedName("name")
  private String name;

  @SerializedName("url")
  private String url;

  @SerializedName("watchers_count")
  private int watchersCount;

  @SerializedName("open_issues_count")
  private int openIssueCount;

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getUrl() {
    return url;
  }

  public void setUrl(String url) {
    this.url = url;
  }

  public int getWatchersCount() {
    return watchersCount;
  }
```

```
  public void setWatchersCount(int watchersCount) {
    this.watchersCount = watchersCount;
  }

  public int getOpenIssueCount() {
    return openIssueCount;
  }

  public void setOpenIssueCount(int openIssueCount) {
    this.openIssueCount = openIssueCount;
  }
}
```

Here you've defined a method that maps a GET request (see the `@GET` annotation) to the API `"users/{user}/repos"` (passed as parameter to the `@GET` annotation). The value of `{user}` is a parameter of the method, and with the annotation `@Path` you're telling Retrofit that the `"user"` parameter must be substituted to the `{user}` part of the API URL.

The method will return an Observable that will do the following:

- Emit a sequence of just a non-null object of type `List<Repo>` if the request is successfull (and then terminates).
- Notify with an error event if some error occurs.

The implementation of the `GitHubService` interface can be generated as follows:

```
Retrofit retrofit = new Retrofit.Builder()
  .baseUrl("https://api.github.com/")
  .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
  .addConverterFactory(GsonConverterFactory.create())
  .build();

GitHubService service = retrofit.create(GitHubService.class);
```

The support for RxJava must be enabled by including the following dependency:

```
<dependency>
      <groupId>com.squareup.retrofit2</groupId>
      <artifactId>adapter-rxjava</artifactId>
      <version>2.1.0</version>
</dependency>
```

You must also register the adapter with the method

```
addCallAdapterFactory(RxJavaCallAdapterFactory.create())
```

You also need to add support for converting the response body from JSON to your `Repo` object using Gson. To do this, you need to add the following dependency:

```
<dependency>
```

```
        <groupId>com.squareup.retrofit2</groupId>
        <artifactId>converter-gson</artifactId>
        <version>2.1.0</version>
        </dependency>
  </dependencies>
```

You also need to register the converter:

```
  addConverterFactory(GsonConverterFactory.create())
```

Now you can use the service implementation and make a subscription:

```
  String user = ...

  service.listRepos(user)
    .subscribe(new Subscriber<List<Repo>>() {

        @Override
        public void onCompleted() {
          System.out.println("sequence completed");
        }

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

        @Override
        public void onNext(List<Repo> repos) {
          for (Repo repo : repos) {
            System.out.println("Repo: " + repo.getName());
          }
        }
});
```

Putting all together,

```
  import retrofit2.Retrofit;
  import retrofit2.adapter.rxjava.RxJavaCallAdapterFactory;
  import retrofit2.converter.gson.GsonConverterFactory;
  import rx.Observable;
  import rx.Subscriber;
  import rx.functions.Func1;
  import rx.schedulers.Schedulers

  public static void listRepos(String user) {
    retrofit2.Retrofit retrofit =
      new retrofit2.Retrofit.Builder()
```

```
      .baseUrl("https://api.github.com/")
      .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
      .addConverterFactory(GsonConverterFactory.create())
      .build();

  GitHubService service = retrofit.create(GitHubService.class);

  service.listRepos(user)
    .subscribe(new Subscriber<List<Repo>>() {

      @Override
      public void onCompleted() {
        System.out.println("sequence completed");
      }

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

      @Override
      public void onNext(List<Repo> repos) {
        for (Repo repo : repos) {
          System.out.println("Repo: " + repo.getName());
        }
      }
    });
}
```

For example, to print a list of the GitHub repositories of user "octocat", you can call method

```
listRepos("octocat");
```

and the results will be

```
Repo: git-consortium
Repo: hello-worId
Repo: Hello-World
Repo: linguist
Repo: octocat.github.io
Repo: Spoon-Knife
Repo: test-repo1
sequence completed
```

Filter Results

Now that you've got the list of repositories, you want to filter that list and show only the repositories with at most two open issues. You can do this by applying the `filter()` operator.

The Observable returned by the method `listRepos()` emits a sequence of only one item of type `List<Repo>`. To apply the `filter()` operator, you need the Observable to emit an item for each `Repo` object. The following code shows how you can transform the Observable into another Observable and then apply a filter:

```
String user = ...
Int maxOpenIssues = ...
service.listRepos(user)
  .flatMap(new Func1<List<Repo>, Observable<Repo>>() {

      @Override
      public Observable<Repo> call(List<Repo> repos) {
        return Observable.from(repos);
      }

  })
  .filter(new Func1<Repo, Boolean>() {

      @Override
      public Boolean call(Repo repo) {
        return repo.getOpenIssueCount() <= maxOpenIssues;
      }

  })
  .subscribe(new Subscriber<Repo>() {

    @Override
    public void onCompleted() {
      System.out.println("sequence completed");
    }

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

    @Override
    public void onNext(Repo repo) {
      System.out.println("Repo: " + repo.getName());
    }
});
```

The complete method is

```
public static void listReposWithMaxIssues(String user, final int
maxOpenIssues) {
  Retrofit retrofit = new Retrofit.Builder()
     .baseUrl("https://api.github.com/")
     .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
     .addConverterFactory(GsonConverterFactory.create())
     .build();

  GitHubService service = retrofit.create(GitHubService.class);

  service.listRepos(user)
   .flatMap(new Func1<List<Repo>, Observable<Repo>>() {

    @Override
    public Observable<Repo> call(List<Repo> repos) {
     return Observable.from(repos);
    }

   })
   .filter(new Func1<Repo, Boolean>() {

    @Override
    public Boolean call(Repo repo) {
     return repo.getOpenIssueCount() <= maxOpenIssues;
    }

   })
   .subscribe(new Subscriber<Repo>() {

    @Override
    public void onCompleted() {
     System.out.println("sequence completed");
    }

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

    @Override
    public void onNext(Repo repo) {
     System.out.println("Repo: " + repo.getName());
    }
   });
 }
```

And here's the output:

```
Repo: hello-worId
Repo: test-repo1
sequence completed
```

Choosing the Right Scheduler

In the previous example, you didn't specify any Scheduler, so the code is executed using the default Scheduler.

Suppose that you're working on an application with a UI and that the network operation is kicked off by clicking on a button. You don't want the UI to be blocked waiting for the response, so you need to execute all the networking stuff on a separate thread. This can be done by applying the `subscribeOn()` operator. A network operation is an I/O operation, so the right scheduler to apply is `Schedulers.io()`.

But you need also to ensure that the result is notified on the UI thread (if the result is used to update the UI), so you also need to apply the `observeOn()` operator. The right scheduler here depends on the library or framework you are using to build the UI. For example, in JavaFx, you can use the JavaFxScheduler provided by RxJavaFX (https://github.com/ReactiveX/RxJavaFX); in Android use `AndroidSchedulers.mainThread()` provided by RxAndroid (https://github.com/ReactiveX/RxAndroid).

```
  service.listRepos(user)
    .flatMap(new Func1<List<Repo>, Observable<Repo>>() {

      @Override
      public Observable<Repo> call(List<Repo> repos) {
        return Observable.from(repos);
      }
    })
    .filter(new Func1<Repo, Boolean>() {

      @Override
      public Boolean call(Repo repo) {
        return repo.getOpenIssueCount() <= maxOpenIssues;
      }
    })
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Subscriber<Repo>() {

      @Override
      public void onCompleted() {
        System.out.println("sequence completed");
      }
```

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

    @Override
    public void onNext(Repo repo) {
      System.out.println("Repo: " + repo.getName());
    }
  });
```

By adding `subscribeOn(Schedulers.io())` right before `subscribe()` call, methods `listRepos()`, `flatMap()`, and `filter()` will be executed on a different thread.

The results (`onNext`, `onError`, and `onCompleted`) will be notified on the main thread.

Chaining Multiple Network Calls

Chaining multiple network calls often means dealing with nested callbacks, which make your code hard to read and maintain. Moreover, you have to handle errors in each callback, increasing the code complexity.

From what you've seen so far, it's easy to understand how to use RxJava to substitute the callback chain with a combination of observables.

In this simple scenario, you need to call a remote API to authenticate a user, then another one to get the user's data, and again another API to get the user's contacts.

The following code shows nested API calls with callbacks:

```
import retrofit2.Callback;
import retrofit2.Call;
import retrofit2.Response;

User user = null;

myService.login(username, password,
      new Callback<AccessToken>() {

  @Override
  public void success(User user, Response response) {

    storeCredentials(response.getAccessToken())

    myService.getUser(accessToken, new Callback<User>() {
      @Override
      public void onResponse(Call<User> call,
              Response<User> response) {

        user = response.getBody();
```

```
        myService.getUserContact(user.getId(),
                        new Callback<Contact>() {
          @Override
          public Contact onResponse(Call<Contact> contact,
              Response<Contact> response) {
                   user.setContact(response.getBody());
           }

           @Override
           public void onFailure(RetrofitError error) {
             // handle error here...
           }
         });
       }

       @Override
       public void onFailure(RetrofitError error) {
         // handle error here...
       }
     });
   }

   @Override
   public void failure(RetrofitError error) {
     // handle error here...
   }
});
```

And this is how you can transform this code using RxJava:

```
  myService.login()
    .doOnNext(accessToken -> storeCredentials(accessToken))
    .flatMap(accessToken -> myService.getUser(accessToken))
    .flatMap(user -> myService.getUserContact(user.getId()))
    .subscribe(new Subscriber<Contact>() {
      @Override
      public void onCompleted() {
      }

      @Override
      public void onError(Throwable e) {
      }

      @Override
      public void onNext(Contact contact) {
      }
    });
```

where

- `myService.login()` returns an `Observable<AccessToken>`.
- The `doOnNext()` method is applied to the result of `myService.login()` to invoke `storeCredentials()` every time an instance of `AccessToken` is emitted.
- The same instance of `AccessToken` is used by the first `flatMap` operator to invoke the method `myService.getUser(accessToken)` and thus return an `Observable<User>`.
- Another `flatMap` operator is applied to use the emitted instance of `User` to call `myService.getUserContact(user.getId())`.

The code becomes simpler to read and maintain because it's built using reusable blocks of code.

With the nested callbacks version, there are three different places where you have to handle errors; in the RxJava version, you can handle errors in just one place, since every error will be forwarded to the observer and notified in the `onError` method.

Caching Data

When working with network calls, an important topic is caching. Caching is a mechanism to save a local copy of the network response to be used when the network is unreachable or to avoid doing too many subsequent network calls.

Retrofit supports caching network responses using `OkHttpClient`'s cache mechanism. In some cases, this is not enough and you need to cache data by applying your own custom logic.

Again, RxJava can help you build a complex caching mechanism with simple and readable code. The following code shows an abstract implementation of a class that caches a network response on disk and in memory. Note that in this example you take advantage of lambda expressions (provided by Java 8) to make your code less verbose.

```
abstract class CacheManager<T> {

  private T mMemoryCache;

  public Observable<T> getData() {
    return Observable.concat(fromMemory(),
          fromDisk(), fromNetwork())
      .first(response -> isValid(response))
      .onErrorResumeNext(t -> fallbackToDiskCache());
  }

  protected Observable<T> fromNetwork() {
    return doNetworkCall()
      .doOnNext(response -> {
```

```
      cacheInMemory(response);
      cacheOnDisk(response);
    })
    .compose(logSource("network"));
}

private Observable<T> fallbackToDiskCache() {
  System.out.println("fallback to disk cache");
  return fromDisk();
}

protected void cacheOnDisk(T response) {
  System.out.println("saving data on disk: " + response);
  persistOnDisk(response);
}

protected Observable<T> fromDisk() {
  Observable<T> observable = Observable.create(subscriber -> {
    T response = readFromDisk();
    if (response != null) {
      subscriber.onNext(response);
      cacheInMemory(response);
    }
    subscriber.onCompleted();
  });

  return observable.compose(logSource("disk"));
}

protected void cacheInMemory(T response) {
  mMemoryCache = response;
}

protected Observable<T> fromMemory() {
  Observable<T> observable = Observable.create(subscriber -> {
    subscriber.onNext(mMemoryCache);
    subscriber.onCompleted();
  });

  return observable.compose(logSource("memory"));
}

public void deleteAll() {
  mMemoryCache = null;
  deleteFromDisk();
}

private Observable.Transformer<T, T> logSource(final String source) {
```

```
    return dataObservable -> dataObservable.doOnNext(data -> {
      if (data == null) {
        System.out.println(source + " does not have any data.");
      } else if (!isValid(data)) {
        System.out.println(source + " has stale data.");
      } else {
        System.out.println(source + " has the data you are looking for!");
      }
    });
  }

  abstract boolean isValid(T data);

  abstract Observable<T> doNetworkCall();

  abstract void persistOnDisk(T response);

  abstract T readFromDisk();

  abstract void deleteFromDisk();

}
```

Let's break down this code:

- The main idea is to use the `Observable.concat()` operator to concatenate three operations: getting data from memory, getting data from the disk, and getting data from the network, in this exact order.
- Then you apply the `first()` operator with a function that filters out data that is not valid (i.e. it's not null or it's not too old). So if the memory returns valid data, it will be forwarded to the subscriber and no data will be read from the disk or the network. If the memory returns invalid data, the data will be read from the disk, checking if it's valid, and so on.
- If some error occurs (for example, no network connection or server unreachable), you fall back to disk cache. This is done with the help of operator `onErrorResumeNext()`, which makes the source `Observable` emit items from the given `Observable` (`fallbackToDiskCache()`) in case of an error.
- The `fromNetwork()` Observable retrieves data from the network and uses the `onNext()` operator to cache data on disk and in memory when the subscriber's `onNext()` method is notified. Remember that `fromNetwork()` is called only if memory and the disk contain invalid data.

- The same logic is applied to the `fromDisk()` method, where you save data in memory before notifying the subscriber's `onNext()` method.
- Finally, you use the `Observable.compose()` operator to apply the `logSource()` operator that prints informations about the current source of data.

Note how you used `Observable.create()` to create the observers for getting data from memory and from disk.

The details about getting data from the network and saving and reading data from disk are left to the concrete implementation.

In these examples, you've used only a few of the many features that the Retrofit library provides, because the focus was on the RxJava integration part. With Retrofit, you can manage not only GET requests but also POST, DELETE, PUT, and PATCH requests. You can specify header and query parameters, manipulate a URL's path, send form-encoded and multipart data. Go to `http://square.github.io/retrofit/` for the full documentation.

CHAPTER 6

RxJava and Android

In the actual mobile-driven world, talking about Java also means also talking about Android.

The RxJava library can be used in Android development without any other requirements. This means that all the code examples that you have seen so far can be included in an Android project; all you need to do is add the dependency for RxJava in your `build.gradle` file.

There are some libraries that provide additional support for applying RxJava concepts in the Android environment. The fundamentals are

- RxAndroid
- RxBindings
- RxLifecycle

RxAndroid

RxAndroid is the first open-source library for Android that you'll need. In fact, it provides a very useful instance of Scheduler that schedules on the main thread (or a given looper).

To install the dependency, you just need to add the two dependency declarations to your `build.gradle` file:

```
apply plugin: 'com.android.application'

android {
  compileSdkVersion 24
    buildToolsVersion "24.0.1"

    defaultConfig {
      applicationId "com.example"
      minSdkVersion 14
      targetSdkVersion 24
      versionCode 1
      versionName "1.0"
    }
```


A. Maglie, *Reactive Java Programming*, DOI 10.1007/978-1-4842-1428-2_6

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

  dependencies {
    compile 'com.android.support:appcompat-v7:24.1.1'
    compile 'io.reactivex:rxjava:1.2.0'
    compile 'io.reactivex:rxandroid:1.2.1'
  }
```

RxJava requires `minSdkVersion 9`, but RxAndroid requires `minSdkVersion 14`.

Refer to the project web site (https://github.com/ReactiveX/RxAndroid) to know the latest version of the library.

As you've already seen, you can use the provided `AndroidSchedulers.mainThread()` as a parameter to the `observeOn()` operator to make your Subscriber notified of the `onCompleted()`/`onError()`/`onNext()` events on the UI thread:

```
Observable.range(1, 10)
  .subscribeOn(Schedulers.newThread())
  .observeOn(AndroidSchedulers.mainThread())
  .subscribe(new Subscriber<Integer>() {

    @Override
    public void onCompleted() {
      // notified on UI thread
    }

    @Override
    public void onError(Throwable e) {
      // notified on UI thread
    }

    @Override
    public void onNext(Integer item) {
      // notified on UI thread
    }
});
```

You can also subscribe on another thread; the corresponding Scheduler can be instantiated with the `AndroidSchedulers.from()` method:

```
BackgroundThread myThread = new myThread();
myThread.start();
```

```
Observable.range(1, 10)
  .subscribeOn(AndroidSchedulers.from(myThread.getLooper())
  .observeOn(AndroidSchedulers.mainThread())
  .subscribe(new Subscriber<Integer>() {

    @Override
    public void onCompleted() {
      // notified on UI thread
    }

    @Override
    public void onError(Throwable e) {
      // notified on UI thread
    }

    @Override
    public void onNext(Integer item) {
      // notified on UI thread
    }
});
```

If you want to build a Scheduler around your Handler, use `HandlerThreadScheduler`.

RxBindings

RxJava can also be used to interact with UI components in a reactive way, instead of using listeners.

For example, to be notified when a `View` is clicked, you must implement an `OnClickListener()`:

```
myView.setOnClickListener(new OnClickListener() {

  @Override
  public void onClick(final View v) {

  }
});
```

Or if you want to listen for changes to the text typed in an `EditText`, you must add a `TextWatcher` with the method `EditText.addTextChangedListener(TextWatcher)`:

```
  editText.addTextChangedListener(new TextWatcher() {

    @Override
    public void onTextChanged(CharSequence s, int start, int before, int count)
{

    }
```

```
    @Override
    public void beforeTextChanged(CharSequence s, int start, int count,
    int after) {

    }

    @Override
    public void afterTextChanged(Editable s) {

    }
  });
```

RxBinding (https://github.com/JakeWharton/RxBinding) is an open source library that provides some methods to avoid using this callback mechanism, applying the reactive logic instead.

To add this library to your Android project, add the dependency into your app's `build.gradle` file:

```
apply plugin: 'com.android.application'

android {
  compileSdkVersion 24
  buildToolsVersion "24.0.1"

  defaultConfig {
    applicationId "com.example"
    minSdkVersion 14
    targetSdkVersion 24
    versionCode 1
    versionName "1.0"
  }

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

dependencies {
  compile 'com.android.support:appcompat-v7:24.1.1'
  compile 'io.reactivex:rxjava:1.2.0'
  compile 'io.reactivex:rxandroid:1.2.1'
  compile 'com.jakewharton.rxbinding:rxbinding:0.4.0'
}
```

EXAMPLE: REACTING TO CLICKS ON A VIEW

Listening for clicks can be implemented with the method

```
RxView.clicks(myView)
  .subscribe(click -> doSomething());
```

`RxView.clicks()` returns a hot Observable (hot because clicks are fired even when there are no subscribers).

You can also apply operators to this Observable.

It could be useful to implement a mechanism to avoid too many consecutive clicks on a view. This can be achieved with the `throttleFirst` operator (see Chapter 3 for the definition of `throttleFirst`):

```
RxView.clicks(myView)
  .throttleFirst(300, TimeUnit.MILLISECONDS)
  .subscribe(click -> doSomething());
```

EXAMPLE: REACTING TO TEXT TYPED IN AN EDITTEXT

The method `RxTextView.afterTextChangeEvents()` provides an Observable that notifies the subscribers when a change occurs in an `EditText`:

```
RxTextView.afterTextChangeEvents(editText)
  .subscribe(new Subscriber<TextViewAfterTextChangeEvent>() {
    @Override
    public void onCompleted() {
    }

    @Override
    public void onError(Throwable e) {
    }

    @Override
    public void onNext(TextViewAfterTextChangeEvent event) {
      CharSequence text = event.view().getText();
      // do something
    }
  });
```

If you want to be notified of text changes only every 500 milliseconds, you can apply the debounce operator:

```
RxTextView.afterTextChangeEvents(editText)
  .debounce(500, TimeUnit.MILLISECONDS)
  .observeOn(AndroidSchedulers.mainThread())
  .subscribe(new Subscriber<TextViewAfterTextChangeEvent>() {
    @Override
    public void onNext(TextViewAfterTextChangeEvent event) {
      CharSequence text = event.view().getText();
      // do something
    }
  });
```

Looking at the source code of RxTextView, you can see that afterTextChangeEvents returns an Observable that implements a TextWatcher for you:

```
Observable<TextViewAfterTextChangeEvent> afterTextChangeEvents(
      @NonNull TextView view) {
  checkNotNull(view, "view == null");
  return Observable
    .create(new TextViewAfterTextChangeEventOnSubscribe(view));
}

class TextViewAfterTextChangeEventOnSubscribe
    implements Observable.OnSubscribe<TextViewAfterTextChangeEvent> {
  final TextView view;

  TextViewAfterTextChangeEventOnSubscribe(TextView view) {
    this.view = view;
  }

  @Override public void call(final Subscriber<? super
  TextViewAfterTextChangeEvent> subscriber) {
    verifyMainThread();

    final TextWatcher watcher = new TextWatcher() {
      @Override public void beforeTextChanged(CharSequence s, int start, int
      count, int after) {
      }

      @Override public void onTextChanged(CharSequence s, int start, int
      before, int count) {
      }

      @Override public void afterTextChanged(Editable s) {
        if (!subscriber.isUnsubscribed()) {
          subscriber.onNext(TextViewAfterTextChangeEvent.create(view, s));
```

```
      }
    }
  };
  view.addTextChangedListener(watcher);

  subscriber.add(new MainThreadSubscription() {
    @Override protected void onUnsubscribe() {
      view.removeTextChangedListener(watcher);
    }
  });

  // Emit initial value.
  subscriber.onNext(TextViewAfterTextChangeEvent.create(view, view.
  getEditableText()));
}
```

In this code,

- A `TextWatcher` is implemented and added to the `EditText`.
- The `TextWatcher` is removed when the observable is unsubscribed.
- The Subscriber's `onNext` method is called every time the `TextWatcher.afterTextChanged()` method is called (if the subscriber is still subscribed).

Activity and Fragment Life Cycle

If you develop on Android, you should be familiar with the concept of the Activity life cycle: during the process of setting up and displaying an Activity, the operating system calls a series of methods on the Activity; other methods are called when a user performs an action (like rotating the device); some others are called in response to external events (as when an incoming call pauses your app's Activity).

How do subscriptions behave in these situations? There is no built-in mechanism in RxJava (nor in RxAndroid) that manages those cases for you.

Let's take a look at some examples. Suppose that you subscribe to an observable inside an Activity's `onCreate` method:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_layout)
    TextView textView = findViewById(R.id.text_view)

    Observable.timer(2, TimeUnit.MINUTES)
            .subscribeOn(Schedulers.newThread())
            .observeOn(AndroidSchedulers.mainThread())
```

```
            .subscribe(new Subscriber<Long>() {
                      @Override
                      public void onCompleted() {

                      }

          @Override
          public void onError(Throwable e) {

          }

          @Override
          public void onNext(Long item) {
            textView.setText("Timeout!");
          }
            });
}
```

A timer observable is subscribed in the `onCreate` method; after 2 minutes, the `TextView` will be updated with the message "Timeout!"

What happens if user rotates the screen before 2 minutes has passed?

When the screen is rotated, the Activity is destroyed and recreated. This leads to two problems in your situation:

- The observable will continue its execution at the activity rotation, but when it completes, it will try to update a wrong instance of `TextView`.
- The observable will keep a reference to the the old Activity, causing a memory leak.

The solution here is simple: you must unsubscribe according to the life cycle. For example, let's keep a reference to the subscription and unsubscribe it when Activity is destroyed:

```
Subscription subscription = null;

@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_layout)
  TextView textView = findViewById(R.id.text_view)

  subscription = Observable.timer(2, TimeUnit.MINUTES)
    .subscribeOn(Schedulers.newThread())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Subscriber<Long>() {
      @Override
      public void onCompleted() {
```

```
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onNext(Long item) {
            textView.setText("Timeout!");
    }
  });
}

@Override
protected void onDestroy() {
  super.onDestroy();

  if (subscription != null && !subscription.isUnsubscribed()) {
    subscription.unsubscribe();
  }
}
```

You can also create a `CompositeSubscription` object and add all subscriptions to it. Then you can unsubscribe all of them at once:

```
CompositeSubscription compositeSubscription
  = new CompositeSubscription();

@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_layout)
  TextView textView = findViewById(R.id.text_view)

      subscription = Observable.timer(2, TimeUnit.MINUTES)
              .subscribeOn(Schedulers.newThread())
              .observeOn(AndroidSchedulers.mainThread())
              .subscribe(new Subscriber<Long>() {
                        @Override
                        public void onCompleted() {

                        }

            @Override
            public void onError(Throwable e) {

            }
```

```
            @Override
            public void onNext(Long item) {
              textView.setText("Timeout!");
            }
              });
}

@Override
protected void onDestroy() {
  super.onDestroy();

  compositeSubscription.unsubscribe();
}
```

Note that once `CompositeSubscription.unsubscribe()` is called, the object becomes unusable, because it will automatically unsubscribe any other subscription that will be added.

Trello made an open source library called RxLifecycle (https://github.com/trello/RxLifecycle) that simplifies managing subscription/unsubscription in the Activity (or Fragment) life cycle.

To include it into your project, add the dependency to your app's `build.gradle` file:

```
dependencies {
  compile 'com.android.support:appcompat-v7:24.1.1'
  compile 'io.reactivex:rxjava:1.2.0'
  compile 'io.reactivex:rxandroid:1.2.1'
  compile 'com.jakewharton.rxbinding:rxbinding:0.4.0'
  compile 'com.trello:rxlifecycle:0.7.0'
  compile 'com.trello:rxlifecycle-android:0.7.0'
}
```

With this library, you can use the `compose()` operator to specify when to unsubscribe from Observable:

```
myObservable
  .compose(RxLifecycle.bindUntilEvent(lifecycle,
      ActivityEvent.DESTROY))
  .subscribe();
```

Or you can let the library auto-detect when to unsubscribe:

```
myObservable
  .compose(RxLifecycleAndroid.bindActivity(lifecycle))
  .subscribe();
```

In the latter case, the unsubscription is managed in the life cycle event opposed to the subscription (for example, if subscription occurs in the Activity's `onStart()` method, the unsubscription will happen in `onStop()`).

Android development with RxJava is a hot topic and many other open source libraries have been developed, covering many areas. Some of those libraries are

- Android-ReactiveLocation (https://github.com/mcharmas/Android-ReactiveLocation)
- rx-android-permissions (https://github.com/beworker/rx-android-permissions)
- RxSensor (https://github.com/wandup/RxSensor)

Index

A. Maglie, *Reactive Java Programming*, DOI 10.1007/978-1-4842-1428-2

P, Q

R

S

T

U, V

W, X, Y, Z