Reactive Programming with Kotlin

FIRST EDITION

Learn Rx with RxJava, RxKotlin, and RxAndroid

By the raywenderlich.com Tutorial Team

Alex Sullivan

Table of Contents: Overview

Table of Contents: Extended

About This Book Sample

Welcome to *Reactive Programming with Kotlin*!

The popularity of reactive programming continues to grow on an ever-increasing number of platforms and languages. Rx lets developers easily and quickly build apps with code that can be understood by other Rx developers—even over different platforms.

Not only will you learn how to use RxJava to create complex reactive applications on Android, you'll also see how to solve common application design issues by using RxJava. Finally, you'll discover how to exercise full control over the library and leverage the full power of reactive programming in your apps.

We are pleased to offer you this sample from the full *Reactive Programming with Kotlin* book that will introduce you to these concepts and give you a chance to practice them in our hands-on By Tutorials style.

The chapters that follow come from the first section of the book, "Getting Started with RxJava."

In this part of the book, you're going to learn about the basics of RxJava. You are going to have a look at what kinds of asynchronous programming problems RxJava addresses, and what kind of solutions it offers.

Further, you will learn about the few basic classes that allow you to create and observe event sequences, which are the foundation of the Rx framework.

You are going to start slow by learning about the basics and a little bit of theory. Please don't skip these chapters! This will allow you to make good progress in the following sections when things get more complex.

This sample includes:

1. **Hello RxJava!**: Learn about the reactive programming paradigm and what RxJava can bring to your app.
2. **Observables**: Now that you're ready to use RxJava and have learned some of the basic concepts, it's time to play around with observables.

The book is ready for purchase at:

- https://store.raywenderlich.com/products/reactive-programming-with-kotlin.

Enjoy!

The *Reactive Programming with Kotlins* Team

Reactive Programming with Kotlin

By Alex Sullivan

Notice of Rights

Notice of Liability

Trademarks

Dedications

"To my wonderful partner Pallavi, without whom I would have never been able to start this undertaking. Your support and encouragement mean the world to me."

— *Alex Sullivan*

About the Author

Alex Sullivan is an author of this book. Alex is a mobile developer who works at Thoughtbot in Boston, where he enjoys reactive programming, experimenting with different programming languages, and tinkering with fun approaches to building mobile applications. In his spare time, Alex enjoys traveling and relaxing with his partner, binging unhealthy amounts of Netflix and reading. Alex hopes to one day find a cat he's not allergic to and rant about bracket placement to him or her.

About the Editors

Joe Howard is the final pass editor for this book. Joe is a former physicist that studied computational particle physics using parallel Fortran simulations. He gradually shifted into systems engineering and then ultimately software engineering around the time of the release of the iOS and Android SDKs. He's been a mobile software developer on iOS and Android since 2009, working primarily at two agencies in Boston, MA since 2011. He's now the Pillar Lead for raywenderlich.com. Twitter: @orionthewake.

Manda Frederick is the editor of this book. She has been involved in publishing for over ten years through various creative, educational, medical and technical print and digital publications, and is thrilled to bring her experience to the raywenderlich.com family as Managing Editor. In her free time, you can find her at the climbing gym, backpacking in the backcountry, hanging with her dog, working on poems, playing guitar and exploring breweries.

Victoria Gonda is a tech editor for this book. Victoria is a software developer working mostly on Android apps. when she's not traveling to speak at conferences, she works remotely from Chicago. Her interest in tech started while studying computer science and dance production in college. In her spare time, you can find Victoria relaxing with a book, her partner, and her pets. You can connect with her on Twitter at @TTGonda.

Ellen Shapiro is a tech editor for this book. Ellen is an iOS developer for Bakken & Bæck's Amsterdam office who also occasionally writes Android apps. She is working in her spare time to help bring songwriting app Hum to life. She's also developed several independent applications through her personal company, Designated Nerd Software. When she's not writing code, she's usually tweeting about it.

Amanjeet Singh is a tech editor for this book. Amanjeet is an Android Engineer in Delhi, India and an open source enthusiast. As a developer he always tries to build apps with optimized performance and good architectures which can be used on a large scale. Currently Android Engineer at 1mg, he helps in building apps for one of the leading healthcare platform in India. Also a technical editor and author in android team at raywenderlich.com.

Matei Suica is a tech editor for this book. Matei is a software developer that dreams about changing the world with his work. From his small office in Romania, Matei is trying to create an App that will help millions. When the laptop lid closes, he likes to go to the gym and read. You can find him on Twitter or LinkedIn: @mateisuica

About the Artist

Vicki Wenderlich is the designer and artist of the cover of this book. She is Ray's wife and business partner. She is a digital artist who creates illustrations, game art and a lot of other art or design work for the tutorials and books on raywenderlich.com. When she's not making art, she loves hiking, a good glass of wine and attempting to create the perfect cheese plate.

Acknowldegments

We'd also like to thank the *RxSwift: Reactive Programming with Swift* authors, whose work served as the basis for parts of this book:

- **Florent Pillet** has been developing for mobile platforms since the last century and moved to iOS on day 1. He adopted reactive programming before Swift was announced and has been using RxSwift in production since 2015. A freelance developer, Florent also uses Rx on Android and likes working on tools for developers like the popular NSLogger when he's not contracting for clients worldwide. Say hello to Florent on Twitter at @fpillet.
- **Junior Bontognali** has been developing on iOS since the first iPhone and joined the RxSwift team in the early development stage. Based in Switzerland, when he's not eating cheese or chocolate, he's doing some cool stuff in the mobile space, without denying to work on other technologies. Other than that he organizes tech events, speaks and blogs. Say hello to Junior on Twitter at @bontoJR.
- **Marin Todorov** is one of the founding members of the raywenderlich.com team and has worked on seven of the team's books. Besides crafting code, Marin also enjoys blogging, teaching, and speaking at conferences. He happily open-sources code. You can find out more about Marin at www.underplot.com.
- **Scott Gardner** has been developing iOS apps since 2010, Swift since the day it was announced, and RxSwift since before version 1. He's authored several video courses, tutorials, and articles on iOS app development, presented at numerous conferences, meetups, and online events, and this is his second book. Say hello to Scott on Twitter at @scotteg.

What You Need

To follow along with the tutorials in this book, you'll need the following:

- **A PC running Windows 10 or a recent Linux such as Ubuntu 18.04 LTS, or a Mac running the latest point release of macOS Mojave or later**: You'll need one of these to be able to install the latest versions of IntelliJ IDEA and Android Studio.
- **IntelliJ IDEA Community 2019.1 or later**: IntelliJ IDEA is the IDE upon which Android Studio is based, and it's used in the book to look at pure Kotlin projects that demonstrate techniques in RxJava. You can download the latest version of IntelliJ IDEA Community for free here: https://www.jetbrains.com/idea/
- **Android Studio 3.3.2 or later**: Android Studio is the main development tool for Android. You can download the latest version of Android Studio for free here: https://developer.android.com/studio
- **An intermediate level** knowledge of Kotlin and Android development. This book is about learning RxJava specifically; to understand the rest of the project code and how the accompanying demo projects work you will need at least an intermediate understanding of Kotlin and the Android SDK.

All the Android sample projects in this book will work just fine in an Android emulator bundled with Android Studio, or you can also use a physical Android device.

Book License

By purchasing *Reactive Programming with Kotlin*, you have the following license:

- You are allowed to use and/or modify the source code in *Reactive Programming with Kotlin* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Reactive Programming with Kotlin* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *Reactive Programming with Kotlin*, available at www.raywenderlich.com".
- The source code included in *Reactive Programming with Kotlin* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Reactive Programming with Kotlin* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action or contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

B Book Source Code & Forums

This book comes with the source code for the starter and completed projects for each chapter. These resources are shipped with the digital edition you downloaded from https://store.raywenderlich.com/products/reactive-programming-with-kotlin.

We've also set up an official forum for the book at forums.raywenderlich.com. This is a great place to ask questions about the book or to submit any errors you may find.

Chapter 1: Hello, RxJava!

By Alex Sullivan & Marin Todorov

This book aims to introduce you, the reader, to the RxJava, RxKotlin and RxAndroid libraries and to writing reactive Android apps with Kotlin.

RxJava and RxKotlin

You may be asking yourself "Wait, why am I reading about Rx**Java** when I'm using Kotlin to build Android apps?" Great question! RxJava has been around since 2013, well before Kotlin began to be accepted as a mainstream programming language, and is part of a long list of **Rx**-based libraries written for different platforms and systems. Since Kotlin has such excellent interopability with Java, it wouldn't make sense to completely rewrite RxJava for Kotlin — you can just use the existing RxJava library instead!

However, just because RxJava doesn't need to be completely rewritten to work in Kotlin doesn't mean that it couldn't benefit from all of the great features in the Kotlin programming language.

That's where **RxKotlin** comes into play. **RxKotlin** is a library that expands RxJava by adding a ton of utilities and extension methods that make working with RxJava much more pleasant in Kotlin. However, you absolutely do not need RxKotlin to use the RxJava library in a Kotlin-based Android app.

But what exactly *is* RxJava? Here's a good definition:

> ***RxJava** is a library for composing asynchronous and event-based code by using observable sequences and functional style operators, allowing for parameterized execution via schedulers.*

Sound complicated? Don't worry if it does. Writing reactive programs, understanding the many concepts behind them and navigating a lot of the relevant, commonly used lingo might be intimidating — especially if you try to take it all in at once, or when you haven't been introduced to it in a structured way.

That's the goal of this book: to gradually introduce you to the various RxJava APIs and Rx concepts by explaining how to use each of the APIs, and then covering their practical usage in Android apps.

You'll start with the basic features of RxJava, and then gradually work through intermediate and advanced topics. Taking the time to exercise new concepts extensively as you progress will make it easier to master RxJava by the end of the book. Rx is too broad of a topic to cover completely in a single book; instead, we aim to give you a solid understanding of the library so that you can continue developing Rx skills on your own.

We still haven't quite established what RxJava *is* though, have we? Start with a simple, understandable definition and progress to a better, more expressive one as we waltz through the topic of reactive programming later in this chapter.

> ***RxJava,*** *in its essence, simplifies developing asynchronous programs by allowing your code to react to new data and process it in a sequential, isolated manner. In other words, RxJava lets you observe sequences of asynchronous events in an app and respond to each event accordingly. Examples are taps by a user on the screen and listening for the results of asynchronous network calls.*

As an Android app developer, this should be much more clear and tell you more about what RxJava is, compared to the first definition you read earlier in this chapter.

Even if you're still fuzzy on the details, it should be clear that RxJava helps you write asynchronous code. And you know that developing good, deterministic, asynchronous code is *hard*, so any help is quite welcome!

Introduction to asynchronous programming

If you tried to explain asynchronous programming in a simple, down-to-earth language, you might come up with something along the lines of the following:

An Android app, at any moment, might be doing any of the following things and more:

- Reacting to button taps
- Animating a view across the screen
- Downloading a large photo from the internet
- Saving bits of data to disk
- Playing audio

All of these things seemingly happen at the same time. Whenever the keyboard animates out of the screen, the audio in your app doesn't pause until the animation has finished, right?

All the different bits of your program don't block each other's execution. Android offers you several different APIs that allow you to perform different pieces of work on different threads and perform them across the different cores of the device's CPU.

Writing code that truly runs in parallel, however, is rather complex, especially when different bits of code need to work with the same pieces of data. It's hard to determine which piece of code updates the data first or which code has read the latest value.

Android asynchronous APIs

Google has provided several different APIs that help you write asynchronous code. You've probably used a few of them before, and chances are they left you leaving a bit frustrated or maybe even *scared*.

You've probably used at least one of the following:

- **`AsyncTask`**: To do some work on the background and then update elements in your UI with the result of that background work. You have to make sure to properly handle cancelling a running `AsyncTask` when your `Activity` or `Fragment` shuts down, since you could otherwise get a `NullPointerException` when the `AsyncTask` tries to update UI elements that don't exist anymore.

- **`IntentService`**: To start a fire-and-forget background job using an Intent. You typically use an `IntentService` if you want to do some work that doesn't need to touch the UI at all — saving an object to a database, for example.
- **`Thread`**: To start background work in a purely Java way without interacting with any Android APIs. `Threads` come with the downside of being expensive and not bound to any sort of `ThreadPool`.
- **`Future`**: To clearly chain work which will complete at some undetermined point in the future. `Futures` are considerably clearer to use than `AsyncTasks`, but run into some of the same problems around null pointers when a `Fragment` or `Activity` has been destroyed.

The above isn't an exhaustive list — there's also `Handler`, `JobScheduler`, `WorkManager`, `HandlerThread` and **Kotlin coroutines**.

Coroutines and RxJava

Now that Kotlin coroutines have started to become popular in the Android development world, you may be asking yourself if it's still worthwhile to learn about RxJava.

Many comparisons have been made between using RxJava and using coroutines for Android development. Each review will give you a different answer about which tool you should use.

In reality, RxJava and coroutines work at different levels of abstractions. Coroutines offer a more lightweight approach to threading and allow you to write asynchronous code in a synchronous manner. Rx, on the other hand, is used primarily to create the event-driven architecture mentioned above, and to allow you to write reactive applications. So, while they both offer an answer for doing asynchronous work off the main thread, they're really different tools that are both useful depending on the context.

If you're simply looking for an easy way to replace `AsyncTask`, then coroutines may make more sense than pulling RxJava into your application. However, if you do want to move towards a reactive, event-driven architecture, then RxJava is your best bet!

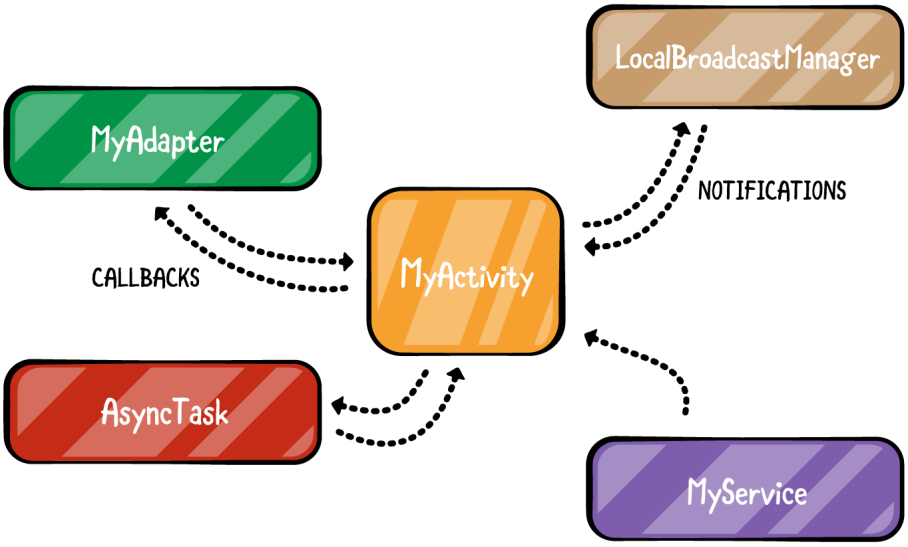
Asynchronous programming challenges

Since most of your typical classes would do something asynchronously, and all UI components are inherently asynchronous, it’s impossible to make assumptions in what order the **entirety** of your app code will get executed.

After all, your app's code runs differently depending on various external factors, such as user input, network activity, or other OS events. Each time the user fires up your app, the code may run in a completely different order depending on those external factors. (Well, except for the case when you have an army of robots testing your app, then you can expect all events to happen with precise, kill-bot synchronization.)

We're definitely not saying that writing good asynchronous code is impossible. After all, there's a litany of tools — like the ones listed above — that Android developers have been using to write asynchronous apps since well before RxJava hit the scene.

The issue is that complex asynchronous code becomes very difficult to write in part because of the variety of APIs that you as an Android developer will end up using:



You may be using an `AsyncTask` to update your UI, an `IntentService` to save something to a database, a `WorkManager` task to sync your app to a server, and other various asynchronous APIs. Since there is no universal language across all the asynchronous APIs, reading and understanding the code, and reasoning about its execution, becomes difficult.

To wrap up this section and put the discussion into a bit more context, you'll compare two pieces of code: one synchronous and one asynchronous.

Synchronous code

Performing an operation for each element of a list is something you've done plenty of times. It's a very simple yet solid building block of app logic because it guarantees two things: It executes **synchronously**, and the collection is **immutable** from the outside world while you iterate over it.

Take a moment to think about what this implies. When you iterate over a collection, you don't need to check that all elements are still there, and you don't need to rewind back in case another thread inserts an element at the start of the collection. You assume you always iterate over the collection in *its entirety* at the beginning of the loop.

If you want to play a bit more with these aspects of the `for` loop, try this in an app or IntelliJ IDEA project:

```
var list = listOf(1, 2, 3)
for (number in list) {
  println(number)
  list = listOf(4, 5, 6)
}
print(list)
```

Is `list` mutable inside the `for` body? Does the collection that the loop iterates over ever change? What's the sequence of execution of all commands? Can you modify `number` if you need to? You may be surprised by what you see if you run this code.

Asynchronous code

Consider similar code, but assume each iteration happens as a reaction to a click on a button. As the user repeatedly clicks on the button, the app prints out the next element in a list:

```
var list = listOf(1, 2, 3)
var currentIndex = 0
button.setOnClickListener {
  println(list[currentIndex])

  if (currentIndex != list.lastIndex) {
    currentIndex++
  }
}
```

Think about this code in the same context as you did for the previous one. As the user clicks the button, will that print all of the list's elements? You really can't say. Another piece of asynchronous code might remove the last element, *before* it's been printed.

Or another piece of code might insert a new element at the start of the collection *after* you've moved on.

Also, you assume only that the click listener will ever change `currentIndex`, but another piece of code might modify `currentIndex` as well — perhaps some clever code you added at some point after crafting the above function.

You've likely realized that some of the core issues with writing asynchronous code are: a) the order in which pieces of work are performed and b) shared mutable data.

These are some of RxJava's strong suits!

Next, you need a good primer on the language that will help you start understanding how RxJava works, what problems it solves, and ultimately let you move past this gentle introduction and into writing your first Rx code in the next chapter.

Asynchronous programming glossary

Some of the language in RxJava is so tightly bound to asynchronous, reactive and/or functional programming that it will be easier if you first understand the following foundational terms.

In general, RxJava tries to address the following aspects of app development:

1. State, and specifically, shared mutable state

State is somewhat difficult to define. To understand state, consider the following practical example.

When you start your laptop it runs just fine, but after you use it for a few days or even weeks, it might start behaving weirdly or abruptly hang and refuse to speak to you. The hardware and software remains the same, but what's changed is the state. As soon as you restart, the same combination of hardware and software will work just fine once more.

The data in memory, the data stored on disk, all the artifacts of reacting to user input, all traces that remain after fetching data from cloud services — the sum of these and more is the state of your laptop.

Managing the state of your Android apps, especially when shared between multiple asynchronous components, is one of the issues you'll learn how to handle in this book.

2. Imperative programming

Imperative programming is a programming paradigm that uses statements to change the program's state. Much like you would use imperative language while playing with your dog — "Fetch! Lay down! Play dead!" — you use imperative code to tell the app exactly *when* and *how* to do things.

Imperative code is similar to the code that your computer understands. All the CPU does is follow lengthy sequences of simple instructions. The issue is that it gets challenging for humans to write imperative code for complex, asynchronous apps — especially when shared, mutable state is involved.

For example, take this code, found in `onCreate()` of an Android Activity:

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)

  setupUI()
  bindClickListeners()
  createAdapter()
  listenForChanges()
}
```

There's no telling what these methods do. Do they update properties of the Activity itself? More disturbingly, are they called in the right order? Maybe somebody inadvertently swapped the order of these method calls and committed the change to source control. Now the app might behave differently due to the swapped calls.

3. Side effects

Now that you know more about mutable state and imperative programming, you can pin down most issues with those two things to **side effects**.

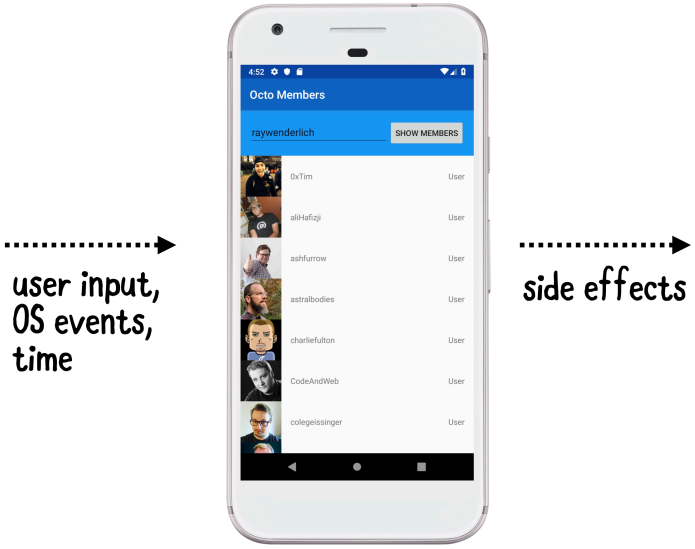
Side effects are any change to the state outside of the current scope. For example, consider the piece of code in the example above. `bindClickListeners()` probably attaches some kind of event handlers to some widgets. This causes a side effect, as it changes the state of the view: the app behaves one way *before* executing `bindClickListeners()`, and differently *after* that.

Side effects are also defined at the level of individual functions in your code. If a function modifies any state other than the local variables defined inside the function, then the function has introduced a side effect.

Any time you modify data stored on disk or update the text of a `TextView` on screen, you cause side effects.

Side effects are not bad in themselves. After all, causing side effects is the ultimate goal of *any* program! You need to change the state of the world somehow after your program has finished executing.

Running for a while and doing nothing makes for a pretty useless app.



The issue with producing side effects is doing it in a controlled way. You need to be able to determine which pieces of code cause side effects, and which simply process and output data.

RxJava tries to address the issues (or problems) listed above by utilizing the remaining two concepts.

4. Declarative code

In imperative programming, you change state at will. An alternative style of programming to imperative is **functional programming**. In functional code, you don't cause any side effects.

Since we don't live in a perfect world, the balance lies somewhere in the middle of these two extremes. RxJava combines some of the best aspects of imperative code and functional code.

In addition to not causing side effects, functional code tends to be **declarative**. Code is declarative when it focuses on the *what* that you want to do, instead of the *how* that encompasses the imperative way of programming. Declarative code lets you define pieces of behavior, and RxJava will run these behaviors any time there's a relevant event and then provide the behaviors an immutable, isolated data input to work with.

By programming declaratively, you can work with asynchronous code, but make the same assumptions as in a simple `for` loop: that you're working with immutable data and you can execute code in a sequential, deterministic way.

5. Reactive systems

"Reactive systems" is a rather abstract term and covers web or Android apps that exhibit most or all of the following qualities:

- **Responsive**: Always keep the UI up to date, representing the latest app state.
- **Resilient**: Each behavior is defined in isolation and provides for flexible error recovery.
- **Elastic**: The code handles varied workload, often implementing features such as lazy pull-driven data collections, event throttling, and resource sharing.
- **Message driven**: Components use message-based communication for improved reusability and isolation, decoupling the lifecycle and implementation of classes.

In short, reactive systems *react* to user and other events in a flexible and coherent fashion.

The terms and concepts defined above are just the start of your RxJava vocabulary. You'll see more terms as you progress through the book. Now that you have a start on understanding the problems RxJava helps solve and how it approaches these issues, it's time to talk about the building blocks of Rx and how they play together.

Foundations of RxJava

Reactive programming isn't a new concept; it's been around for a fairly long time, but its core concepts have made a noticeable comeback over the last decade.

In that period, web applications have became more involved and are facing the issue of managing complex asynchronous UIs. On the server side, reactive systems (as described above) have become a necessity.

A team at Microsoft took on the challenge of solving the problems of asynchronous, scalable, real-time application development that we've discussed in this chapter. They worked on a library, independently from the core teams in the company, and sometime around 2009, offered a new client and server-side framework called Reactive Extensions for .NET (Rx).

It was an installable add-on for .NET 3.5 and later became a built-in core library in .NET 4.0. It's been an open-source component since 2012. Open sourcing the code permitted other languages and platforms to reimplement the same functionality, which turned Rx into a cross-platform standard.

Today you have RxJS, RxSwift, Rx.NET, RxScala, RxJava, and more. All these libraries strive to implement the same behavior and same expressive APIs. Ultimately, a developer creating an Android app with RxJava can freely discuss app logic with another programmer using RxJS on the web or RxSwift on iOS.

Like the original Rx, RxJava works with all the concepts you've covered so far: It tackles mutable state, it allows you to compose event sequences and improves on architectural concepts such as code isolation, reusability and decouplings.

Let's revisit that definition:

> ***RxJava*** *finds the sweet spot between traditionally imperative Java/Kotlin code and purist functional code. It allows you to react to events by using immutable code definitions to asynchronously process pieces of input in a deterministic, composable way.*

You can read more about the family of Rx implementations at http://reactivex.io. This is the central repository of documentation about Rx's operators and core classes. It's also probably the first place you'll notice the Rx logo, the electric eel:

> **Note**: I personally thought for some time that it was a piece of seaweed, but research shows that it is, in fact, an electric eel. (The Rx project used to be called Volta.)

In this book, you are going to cover both the cornerstone concepts of developing with RxJava as well as real-world examples of how to use them in your apps.

The three building blocks of Rx code are **observables**, **operators** and **schedulers**. The sections below cover each of these in detail.

Observables

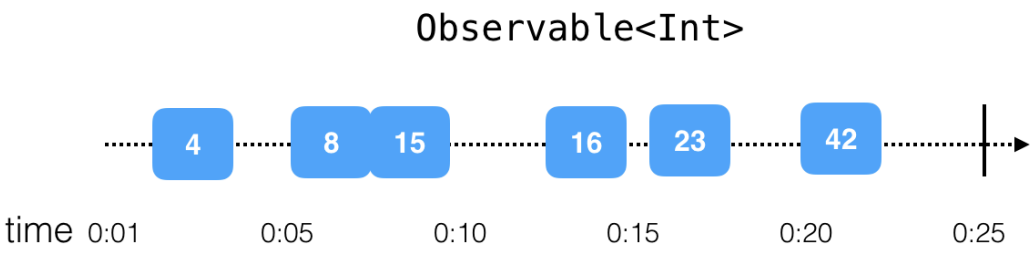
The `Observable<T>` class provides the foundation of Rx code: the ability to asynchronously produce a sequence of events that can "carry" an immutable snapshot of data `T`. In the simplest words, it allows classes to subscribe for values emitted by another class over time.

The `Observable<T>` class allows one or more observers to react to any events in real time and update the app UI, or otherwise process and utilize new and incoming data.

The `ObservableSource<T>` Interface (which the `Observable<T>` class implements) is extremely simple. An `Observable` can emit (and observers can receive) only three types of events:

- **A `next` event**: An event which "carries" the latest (or *next*) data value. This is the way observers "receive" values.
- **A `complete` event**: This event terminates the event sequence with success. It means the `Observable` completed its life-cycle successfully and won't emit any other events.
- **An `error` event**: The `Observable` terminates with an error and will not emit other events.

When talking about asynchronous events emitted over time, you can visualize an observable sequence of integers on a timeline, like so:

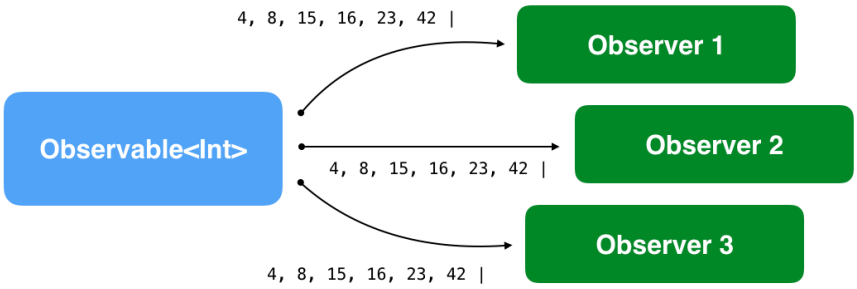


The blue boxes are the `next` events being emitted by the `Observable`. The vertical bar on the right represents the `complete` event. An `error` event would be represented by an x on the timeline.

This simple contract of three possible events an `Observable` can emit is anything and everything in Rx. Because it is so universal, you can use it to create even the most complex app logic.

Because the observable contract does not make any assumptions about the nature of the `Observable` or the `Observer`, using event sequences is the ultimate decoupling practice.

You don't ever need to use callbacks to allow your classes to talk to each other.



To get an idea about some real-life situations, you'll look at two different kinds of observable sequences: **finite** and **infinite**.

Finite observable sequences

Some observable sequences emit zero, one or more values, and, at a later point, either terminate successfully or terminate with an error.

In an Android app, consider code that downloads a file from the internet:

- First, you start the download and start observing for incoming data.
- Then you repeatedly receive chunks of data as parts of the file come in.
- In the event the network connection goes down, the download will stop and the connection will time-out with an error.
- Alternatively, if the code downloads all the file's data, it will complete with success.

This workflow accurately describes the lifecycle of a typical observable. Take a look at the related code below:

```
API.download(file = "http://www...")
  .subscribeBy(
    onNext = {
      // append data to a file
    },
    onComplete = {
      // use downloaded file
    },
    onError = {
      // display error to user
    }
  )
```

`API.download()` returns an `Observable<String>` instance, which emits `String` values as chunks of data come over the network. Calling `subscribeBy` tells the observable that you'd like to subscribe for events that you're going to provide lambdas for.

You subscribe to `next` events by providing the `onNext` lambda. In the downloading example, you append the data to a temporary file stored on disk.

You subscribe to an `error` event by providing the `onError` lambda. In the lambda, you can display a `Throwable.message` in an alert box or do something else.

Finally, to handle a `complete` event, you provide the `onComplete` lambda, where you can do something like start a new Activity to display the downloaded file or anything else your app logic dictates.

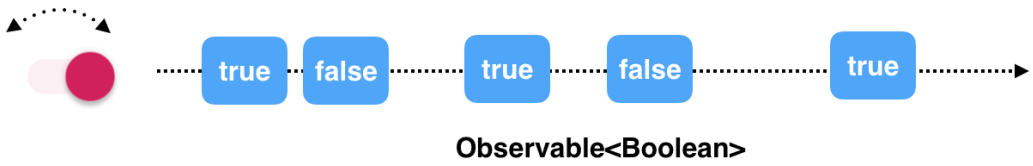
Infinite observable sequences

Unlike file downloads or similar activities, which are supposed to terminate either naturally or forcefully, there are other sequences which are simply infinite. Often, UI events are such infinite observable sequences.

For example, consider the code you need to react to a `Switch` being toggled in your app:

- You add an `OnCheckedChangedListener` to the switch you want to listen to.
- You then need to provide a lambda callback to the `OnCheckedChangeListener`. It looks at the `isChecked` value and updates the app state accordingly.

This sequence of switch checked changes does not have a natural end. As long as there is a switch on the screen, there is a possible sequence of switch checked changes. Further, since the sequence is virtually infinite, you always have an initial value at the time you start observing it — namely, whether the switch is on or off.



It may happen that the user never toggles the switch, but that doesn't mean the sequence of events is terminated. It just means that there were no events emitted.

In RxJava, you could write code like this to react to the switch changing:

```
switch.checkedChanges()
  .subscribeBy(
    onNext = { isOn ->
      if (isOn) {
        // toggle a setting on
      } else {
        // toggle a setting off
      }
```

`checkedChanges()` is a soon-to-be-discovered extension method on `CompoundButton` that produces an `Observable<Boolean>`. (This is very easy to code yourself; you'll learn how in upcoming chapters).

You subscribe to the `Observable` returned from `checkedChanges()` and update the app settings according to the current state of the switch. Note that you skip the `onError` and `onComplete` parameters to `subscribeBy`, since these events will not be emitted from that observable — a switch is either on or it's not.

Operators

`ObservableSource<T>` and the implementation of the `Observable` class include plenty of methods that abstract discrete pieces of asynchronous work, which can be composed together to implement more complex logic.

Because they are highly decoupled and composable, these methods are most often referred to as **operators**. Since these operators mostly take in asynchronous input and only produce output without causing side effects, they can easily fit together, much like puzzle pieces, and work to build a bigger picture.

For example, take the mathematical expression `(5 + 6) * 10 - 2`.

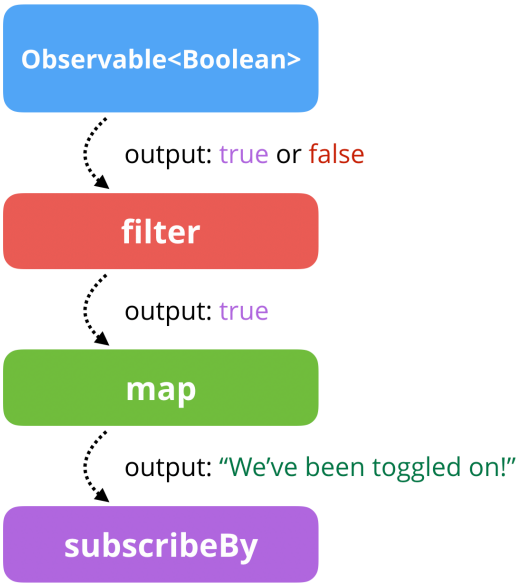
In a clear, deterministic way, you can apply the operators `*`, `( )`, `+` and `-` in their predefined order to the pieces of data that are their input, take their output and keep processing the expression until it's resolved.

In a somewhat similar manner, you can apply Rx operators to the pieces of input emitted by an `Observable` to deterministically process inputs and outputs until the expression has been resolved to a final value, which you can then use to cause side effects.

Here's the previous example about observing switch changes, adjusted to use some common Rx operators:

```
switch.checkedChanges()
  .filter { it == true }
  .map { "We've been toggled on!" }
  .subscribeBy(
    onNext = { message ->
      updateTextView(message)
    }
  )
```

Each time `checkedChanges()` produces either a `true` or `false` value, Rx will apply the `filter` and `map` operators to that emitted piece of data.



First, `filter` will only let through values that are `true`. If the switch has been toggled off the subscription code will not be executed because `filter` will restrict those values.

In case of `true` values, the `map` operator will take the `Boolean` type input and convert it to a `String` output — the text `"We've been toggled on!"`.

Finally, with `subscribeBy` you subscribe for the resulting `next` event, this time carrying a `String` value, and you call a method to update some text view with that text onscreen.

The operators are also highly **composable** — they always take in data as input and output their result, so you can easily chain them in many different ways, achieving much more than what a single operator can do on its own!

As you work through the book, you will learn about more complex operators that abstract even more-involved pieces of asynchronous work.

Schedulers

Schedulers are similar to the `ThreadPools` that you see in normal Java and Kotlin code. If you're not familiar with `ThreadPools`, you can think of them as a collection of `Threads` that are all joined together and available to use.

RxJava comes with a number of predefined schedulers, which cover 99% of use cases. Hopefully, this means you will never have to go about creating your own scheduler.

In fact, most of the examples in the first half of this book are quite simple and generally deal with observing data and updating the UI, so you won't look into schedulers at all until you've covered the basics.

That being said, schedulers are very powerful.

For example, you can specify that you'd like to observe for `next` events on the `IO` scheduler, which makes your Rx code run on a background thread pool — you may want to use this scheduler if you're downloading files from the network or saving something to a database.

`TrampolineScheduler` will run your code concurrently. The `ComputationScheduler` will allow you to schedule your subscriptions on a separate set of `Threads` that are reserved for heavy lifting computation tasks.

Thanks to RxJava, you can schedule the different pieces of work of the same subscription on different schedulers to achieve the best performance. Even if they sound very interesting and quite handy, don't bother too much with schedulers for now. You'll return to them later in the book.

App architecture

It's worth mentioning that RxJava doesn't alter your app's architecture in any way; it mostly deals with events, asynchronous data sequences and a universal communication contract.

You can create apps with Rx by implementing a normal Model-View-Controller (MVC) architecture. You can also choose to implement a Model-View-Presenter (MVP) architecture or Model-View-ViewModel (MVVM) if that's what you prefer.

In case you'd like to go that way, RxJava is also very useful for implementing your own unidirectional data-flow architecture.

It's important to note that you definitely do *not* have to start a project from scratch to make it a reactive app; you can iteratively refactor pieces of an exiting project or simply use RxJava when appending new features to your app.

The MVVM architecture was originally developed by Microsoft specifically for event-driven software created on platforms which offers data bindings. RxJava and MVVM definitely do play nicely together, and towards the end of this book you'll look into that pattern and how to implement it with RxJava.

The reason MVVM and RxJava go great together is that a ViewModel allows you to

expose `Observable<T>` properties, which you can bind directly to UI widgets in your Activity, or translate them into `LiveData` objects from **Android Jetpack** and then subscribe to those instead. This makes binding model data to the UI very simple to represent, and to code. You'll see how to integrate the use of RxJava with `LiveData` later in the book.

RxAndroid and RxBinding

RxJava is the implementation of the common Rx API. Therefore, it doesn't know anything about any Android-specific classes.

There are two companion libraries that can be used to fill in a few of the gaps between Android and RxJava.

The first is a tiny library called **RxAndroid**. RxAndroid has one specific purpose: to provide a bridge between Androids `Looper` class and RxJava's schedulers. Chances are, you'll use this library simply to receive the results of an `Observable` on the UI thread so that you can update your views.

The second library is a broader library called **RxBinding**. RxBinding provides a large number of utility methods to turn callback-styled view listeners into observables. You actually already saw an example of this library being used, the `checkedChanges()` method used earlier on a `Switch`:

```
switch.checkedChanges()
  .subscribeBy(
    onNext = { boolean ->
      println("Switch is on: $boolean")
    }
  )
```

`checkedChanges()` is an extension method provided by the RxBinding library to turn a normal `CompoundButton` like `Switch` into a stream of on or off states.

RxBinding provides similar bindings for many of the Android view classes, such as listening for clicks on a `Button` and changes to the text in an `EditText`.

Installing RxJava

RxJava is available for free at https://github.com/ReactiveX/RxJava.

RxJava is distributed under the Apache-2.0 license, which, in short, allows you to include the library in free or commercial software, on an as-is basis. As with all other

Apache-2.0 licensed software, the copyright notice should be included in all apps you distribute.

Including `RxJava` in a Gradle-based project, such as an Android app, is simple — just add the following to the `dependencies` block in your module's `build.gradle` file:

```
implementation "io.reactivex.rxjava2:rxjava:2.2.2"
implementation 'io.reactivex:rxkotlin:2.3.0'
```

The first `implementation` line is for `RxJava`. The second is for including the `RxKotlin` extensions. You can omit the `RxJava` import if you include `RxKotlin`, but since the `RxKotlin` library may not include the latest `RxJava` library, it's good practice to include both. You'll generally want to include the latest versions of both libraries.

> **Note**: You may have noticed that the dependency for `RxJava` actually says `rxjava2` in it. There's two major versions of RxJava: RxJava1 and RxJava2. RxJava2 added a lot of useful new tricks and types to the library, and this book will be using RxJava2. You can find some of the differences in the What's different in 2.0 article.

Community

The RxJava project is alive and buzzing with activity, not only because Rx is inspiring programmers to create cool software with it, but also due to the positive nature of the community that formed around this project.

The RxJava community is very friendly, open minded, and enthusiastic about discussing patterns, common techniques, or just helping each other.

You can find channels dedicated to talking about RxJava in both the Android United Slack and the official Kotlin Slack.

The first can be found, here: http://android-united.community/. If you request an invite you should be approved quickly.

The official Kotlin Slack can be found here: https://kotlinlang.slack.com/.

Search for **rx** in both Slacks and you should find what you're looking for!

Both Slacks are friendly and inviting, and people are always available to troubleshoot some particularly tricky Rx code, or to just discuss the latest and greatest in the world of RxJava and RxKotlin.

Key points

- **RxJava** is a library that provides an Rx framework for Java-based projects such as Android apps.
- RxJava can be used even when using the Kotlin language for app development.
- The **RxKotlin** library adds some Kotlin related utilities and extensions on top of RxJava.
- RxJava and all Rx frameworks provide for a way to program using **asynchronous**, **event-based** code.
- RxJava helps you build **reactive systems** in a **declarative** style.
- The main elements you'll use in RxJava are **observables**, **operators**, and **schedulers**.
- The **RxAndroid** and **RxBinding** libraries assist you in using RxJava on Android.

Where to go from here?

This chapter introduced you to many of the problems that RxJava addresses. You learned about the complexities of asynchronous programming, sharing mutable state, causing side effects and more.

You haven’t written any RxJava yet, but you now understand why RxJava is a good idea and you’re aware of the types of problems it solves. This should give you a good start as you work through the rest of the book.

And there is plenty to work through! You’ll start by creating very simple observables and work your way up to complete real-world Android apps using the MVVM architecture.

Move right on to Chapter 2, “Observables”!

Chapter 2: Observables

By Alex Sullivan & Scott Gardner

Now that you're all setup with RxJava, it's time to jump in and start building some observables!

In this chapter, you're going to go over a few different examples of creating and subscribing to observables. Things are going to be pretty theoretical for now, but rest assured that the skills you pick up in this chapter will come in very handy as you start working through real-world projects.

Getting started

You'll work through these theoretical examples of observables using a normal IntelliJ IDEA project. You'll move on to Android Studio projects once you switch to working on real-world Android applications.

Use the **File ▸ Open** command in IntelliJ IDEA to open the root folder of the starter project. Accept the defaults in any pop-ups that occur, and the project will then be opened. You'll primarily be working in the **main.kt** file in the **src/main/kotlin** folder of the project. For now, there's just an empty `main` function. You'll fill it out as you progress through the chapter.

Before you start diving into some RxJava code, take a look at the **SupportCode.kt** file. It contains the following helper function `exampleOf(description: String, action: () -> Unit)`:

```
fun exampleOf(description: String, action: () -> Unit) {
  println("\n--- Example of: $description ---")
  action()
}
```

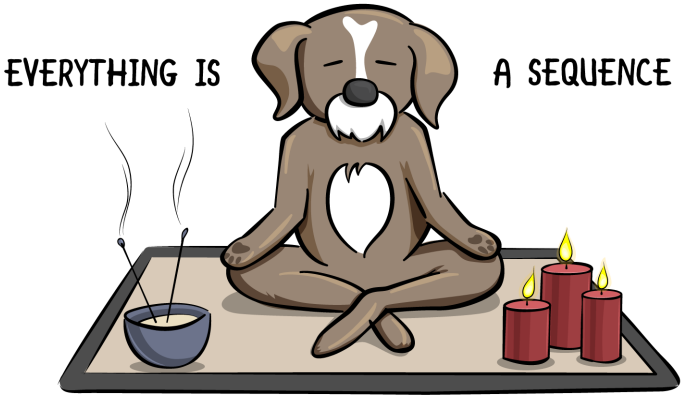
You'll use this function to encapsulate different examples as you work your way through this chapter. You'll see how to use this function shortly.

But, before you get too deep into that, now would probably be a good time to answer the question: What *is* an observable?

Observables are the heart of Rx. You're going to spend some time discussing what observables are, how to create them and how to use them.

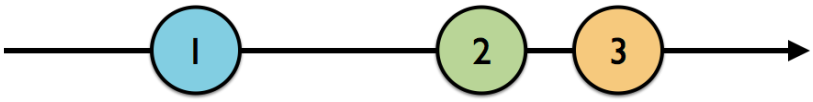
What is an observable?

You'll see "observable," "observable sequence," and "stream" used interchangeably in Rx. And, really, they're all the same thing. In RxJava...



...or something that *works* with a sequence. And an `Observable` is just a sequence with special powers. One of them, in fact the most important one, is that it is *asynchronous*. Observables produce events, the process of which is referred to as *emitting*, over a period of time. Events can contain values, such as numbers or instances of a custom type, or they can be recognized user gestures, such as taps.

One of the best ways to conceptualize this is by using marble diagrams, which are just values plotted on a timeline.

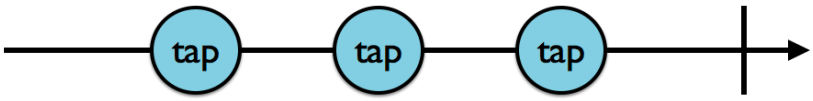


The left-to-right arrow represents time, and the numbered circles represent elements of a sequence. Element 1 will be emitted, some time will pass, and then 2 and 3 will be emitted. How much time, you ask? It could be at *any* point throughout the life of the observable — which brings you to the lifecycle of an observable.

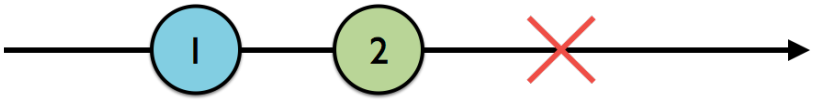
Lifecycle of an observable

In the previous marble diagram, the observable emitted three elements. When an observable emits an element, it does so in what's known as a **next** event.

Here's another marble diagram, this time including a vertical bar that represents the end of the road for this observable.



This observable emits three tap events, and then it ends. This is called a **complete** event, as it's been **terminated**. For example, perhaps the taps were on a view that had been dismissed. The important thing is that the observable has terminated, and it can no longer emit anything. This is normal termination. However, sometimes things can go wrong.



An error has occurred in this marble diagram; it's represented by the red X. The observable emitted an **error** event containing the error. This is no different than when an observable terminates normally with a **complete** event. If an observable emits an **error** event, it is also terminated and can no longer emit anything else.

Here's a quick recap:

- An observable emits **next** events that contain elements. It can continue to do this until it either:
- ...emits an **error** event and is terminated.
- ...emits a **complete** event and is terminated.
- Once an observable is terminated, it can no longer emit events.

Now that you understand what an observable is and what it does, you'll create some observables to see them in action.

Creating observables

Switch back from the current file to **main.kt** and add the code below:

```
exampleOf("just") {
  val observable: Observable<Int> = Observable.just(1)
}
```

In the code above, you used the `just` static method to create an observable with **just** one item: the *Integer* 1.

In Rx, methods that operate on observables are referred to as **operators** — so you just utilized the `just` operator!

`just` is aptly named, since all it does is create an observable sequence containing *just* the provided elements. `just` can take more than one item as well — try updating the previous line to take in a few more items:

```
val observable = Observable.just(1,2,3)
```

This time, you didn't explicitly specify the type. You *might* think that because you gave it several integers, the type is `Observable<List<Int>>`. However, if you hover over the `Observable.just(1,2,3)` expression and click **View ▸ Expression Type** you'll see that the type is actually `Observable<Int>`.

`just` has ten overloaded methods that take a variable number of arguments, each of which are eventually emitted by the observable. If you want to create an observable of type `Observable<List<Int>>`, then you can pass a `List<Int>` into the `just` operator. Replace the observable you previously defined with the following:

```
val observable = Observable.just(listOf(1))
```

Now, hover over the `Observable.just(listOf(1))` expression and click **View ▸ Expression Type** again. You'll see that the type is now `Observable<List<Int>>`. That means that this new observable will emit one item — and that single item will be a list of `Int` values. It can be a little tough to wrap your mind around an observable that emits lists, but with time it will become second nature!

Another operator you can use to create observables is `fromIterable`. Add this code tothe bottom of the `main()` function:

```
exampleOf("fromIterable") {
  val observable: Observable<Int> =
    Observable.fromIterable(listOf(1, 2, 3))
}
```

The `fromIterable` operator creates an observable of individual objects from a regular list of elements. That is, it takes all of the items in the provided list and emits those elements as if you had instead written `Observable.just(1, 2, 3)`.

Hover over the `Observable.fromIterable(listOf(1, 2, 3))` expression and click **View ▸ Expression Type** again. You'll see that the type of this observable is `Observable<Int>` rather than `Observable<List<Int>>`.

`fromIterable` can be handy if you have a list of objects you want to convert into an observable sequence.

The IntelliJ IDEA console is probably looking pretty bare at the moment. That's because you haven't printed anything except the example header. Time to change that by *subscribing* to observables.

Subscribing to observables

As an Android developer, you may be familiar with `LocalBroadcastManager`; it broadcasts notifications to observers, which are different than RxJava `Observables`. Here's an example of of a broadcast receiver that listens for a `custom-event` Intent:

```
LocalBroadcastManager.getInstance(this)
    .registerReceiver(object : BroadcastReceiver() {
  override fun onReceive(context: Context?, intent: Intent?) {
    println("We got an intent!")
  }
}, IntentFilter("custom-event"))
```

Subscribing to an RxJava observable is fairly similar; you call observing an observable *subscribing* to it. So instead of `registerReceiver()`, you use `subscribe()`. Unlike `LocalBroadcastManager`, where developers typically use only the `getInstance()` singleton instance, each observable in Rx is different.

More importantly, an observable won't send events until it has a subscriber. Remember that an observable is really a sequence definition; subscribing to an observable is more like calling `next()` on an `Iterator` in the Kotlin Standard Library:

```
val sequence = 0 until 3
val iterator = sequence.iterator()
while (iterator.hasNext()) {
  println(iterator.next())
}

/* Prints:
0
1
```

```
2
*/
```

Subscribing to observables is more streamlined than this, though. You can also add handlers for each event type an observable can emit. Recall that an observable emits `next`, `error`, and `complete` events. A `next` event passes the element being emitted to the handler, and an `error` event contains a throwable instance.

To see this in action, add this new example to the IntelliJ project (insert the code somewhere *after* the closing curly bracket of the previous example):

```
exampleOf("subscribe") {
  val observable = Observable.just(1, 2, 3)
}
```

This is similar to the previous example, except, this time, you're simply using the `just` operator. Now add this code at the bottom of this example's lambda, to subscribe to the observable:

```
observable.subscribe { println(it) }
```

> **Note**: The console should automatically appear whenever you run the project, but you can manually show it by clicking the **Run** tab in the bottom left of the IntelliJ IDEA window after you run the `main()` function. This is where the `println` statements display their output.

Option-click on the `subscribe` operator, and you'll see that it takes a `Consumer` of type `Int`. `Consumer` is a simple `interface` that has one method, `accept()`, which takes a value and returns nothing. You'll also see that `subscribe` returns a `Disposable`. You'll cover disposables shortly.

The result of this subscription is that each event emitted by the `observable` prints out:

```
--- Example of: subscribe ---
1
2
3
```

You've seen how to create observables of one element and of many elements. But what about an observable of zero elements? The `empty` operator creates an empty observable sequence with zero elements; it will only emit a `complete` event.

Add this new example to the project:

```
exampleOf("empty") {
  val observable = Observable.empty<Unit>()
}
```

An observable must be defined as a specific type if it can't be inferred. So, since `empty` has nothing from which to infer the type, the type must be defined explicitly. In this case, `Unit` is as good as anything else. Add this code to the example to subscribe to it:

```
observable.subscribeBy(
  // 1
  onNext = { println(it) },
  // 2
  onComplete = { println("Completed") }
)
```

You're using a new `subscribeBy` method here instead of the `subscribe` method you used previously. `subscribeBy` is a handy extension method defined in the `RxKotlin` library, which we'll touch on later in the book. Unlike the `subscribe` method you used previously, `subscribeBy` lets you explicitly state what event you want to handle — `onNext`, `onComplete`, or `onError`. If you were to only supply the `onNext` field of `subscribeBy`, you'd be recreating the `subscribe` functionality you used above.

Taking each numbered comment in turn:

1. Explicitly handle the `onNext` callback, just like before.

2. A `complete` event doesn't carry any value, so just print "Completed" instead.

In the console, you'll see that `empty` simply emits a `complete` message:

```
--- Example of: empty ---
Completed
```

But what use is an *empty* observable? Well, they're handy when you want to return an observable that immediately terminates or intentionally has zero values. As opposed to the `empty` operator, the `never` operator creates an observable that doesn't emit anything and *never* terminates. It can be used to represent an infinite duration. Add this example to the project:

```
exampleOf("never") {
  val observable = Observable.never<Any>()

  observable.subscribeBy(
      onNext = { println(it) },
      onComplete = { println("Completed") }
  )
}
```

Nothing is printed, except for the example header. Not even `"Completed"`. How do you know if this is even working? Hang on to that inquisitive spirit until the **Challenges** section of this chapter.

So far, you've been working mostly with observables of explicit variables, but it's also possible to generate an observable from a range of values.

Add this example to the project:

```
exampleOf("range") {
  // 1
  val observable: Observable<Int> = Observable.range(1, 10)

  observable.subscribe {
    // 2
    val n = it.toDouble()
    val fibonacci = ((Math.pow(1.61803, n) - Math.pow(0.61803, n)) /
                     2.23606).roundToInt()
    println(fibonacci)
  }
}
```

Taking it section by section:

1. Create an observable using the `range` operator, which takes a `start` integer value and a `count` of sequential integers to generate.
2. Calculate and print the *nth* Fibonacci number for each emitted element. (The *Fibonacci sequence* is generated by adding each of the previous two numbers in the sequence, starting with 0 and 1.)

There's actually a better place than in the `subscribe` method, to put code that transforms the emitted element. You'll learn about that in Chapter 7, "Transforming Operators."

Except for the `never()` example, up to this point, you've been working with observables that automatically emit a `completed` event and naturally terminate. This permitted you to focus on the mechanics of creating and subscribing to observables, but that swept an important aspect of subscribing to observables under the rug.

It's time to do some housekeeping and deal with that aspect before moving on.

Disposing and terminating

Remember that an observable doesn't do anything until it receives a subscription. It's the subscription that triggers an observable to begin emitting events, up until it emits an `error` or `completed` event and is terminated. You can manually cause an observable to terminate by canceling a subscription to it.

Add this new example to the project:

```
  exampleOf("dispose") {
    // 1
    val mostPopular: Observable<String> = Observable.just("A", "B", "C")
    // 2
    val subscription = mostPopular.subscribe {
      // 3
      println(it)
    }
  }
```

Quite simply:

1. Create an observable of strings.
2. Subscribe to the observable, this time saving the returned `Disposable` as a local constant called `subscription`.
3. Print each emitted `event` in the handler.

To explicitly cancel a subscription, call `dispose()` on it. After you cancel the subscription, or **dispose** of it, the observable in the current example will stop emitting events.

Add this code to the bottom of the example:

```
  subscription.dispose()
```

Managing each subscription individually would be tedious, so RxJava includes a `CompositeDisposable` type. A `CompositeDisposable` holds disposables — typically added using the `add()` method — and will call `dispose()` on all of them when you call `dispose()` on the `CompositeDisposable` itself. Add this new example to the project:

```
  exampleOf("CompositeDisposable") {
    // 1
    val subscriptions = CompositeDisposable()
    // 2
    val disposable = Observable.just("A", "B", "C")
        .subscribe {
          // 3
          println(it)
```

```
        }
    // 4
    subscriptions.add(disposable)
}
```

Here's how this disposable code works:

1. Create a `CompositeDisposable`.
2. Create an observable and disposable.
3. Subscribe to the observable and print out the emitted item.
4. Add the `Disposable` return value from `subscribe` to the `subscriptions CompositeDisposable`.

This is the pattern you'll use most frequently: creating and subscribing to an observable and immediately adding the subscription to a `CompositeDisposable`.

Why bother with disposables at all? If you forget to call `dispose()` on a `Disposable` when you're done with the subscription, or in some other way cause the observable to terminate at some point, you will *probably* leak memory.

If you forget to utilize the `Disposable` returned by calling `subscribe` on an `Observable`, **Android Studio** will make it very clear that something is not right in an Android project!

The create operator

In the previous examples, you've created observables with specific `next` event elements. Another way to specify all events that an observable will emit to subscribers is by using the `create` operator.

Add this new example to the project:

```
exampleOf("create") {

  val disposables = CompositeDisposable()

  Observable.create<String> { emitter ->

  }
}
```

The `create` operator takes a single parameter named `source`. Its job is to provide the implementation of calling `subscribe` on the observable. In other words, it defines all the events that will be emitted to subscribers. **Command-click** on `create` to see it's definition:

```
 * @param <T> the element type
 * @param source the emitter that is called when an Observer subscribes to the returned {@code Observable}
 * @return the new Observable instance
 * @see ObservableOnSubscribe
 * @see ObservableEmitter
 * @see Cancellable
 */
@CheckReturnValue
@SchedulerSupport(SchedulerSupport.NONE)
public static <T> Observable<T> create(ObservableOnSubscribe<T> source) {
    ObjectHelper.requireNonNull(source,  message: "source is null");
    return RxJavaPlugins.onAssembly(new ObservableCreate<T>(source));
}

/**
 * Returns an Observable that calls an ObservableSource factory to create an ObservableSource for each new Observer
 * that subscribes. That is, for each subscriber, the actual ObservableSource that subscriber observes is
```

The `source` parameter is an `ObservableOnSubscribe<T>`. `ObservableOnSubscribe` is a SAM (Single Abstract Method) interface that exposes one method — `subscribe`. That `subscribe` method takes in an `Emitter<T>`, which has a few methods that you'll use to build up the actual `Observable`. Specifically, it has `onNext`, `onComplete`, and `onError` methods that you can invoke.

Change the implementation of `create` to the following:

```
val observable = Observable.create<String> { emitter ->
    // 1
    emitter.onNext("1")

    // 2
    emitter.onComplete()

    // 3
    emitter.onNext("?")
}
```

Here's the play by play:

1. Emit the string `1` via the `onNext` method.
2. Emit a `completed` event.
3. Emit another string `?` via the `onNext` method again.

Do you think the second `onNext` element (`?`) could ever be emitted to subscribers? Why or why not?

To see if you guessed correctly, subscribe to the observable by adding the following code on the next line after the `create` implementation:

```
.subscribeBy(
    onNext = { println(it) },
    onComplete = { println("Completed") },
    onError = { println(it) }
)
```

You've subscribed to the observable. The result is that the first `next` event element and `"Completed"` print out. The second `next` event doesn't print because the observable emitted a `completed` event and terminated before it.

```
 --- Example of: create ---
1
Completed
```

Add the following line of code between the `emitter.onNext` and `emitter.onComplete` calls:

```
 emitter.onError(RuntimeException("Error"))
```

The observable emits the error and then is terminated.

```
--- Example of: create ---
1
Error
```

What would happen if you emitted neither a `completed` nor an `error` event? Comment out the `onComplete` and `onError` lines of code to find out.

Here's the complete implementation:

```
exampleOf("create") {
  Observable.create<String> { emitter ->
    // 1
    emitter.onNext("1")

//    emitter.onError(RuntimeException("Error"))
    // 2
//    emitter.onComplete()

    // 3
    emitter.onNext("?")
  }.subscribeBy(
      onNext = { println(it) },
      onComplete = { println("Completed") },
      onError = { println("Error") }
  )
}
```

Congratulations, you've just leaked memory! The observable will never finish, and since you never disposed of the `Disposable` returned by `Observable.create` the sequence will never be canceled.

```
 --- Example of: create ---
1
?
```

Feel free to uncomment the line adding the `complete` event or dispose of the returned `Disposable` if you can't stand leaving the code in a leaky state.

Creating observable factories

Rather than creating an observable that waits around for subscribers, it's possible to create observable factories that vend a new observable to each subscriber.

Add this new example to the project:

```
exampleOf("defer") {

  val disposables = CompositeDisposable()
  // 1
  var flip = false
  // 2
  val factory: Observable<Int> = Observable.defer {
    // 3
    flip = !flip
    // 4
    if (flip) {
      Observable.just(1, 2, 3)
    } else {
      Observable.just(4, 5, 6)
    }
  }
}
```

Here's the explanation:

1. Create a `Boolean` flag to flip which observable to return.
2. Create an observable of `Int` factory using the `defer` operator.
3. Invert `flip`, which will be used each time `factory` is subscribed to.
4. Return different observables based on whether `flip` is `true` or `false`.

Externally, an observable factory is indistinguishable from a regular observable. Add this code to the bottom of the example to subscribe to `factory` four times:

```
for (i in 0..3) {
  disposables.add(
      factory.subscribe {
        println(it)
      }
  )
}

disposables.dispose()
```

Each time you subscribe to `factory`, you get the opposite observable. You get `123`, then `456`, and the pattern repeats each time a new subscription is created:

```
 --- Example of: defer ---
1
2
3
4
5
6
1
2
3
4
5
6
```

Using other observable types

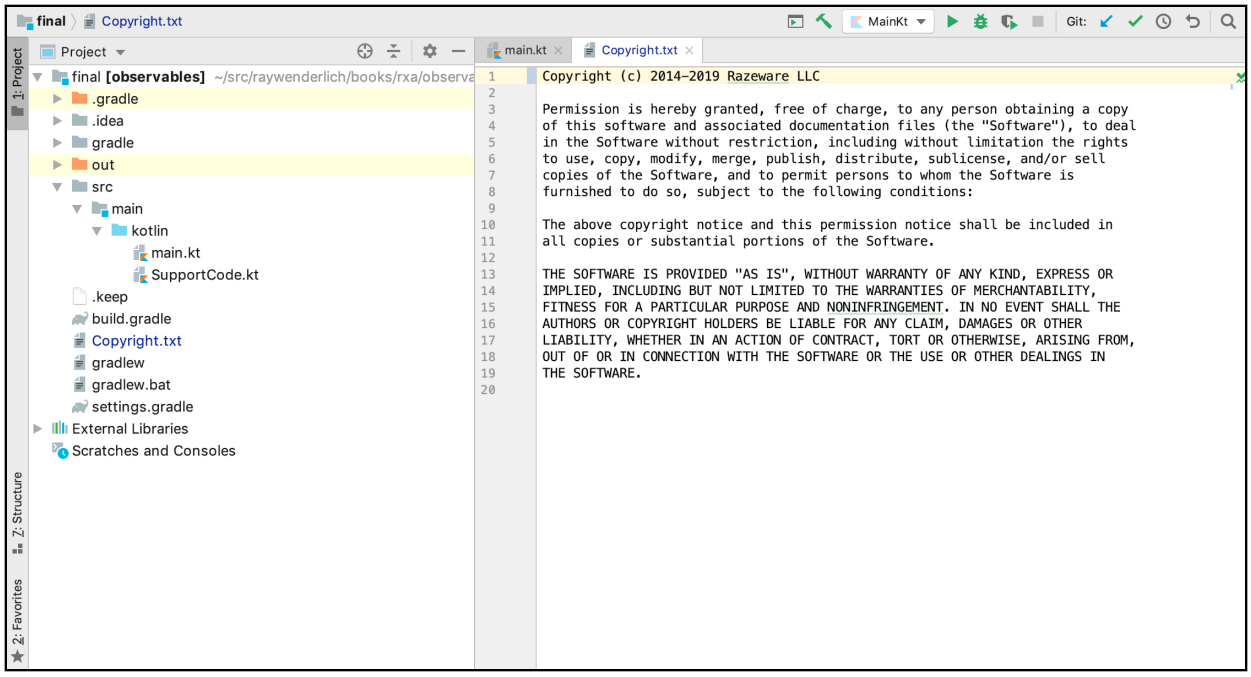
In addition to the normal `Observable` type, there are a few other types of observables with a narrower set of behaviors than regular observables. Their use is optional; you can use a regular observable anywhere you might use one of these specialized observables. Their purpose is to provide a way to more clearly convey your intent to readers of your code or consumers of your API. The context implied by using them can help make your code more intuitive.

There are three special types of observables in RxJava: `Single`, `Maybe` and `Completable`. Without knowing anything more about them yet, can you guess how each one is specialized?

- `Singles` will emit either a `success(value)` or `error` event. `success(value)` is actually a combination of the `next` and `completed` events. This is useful for one-time processes that will either succeed and yield a value or fail, such as downloading data or loading it from disk.

- A `Completable` will only emit a `completed` or `error` event. It doesn't emit any value. You could use a `Completable` when you only care that an operation completed successfully or failed, such as a file write.
- And `Maybe` is a mash-up of a `Single` and `Completable`. It can either emit a `success(value)`, `completed`, or `error`. If you need to implement an operation that could either succeed or fail, and optionally return a value on success, then `Maybe` is your ticket.

You'll have an opportunity to work more with these special observable types in Chapter 4, "Observables and Subjects in Practice," and beyond. For now, you'll run through a basic example of using a `Single` to load some text from a text file named **Copyright.txt** in the **src** folder of the project, because who doesn't love some legalese once in a while?



Add this example to `main()`:

```
exampleOf("Single") {
  // 1
  val subscriptions = CompositeDisposable()
  // 2
  fun loadText(filename: String): Single<String> {
    // 3
    return Single.create create@{ emitter ->

    }
  }
}
```

Here's what you do in this code:

1. Create a composite disposable to use later.
2. Implement a function to load text from a file on disk that returns a `Single`.
3. Create and return a `Single`.

Add this code inside the `create` lambda to complete the implementation:

```
// 1
val file = File(filename)
// 2
if (!file.exists()) {
  emitter.onError(FileNotFoundException("Can't find $filename"))
  return@create
}
// 3
val contents = file.readText(Charsets.UTF_8)
// 4
emitter.onSuccess(contents)
```

From the top:

1. Create a new `File` from the filename.
2. If the file doesn't exist, emit a `FileNotFoundException` via the `onError` method and return from the `create` method.
3. Get the data from the file.
4. Emit the contents of the file.

Now you can put this function to work. Add this code to the example:

```
// 1
val observer = loadText("Copyright.txt")
    // 2
    .subscribeBy(
        // 3
        onSuccess = { println(it) },
        onError = { println("Error, $it") }
    )

subscriptions.add(observer)
```

Here, you:

1. Call `loadText()`, passing the root name of the text file.
2. Subscribe to the `Single` it returns.

3. Pass `onSuccess` and `onError` lambdas to the `subscribeBy` method, either printing the contents of the file or printing the error.

If you run the example, you should see the text from the file printed to the console, the same as the copyright comment at the top of the project:

```
 --- Example of: Single ---
Copyright (c) 2014-2019 Razeware LLC
...
```

Try changing the filename to something else, and you should get the file not found exception printed instead.

Key points

- Everything is a **sequence** in RxJava, and the primary sequence type is `Observable`.
- Observables start emitting when they are **subscribed** to.
- You must **dispose** of subscriptions when done with them, and you'll often use a `CompositeDisposable` to do so.
- `Single`, `Completable` and `Maybe` are specialized observable types that are handy in certain situations.

Challenges

Practice makes *permanent*. By completing challenges in the book, you'll practice what you've learned in each chapter and pick up a few more tidbits of knowledge about working with observables. A starter project as well as a finished version are provided for each challenge. Enjoy!

Challenge: Perform side effects

In the `never` operator example earlier, nothing printed out. That was before you were adding your subscriptions to composite disposables, but if you *had* added it to one, you could've used a handy operator to print a message when the disposable was disposed.

Operators that begin with `doOn`, such as the `doOnDispose` operator, allows you to insert **side effects**; that is, you add handlers that take some action but that won't affect the observable. For `doOnDispose`, that is whenever the disposable is disposed of.

There's a few other handy `doOn` methods that you can use. There's a `doOnNext` method, a `doOnComplete` method, a `doOnError` method and a `doOnSubscribe` method that you can also use to perform some side effect at the right moment.

To complete this challenge, insert the `doOnSubscribe` operator in the `never` example. Feel free to include any of the other handlers if you'd like; they work just like `doOnSubscribe`'s handler does.

And while you're at it, create a composite disposable and add the subscription to it.

Don't forget you can always peek into the finished challenge project for "inspiration."

Where to Go From Here?

We hope you enjoyed this sample of *Reactive Programming with Kotlin: Learn Rx with RxJava, RxKotlin, and RxAndroid*!

If you enjoyed this sample, be sure to check out the full book, which will contain the following chapters:

1. **Hello RxJava!**: Learn about the reactive programming paradigm and what RxJava can bring to your app.
2. **Observables**: Now that you're ready to use RxJava and have learned some of the basic concepts, it's time to play around with observables.
3. **Subjects**: In this chapter, you're going to learn about the different types of subjects in RxJava, see how to work with each one and why you might choose one over another based on some common use cases.
4. **Observables & Subjects in Practice**: In this chapter, you'll use RxJava and your new observable super-powers to create an app that lets users to create nice photo collages — the reactive way.
5. **Filtering Operators**: This chapter will teach you about RxJava's filtering operators that you can use to apply conditional constraints to "next" events, so that the subscriber only receives the elements it wants to deal with.
6. **Filtering Operators in Practice**: In the previous chapter, you began your introduction to the functional side of RxJava. In this chapter, you're going to try using the filtering operators in a real-life app.
7. **Transforming Operators**: In this chapter, you're going to learn about one of the most important categories of operators in RxJava: transforming operators.

8. **Transforming Operators in Practice:** In this chapter, you'll take an existing app and add RxJava transforming operators as you learn more about `map` and `flatMap`, and in which situations you should use them in your code.
9. **Combining Operators**: This chapter will show you several different ways to assemble sequences, and how to combine the data within each sequence.
10. **Combining Operators in Practice**: You'll get an opportunity to try some of the most powerful RxJava operators. You'll learn to solve problems similar to those you'll face in your own applications.
11. **Time-Based Operators**: Managing the time dimension of your sequences is important. To learn about time-based operators, you'll practice with an animated app that visually demonstrates how data flows over time.
12. **Error Handling in Practice**: Even the best RxJava developers can't avoid encountering errors. You'll learn how to deal with errors, how to manage error recovery through retries, or just surrender yourself to the universe and letting the errors go.
13. **Intro to Schedulers**: This chapter will cover the beauty behind schedulers, where you'll learn why the Rx abstraction is so powerful and why working with asynchronous programming is far less less painful than using locks or queues.
14. **Flowables & Back Pressure**: Observables are very powerful, but what happens if a subscriber can't keep up with the `next` events? You'll see how to handle this situation using Flowables.
15. **Testing RxJava Code**: Testing your code is at the heart of writing good software — RxJava comes with lots of nifty tricks for testing everything under the sun.
16. **Creating Custom Reactive Extensions**: Beyond using the elements made available directly by RxJava, you can also create RxJava wrappers around existing non-Rx frameworks. You'll learn how to create and incorporate such wrappers into your reactive application.
17. **RxBinding**: You'll learn how the extremely handy library RxBinding takes care of making reactive bindings for the Android View classes, and see how to use RxBinding in an app.
18. **Retrofit**: In earlier chapters, you've used Retrofit to add networking to your reactive apps. In this chapter, explore exactly how Retrofit interfaces with the Rx world and see how you can take advantage of all that it offers.

19. **RxPreferences**: The RxPreferences library provides a reactive wrapper around SharedPreferences. In this chapter, you'll learn how the library works and how you can use it to effectively stream preference changes.
20. **RxPermissions**: There's a fantastic library called RxPermissions that you'll use in this chapter to help alleviate the pain points of asking the user for permissions at runtime, giving you a reactive flow when requesting permissions.
21. **RxJava & Jetpack**: Android Jetpack is a suite of libraries provided by the Android team to make developing Android apps a breeze. You've already seen ViewModel and LiveData used with RxJava. In this chapter, you'll explore using the Room and Paging Library components from Jetpack in a reactive app.
22. **Building a Complete RxJava App**: To conclude this book, you'll architect and code a small RxJava application. The goal is not to use Rx "at all costs", but rather to make design decisions that lead to a tidy architecture with stable, predictable and modular behavior. The application is simple by design, to clearly present ideas you can use to architect your own applications.

You can find the book on the raywenderlich.com store here: https://store.raywenderlich.com/products/reactive-programming-with-kotlin

We hope you enjoy the book!

— The *Reactive Programming with Kotlin: Learn Rx with RxJava, RxKotlin, and RxAndroid* team

Learn Reactive Programming in Kotlin with RxJava!

The popularity of reactive programming continues to grow on an ever-increasing number of platforms and languages. Rx lets developers easily and quickly build apps with code that can be understood by other Rx developers—even over different platforms.

Not only will you learn how to use RxJava to create complex reactive applications on Android, you'll also see how to solve common application design issues by using RxJava. Finally, you'll discover how to exercise full control over the library and leverage the full power of reactive programming in your apps.

Who This Book Is For

This book is for Android developers who already feel comfortable with the Android SDK and Kotlin, and want to dive deep into development with RxJava, RxKotlin, and RxAndroid.

Topics Covered in Reactive Programming with Kotlin:

- **Getting Started:** Get an introduction to the reactive programming paradigm, learn the terminology involved, and see how to begin using RxJava in your projects.
- **Event Management:** Learn how to handle asynchronous event sequences via two key concepts in Rx—Observables and Observers.
- **Being Selective:** See how to work with various events using tools such as filtering, transforming, combining, and timing operators.
- **UI Development:** RxJava and companion libraries make it easy to work with the UI of your apps, providing a reactive approach to handling user events.
- **Intermediate Topics:** Level up your RxJava knowledge with chapters on reactive networking, error handling, and schedulers.
- **Advanced Topics:** Round out your RxJava education by learning about app architecture, repositories, and integrating RxJava with Android Jetpack.

...and much, much more! By the end of the book, you'll have hands-on experience solving common issues in a reactive paradigm—and you'll be well on your way to coming up with your own Rx patterns and solutions!

About the Tutorial Team

The Tutorial Team is a group of app developers and authors who write tutorials at the popular website **raywenderlich.com.** We take pride in making sure each tutorial we write holds to the highest standards of quality. We want our tutorials to be well written, easy to follow, and fun.

If you've enjoyed the tutorials we've written in the past, you're in for a treat. The tutorials we've written for this book are some of our best yet — and this book contains detailed technical knowledge you simply won't be able to find anywhere else.