# 1 Chapter 1: Hello, RxSwift!

By Marin Todorov

This book aims to introduce you, the reader, to the RxSwift library and to writing reactive iOS apps with RxSwift.

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

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

Sounds 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 RxSwift APIs and general Rx concepts by explaining how to use each of the APIs and build intuition about how reactive programming can serve you, all while covering RxSwift's practical usage in iOS apps.

You'll start with the basic features of RxSwift, 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 RxSwift 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 RxSwift *is* though, have we? Let's 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.

> ***RxSwift**, in its essence, simplifies developing asynchronous programs by allowing your code to react to new data and process it in a sequential, isolated manner.*

As an iOS app developer, this should be much more clear and tell you more about what RxSwift 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 RxSwift 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

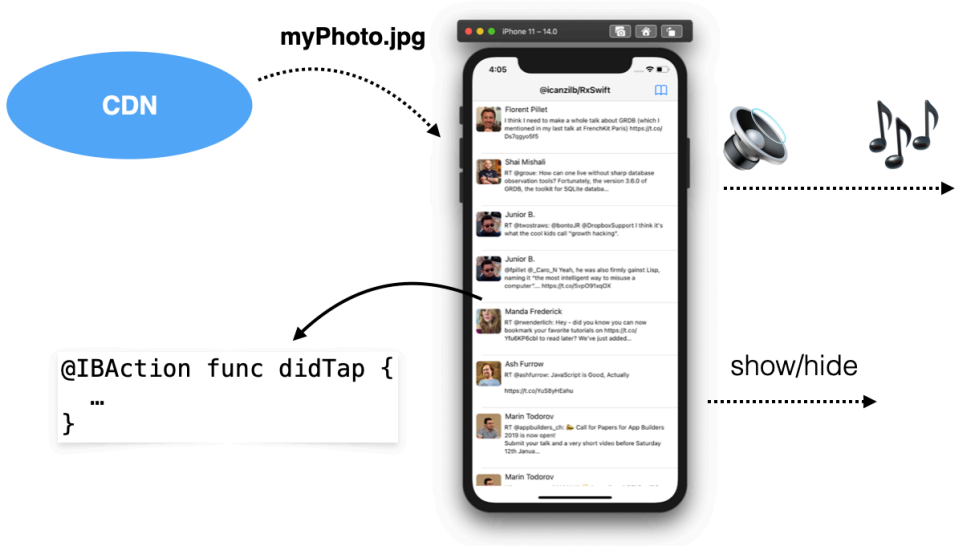
If you tried to explain asynchronous programming in a simple, down to earth language, you might come up with something along the lines of the following.

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

- Reacting to button taps
- Animating the keyboard as a text field loses focus
- 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. iOS offers you various kinds of APIs that allow you to perform different pieces of work on different threads, across different execution contexts, 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 know for sure which piece of code updates the data first, or which code read the latest value.

## Cocoa and UIKit asynchronous APIs

Apple has always provided numerous APIs in the iOS SDK that help you write asynchronous code. In fact the best practices on how to write asynchronous code on the platform have evolved many times over the years.

You've probably used many of these in your projects and probably haven't given them a second thought because they are so fundamental to writing mobile apps.

To mention few, you have a choice of:

- **`NotificationCenter`**: To execute a piece of code any time an event of interest happens, such as the user changing the orientation of the device or the software keyboard showing or hiding on the screen.
- **The delegate pattern**: Lets you define an object that acts on behalf, or in coordination with, another object.
- **Grand Central Dispatch**: To help you abstract the execution of pieces of work. You can schedule blocks of code to be executed sequentially, concurrently, or after a given delay.
- **Closures**: To create detached pieces of code that you can pass around in your code, and finally
- **Combine**: Apple's own framework for writing reactive asynchronous code with Swift, introduced in and available from iOS 13.

Depending on which APIs you chose to rely on, the degree of difficulty to maintain your app in a coherent state varies largely.

For example if you're using some of the older Apple APIs like the delegate pattern or notification center you need to do a lot of hard work to keep your app's state consistent at any given time.

If you have a shiny new codebase using Apple's Combine, then (of course) you're already verse with reactive programming - congrats and kudos!

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 an array 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** 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 a playground:

```
var array = [1, 2, 3]
for number in array {
  print(number)
  array = [4, 5, 6]
}
print(array)
```

Is `array` 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?

## Asynchronous code

Consider similar code, but assume each iteration happens as a reaction to a tap on a button. As the user repeatedly taps on the button, the app prints out the next element in an array:

```
var array = [1, 2, 3]
var currentIndex = 0

// This method is connected in Interface Builder to a button
@IBAction private func printNext() {
  print(array[currentIndex])

  if currentIndex != array.count - 1 {
    currentIndex += 1
  }
}
```

Think about this code in the same context as you did for the previous one. As the user taps the button, will that print all of the array'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 `currentIndex` is only mutated by `printNext()`, but another piece of code might modify `currentIndex` as well — perhaps some clever code you added at some point after crafting the above method.

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.

Luckily, these are some of RxSwift's strong suits!

Next, you need a good primer on the language that will help you start understanding how RxSwift works and what problems it solves; this will 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 RxSwift 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, RxSwift tries to address the following issues:

## 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 one 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 is the state of your laptop.

Managing the state of your app, 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 `viewDidAppear(_:)` of an iOS view controller:

```
override func viewDidAppear(_ animated: Bool) {
  super.viewDidAppear(animated)

  setupUI()
  connectUIControls()
  createDataSource()
  listenForChanges()
}
```

There's no telling what these methods do. Do they update properties of the view controller 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 represent any changes to the state outside of your code's current scope. For example, consider the last piece of code in the example above. `connectUIControls()` probably attaches some kind of event handler to some UI components. This causes a side effect, as it changes the state of the view: The app behaves one way *before* executing `connectUIControls()` and differently *after* that.

Any time you modify data stored on disk or update the text of a label on screen, you cause a side effect.

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 important aspect of producing side effects is doing so 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.

RxSwift tries to address the issues (or problems) listed above by tackling the following couple of concepts.

## 4. Declarative code

In imperative programming, you change state at will. In functional programming, you aim to minimize the code that causes side effects. Since you don't live in a perfect world, the balance lies somewhere in the middle. RxSwift combines some of the best aspects of imperative code and functional code.

Declarative code lets you define pieces of behavior. RxSwift will run these behaviors any time there's a relevant event and provide an immutable, isolated piece of data to work with.

This way, 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 can execute code in a sequential, deterministic way.

### 5. Reactive systems

Reactive systems is a rather abstract term and covers web or iOS 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.

Now that you have a good understanding of the problems RxSwift helps solve and how it approaches these issues, it's time to talk about the building blocks of Rx and how they play together.

## Foundation of RxSwift

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 apps 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 app development that we've discussed in this chapter. Sometime around 2009 they offered a new client and server side framework called Reactive Extensions for .NET (Rx).

Rx for .NET has been open source since 2012 permitting other languages and platforms to reimplement the same functionality, which turned Rx into a cross-platform standard.

Today, you have RxJS, RxKotlin, Rx.NET, RxScala, RxSwift and more. All strive to implement the same behavior and same expressive APIs, based on the Reactive Extensions specification. Ultimately, a developer creating an iOS app with RxSwift can freely discuss app logic with another programmer using RxJS on the web.

> **Note**: More about the family of Rx implementations at http://reactivex.io.

Like the original Rx, RxSwift also 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 decoupling.

In this book, you are going to cover both the cornerstone concepts of developing with RxSwift 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

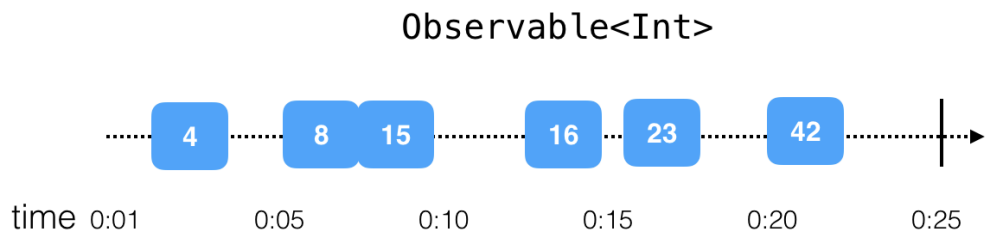
`Observable<Element>` provides the foundation of Rx code: the ability to asynchronously produce a sequence of events that can "carry" an immutable snapshot of generic data of type `Element`. In the simplest words, it allows consumers to *subscribe* for events, or values, emitted by another object over time.

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

The `ObservableType` protocol (to which `Observable` conforms) is extremely simple. An `Observable` can emit (and observers can receive) only three types of events:

- **A `next` event**: An event that "carries" the latest (or "*next*") data value. This is the way observers "receive" values. An `Observable` may emit an indefinite amount of these values, until a terminating event is emitted.
- **A `completed` event**: This event terminates the event sequence with success. It means the `Observable` completed its lifecycle successfully and won't emit additional events.
- **An `error` event**: The `Observable` terminates with an error and will not emit additional events.

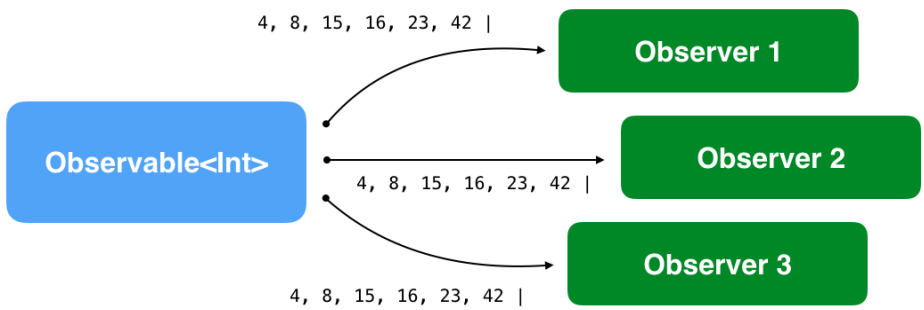
When talking about asynchronous events emitted over time, you can visualize an observable stream of integers on a timeline, like so:




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.

Since 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 delegate protocols or to inject closures 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 iOS app, consider code that downloads a file from the internet:

- First, you start the download and start observing for incoming data.

- You then repeatedly receive chunks of data as parts of the file arrive.
- 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...")
   .subscribe(
     onNext: { data in
      // Append data to temporary file
     },
     onError: { error in
       // Display error to user
     },
     onCompleted: {
       // Use downloaded file
     }
   )
```

`API.download(file:)` returns an `Observable<Data>` instance, which emits `Data` values as chunks of data fetched over the network.

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

You subscribe for an `error` by providing the `onError` closure. In this closure, you can display the `error.localizedDescription` in an alert box or otherwise handle your error.

Finally, to handle a `completed` event, you provide the `onCompleted` closure, where you can push a new view controller to display the downloaded file or anything else your app logic dictates.

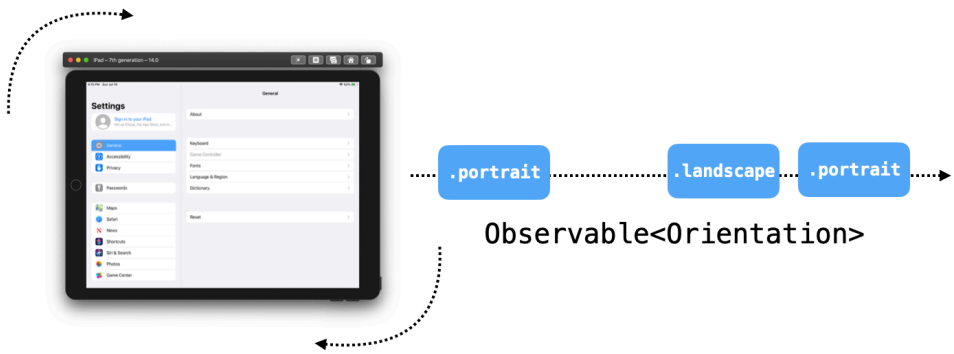
## Infinite observable sequences

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 device orientation changes in your app:

- You add your class as an observer to `UIDeviceOrientationDidChange` notifications from `NotificationCenter`.
- You then need to provide a method callback to handle orientation changes. It needs to grab the current orientation from `UIDevice` and react accordingly to the latest value.

This sequence of orientation changes does not have a natural end. As long as there is a device, there is a possible sequence of orientation changes. Further, since the sequence is virtually infinite and stateful, you always have an initial value at the time you start observing it.




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

In RxSwift, you could write code like this to handle device orientation:

```
UIDevice.rx.orientation
  .subscribe(onNext: { current in
    switch current {
    case .landscape:
      // Re-arrange UI for landscape
    case .portrait:
      // Re-arrange UI for portrait
    }
  })
```

`UIDevice.rx.orientation` is a fictional control property that produces an `Observable<Orientation>` (this is very easy to code yourself; you'll learn how in the next chapters). You subscribe to it and update your app UI according to the current orientation. You skip the `onError` and `onCompleted` arguments, since these events can never be emitted from that observable.

# Operators

`ObservableType` and the implementation of the `Observable` class include plenty of methods that abstract discrete pieces of asynchronous work and event manipulations, 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`.

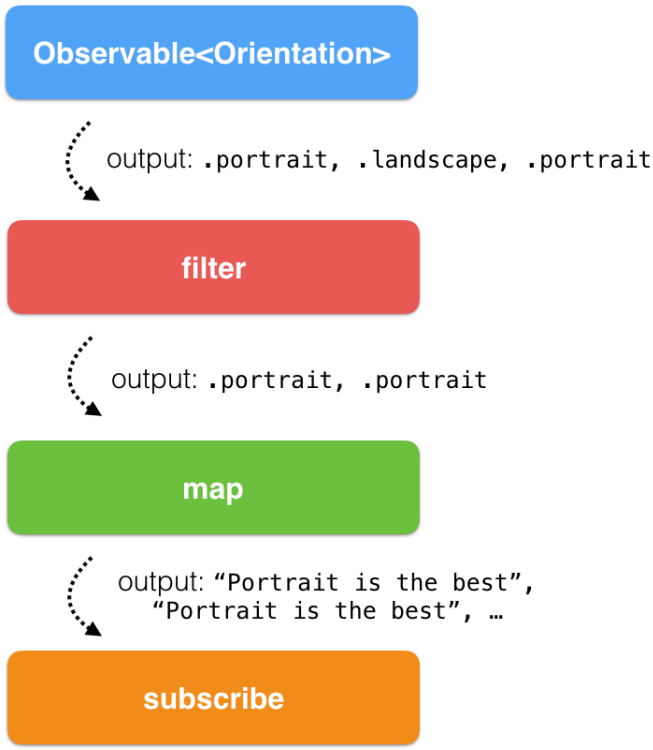
In a clear, deterministic way, you can apply the operators `*`, `( )`, `+` and `–` in their predefined order to the pieces of data that are their input, take their output and keep processing the expression until it's resolved.

In a somewhat similar manner, you can apply Rx operators to the events 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 orientation changes, adjusted to use some common Rx operators:

```
UIDevice.rx.orientation
  .filter { $0 != .landscape }
  .map { _ in "Portrait is the best!" }
  .subscribe(onNext: { string in
    showAlert(text: string)
  })
```

Each time `UIDevice.rx.orientation` produces either a `.landscape` or `.portrait` value, RxSwift will apply `filter` and `map` to that emitted piece of data.




First, `filter` will only let through values that are not `.landscape`. If the device is in landscape mode, the subscription code will not get executed because `filter` will suppress these events.

In case of `.portrait` values, the `map` operator will take the `Orientation` type input and convert it to a `String` output — the text `"Portrait is the best!"`

Finally, with `subscribe`, you subscribe for the resulting `next` event, this time carrying a `String` value, and you call a method to display an alert 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 so 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 more involved pieces of asynchronous work.

## Schedulers

Schedulers are the Rx equivalent of dispatch queues or operation queues — just on steroids and much easier to use. They let you define the execution context of a specific piece of work.

RxSwift comes with a number of predefined schedulers, which cover 99% of use cases and hopefully 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.

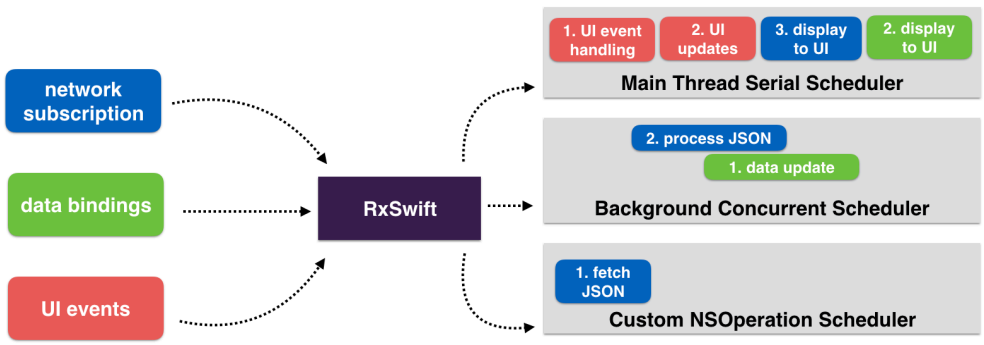
That being said, schedulers are very powerful.

For example, you can specify that you'd like to observe `next` events on a `SerialDispatchQueueScheduler`, which uses Grand Central Dispatch to run your code serially on a given queue.

`ConcurrentDispatchQueueScheduler` will run your code concurrently, while `OperationQueueScheduler` will allow you to schedule your subscriptions on a given `OperationQueue`.

Thanks to RxSwift, you can schedule your different pieces of work of the same subscription on different schedulers to achieve the best performance fitting your use-case.

RxSwift will act as a dispatcher between your subscriptions (on the left-hand side below) and the schedulers (on the right-hand side), sending the pieces of work to the correct context and seamlessly allowing them to work with each other's output.



To read this diagram, follow the colored pieces of work in the sequence they were scheduled `(1, 2, 3, ...)` across the different schedulers. For example:

- The blue network subscription starts with a piece of code `(1)` that runs on a custom `OperationQueue`-based scheduler.
- The data output by this block serves as the input of the next block `(2)`, which runs on a different scheduler, which is on a concurrent background GCD queue.
- Finally, the last piece of blue code `(3)` is scheduled on the Main thread scheduler in order to update the UI with the new data.

Even if it looks very interesting and quite handy, don't bother too much with schedulers right now. You'll return to them later in this book.

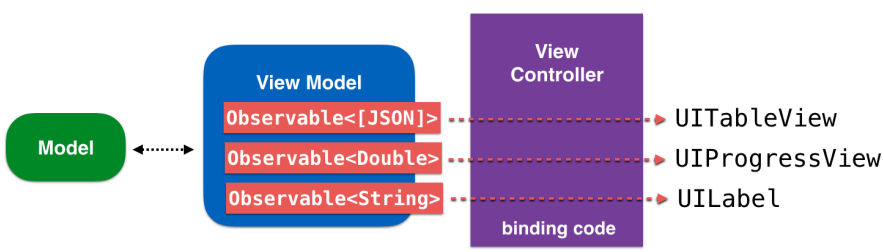
# App architecture

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

It's also 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 RxSwift when building new features for your app.

You can create apps with Rx by implementing a Model-View-Controller architecture, Model-View-Presenter, or Model-View-ViewModel (MVVM), or any other pattern that makes your life easier.

RxSwift and MVVM specifically do play nicely together. The reason is that a ViewModel allows you to expose `Observable` properties, which you can bind directly to UIKit controls in your View controller's glue code. This makes binding model data to the UI very simple to represent and to code:



Towards the end of this book, you'll look into that pattern and how to implement it with RxSwift. All other examples in the book use the MVC architecture in order to keep the sample code simple and easy to understand.

# RxCocoa

RxSwift is the implementation of the common, platform-agnostic, Rx specification. Therefore, it doesn't know anything about any Cocoa or UIKit-specific classes.

RxCocoa is RxSwift's companion library holding all classes that specifically aid development for UIKit and Cocoa. Besides featuring some advanced classes, RxCocoa adds reactive extensions to many UI components so that you can subscribe to various UI events out of the box.

For example, it's very easy to use RxCocoa to subscribe to the state changes of a `UISwitch`, like so:

```
toggleSwitch.rx.isOn
  .subscribe(onNext: { isOn in
    print(isOn ? "It's ON" : "It's OFF")
  })
```

RxCocoa adds the `rx.isOn` property (among others) to the `UISwitch` class so you can subscribe to useful events as reactive `Observable` sequences.


Further, RxCocoa adds the `rx` namespace to `UITextField`, `URLSession`, `UIViewController` and many more, and even lets you define your own reactive extensions under this namespace, which you'll learn more about later in this book.

# Installing RxSwift

RxSwift is open-source and available for free at https://bit.ly/2ZOzK2i.

RxSwift is distributed under the MIT license, which in short allows you to include the library in free or commercial software, on an as-is basis. As with all other MIT-licensed software, the copyright notice should be included in all apps you distribute.

There is plenty to explore in the RxSwift repository. It includes the **RxSwift**, **RxCocoa**, and **RxRelay** libraries, but you will also find **RxTest** and **RxBlocking** in there, which allow you to write tests for your RxSwift code.

Besides all the great source code (definitely worth peeking into), you will find **Rx.playground**, which interactively demonstrates many of the operators. Also check out **RxExample**, which is a great showcase app that demonstrates many of the concepts in practice.

You can install RxSwift/RxCocoa in few different ways - either via Xcode's built-in dependency management, via Cocoapods, or Carthage.

## RxSwift via CocoaPods

You can install RxSwift via CocoaPods like any other CocoaPod. A typical **Podfile** would look something like this:

```
use_frameworks!

target 'MyTargetName' do
  pod 'RxSwift', '~> 5.1'
  pod 'RxCocoa', '~> 5.1'
end
```

Of course, you can include just RxSwift, both RxSwift and RxCocoa, or even all the libraries found in the GitHub repository.

## RxSwift via Carthage

Installing RxSwift via Carthage is almost equally streamlined. First, make sure you've installed the latest version of Carthage from here: https://bit.ly/3cd1fF5.

In your project, create a new file named **Cartfile** and add the following line to it:

```
github "ReactiveX/RxSwift" ~> 5.1
```

Next, within the folder of your project execute `carthage update`.

This will download the source code of **all libraries** included in the RxSwift repository and build them, which might take some time. Once the process finishes, find the resulting framework files in the Carthage subfolder created inside the current folder and link them in your project.

Build once more to make sure Xcode indexes the newly added frameworks, and you're ready to go.

# Installing RxSwift in the book projects

The projects in this book all come with a completed **Podfile** to use with CocoaPods, but without RxSwift itself installed, to keep the download size of the book projects light.

Before you start working on the book, make sure you have the latest version of CocoaPods installed. You need to do that just once before starting to work on the book's projects. Usually executing this in Terminal will suffice:

```
sudo gem install cocoapods
```

If you want to know more, visit the CocoaPods website: https://bit.ly/2XGIvIN.

At the start of **each chapter**, you will be asked to open the starter project for that chapter and install RxSwift in the starter project. This is an easy operation:

1. In the book folder, find the directory matching the name of the chapter you are working on.
2. Copy the **starter** folder in a convenient location on your computer. A location in your user folder is a good idea.
3. Open the built-in Terminal.app or another one you use on daily basis and navigate to the **starter** folder. Type `cd /users/yourname/path/to/starter`, replacing the example path with the actual path on your computer.
4. In the chapters you'll be using a Playground, simply run `./bootstrap.sh`, which will fetch RxSwift from GitHub, pre-build the framework, and then automatically open Xcode for you so you can start writing some code.
5. In chapters you'll be using a standard Xcode project, type `pod install` to fetch RxSwift from GitHub and install it in the chapter project. Find the newly created `.xcworkspace` file and launch it. Build the workspace one time in Xcode.

You're now ready to work through the chapter!

> **Note**: While all playgrounds were tested under Xcode 11, Xcode 12 suffers from a myriad of issues related to playground support with third-party dependencies, such as RxSwift. If one of the provided playgrounds in this book doesn't work for you, we suggest copy and pasting the code from the playgrounds into a regular project with RxSwift embedded into it, or working with Xcode 11 in regards to these specific chapters.

# RxSwift and Combine

In this introductory chapter you got a taste of what RxSwift is all about. We spoke about some of the benefits of writing reactive code with RxSwift over using more traditional APIs like notification center and delegates.

Before wrapping up, it's definitely worth expanding a bit on what we already mentioned earlier - Apple's own reactive framework called Combine.

RxSwift and Combine (as well as other reactive programming frameworks in Swift) share a lot of common language and very similar concepts.

**RxSwift** is an older, well established framework with some of its own, original concepts, operator names and type variety mainly due to its multi-platform cross-language standard, which works also on Linux which is great for Server-Side Swift. It's also open source so you can, if you so wish, contribute directly to its core, and see exactly how specific portions of it work. It's compatible with all Apple platform versions that support Swift all the way back to iOS 8.

**Combine** is Apple's new and shiny framework that covers similar concepts but tailored specifically towards Swift and Apple's own platforms. It shares a lot of the common language with the Swift standard library so the APIs feel very familiar even to newcomers. It supports only the newer Apple platforms starting at iOS 13, macOS 10.15, etc. It is unfortunately not open-source as of today, and does not support Linux.

Luckily, since RxSwift and Combine resemble each other so closely, your RxSwift knowledge is easily transferable to Combine, and vice-versa. And projects such as RxCombine (https://github.com/CombineCommunity/RxCombine) allow you to mix-and-match RxSwift Observables and Combine Publishers based on your needs.

If you'd like to learn more about Combine - we've created the definitive book on that framework too "*Combine: Asynchronous Programming with Swift*" which you can check out here:

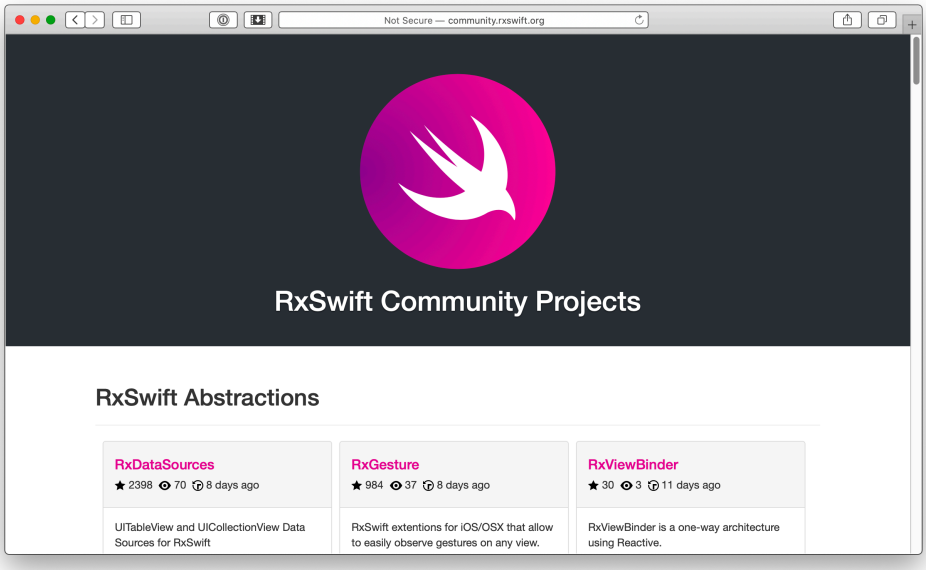
https://bit.ly/3dgOGds.

# Community

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

Besides the official RxSwift repository, you'll find plenty of projects created by Rx enthusiasts here: http://community.rxswift.org.




Even more Rx libraries and experiments, which spring up like mushrooms after the rain, can be found, here: https://github.com/RxSwiftCommunity

Probably the best way to meet many of the people interested in RxSwift is the Slack channel dedicated to the library: http://slack.rxswift.org.

The Slack channel has almost 8,000 members! Day-to-day topics are: helping each other, discussing potential new features of RxSwift or its companion libraries, and sharing RxSwift blog posts and conference talks.

# Where to go from here?

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

You haven't written any RxSwift code yet, but you now understand why RxSwift 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 apps using MVVM architecture.

Move right on to Chapter 2, "Observables"!

# Chapter 2: Observables

By Scott Gardner

Now that you've learned some of the basic concepts of RxSwift, it's time to take the jump and play with observables.

In this chapter, you'll go over several examples of creating and subscribing to observables. The real-world use of some of the observables may seem a bit obscure, but rest assured that you'll acquire important skills and learn a lot about the types of observables available to you in RxSwift. You'll use these skills throughout the rest of this book — and beyond!

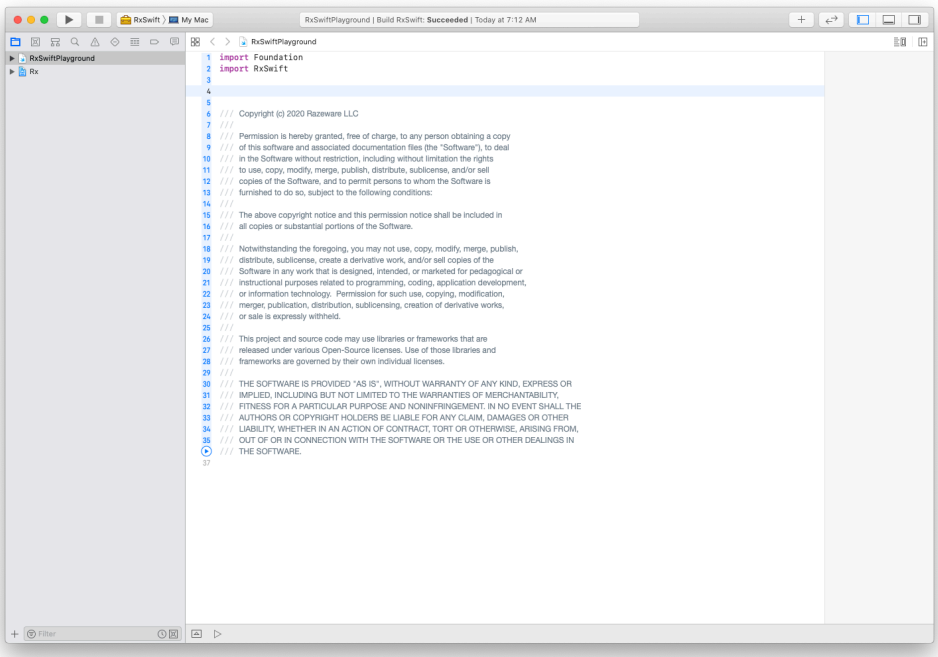
## Getting started

For this chapter, you're going to use an Xcode project that's already been set up to include a playground and the RxSwift framework. To get started, open up the Terminal app, navigate to this chapter's starter folder and then to the **RxPlayground** project folder. Finally, run the `bootstrap.sh` script by entering this command:

```
./bootstraph.sh
```

The bootstrap process will take a few seconds; remember that, every time you want to open this playground project, you will need to repeat the above steps. You cannot just open the playground file or workspace directly.

Select **RxSwiftPlayground** in the **Project navigator**, and you should see the following:




Twist down the playground page, through the **Sources** folder in the **Project navigator**, and select **SupportCode.swift**. It contains the following helper function `example(of:)`:

```
public func example(of description: String, action: () -> Void)
{
  print("\n--- Example of:", description, "---")
  action()
}
```

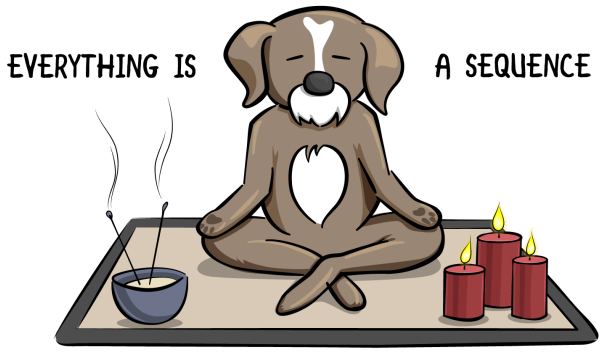
You're going to use this function to encapsulate different examples as you work your way through this chapter. You'll see how to use this function shortly.

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

# What is an observable?

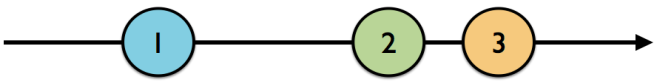
**Observables** are the heart of Rx. You're going to spend some time discussing what observables are, how to create them, and how to use them.

You'll see "observable", "observable sequence" and "sequence" used interchangeably in Rx. And, really, they're all the same thing. You may even see an occasional "stream" thrown around from time to time, especially from developers that come to RxSwift from a different reactive programming environment. "Stream" also refers to the same thing, but, in RxSwift, all the cool kids call it a sequence, not a stream. In RxSwift...




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

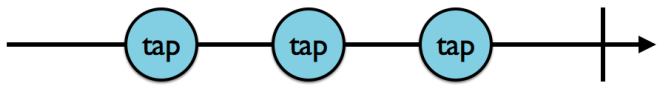



The left-to-right arrow represents time, and the numbered circles represent elements of a sequence. Element 1 will be emitted, some time will pass, and then 2 and 3 will be emitted. How much time, you ask? It could be at *any* point throughout the lifetime of the observable — which brings you to the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

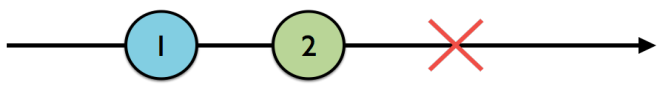
# Lifecycle of an observable

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 **completed** event, that is, it's **terminated**. For example, perhaps the taps were on a view that was dismissed. The important thing is that the observable is terminated, and can no longer emit anything. This is normal termination. However, sometimes things can go wrong.




An error occurred in this marble diagram, represented by the red X. The observable emitted an **error** event containing the error. This is the same as when an observable terminates normally with a **completed** 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 a **terminating event** is emitted, i.e., an **error** or **completed** event.
- Once an observable is terminated, it can no longer emit events.

Events are represented as enumeration cases. Here's the actual implementation in the RxSwift source code:

```
/// Represents a sequence event.
///
/// Sequence grammar:
/// **next\* (error | completed)**
public enum Event<Element> {
    /// Next element is produced.
    case next(Element)

    /// Sequence terminated with an error.
```

```
    case error(Swift.Error)

    /// Sequence completed successfully.
    case completed
}
```

Here, you see that `next` events contain an instance of some `Element`, `error` events contain an instance of `Swift.Error` and `completed` events are simply stop events that don't contain any data.

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 **RxSwiftPlayground** and add the code below:

```
example(of: "just, of, from") {
  // 1
  let one = 1
  let two = 2
  let three = 3

  // 2
  let observable = Observable<Int>.just(one)
}
```

In the above code, you:

1. Define integer constants you'll use in the following examples.
2. Create an observable sequence of type `Int` using the `just` method with the `one` integer constant.

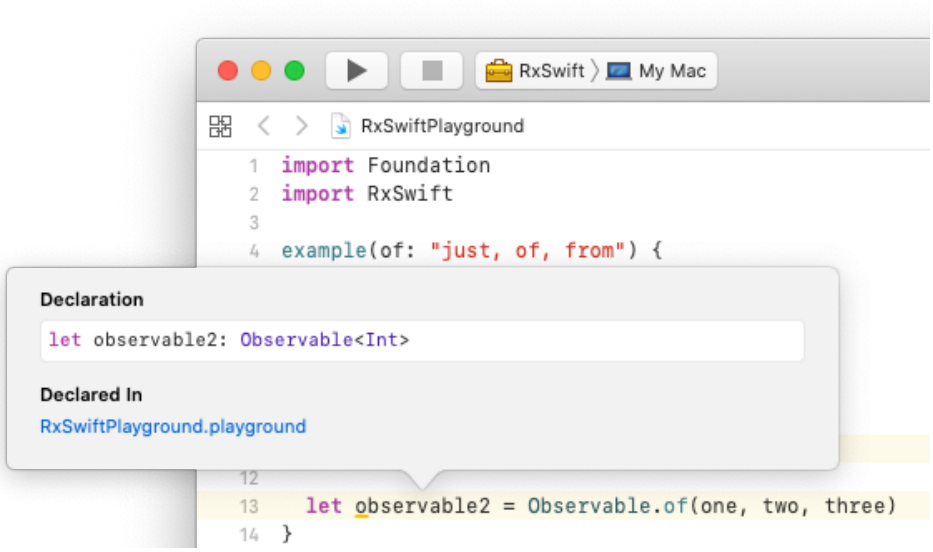
The `just` method is aptly named, because all it does is create an observable sequence containing *just* a single element. It is a static method on `Observable`. However, in Rx, methods are referred to as "operators."And the eagle-eyed among you can probably guess which operator you're going to check out next.

Add this code into the same example:

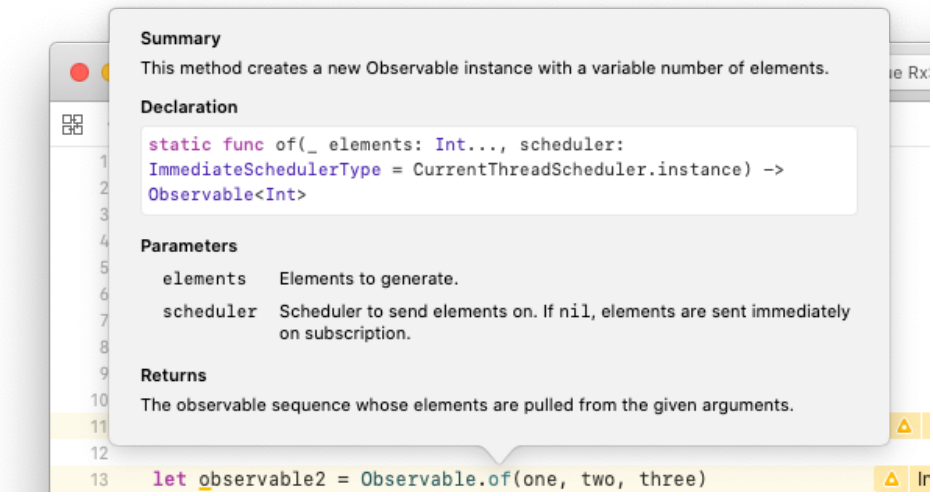
```
let observable2 = Observable.of(one, two, three)
```

This time, you didn't explicitly declare the type. You *might* think, because you give it several integers, the type is `Observable` of `[Int]`.

However, **Option-click** on `observable2` to show its inferred type, and you'll see that it's an `Observable` of `Int`, not an array:




That's because the `of` operator has a **variadic** parameter, and Swift can infer the `Observable`'s type based on it.




Pass an array to `of` when you want to create an observable array. Add this code to the bottom of the example:

```
let observable3 = Observable.of([one, two, three])
```

Option-click on `observable3` and you'll see that it is indeed an `Observable` of `[Int]`. The `just` operator can also take an array as its single element, which may seem a little weird at first. However, it's the *array* that is the single element, not its contents.

Another operator you can use to create observables is `from`. Add this code to the bottom of the example:

```
let observable4 = Observable.from([one, two, three])
```

The `from` operator creates an observable of individual elements from an array of typed elements. Option-click on `observable4` and you'll see that it is an `Observable` of `Int`, not `[Int]`. The `from` operator *only* takes an array.

Your console is probably looking quite 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

From your experience as an iOS developer, you are likely familiar with `NotificationCenter`; it broadcasts notifications to observers. However, these observed notifications are different than RxSwift `Observables`. Here's an example of an observer of the `UIKeyboardDidChangeFrame` notification, with a handler as a trailing closure (don't add this code to your playground):

```
let observer = NotificationCenter.default.addObserver(
  forName: UIResponder.keyboardDidChangeFrameNotification,
  object: nil,
  queue: nil) { notification in
  // Handle receiving notification
}
```

Subscribing to an RxSwift observable is fairly similar; you call observing an observable *subscribing* to it. So instead of `addObserver()`, you use `subscribe()`. Unlike `NotificationCenter`, where developers typically use only its `.default` singleton instance, each observable in Rx is different.

More importantly, an observable won't send events, or perform any work, until it has a subscriber.

Remember, an observable is really a sequence definition, and subscribing to an observable is really more like calling `next()` on an `Iterator` in the Swift standard library (don't add this code to your playground):

```
let sequence = 0..<3

var iterator = sequence.makeIterator()

while let n = iterator.next() {
  print(n)
}

/* Prints:
 0
 1
 2
 */
```

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

To see this in action, add this new example to your playground. Remember to insert each new example on its own, *after* the closing curly bracket of the previous example.

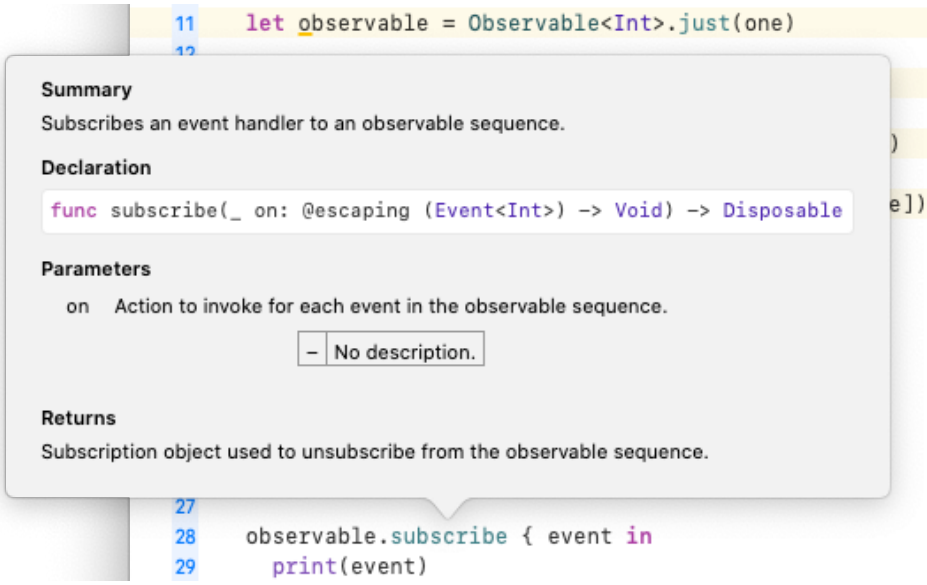
```
example(of: "subscribe") {
  let one = 1
  let two = 2
  let three = 3

  let observable = Observable.of(one, two, three)
}
```

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

```
observable.subscribe { event in
  print(event)
}
```

> **Note**: The Console should automatically appear whenever there is output, but you can manually show it by selecting **View ▸ Debug Area ▸ Activate Console** from the menu. This is where the `print` statements in the playground display their output.

Option-click on the `subscribe` operator, and observe it takes a closure parameter that receives an `Event` of type `Int` and doesn't return anything, and `subscribe` returns a `Disposable`. You'll learn about disposables shortly.




This subscription will print out each event emitted by `observable`:

```
--- Example of: subscribe ---
next(1)
next(2)
next(3)
completed
```

The observable emits a `next` event for each element, then emits a `completed` event and is terminated. When working with observables, you'll usually be primarily interested in the **elements** emitted by `next` events, rather than the events themselves.

To see one way to access the elements directly, replace the subscribing code from above with the following code:

```
observable.subscribe { event in
  if let element = event.element {
    print(element)
  }
}
```

`Event` has an `element` property. It's an optional value, because only `next` events have an element. So you use optional binding to unwrap the element if it's not `nil`. Now only the elements are printed, not the events *containing* the elements, nor the `completed` event:

```
1
2
3
```

That's a nice pattern, and it's so frequently used that there's a shortcut for it in RxSwift. There's a `subscribe` operator for each type of event an observable emits: **next**, **error** and **completed**.

Replace the previous subscription code with this:

```
observable.subscribe(onNext: { element in
  print(element)
})
```

> **Note**: If you have code completion suggestions turned on in Xcode preferences, you may be asked for handlers for the other events. Ignore these for now.

Now you're only handling `next` event elements and ignoring everything else. The `onNext` closure receives the `next` event's element as an argument, so you don't have to manually extract it from the event like you did before.

Now you know how to create observable 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 `completed` event.

Add this new example to your playground:

```
example(of: "empty") {
  let observable = Observable<Void>.empty()
}
```

An observable must be defined as a specific type if it cannot be inferred. So the type must be explicitly defined, because `empty` has nothing from which to infer the type. `Void` is typically used because nothing is going to be emitted.

Add this code to the example to subscribe to the empty observable:

```
observable.subscribe(
  // 1
  onNext: { element in
    print(element)
  },

  // 2
  onCompleted: {
    print("Completed")
  }
)
```

In the above code, you:

1. Handle `next` events, just like you did in the previous example.
2. Simply print a message, because a `.completed` event does not include an element.

In the console, you'll see that `empty` only emits a `.completed` event:

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

What use is an *empty* observable? 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 use to represent an infinite duration. Add this example to your playground:

```
example(of: "never") {
  let observable = Observable<Void>.never()

  observable.subscribe(
    onNext: { element in
      print(element)
    },
    onCompleted: {
      print("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.

So far, you've worked with observables of specific elements or values. However, it's also possible to generate an observable from a range of values.

Add this example to your playground:

```
example(of: "range") {
  // 1
  let observable = Observable<Int>.range(start: 1, count: 10)

  observable
    .subscribe(onNext: { i in
      // 2
      let n = Double(i)

      let fibonacci = Int(
        ((pow(1.61803, n) - pow(0.61803, n)) /
          2.23606).rounded()
      )

      print(fibonacci)
  })
}
```

What you just did:

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.

There's actually a better place than in the `onNext` handler 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 worked with observables that automatically emit a `completed` event and naturally terminate. Doing so allowed you to focus on the mechanics of creating and subscribing to observables, but this brushed an important aspect of subscribing to observables under the carpet. It's time to do some housekeeping 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's work, causing it to emit new events until an `error` or `completed` event terminates the observable. However, you can also manually cause an observable to terminate by canceling a subscription to it.

Add this new example to your playground:

```
example(of: "dispose") {
  // 1
  let observable = Observable.of("A", "B", "C")

  // 2
  let subscription = observable.subscribe { event in
    // 3
    print(event)
  }
}
```

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 RxSwift includes a `DisposeBag` type. A dispose bag holds disposables — typically added using the `disposed(by:)` method — and will call `dispose()` on each one when the dispose bag is about to be deallocated.

Add this new example to your playground:

```
example(of: "DisposeBag") {
  // 1
  let disposeBag = DisposeBag()
```

```
  // 2
  Observable.of("A", "B", "C")
    .subscribe { // 3
      print($0)
    }
    .disposed(by: disposeBag) // 4
}
```

Step-by-step, you:

1. Create a dispose bag.
2. Create an observable.
3. Subscribe to the observable and print out the emitted events using the default argument name `$0`.
4. Add the returned `Disposable` from `subscribe` to the dispose bag.

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

Why bother with disposables at all?

If you forget to add a subscription to a dispose bag, or manually call `dispose` on it 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.

Don't worry if you forget; the Swift compiler should warn you about unused disposables.

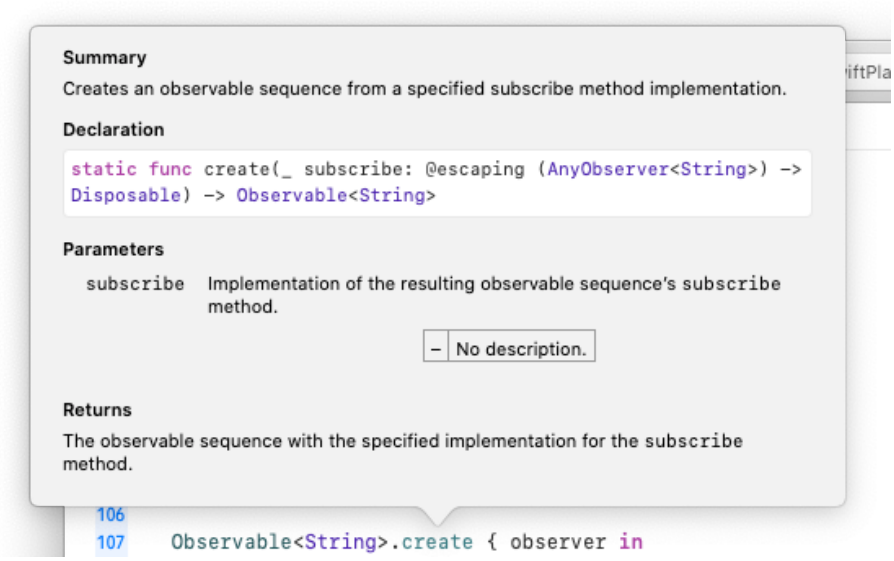
In the previous examples, you created observables with specific `next` event elements. Using the `create` operator is another way to specify all the events an observable will emit to subscribers.

Add this new example to your playground:

```
example(of: "create") {
  let disposeBag = DisposeBag()

  Observable<String>.create { observer in

  }
}
```

The `create` operator takes a single parameter named `subscribe`. 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.

If you option-click on `create` right now you will not get the Quick Help documentation, because this code won't compile yet. So here's a preview:




The `subscribe` parameter is an escaping closure that takes an `AnyObserver` and returns a `Disposable`. `AnyObserver` is a generic type that facilitates adding values *onto* an observable sequence, which will then be emitted to subscribers.

Change the implementation of `create` to the following:

```
Observable<String>.create { observer in
  // 1
  observer.onNext("1")

  // 2
  observer.onCompleted()

  // 3
  observer.onNext("?")

  // 4
  return Disposables.create()
}
```

Here's what you do with this code:

1. Add a `next` event onto the observer. `onNext(_:)` is a convenience method for `on(.next(_:))`.
2. Add a `completed` event onto the observer. Similarly, `onCompleted` is a convenience method for `on(.completed)`.
3. Add another `next` event onto the observer.
4. Return a disposable, defining what happens when your observable is terminated or disposed of; in this case, no cleanup is needed so you return an empty disposable.

> **Note**: The last step, returning a `Disposable`, may seem strange at first. Remember that `subscribe` operators must return a disposable representing the subscription, so you use `Disposables.create()` to create a disposable.

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:

```
.subscribe(
  onNext: { print($0) },
  onError: { print($0) },
  onCompleted: { print("Completed") },
  onDisposed: { print("Disposed") }
)
.disposed(by: disposeBag)
```

You subscribed to the observable, and implemented all the handlers using default argument names for element and error arguments passed to the `onNext` and `onError` handlers, respectively. The result is, the first `next` event element, `"Completed"` and `"Disposed"` are printed out. The second `next` event is not printed because the observable emitted a `completed` event and terminated before it is added.

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

What would happen if you add an error to the `observer`? Add this code at the top of the example to define an error type with a single case:

```
enum MyError: Error {
  case anError
}
```

Next, add the following line of code between the `observer.onNext` and `observer.onCompleted` calls:

```
observer.onError(MyError.anError)
```

Now the observable emits the error and then terminates:

```
--- Example of: create ---
1
anError
Disposed
```

What would happen if you did not add a `completed` or `error` event, and also didn't add the subscription to `disposeBag`? Comment out the `observer.onError`, `observer.onCompleted` and `disposed(by: disposeBag)` lines of code to find out.

Here's the complete implementation:

```
example(of: "create") {
  enum MyError: Error {
    case anError
  }

  let disposeBag = DisposeBag()

  Observable<String>.create { observer in
    // 1
    observer.onNext("1")

//    observer.onError(MyError.anError)

    // 2
//    observer.onCompleted()

    // 3
    observer.onNext("?")

    // 4
    return Disposables.create()
  }
  .subscribe(
    onNext: { print($0) },
```

```
      onError: { print($0) },
      onCompleted: { print("Completed") },
      onDisposed: { print("Disposed") }
    )
//  .disposed(by: disposeBag)
}
```

Congratulations, you've just leaked memory! The observable will never finish, and the disposable will never be disposed.

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

Feel free to uncomment the line that adds the `completed` event or the code that adds the subscription to the `disposeBag` if you just can't stand to leave this example 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 your playground:

```
example(of: "deferred") {
  let disposeBag = DisposeBag()

  // 1
  var flip = false

  // 2
  let factory: Observable<Int> = Observable.deferred {

    // 3
    flip.toggle()

    // 4
    if flip {
      return Observable.of(1, 2, 3)
    } else {
      return Observable.of(4, 5, 6)
    }
  }
}
```

From the top, you:

1. Create a `Bool` flag to flip which observable to return.
2. Create an observable of `Int` factory using the `deferred` operator.

$

3. Toggle `flip`, which happens 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 _ in 0...3 {
  factory.subscribe(onNext: {
    print($0, terminator: "")
  })
  .disposed(by: disposeBag)

  print()
}
```

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

```
--- Example of: deferred ---
123
456
123
456
```

# Using Traits

Traits are observables with a narrower set of behaviors than regular observables. Their use is optional; you can use a regular observable anywhere you might use a trait instead. 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 a trait can help make your code more intuitive.

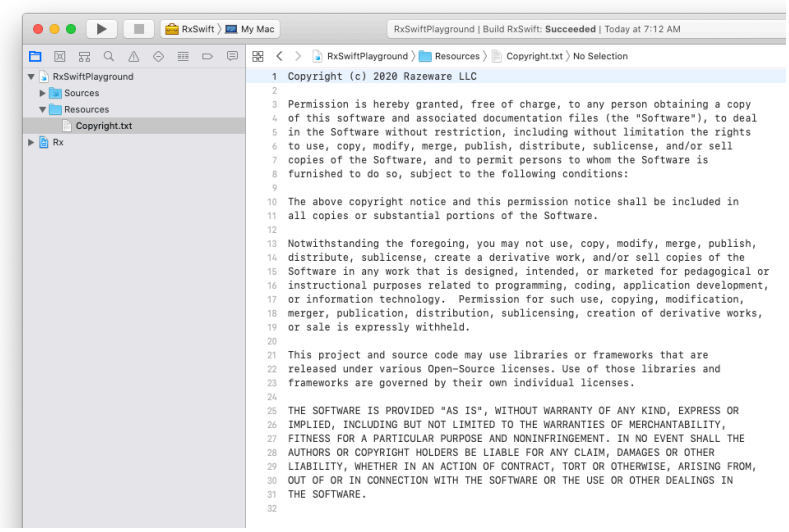
There are three kinds of traits in RxSwift: `Single`, `Maybe` and `Completable`. Without knowing anything more about them yet, can you guess how each one is specialized?

`Singles` will emit either a `success(value)` or `error(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 when downloading data or loading it from disk.

A `Completable` will only emit a `completed` or `error(error)` event. It will not emit any values. You could use a completable when you only care that an operation completed successfully or failed, such as a file write.

Finally, `Maybe` is a mashup of a `Single` and `Completable`. It can either emit a `success(value)`, `completed` or `error(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 traits in Chapter 4, "Observables & 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 **Resources** folder for this playground — because who doesn't love some legalese once in a while?




Add this example to your playground:

```
example(of: "Single") {
  // 1
  let disposeBag = DisposeBag()

  // 2
  enum FileReadError: Error {
```

```
    case fileNotFound, unreadable, encodingFailed
  }

  // 3
  func loadText(from name: String) -> Single<String> {
    // 4
    return Single.create { single in

    }
  }
}
```

In the above code, you:

1. Create a dispose bag to use later.
2. Define an `Error` enum to model some possible errors that can occur in reading data from a file on disk.
3. Implement a function to load text from a file on disk that returns a `Single`.
4. Create and return a `Single`.

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

```
// 1
let disposable = Disposables.create()

// 2
guard let path = Bundle.main.path(forResource: name, ofType:
"txt") else {
  single(.error(FileReadError.fileNotFound))
  return disposable
}

// 3
guard let data = FileManager.default.contents(atPath: path) else
{
  single(.error(FileReadError.unreadable))
  return disposable
}

// 4
guard let contents = String(data: data, encoding: .utf8) else {
  single(.error(FileReadError.encodingFailed))
  return disposable
}

// 5
single(.success(contents))
return disposable
```

With this code, you:

1. Create a `Disposable`, because the `subscribe` closure of `create` expects it as its return type.
2. Get the path for the filename, or else add a file not found error onto the `Single` and return the disposable you created.
3. Get the data from the file at that path, or add an unreadable error onto the `Single` and return the disposable.
4. Convert the data to a string; otherwise, add an encoding failed error onto the `Single` and return the disposable. Starting to see a pattern here?
5. Made it this far? Add the contents onto the `Single` as a success, and return the disposable.

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

```
// 1
loadText(from: "Copyright")
// 2
  .subscribe {
    // 3
    switch $0 {
    case .success(let string):
      print(string)
    case .error(let error):
      print(error)
    }
  }
  .disposed(by: disposeBag)
```

Here, you:

1. Call `loadText(from:)` and pass the root name of the text file.
2. Subscribe to the `Single` it returns.
3. Switch on the event and print the string if it was successful, or print the error if not.

You should see the text from the file printed to the console, which is the same as the copyright comment at the bottom of the playground:

```
--- Example of: Single ---
Copyright (c) 2020 Razeware LLC
...
```

# 3 Chapter 3: Subjects

By Scott Gardner

At this point, you know what an observable is, how to create one, how to subscribe to it, and how to dispose of things when you're done. Observables are a fundamental part of RxSwift, but they're essentially read-only. You may only subscribe to them to get notified of new events they produce.

A common need when developing apps is to manually add new values onto an observable during runtime to emit to subscribers. What you want is something that can act as both an observable and as an **observer**. That something is called a **Subject**.

In this chapter, you'll learn about the different types of subjects in RxSwift, see how to work with each one and why you might choose one over another based on some common use cases. You'll also learn about relays, which are wrappers around subjects. We'll unwrap that later!

## Getting started

Run `./bootstrap.sh` in the starter project folder **RxPlayground**, which will open the project for this chapter, and select **RxSwiftPlayground** in the Project navigator. You'll start out with a quick example to prime the pump. Add the following code to your playground:

```
example(of: "PublishSubject") {
  let subject = PublishSubject<String>()
}
```

You just created a `PublishSubject`. It's aptly named, because, like a newspaper publisher, it will receive information and then publish it to subscribers. It's of type `String`, so it can only receive and publish strings. After being initialized, it's ready to receive strings.

Add the following code to your example:

```
subject.on(.next("Is anyone listening?"))
```

This puts a new string onto the subject. Nothing is printed out yet, because there are no observers. Create one by subscribing to the subject. Add the following code to the example:

```
let subscriptionOne = subject
  .subscribe(onNext: { string in
    print(string)
  })
```

You created a subscription to `subject` just like in the last chapter, printing `next` events. But still, nothing shows up in Xcode's output console. What gives?

What's happening here is that a `PublishSubject` only emits to *current* subscribers. So if you weren't subscribed to it when an event was added to it, you won't get it when you do subscribe. Think of the tree-falling-in-the-woods analogy. If a tree falls and no one's there to hear it, does that make your illegal logging business a success? :]

To fix things, add this code to the end of the example:

```
subject.on(.next("1"))
```

Notice that, because you defined the publish subject to be of type `String`, only strings may be added to it. Now, because `subject` *has* a subscriber, it will emit the added value:

```
--- Example of: PublishSubject ---
1
```

In a similar fashion to the `subscribe` operators, `on(.next(_:))` is how you add a new `next` event onto a subject, passing the element as the parameter. And just like `subscribe`, there's shortcut syntax for subjects. Add the following code to the example:

```
subject.onNext("2")
```

`onNext(_:)` does the same thing as `on(.next(_))`. It's just a bit easier on the eyes. And now the 2 is also printed:

```
--- Example of: PublishSubject ---
1
2
```

With that gentle intro, now it's time to dig in and learn all about subjects.

# What are subjects?

Subjects act as both an observable **and** an observer. You saw earlier how they can receive events and also be subscribed to. In the above example, the subject received `next` events, and for each of them, it turned around and emitted it to its subscriber.

There are four subject types in RxSwift:

- `PublishSubject`: Starts empty and only emits new elements to subscribers.
- `BehaviorSubject`: Starts with an initial value and replays it or the latest element to new subscribers.
- `ReplaySubject`: Initialized with a buffer size and will maintain a buffer of elements up to that size and replay it to new subscribers.
- `AsyncSubject`: Emits *only the last* `next` event in the sequence, and only when the subject receives a `completed` event. This is a seldom used kind of subject, and you won't use it in this book. It's listed here for the sake of completeness.

RxSwift also provides a concept called Relays. RxSwift provides two of these, named `PublishRelay` and `BehaviorRelay`. These wrap their respective subjects, but only accept and *relay* `next` events. You cannot add a `completed` or `error` event onto relays at all, so they're great for non-terminating sequences.

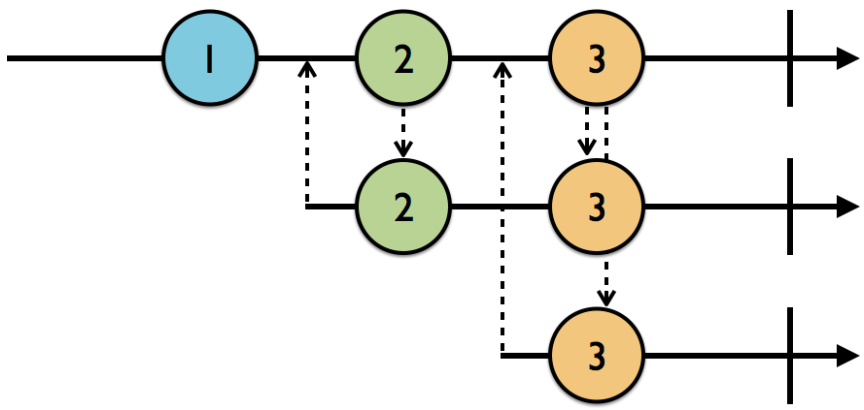
> **Note**: Did you notice the additional `import RxRelay` in this chapter's playground? Originally, relays were part of RxCocoa, RxSwift's suite of reactive Cocoa extensions and utilities. However, relays are a general-use concept that are also useful in non-Cocoa development environments such as Linux and command line tools. So it was split into its own consumable module, which RxCocoa depends on.

Next, you'll learn more about these subjects and relays and how to work with them, starting with publish subjects.

# Working with publish subjects

Publish subjects come in handy when you simply want subscribers to be notified of new events from the point at which they subscribed, until either they unsubscribe, or the subject has terminated with a `completed` or `error` event.

In the following marble diagram, the top line is the publish subject and the second and third lines are subscribers. The upward-pointing arrows indicate subscriptions, and the downward-pointing arrows represent emitted events.




The first subscriber subscribes after 1 is added to the subject, so it doesn't receive that event. It does get 2 and 3, though. And because the second subscriber doesn't join in until after 2 is added, it only gets 3.

Returning to the playground, add this code to the bottom of the same example:

```
let subscriptionTwo = subject
  .subscribe { event in
    print("2)", event.element ?? event)
  }
```

Events have an optional `element` property that contains the emitted element for `next` events. You use the nil-coalescing operator here to print the element if there is one; otherwise, you print the event.

As expected, `subscriptionTwo` doesn't print anything out yet because it subscribed after the 1 and 2 were emitted. Now add this code:

```
subject.onNext("3")
```

The 3 is printed twice, once for `subscriptionOne` and once for `subscriptionTwo`.

```
3
2) 3
```

Add this code to terminate `subscriptionOne` and then add another `next` event onto the subject:

```
subscriptionOne.dispose()

subject.onNext("4")
```

The value 4 is only printed for subscription 2), because `subscriptionOne` was disposed.

```
2) 4
```

When a publish subject receives a `completed` or `error` event, also known as a *stop* event, it will emit that stop event to new subscribers and it will no longer emit `next` events. However, it will *re-emit* its stop event to future subscribers. Add this code to the example:

```
// 1
subject.onCompleted()

// 2
subject.onNext("5")

// 3
subscriptionTwo.dispose()

let disposeBag = DisposeBag()

// 4
subject
  .subscribe {
    print("3)", $0.element ?? $0)
  }
  .disposed(by: disposeBag)

subject.onNext("?")
```

From the top, you:

1. Add a `completed` event onto the subject, using the convenience method for `on(.completed)`. This terminates the subject's observable sequence.
2. Add another element onto the subject. This won't be emitted and printed, though, because the subject has already terminated.
3. Dispose of the subscription.
4. Subscribe to the subject, this time adding its disposable to a dispose bag.

Maybe the new subscriber `3)` will kickstart the subject back into action? Nope, but you do still get the `completed` event replayed.

```
2) completed
3) completed
```

Actually, subjects, once terminated, will re-emit their stop event to future subscribers. So it's a good idea to include handlers for stop events in your code, not just to be notified when it terminates, but also in case it is already terminated when you subscribe to it. This can sometimes be the cause of subtle bugs, so watch out!

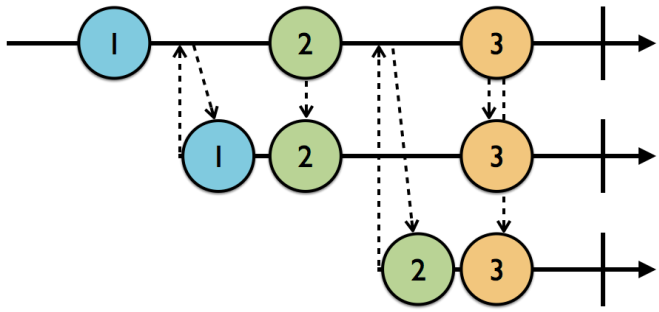
You might use a publish subject when you're modeling time-sensitive data, such as in an online bidding app. It wouldn't make sense to alert the user who joined at 10:01 am that at 9:59 am there was only 1 minute left in the auction. That is, of course, unless you like 1-star reviews of your bidding app.

Sometimes you want to let new subscribers know what was the latest emitted element, even though that element was emitted before the subscription. For that, you've got some options.

Publish subjects don't replay values to new subscribers. This makes them a good choice to model **events** such as "user tapped something" or "notification just arrived."

# Working with behavior subjects

Behavior subjects work similarly to publish subjects, except they will *replay* the latest `next` event to new subscribers. Check out this marble diagram:




The first line at the top is the subject. The first subscriber on the second line down subscribes after 1 but before 2, so it receives 1 immediately upon subscription, and then 2 and 3 when they're emitted by the subject. Similarly, the second subscriber subscribes after 2 but before 3, so it receives 2 immediately and then 3 when it's emitted.

Add this code to your playground, after the last example:

```
// 1
enum MyError: Error {
  case anError
}

// 2
func print<T: CustomStringConvertible>(label: String, event:
Event<T>) {
  print(label, (event.element ?? event.error) ?? event)
}

// 3
example(of: "BehaviorSubject") {
  // 4
  let subject = BehaviorSubject(value: "Initial value")
  let disposeBag = DisposeBag()
}
```

Here's the play-by-play:

1. Define an error type to use in upcoming examples.
2. Expanding upon the use of the ternary operator in the previous example, you create a helper function to print the element if there is one, an error if there is one, or else the event itself. How convenient!
3. Start a new example.
4. Create a new `BehaviorSubject` instance. Its initializer takes an initial value.

> **Note**: Because `BehaviorSubject` *always* emits its latest element, you can't create one without providing an initial value. If you can't provide an initial value at creation time, that probably means you need to use a `PublishSubject` instead, or model your element as an `Optional`.

Next, add the following code to the example:

```
subject
  .subscribe {
    print(label: "1)", event: $0)
  }
  .disposed(by: disposeBag)
```

You subscribe to the subject immediately *after* it was created. Because no other elements have been added to the subject, it replays its initial value to the subscriber.

```
--- Example of: BehaviorSubject ---
1) Initial value
```

Now, insert the following code right *before* the previous subscription code, but *after* the definition of the subject:

```
subject.onNext("X")
```

The `X` is printed, because now *it's* the latest element when the subscription is made.

```
--- Example of: BehaviorSubject ---
1) X
```

Add the following code to the end of the example. But first, look it over and see if you can determine what will be printed:

```
// 1
subject.onError(MyError.anError)

// 2
subject
  .subscribe {
    print(label: "2)", event: $0)
  }
  .disposed(by: disposeBag)
```

With this code, you:

1. Add an error event onto the subject.
2. Create a new subscription to the subject.

This prints:

```
1) anError
2) anError
```

Did you figure out that the error event will be printed twice, once for each subscription? If so, right on!

Behavior subjects are useful when you want to pre-populate a view with the most recent data. For example, you could bind controls in a user profile screen to a behavior subject, so that the latest values can be used to pre-populate the display while the app fetches fresh data.

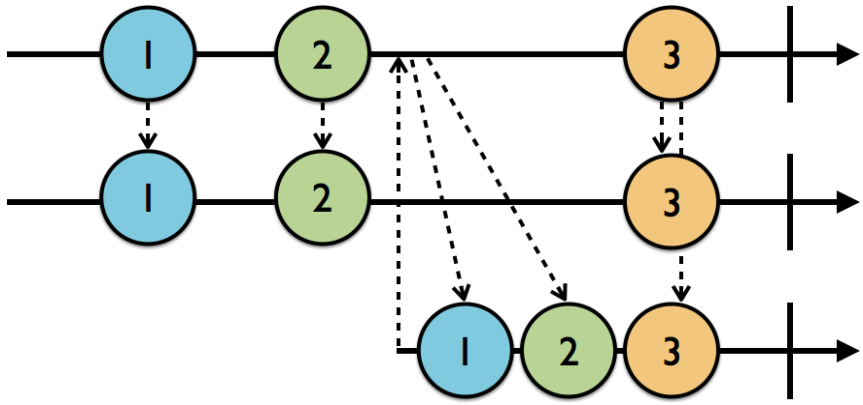
Behavior subjects replay their latest value to new subscribers. This makes them a good choice to model **state** such as "request is currently loading," or "the time is now 9:41."

What if you wanted to show more than the latest value? For example, on a search screen, you may want to show the most recent *five* search terms used. This is where replay subjects come in.

# Working with replay subjects

Replay subjects will temporarily cache, or *buffer*, the latest elements they emit, up to a specified size of your choosing. They will then replay that buffer to new subscribers.

The following marble diagram depicts a replay subject with a buffer size of 2.




The first subscriber (middle line) is already subscribed to the replay subject (top line) so it gets elements as they're emitted. The second subscriber (bottom line) subscribes after 2, so it gets 1 and 2 replayed to it.

Keep in mind, when using a replay subject, that this buffer is held in memory. You can definitely shoot yourself in the foot here, such as if you set a large buffer size for a replay subject of some type whose instances each take up a lot of memory, like images.

Another thing to watch out for is creating a replay subject of an **array** of items. Each emitted element will be an array, so the buffer size will buffer that many arrays. It would be easy to create memory pressure here if you're not careful.

Add this new example to your playground:

```
example(of: "ReplaySubject") {
  // 1
  let subject = ReplaySubject<String>.create(bufferSize: 2)
  let disposeBag = DisposeBag()

  // 2
  subject.onNext("1")
  subject.onNext("2")
  subject.onNext("3")

  // 3
  subject
    .subscribe {
      print(label: "1)", event: $0)
    }
    .disposed(by: disposeBag)
```

```
  subject
    .subscribe {
      print(label: "2)", event: $0)
    }
    .disposed(by: disposeBag)
}
```

From the top, you:

1. Create a new replay subject with a buffer size of 2. Replay subjects are initialized using the type method `create(bufferSize:)`.
2. Add three elements onto the subject.
3. Create two subscriptions to the subject.

The latest two elements are replayed to both subscribers; 1 never gets emitted, because 2 and 3 are added onto the replay subject with a buffer size of 2 before anything subscribed to it.

```
--- Example of: ReplaySubject ---
1) 2
1) 3
2) 2
2) 3
```

Next, add the following code to the example:

```
subject.onNext("4")

subject
  .subscribe {
    print(label: "3)", event: $0)
  }
  .disposed(by: disposeBag)
```

With this code, you add another element onto the subject, and then create a new subscription to it. The first two subscriptions will receive that element as normal because they were already subscribed when the new element was added to the subject, while the new third subscriber will get the last two buffered elements replayed to it.

```
1) 4
2) 4
3) 3
3) 4
```

You're getting pretty good at this stuff by now, so there should be no surprises, here. But what would happen if you threw a wrench into the works? Add this line of code right after adding 4 onto the subject, before creating the third subscription:

```
subject.onError(MyError.anError)
```

This *may* surprise you. And if so, that's OK. Life's full of surprises. :]

```
1) 4
2) 4
1) anError
2) anError
3) 3
3) 4
3) anError
```

What's going on, here? The replay subject is terminated with an error, which it will re-emit to new subscribers — you learned this earlier. But the buffer is also still hanging around, so it gets replayed to new subscribers as well, before the stop event is re-emitted.

Add this line of code immediately after adding the error:

```
subject.dispose()
```

By explicitly calling `dispose()` on the replay subject beforehand, new subscribers will only receive an `error` event indicating that the subject was already disposed.

```
3) Object `RxSwift...ReplayMany<Swift.String>` was already
disposed.
```

Explicitly calling `dispose()` on a replay subject like this isn't something you generally need to do. If you've added your subscriptions to a dispose bag, then everything will be disposed of and deallocated when the owner — such as a view controller or view model — is deallocated.

It's just good to be aware of this little gotcha for those edge cases.

> **Note**: In case you're wondering what is a `ReplayMany`, it's an internal type that is used to create replay subjects.

By using a publish, behavior, or replay subject, you should be able to model almost any need. There may be times, though, when you simply want to go old-school and ask an observable type, "Hey, what's your current value?" Relays FTW here!

# Working with relays

You learned earlier that a relay wraps a subject while maintaining its replay behavior. Unlike other subjects — and observables in general — you add a value onto a relay by using the `accept(_:)` method. In other words, you don't use `onNext(_:)`. This is because relays can only *accept* values, i.e., you cannot add an `error` or `completed` event onto them.

A `PublishRelay` wraps a `PublishSubject` and a `BehaviorRelay` wraps a `BehaviorSubject`. What sets relays apart from their wrapped subjects is that they are *guaranteed* to never terminate.

Add this new example to your playground:

```
example(of: "PublishRelay") {
  let relay = PublishRelay<String>()

  let disposeBag = DisposeBag()
}
```

Nothing new here versus creating a `PublishSubject`, except the name. However, in order to add a new value onto a publish relay, you use the `accept(_:)` method. Add this code to your example:

```
relay.accept("Knock knock, anyone home?")
```

There are no subscribers yet, so nothing is emitted. Create a subscriber and then add another value onto `relay`:

```
relay
  .subscribe(onNext: {
    print($0)
  })
  .disposed(by: disposeBag)

relay.accept("1")
```

The output is the same as if you'd created a publish **subject** instead of a relay:

```
--- Example of: PublishRelay ---
1
```

There is no way to add an `error` or `completed` event onto a relay. Any attempt to do so such as the following will generate a compiler error (don't add this code to your playground, it won't work):

```
relay.accept(MyError.anError)
relay.onCompleted()
```

Remember that publish relays wrap a publish subject and work just like them, except the `accept` part and that they will not terminate. How about something a little more interesting? Say hello to my little friend, `BehaviorRelay`.

Behavior relays also will not terminate with a `completed` or `error` event. Because it wraps a behavior subject, a behavior relay is created with an initial value, and it will replay its latest or initial value to new subscribers. A behavior relay's special power is that you can ask it for its current value at any time. This feature bridges the imperative and reactive worlds in a useful way.

Add this new example to your playground:

```
example(of: "BehaviorRelay") {
  // 1
  let relay = BehaviorRelay(value: "Initial value")
  let disposeBag = DisposeBag()

  // 2
  relay.accept("New initial value")

  // 3
  relay
    .subscribe {
      print(label: "1)", event: $0)
    }
    .disposed(by: disposeBag)
}
```

Here's what you're doing this time:

1. You create a behavior relay with an initial value. The relay's type is inferred, but you could also explicitly declare the type as `BehaviorRelay<String>(value: "Initial value")`.

2. Add a new element onto the relay.

3. Subscribe to the relay.

The subscription receives the latest value.

```
--- Example of: BehaviorRelay ---
1) New initial value
```

Next, add this code to the same example:

```
// 1
relay.accept("1")

// 2
relay
  .subscribe {
    print(label: "2)", event: $0)
  }
  .disposed(by: disposeBag)

// 3
relay.accept("2")
```

From the top:

1. Add a new element onto the relay.
2. Create a new subscription to the relay.
3. Add another new element onto the relay.

The existing subscription `1)` receives the new value `1` added onto the relay. The new subscription receives that same value when it subscribes, because it's the latest value. And both subscriptions receive the `2` when it's added onto the relay.

```
1) 1
2) 1
1) 2
2) 2
```

Finally, add the following piece of code to the last example:

```
print(relay.value)
```

Remember, behavior relays let you directly access their current value. In this case, the latest value added onto the relay is `2`, so that's what is printed to the console.

```
2
```

# Chapter 4: Observables & Subjects in Practice

By Marin Todorov

By this point in the book, you understand how observables and different types of subjects work, and you've learned how to create and experiment with them in a Swift playground.

It could be a bit challenging, however, to see the practical use of observables in everyday development situations such as binding your UI to a data model, or presenting a new controller and getting output back from it.

It's OK to be a little unsure how to apply these newly acquired skills to the real world. In this book, you'll work through theoretical chapters such as Chapter 2, "Observables," and Chapter 3, "Subjects", as well as practical step-by-step chapters — just like this one!

In the "… *in practice*" chapters, you'll work on a complete app. The starter Xcode project will include all the non-Rx code. Your task will be to add the RxSwift framework and add other features using your newly-acquired reactive skills.


That doesn't mean to say you won't learn a few new things along the way — *au contraire*!

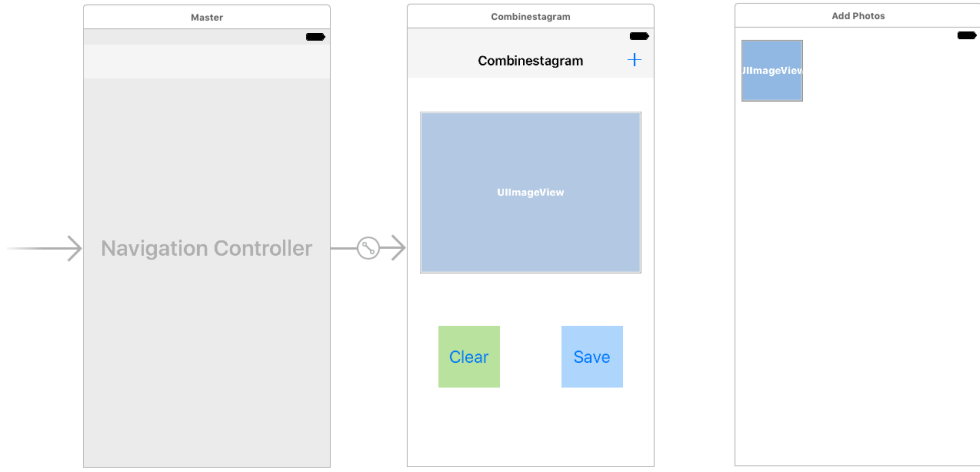
In this chapter, you'll use RxSwift and your new observable superpowers to create an app that lets users create nice photo collages — the reactive way.

# Getting started

Open the starter project for this chapter: **Combinestagram**. It takes a couple of tries to roll your tongue just right to say the name, doesn't it? It's probably not the most marketable name, but it will do.

Install all pods and open **Combinestagram.xcworkspace**. Refer to Chapter 1, "Hello RxSwift," for details on how to do that.

Select **Assets/Main.storyboard** and you'll see the interface of the app you will bring to life:




In the first screen, the user can see the current photo collage and has buttons to either clear the current list of photos or to save the finished collage to disk. Additionally, when the user taps on the + button at the top-right, they will be taken to the second view controller in the storyboard where they will see the list of photos in their Camera Roll. The user can add photos to the collage by tapping on the thumbnails.

The view controllers and the storyboard are already wired up, and you can also peek at **UIImage+Collage.swift** to see how the actual collage is put together.

In this chapter, you are going to focus on putting your new skills to practice. Time to get started!

# Using a subject/relay in a view controller

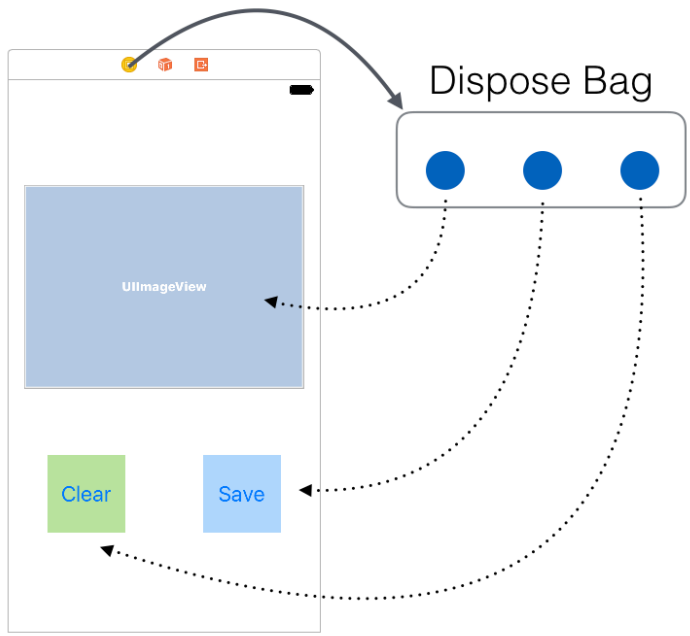
You'll start by adding a `BehaviorRelay<[UIImage]>` property to the controller class and store the selected photos in its `value`. As you learned in Chapter 3, "Subjects", the `BehaviorRelay` class works much like you're used to with plain variables: you can manually change their `value` property any time you want. You will start with this simple example and later move on to subjects and custom observables.

Open **MainViewController.swift** and add the following inside the body of `MainViewController`:

```
private let bag = DisposeBag()
private let images = BehaviorRelay<[UIImage]>(value: [])
```

Since no other class will use those two constants, you define them as `private`. Encapsulation FTW!

The dispose bag is owned by the view controller. As soon as the view controller is released all your observable subscriptions will be disposed as well:




This makes Rx subscription memory management very easy: Simply throw subscriptions in the bag and they will be disposed alongside the view controller's deallocation.

However, that won't happen for this specific view controller, since it's the root view controller and it isn't released before the app quits. You'll see the clever dispose-upon-deallocation mechanism at work later on in this chapter for the other controller in the storyboard.

At first, your app will always build a collage based on the same photo. No worries; it's a nice photo from the Barcelona country side, which is already included in the app's

Asset Catalog. Each time the user taps +, you will add that same photo, one more time, to `images`.

Find `actionAdd()` and add the following to it:

```
let newImages = images.value
  + [UIImage(named: "IMG_1907.jpg")!]
images.accept(newImages)
```

First, you get the latest collection of images emitted by the relay fetching it via its `value` property and then you append one more image to it. Don't mind the force-unwrapping after the `UIImage` initialization, we're keeping things simple by skipping error handling for this chapter.

Next, you use the relay's `accept(_)` to emit the updated set of images to any observers subscribed to the relay.

The initial value of the `images` relay is an empty array, and every time the user taps the + button, the observable sequence produced by `images` emits a new `.next` event with the new array as an element.

To permit the user to clear the current selection, scroll up and add the following to `actionClear()`:

```
images.accept([])
```

With few lines of code in this chapter section, you neatly handled the user input. You can now move on to observing `images` and displaying the result on screen.

## Adding photos to the collage

Now that you have `images` wired up, you can observe for changes and update the collage preview accordingly.

In `viewDidLoad()`, create the following subscription to `images`. Even though its a relay, you can subscribe to it directly, since its conforms to `ObservableType`, much like `Observable` itself does:

```
images
  .subscribe(onNext: { [weak imagePreview] photos in
    guard let preview = imagePreview else { return }

    preview.image = photos.collage(size: preview.frame.size)
  })
  .disposed(by: bag)
```

You subscribe for `.next` events emitted by `images`. For every event, you create a collage with the helper method `collage(images:size:)` provided for arrays of type `UIImage`. Finally, you add this subscription to the view controller's dispose bag.

In this chapter, you are going to subscribe to your observables in `viewDidLoad()`. Later in the book, you will look into extracting these into separate classes and, in the last chapter, structure them into an MVVM architecture. You now have your collage UI together; the user can update `images` by tapping the + bar item (or **Clear**) and you update the UI in turn.

Run the app and give it a try! If you add the photo four times, your collage will look like this:

Wow, that was easy!

Of course, the app is a bit boring right now, but don't worry — you will add the ability to select photos from Camera Roll in just a bit.

## Driving a complex view controller UI

As you play with the current app, you'll notice the UI could be a bit smarter to improve the user experience. For example:

- You could disable the **Clear** button if there aren't any photos selected just yet, or in the event the user has just cleared the selection.
- Similarly, there's no need for the **Save** button to be enabled if there aren't any photos selected.
- You could also disable **Save** for an odd number of photos, as that would leave an empty spot in the collage.

- It would be nice to limit the amount of photos in a single collage to six, since more photos simply look a bit weird.
- Finally, it would be nice if the view controller title reflected the current selection.

If you take a moment to read through the list above one more time, you'll certainly see these modifications could be quite a hassle to implement the non-reactive way.

Thankfully, with RxSwift you simply subscribe to `images` one more time and update the UI from a single place in your code.

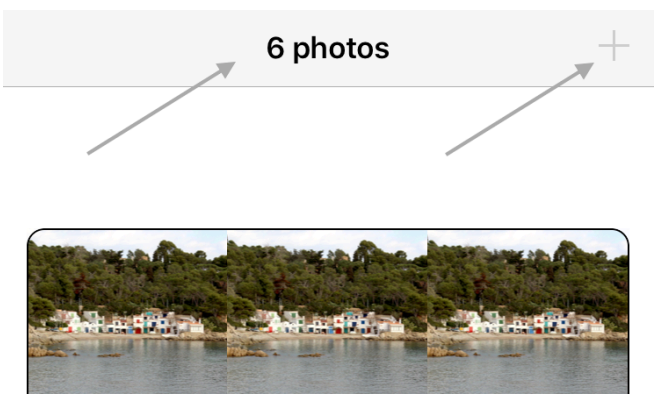
Add this subscription inside `viewDidLoad()`:

```
images
  .subscribe(onNext: { [weak self] photos in
      self?.updateUI(photos: photos)
  })
  .disposed(by: bag)
```

Every time there's a change to the photo selection, you call `updateUI(photos:)`. You don't have that method just yet, so add it anywhere inside the class body:

```
private func updateUI(photos: [UIImage]) {
  buttonSave.isEnabled = photos.count > 0 && photos.count % 2 ==
0
  buttonClear.isEnabled = photos.count > 0
  itemAdd.isEnabled = photos.count < 6
  title = photos.count > 0 ? "\(photos.count) photos" :
"Collage"
}
```

In the above code, you update the complete UI according to the ruleset above. All of the logic is in a single place and easy to read through. Run the app again, and you will see all the rules kick in as you play with the UI:



By now, you're probably starting to see the real benefits of Rx when applied to your iOS apps. If you look through all the code you've written in this chapter, you'll see there are only a few simple lines that drive the entire UI!

# Talking to other view controllers via subjects

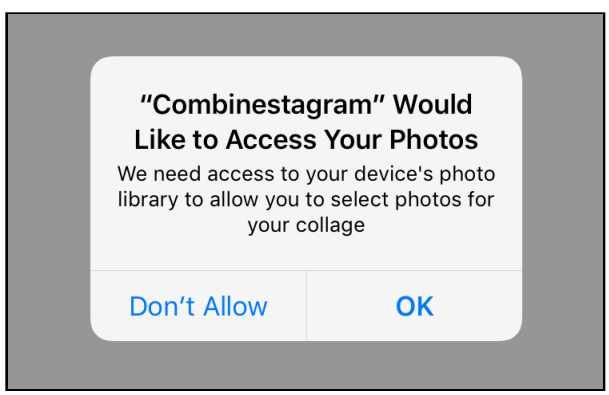
In this section of the chapter, you will connect the `PhotosViewController` class to the main view controller in order to let the user select arbitrary photos from their Camera Roll. That will result in *far* more interesting collages!

First, you need to push `PhotosViewController` to the navigation stack. Open **MainViewController.swift** and find `actionAdd()`. Comment out the existing code and add this code in its place:

```
let photosViewController =
storyboard!.instantiateViewController(
  withIdentifier: "PhotosViewController") as!
PhotosViewController

navigationController!.pushViewController(photosViewController,
animated: true)
```

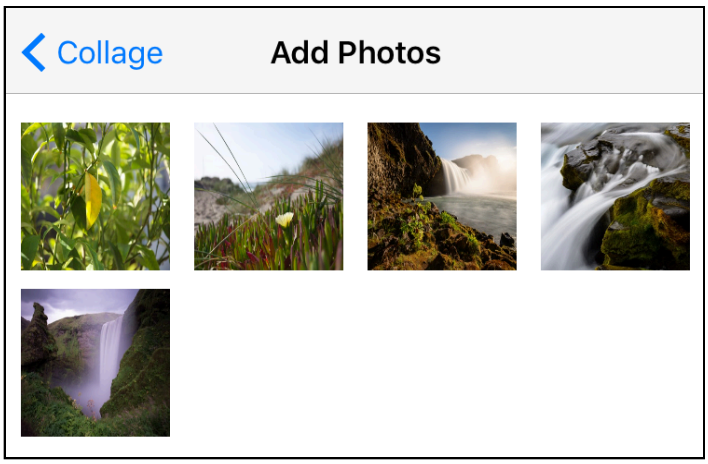
Above, you instantiate `PhotosViewController` from the project's storyboard and push it onto the navigation stack. Run the app and tap + to see the Camera Roll.

The very first time you do this, you'll need to grant access to your Photo Library:

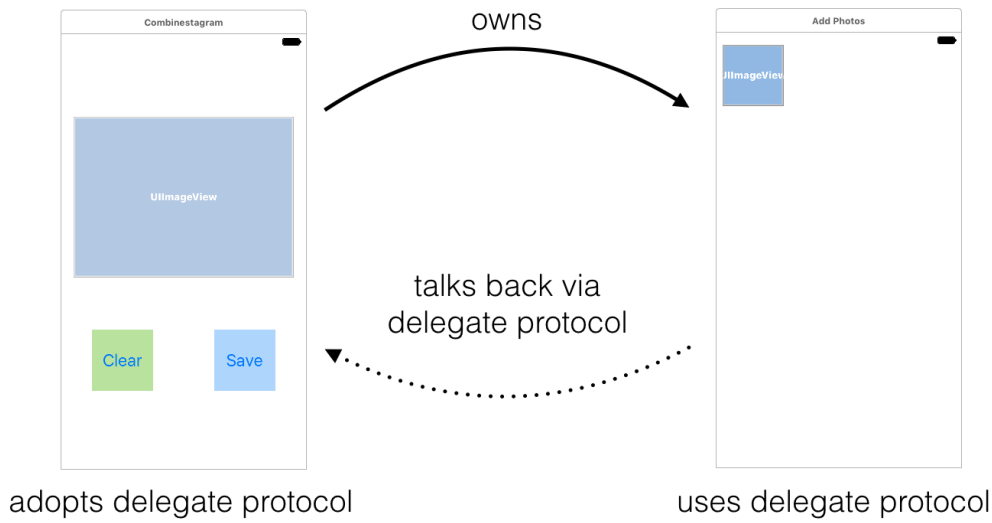


Once you tap **OK** you will see what the photos controller looks like. The actual photos might differ on your device, and you might need to go back and try again after granting access.

The second time around, you should see the sample photos included with the iPhone Simulator.




If you were building an app using the established Cocoa patterns, your next step would be to add a delegate protocol so that the photos controller could talk back to your main controller (that is, the non-reactive way):



With RxSwift, however, you have a universal way to talk between *any* two classes — an `Observable`! There is no need to define a special protocol, because an `Observable` can deliver any kind of message to any one or more interested parties — the observers.

## Creating an observable out of the selected photos

You'll next add a subject to `PhotosViewController` that emits a `.next` event each time the user taps a photo from the Camera Roll.Open **PhotosViewController.swift** and add the following near the top:

```
import RxSwift
```

You'd like to add a `PublishSubject` to expose the selected photos, but you don't want the subject publicly accessible, as that would allow other classes to call `onNext(_)` and make the subject emit values. You might want to do that elsewhere, but not in this case.

Add the following properties to `PhotosViewController`:

```
private let selectedPhotosSubject = PublishSubject<UIImage>()
var selectedPhotos: Observable<UIImage> {
  return selectedPhotosSubject.asObservable()
}
```

Here, you define both a private `PublishSubject` that will emit the selected photos and a public property named `selectedPhotos` that exposes the subject's observable.

Subscribing to this property is how the main controller can observe the photo sequence, without being able to interfere with it.

`PhotosViewController` already contains the code to read photos from your Camera Roll and display them in a collection view. All you need to do is add the code to emit the selected photo when the user taps on a collection view cell.

Scroll down to `collectionView(_:didSelectItemAt:)`. The code inside fetches the selected image and flashes the collection cell to give the user a bit of a visual feedback.

Next, `imageManager.requestImage(...)` gets the selected photo and gives you `image` and `info` parameters to work with in its completion closure. In that closure, you'd like to emit a `.next` event from `selectedPhotosSubject`.

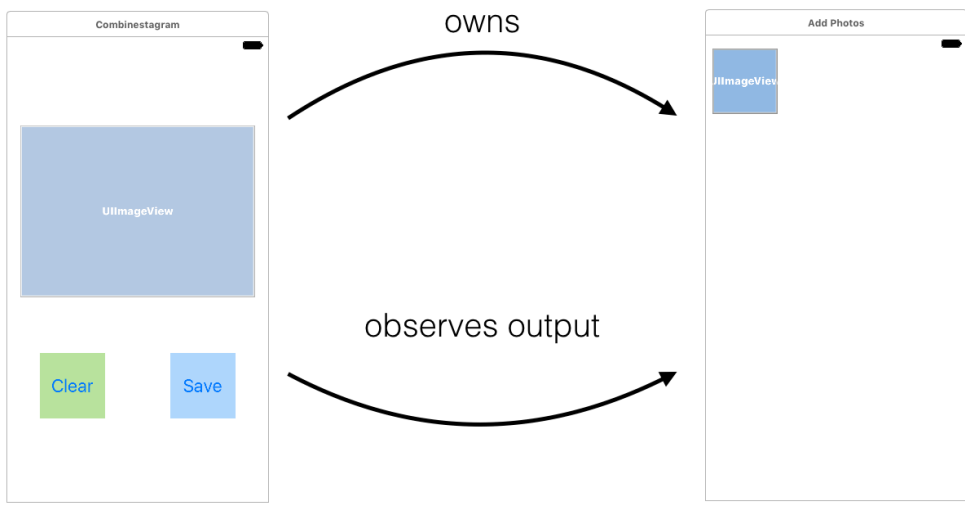
Inside the closure, just after the `guard` statement, add:

```
if let isThumbnail = info[PHImageResultIsDegradedKey as
NSString] as? Bool, !isThumbnail {
  self?.selectedPhotosSubject.onNext(image)
}
```

You use the `info` dictionary to check if the image is the thumbnail or the full version of the asset. `imageManager.requestImage(...)` will call that closure once for each size. In the event you receive the full-size image, you call `onNext(_)` on your subject and provide it with the full photo.

That's all it takes to expose an observable sequence from one view controller to another. There's no need for delegate protocols or any other shenanigans of that sort.

As a bonus, once you remove the protocols, the controllers relationship becomes very simple:




# Observing the sequence of selected photos

Your next task is to return to **MainViewController.swift** and add the code to complete the last part of the schema above: namely, observing the selected photos sequence.

Find `actionAdd()` and add the following just before the line where you push the controller onto the navigation stack:

```
photosViewController.selectedPhotos
  .subscribe(
    onNext: { [weak self] newImage in

    },
    onDisposed: {
      print("Completed photo selection")
    }
  )
  .disposed(by: bag)
```

Before you push the controller, you subscribe for events on its `selectedPhotos` observable. You are interested in two events: `.next`, which means the user has tapped a photo, and also when the subscription is disposed. You'll see why you need that in a moment.

Insert the following code inside the `onNext` closure to get everything working. It's the same code you had before, but this time it adds the photo from Camera Roll:

```
guard let images = self?.images else { return }
images.accept(images.value + [newImage])
```

Run the app, select a few photos from your Camera Roll, and go back to see the result. Cool!

# Disposing subscriptions — review

The code seemingly works as expected, but try the following: Add few photos to a collage, go back to the main screen and inspect the console.

Do you see a message saying, "Completed photo selection"? You added a `print` to your last subscription's `onDispose` closure, but it never gets called! That means the subscription is never disposed and never frees its memory!

How so? You subscribe an observable sequence and throw it in the main screen's dispose bag. This subscription (as discussed in previous chapters) will be disposed of either when the bag object is released, or when the sequence completes via an error or completed event.

Since you neither destroy the main view controller to release its `bag` property, nor complete the photos sequence, your subscription just hangs around for the lifetime of the app!

To give your observers some closure, you could emit a `.completed` event when that controller disappears from the screen. This would notify all observers that the subscription has completed to help with automatic disposal.

Open **PhotosViewController.swift** and add a call to your subject's `onComplete()` method in the controller's `viewWillDisappear(_:)`:

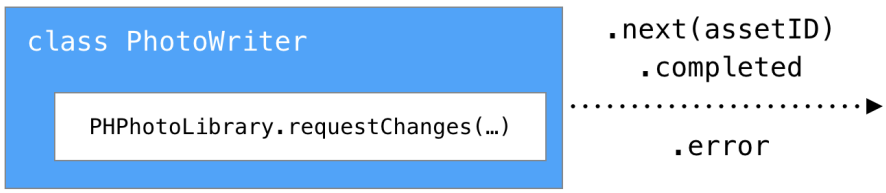
```
selectedPhotosSubject.onCompleted()
```

Perfect! Now, you're ready for the last part of this chapter: taking a plain old boring function and converting it into a super-awesome and fantastical reactive class.

# Creating a custom observable

So far, you've tried `BehaviorRelay`, `PublishSubject`, and an `Observable`. To wrap up, you'll create your own custom `Observable` and turn a plain old callback API into a reactive class. You'll use the `Photos` framework to save the photo collage — and since you're already an RxSwift veteran, you are going to do it the reactive way!

You could add a reactive extension on `PHPhotoLibrary` itself, but to keep things simple, in this chapter you will create a new custom class named `PhotoWriter`:




Creating an `Observable` to save a photo is easy: If the image is successfully written to disk you will emit its asset ID and a `.completed` event, or otherwise an `.error` event.

## Wrapping an existing API

Open **Classes/PhotoWriter.swift** — this file includes a couple of definitions to get you started.

First, as always, add an import of the RxSwift framework:

```
import RxSwift
```

Then, add a new static method to `PhotoWriter`, which will create the observable you will give back to code that wants to save photos:

```
static func save(_ image: UIImage) -> Observable<String> {
  return Observable.create { observer in

  }
}
```

`save(_:)` will return an `Observable<String>`, because, after saving the photo, you will emit a single element: the unique local identifier of the created asset.

`Observable.create(_)` creates a new `Observable`, and you need to add all the meaty logic inside that last closure.

Add the following to the `Observable.create(_)` parameter closure:

```
var savedAssetId: String?
PHPhotoLibrary.shared().performChanges({

}, completionHandler: { success, error in

})
```

In the first closure parameter of `performChanges(_:completionHandler:)`, you will create a photo asset out of the provided image; in the second one, you will emit either the asset ID or an `.error` event.

Add inside the first closure:

```
let request = PHAssetChangeRequest.creationRequestForAsset(from:
image)
savedAssetId =
request.placeholderForCreatedAsset?.localIdentifier
```

You create a new photo asset by using `PHAssetChangeRequest.creationRequestForAsset(from:)` and store its identifier in `savedAssetId`. Next insert into `completionHandler` closure:

```
DispatchQueue.main.async {
  if success, let id = savedAssetId {
    observer.onNext(id)
    observer.onCompleted()
  } else {
    observer.onError(error ?? Errors.couldNotSavePhoto)
  }
}
```

If you got a success response back and `savedAssetId` contains a valid asset ID, you emit a `.next` event and a `.completed` event. In case of an error, you emit either a custom or the default error.

With that, your observable sequence logic is completed.

Xcode should already be warning you that you miss a return statement. As a last step, you need to return a `Disposable` out of that outer closure so add one final line to `Observable.create({})`:

```
return Disposables.create()
```

That wraps up the class nicely. The complete `save()` method should look like this:

```
static func save(_ image: UIImage) -> Observable<String> {
  return Observable.create({ observer in
    var savedAssetId: String?
    PHPhotoLibrary.shared().performChanges({
      let request =
PHAssetChangeRequest.creationRequestForAsset(from: image)
      savedAssetId =
request.placeholderForCreatedAsset?.localIdentifier
    }, completionHandler: { success, error in
      DispatchQueue.main.async {
```

```
          if success, let id = savedAssetId {
            observer.onNext(id)
            observer.onCompleted()
          } else {
            observer.onError(error ?? Errors.couldNotSavePhoto)
          }
        }
      })
      return Disposables.create()
    })
  }
```

If you've been paying attention, you might be asking yourself, "Why do we need an `Observable` that emits just a single `.next` event?"

Take a moment to reflect on what you've learned in the previous chapters. For example, you can create an `Observable` by using any of the following:

- `Observable.never()`: Creates an observable sequences that never emits any elements.
- `Observable.just(_:)`: Emits one element and a `.completed` event.
- `Observable.empty()`: Emits no elements followed by a `.completed` event.
- `Observable.error(_)`: Emits no elements and a single `.error` event.

As you see, observables can produce any combination of zero or more `.next` events, possibly terminated by either a `.completed` or an `.error`.

In the particular case of `PhotoWriter`, you are only interested in one event since the save operation completes just once. You use `.next` + `.completed` for successful writes, and `.error` if a particular write failed.

You get a big bonus point if you're screaming *"But what about Single?"* about now. Indeed, what about `Single`?

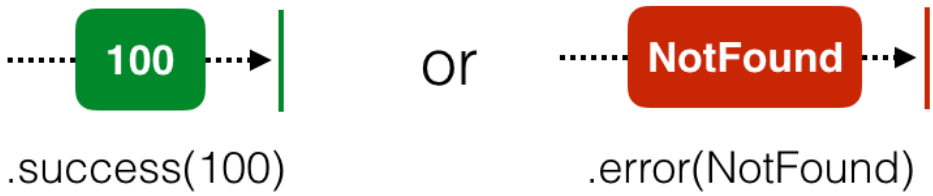
# RxSwift traits in practice

In Chapter 2, "Observables," you had the chance to learn about RxSwift traits: specialized variations of the `Observable` implementation that are very handy in certain cases.

In this chapter, you're going to do a quick review and use some of the traits in the Combinestagram project! Let's start with `Single`.

## Single

As you know from Chapter 2, `Single` is an `Observable` specialization. It represents a sequence, which can emit just once either a `.success(Value)` event or an `.error`. Under the hood, a `.success` is just `.next` + `.completed` pair.




This kind of trait is useful in situations such as saving a file, downloading a file, loading data from disk or basically any asynchronous operation that yields a value. You can categorize two distinct use-cases of `Single`:

1. For wrapping operations that emit exactly one element upon success, just as `PhotoWriter.save(_)` earlier in this chapter.

   You can directly create a `Single` instead of an `Observable`. In fact you will update the `save(_)` method in `PhotoWriter` to create a `Single` in one of this chapter's challenges.

2. To better express your intention to consume a single element from a sequence and ensure if the sequence emits more than one element the subscription will error out.

```
To achieve this, you can subscribe to any observable and use
`.asSingle()` to convert it to a `Single`. You'll try this just
after you've finished reading through this section.
```

## Maybe

`Maybe` is quite similar to `Single` with the only difference that the observable *may* not emit a value upon successful completion.

If we keep to the photograph-related examples imagine this use-case for `Maybe`, your app is storing photos in its own custom photo album. You persist the album identifier in UserDefaults and use that ID each time to "open" the album and write a photo inside. You would design a `open(albumId:) -> Maybe<String>` method to handle the following situations:

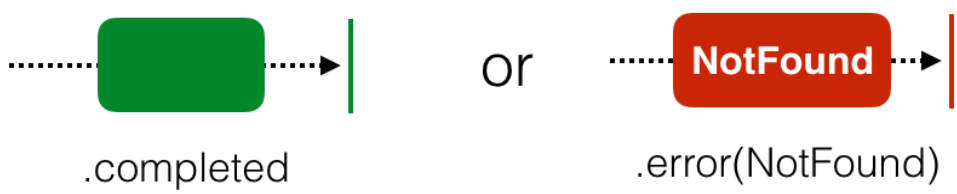
- In case the album with the given ID still exists, just emit a `.completed` event.
- In case the user has deleted the album in the meanwhile, create a new album and emit a `.next` event with the new ID so you can persist it in `UserDefaults`.
- In case something is wrong and you can't access the Photos library at all, emit an `.error` event.

Just like other traits, you can achieve the same functionality with using a "vanilla" `Observable`, but `Maybe` gives more context both to you as you're writing your code and to the programmers coming to alter the code later on.

Just as with `Single`, you can either create a `Maybe` directly by using `Maybe.create({ ... })` or by converting any observable sequence via `.asMaybe()`.

# Completable

The final trait to cover is `Completable`. This variation of `Observable` allows only for a single `.completed` or `.error` event to be emitted before the subscription is disposed of.




You can convert an observable sequence to a completable by using the `ignoreElements()` operator, in which case all next events will be ignored, with only a completed or error event emitted, just as required for a `Completable`.

You can also create a completable sequence by using `Completable.create { ... }` with code very similar to that you'd use to create other observables or traits.

You might notice that `Completable` simply doesn't allow for emitting any values and wonder why would you need a sequence like that. You'd be surprised at the number of use-cases wherein you only need to know whether an async operation succeeded or not.

Let's look at an example before going back to `Combinestagram`. Let's say your app auto-saves the document while the user is working on it. You'd like to asynchronously save the document in a background queue and, when completed, show a small notification or an alert box onscreen if the operation fails.

Let's say you wrapped the saving logic into a function `saveDocument() -> Completable`. This is how easy it is then to express the rest of the logic:

```
saveDocument()
  .andThen(Observable.from(createMessage))
  .subscribe(onNext: { message in
    message.display()
  }, onError: {e in
    alert(e.localizedDescription)
  })
```

The `andThen` operator allows you to chain more completables or observables upon a success event and subscribe for the final result. In case any of them emits an error, your code will fall through to the final `onError` closure.

I'll assume you're delighted to hear that you will get to use `Completable` in two chapters later in the book. And now back to `Combinestagram` and the problem at hand!

## Subscribing to your custom observable

The current feature — saving a photo to the Photos library — falls under one of those special use-cases for which there is a special trait. Your `PhotoWriter.save(_)` observable emits just once (the new asset ID), or it errors out, and is therefore a great case for a `Single`.

Now for the sweetest part of all: making use of your custom-designed `Observable` and kicking serious butt along the way!

Open **MainViewController.swift** and add the following inside the `actionSave()` action method for the **Save** button:

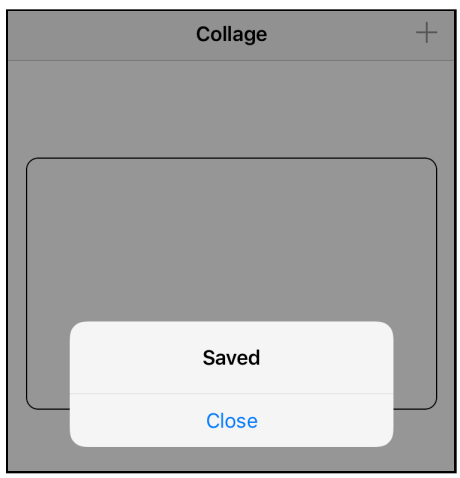
```
guard let image = imagePreview.image else { return }

PhotoWriter.save(image)
```

```
    .asSingle()
    .subscribe(
      onSuccess: { [weak self] id in
        self?.showMessage("Saved with id: \(id)")
        self?.actionClear()
      },
      onError: { [weak self] error in
        self?.showMessage("Error", description:
error.localizedDescription)
      }
    )
    .disposed(by: bag)
```

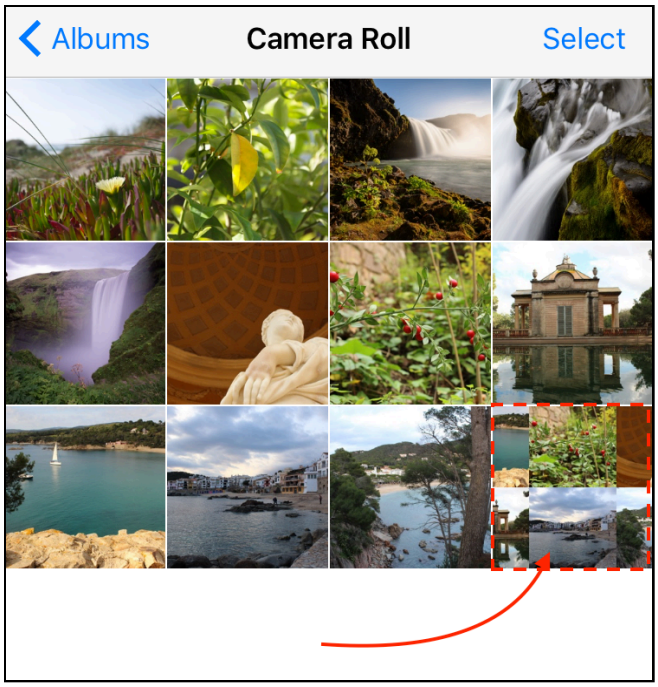
Above you call `PhotoWriter.save(image)` to save the current collage. Then you convert the returned `Observable` to a `Single`, ensuring your subscription will get at most one element, and display a message when it succeeds or errors out. Additionally, you clear the current collage if the write operation was a success.

> **Note**: `asSingle()` ensures that you get at most one element by throwing an error if the source sequence emits more than one.

Give the app one last triumphant run, build up a nice photo collage and save it to the disk.




Don't forget to check your *Photos* app for the result!




With that, you've completed Section 1 of this book — congratulations!

You are not a young Padawan anymore, but an experienced RxSwift Jedi. However, don't be tempted to take on the Dark Side just yet. You will get to battle networking, thread switching, and error handling soon enough!

Before that, you must continue your training and learn about one of the most powerful aspects of RxSwift. In Section 2, "Operators and Best Practices," operators will allow you to take your `Observable` superpowers to a whole new level!

# Challenges

Before you move on to the next section, there are two challenges waiting for you. You will once again create a custom `Observable` — but this time with a little twist.

## Challenge 1: It's only logical to use a Single

You've probably noticed that you didn't gain much by using `.asSingle()` when saving a photo to the Camera Roll. The observable sequence already emits at most one element!

# Chapter 5: Filtering Operators

By Scott Gardner

Learning a new technology stack is a bit like building a skyscraper. You've got to build a solid foundation before you can reach the sky. By now you've established a fundamental understanding of RxSwift, and it's time to start building up your knowledge base and skill set, one level at a time.

This chapter will teach you about RxSwift's filtering operators you can use to apply conditional constraints to emitted events, so that the subscriber only receives the elements it wants to deal with. If you've ever used the `filter(_:)` method in the Swift standard library, you're already half way there. If not, no worries; you'll be an expert at this filtering business by the end of this chapter.

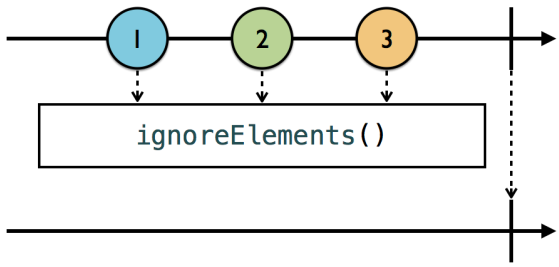
## Getting started

The starter project for this chapter is named **RxPlayground**. After running `./bootstrap.sh` in the project folder, Xcode will open. Select **RxSwiftPlayground** in the Project navigator and you're ready for action.

## Ignoring operators

You're going to jump right in and look at some useful filtering operators in RxSwift, beginning with `ignoreElements`. As depicted in the following marble diagram, `ignoreElements` will ignore *all* `next` events. It will, however, allow stop events through, such as `completed` or `error` events.

Allowing stop events through is usually implied in all marble diagrams. It's explicitly called out this time because that's *all* `ignoreElements` will let through.




> **Note**: Up to now you've seen marble diagrams used for types. This form of marble diagram helps you visualize how *operators* work. The top line is the observable that is being subscribed to. The box represents the operator and its parameters, and the bottom line is the subscriber, or more specifically, what the subscriber will *receive* after the operator does its work.

To see `ignoreElements` in action, add this example to your playground:

```
example(of: "ignoreElements") {
  // 1
  let strikes = PublishSubject<String>()

  let disposeBag = DisposeBag()

  // 2
  strikes
    .ignoreElements()
    .subscribe { _ in
      print("You're out!")
    }
    .disposed(by: disposeBag)
}
```

Here's what you did:

1. Create a `strikes` subject.
2. Subscribe to *all* `strikes`' events, but ignore all `next` events by using `ignoreElements`.

> **Note**: If you don't happen to know much about strikes, batters, and the game of baseball in general, you can read up on that when you decide to take a little break from programming: https://simple.wikipedia.org/wiki/Baseball.

The `ignoreElements` operator is useful when you only want to be notified when an observable has terminated, via a `completed` or `error` event. Add this code to the example:

```
strikes.onNext("X")
strikes.onNext("X")
strikes.onNext("X")
```

Even though this batter can't seem to hit the broad side of a barn and has clearly struck out, nothing is printed, because you're ignoring all `next` events. It's up to you to add a `completed` event to this subject in order to let the subscriber be notified. Add this code to do that:

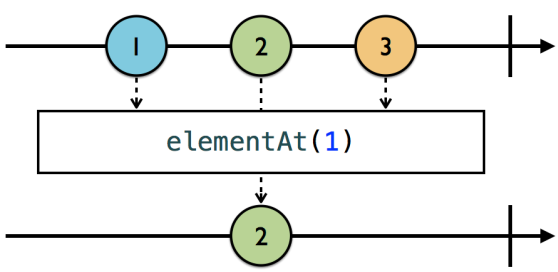
```
strikes.onCompleted()
```

Now the subscriber will receive the `completed` event, and print that catchphrase *no* batter ever wants to hear:

```
--- Example of: ignoreElements ---
You're out!
```

The investigative reader might notice that `ignoreElements` actually returns a `Completable`, which makes sense because it will only emit a `completed` or `error` event.

There may be times when you only want to handle the *nth* (ordinal) element emitted by an observable, such as the third strike. For that, you can use `elementAt`, which takes the index of the element you want to receive, and ignores everything else. In the marble diagram, `elementAt` is passed an index of `1`, so it only lets through the second element.



Add this new example:

```
example(of: "elementAt") {

  // 1
  let strikes = PublishSubject<String>()

  let disposeBag = DisposeBag()

  //  2
  strikes
    .elementAt(2)
    .subscribe(onNext: { _ in
      print("You're out!")
    })
    .disposed(by: disposeBag)
}
```

Here's the play-by-play:

1. You create a subject.
2. You subscribe to the `next` events, ignoring all but the 3rd `next` event, found at index 2.

Now you can simply add new strikes onto the subject, and your subscription will take care of letting you know when the batter has struck out. Add this code:

```
strikes.onNext("X")
strikes.onNext("X")
strikes.onNext("X")
```

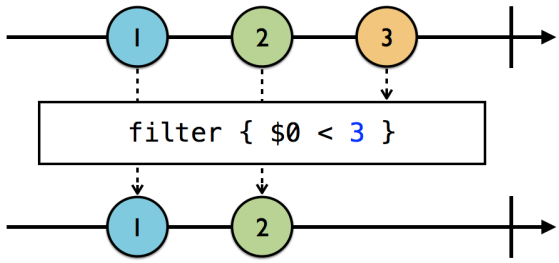
*"Hey batta, batta, batta — swing batta!"*

```
--- Example of: elementAt ---
You're out!
```

An interesting fact about `element(at:)`: As soon as an element is emitted at the provided index, the subscription is terminated.

`ignoreElements` and `elementAt` are filtering elements emitted by an observable. When your filtering needs go beyond all or one, use the `filter` operator. It takes a predicate closure and applies it to every element emitted, allowing through only those elements for which the predicate resolves to `true`.

Check out this marble diagram, where only `1` and `2` are let through because the filter's predicate only allows elements that are less than `3`.




Add this example to your playground:

```
example(of: "filter") {
  let disposeBag = DisposeBag()

  // 1
  Observable.of(1, 2, 3, 4, 5, 6)
    // 2
    .filter { $0.isMultiple(of: 2) }
    // 3
    .subscribe(onNext: {
      print($0)
    })
    .disposed(by: disposeBag)
}
```

From the top:

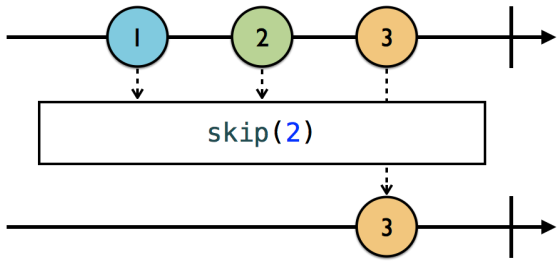
1. You create an observable of some predefined integers.
2. You use the `filter` operator to apply a conditional constraint to prevent odd numbers from getting through.
3. You subscribe and print out the elements that pass the filter predicate.

The result of applying this filter is that only even numbers are printed:

```
--- Example of: filter ---
2
4
6
```

# Skipping operators

When you want to skip a certain number of elements, use the `skip` operator. It lets you ignore the first *n* elements, where *n* is the number you pass as its parameter. This marble diagram shows `skip` is passed 2, so it ignores the first 2 elements.




Add this new example to your playground:

```
example(of: "skip") {
  let disposeBag = DisposeBag()

  // 1
  Observable.of("A", "B", "C", "D", "E", "F")
    // 2
    .skip(3)
    .subscribe(onNext: {
      print($0)
    })
    .disposed(by: disposeBag)
}
```

With this code, you:

1. Create an observable of letters.
2. Use `skip` to skip the first 3 elements and subscribe to `next` events.

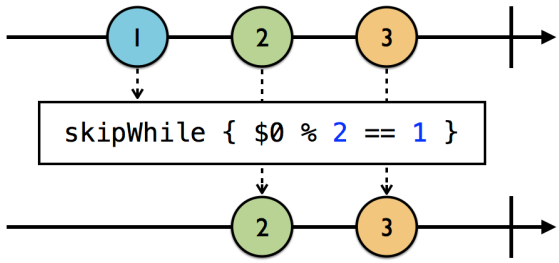
After skipping the first 3 elements, only `D`, `E`, and `F` are printed:

```
--- Example of: skip ---
D
E
F
```

There's a small family of `skip` operators. Like `filter`, `skipWhile` lets you include a predicate to determine what is skipped. However, unlike `filter`, which filters elements for the life of the subscription, `skipWhile` only skips up until something is *not* skipped, and then it lets everything else through from that point on.

And with `skipWhile`, returning `true` will cause the element to be *skipped*, and returning `false` will let it through. It's the opposite of `filter`.

In this marble diagram, `1` is prevented because `1 % 2` equals `1`, but then `2` is allowed through because it fails the predicate, and `3` (and everything else going forward) gets through because `skipWhile` is no longer skipping.




Add this new example to your playground:

```
example(of: "skipWhile") {
  let disposeBag = DisposeBag()

  // 1
  Observable.of(2, 2, 3, 4, 4)
    // 2
    .skipWhile { $0.isMultiple(of: 2) }
    .subscribe(onNext: {
      print($0)
    })
    .disposed(by: disposeBag)
}
```

Here's what you did:

1. Create an observable of integers.
2. Use `skipWhile` with a predicate that skips elements until an odd integer is emitted.

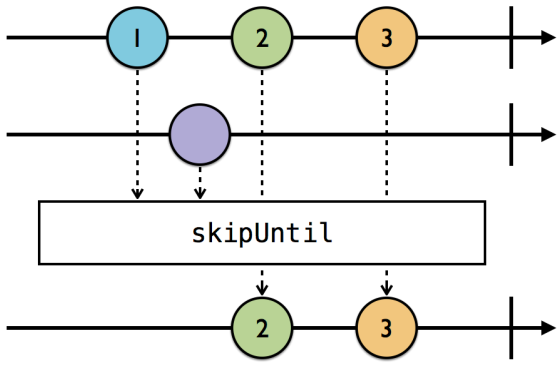
Remember, `skip` only skips elements up until the first element is let through, and then *all* remaining elements are allowed through. So this example prints:

```
--- Example of: skipWhile ---
3
4
4
```

For example, if you are developing an insurance claims app, you could use `skipWhile` to deny coverage until the deductible is met.

So far, you've filtered based on a static condition. What if you wanted to dynamically filter elements based on another observable? There are a couple of operators to choose from.

The first is `skipUntil`, which will keep skipping elements from the source observable — the one you're subscribing to — until some other *trigger* observable emits. In this marble diagram, `skipUntil` ignores elements emitted by the source observable on the top line until the trigger observable on second line emits a `next` event. Then it stops skipping and lets everything through from that point on.




Add this example to see how `skipUntil` works in code:

```
example(of: "skipUntil") {
  let disposeBag = DisposeBag()

  // 1
  let subject = PublishSubject<String>()
  let trigger = PublishSubject<String>()

  // 2
  subject
    .skipUntil(trigger)
    .subscribe(onNext: {
      print($0)
    })
    .disposed(by: disposeBag)
}
```

In this code, you:

1. Create a subject to model the data you want to work with, and another subject to act as a trigger.
2. Use `skipUntil` and pass the `trigger` subject. When `trigger` emits, `skipUntil` stops skipping.

Add a couple of `next` events onto `subject`:

```
subject.onNext("A")
subject.onNext("B")
```

Nothing is printed, because you're skipping. Now add a new `next` event onto `trigger`:

```
trigger.onNext("X")
```

This causes `skipUntil` to stop skipping. From this point onward, all elements are let through. Add another `next` event onto subject:

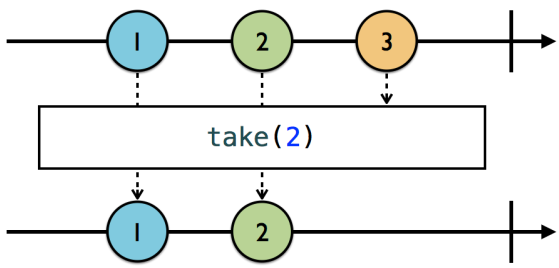
```
subject.onNext("C")
```

Sure enough, it's printed out:

```
--- Example of: skipUntil ---
C
```

# Taking operators

Taking is the opposite of skipping. When you want to take elements, RxSwift has you covered. The first taking operator you'll learn about is `take`, which as this marble diagram depicts, will take the first of the number of elements you specified.




Add this example to your playground to explore the first of the `take` operators:

```
example(of: "take") {
  let disposeBag = DisposeBag()

  // 1
  Observable.of(1, 2, 3, 4, 5, 6)
    // 2
    .take(3)
    .subscribe(onNext: {
      print($0)
    })
    .disposed(by: disposeBag)
}
```