Eimantas Vaičiūnas

ReactiveCocoa Book

Streams of values over time.

Eimantas Vaičiūnas

This book is for sale at http://leanpub.com/reactivecocoa-book

This version was published on 2017-07-18

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Eimantas Vaičiūnas by spreading the word about this book on Twitter!

The suggested hashtag for this book is #ReactiveCocoaBook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#ReactiveCocoaBook

To my family and friends.

Contents

Overview

ReactiveCocoa is a framework that sports extensions to Cocoa and CocoaTouch frameworks. However it is built on top of functionality of ReactiveSwift framework. The latter holds all of the building blocks that make ReactiveCocoa oh so special. This is why this book is split into two major parts: ReactiveSwift and ReactiveCocoa. The first part introduces you to main building blocks used to implement FRP in ReactiveSwift. The second - how all those building blocks are used in Cocoa & Cocoa Touch frameworks as well as how they can be used to make your very own code reactive. Before diving into the details about both frameworks, let's review them shortly and see how they came to be.

The first two major ReactiveCocoa versions had Objective-C API only. And when Swift came out - you could only use Objective-C API. And that wasn't very Swifty[1]. At that time version 3 was developed that introduced first attempts at creating Swift API with custom operators, generic `Signal` class and `SignalProducer` struct that shaped the future of how the ReactiveCocoa API would look. The one major thing that lacked from this framework in version 3 (and 4) was UIKit and AppKit extensions (also known as UI bindings) that it was famous for in version 2. So version 3 and 4 can be looked at as versions that crystalized how ReactiveSwift and ReactiveCocoa API should look. There were still workarounds that many people used on their own to create reactive extensions to UIKit and AppKit frameworks.

Version 5 has been a major overhaul of the framework as a whole. The codebase grew quite a lot through last four major framework versions and contained a lot of code. So it only made sense to split it into separate frameworks that deliver less, but more focused functionality. The framework was split into four.

ReactiveSwift

ReactiveSwift is pure Swift implementation of ReactiveCocoa without any legacy classes/methods coming from the Objective-C API. This framework forms the foundation for the ReactiveCocoa.

ReactiveCocoa

A framework that provides reactive UI bindings for UIKit and AppKit framework. It relies on all of the FRP concepts implemented in ReactiveSwift framework.

[1] this is a default method for observing sent values through a signal;

Objective-C frameworks

ReactiveObjC

Framework for Objective-C users. This framework holds the code from the latest stable 2.x version that dragged along in version 3 and 4. Given it has separate user-base and is independent of Swift code - it has been extracted to a separate framework under different name.

ReactiveObjCBridge

A bridging framework for users with projects panning both, Objective-C and Swift, languages. As authors state in the documentation - it's best if you already have an Objective-C project written with ReactiveCocoa 2 and start migrating to Swift and want to keep reactivity. This framework helps with transition and bridging Objective-C and Swift worlds under single project's roof.

Signals

Let's get back to the "functional reactive programming" term. In the first chapter we saw the building blocks that bring "functional" into functional reactive programming. Now let's talk about "reactive" (also known as "asynchronous") part. The main functionality in ReactiveCocoa is built around the concept of a *Signal*. The concept (according to the authors) is taken from Elm language[2]. Signal is an object that can deliver *Events*. And events can be of different types. So let's get into this slowly. If at first it is hard for you to imagine how things work - think of signals as tubes that have stuff (events) traveling through them. It will make writing the code way easier.

In ReactiveSwift signal is represented as a `class`. The class is generic over two parameters, that define what data will be traveling along with events the signal will deliver. One is for the actual value, the other one is for the type of error in case of failure. Creating a signal is rather easy. You define what type of values/errors its events will carry and init it with a closure that will accept an observer receiving those events:

Example 1.1: Our first signal sends one and two

```
let oneTwoSignal = Signal<Int, AnyError>({ observer in
    /* let's make a signal emit the event */
    observer.send(value: 1)

    /* how about another one */
    observer.send(value: 2)

    /* do some other stuff */

    return nil
})
```

As you can see, the closure that is provided to the initialiser accepts a single argument: *implicitly* created observer for, *not "of"*, the signal. You must use this observer to send it events so that the initialised signal can emit them. Yes, it sounds confusing, but bear with me through this chapter and it will make a lot more sense later on. Let's ignore for now the return result of the closure even though it is `nil`. I promise, you will read about this in the later section.

[2] http://package.elm-lang.org/packages/elm-lang/core/2.1.0/Signal

Events

There are four types of events that a signal can emit. One type contains a value of type `Signal.Value`. Other three types of events are mutually exclusive and terminating (meaning no other events can be emitted after any one of them): `completed`, `failed` and `interrupted`.

Remember when I said `Signal` is a generic class with two parameters? Here is how the signature of the `Signal` looks:

```
public final class Signal<Value, Error: Swift.Error>
```

The `Value` type here describes the type of value that will be delivered with `value` events (e.g. `Int`, `String`, `MyStruct`, etc.). The `Error` type here (that should conform to Swift STL's `Error` type) is a type of error that will be delivered by `failed` events. Here's a comprehensive list of events and what they mean:

- `value(Value)` events are most common ones. They carry a certain value;
- `failed(Error)` event carries an error object that reflects the reason of failure. It is a terminating event and no other events can be emitted after it;
- `completed` event signals (no pun intended) the successful completion of a signal. It is a terminating event and no other events can be emitted after it;
- `interrupted` is a termination event that signifies neither a successful, nor failed completion of a signal but rather a failed subscription to observe a signal. One of the scenarios when `interrupted` event is sent is when trying to observe a signal that has already terminated (in any of the three ways);

Timeline of events

If we consider events on a timeline as a strings of characters, then the general pattern by which the events can arrive from the signal can be described using the following regular expression:

```
/^v*(f|c|i)?$/
```

Here `v` is `value` event bearing a value. Other three letters represent `failed`, `completed` and `interrupted` events accordingly. The `failed` event comes with an error object describing why signal has failed. These three event types terminate the signal and no other events can be emitted by that signal ever again. In terms of pipes it would be as if the pipe has been transferring its own cap. And the `failed` cap would have a label on it saying why it's there (e.g. "Out of stuff" or "Sorry we're closed").

> Tip: There can also be times when you only need to know that event happened without any value transfer. It's perfectly fine to define a signal that emits void events.

> Note about errors: Throughout the book we will be using `AnyError` type for errors. This type is defined in `Result` framework that is one of the `ReactiveSwift` framework's dependencies.

It is also important to note that you can't make a signal send events when you want it to (actually you can, but not at the receiving end). Its code (a closure passed-in during initialisation) decides when and what kind of events should be sent. In other words - it's a *push-driven* stream of events. The event generator decided when the events will be delivered by the signal. You can't ask. You are being told!

The signal is rather useless on its own. That's why we also need to talk about *Observers*.

Observers

Observers are class instances that are generic over the very same two parameters as a signal. So when you try to observe a signal, make sure the observer has same types for values and errors. Observers can be made to observe a single type of event (e.g. `completed` which does not carry any type of value). They perform two completely opposite functions in ReactiveCocoa: observing events (subscribing to signals) and make signals emit them (being event sinks).

Observing events

You can think of observers as user-defined functions that do something with event when it arrives. You can also call them *subscribers* (because they subscribe to signal's events). They are usually implemented as a function per event type or a single function for all possible events:

Example 1.2: Creating observers for `completed` and all events

```
/*
 * Observer that responds only to `Completed` events
 */
let completionObserver = Signal<Int, AnyError>.Observer(completed: {
    print("Show is over, nothing to see here anymore.")
})

/*
 * Observer that handles all of the signal's events
 */
let bigBrother = Signal<Int, AnyError>.Observer({ event in
```

```
    switch(event) {
    case let .value(someInteger):
        print("Got value: \(someInteger)")
    case let .failed(error):
        print("Error: \(error)")
    case .completed:
        print("Signal has successfully completed!")
    case .interrupted:
        print("Attempting to observe completed signal!")
    }
})
```

The latter example above uses actual designated initialiser of `Observer` class. You can also create an observer that observes a select list of events. This is how an observer for `value` and `failed` events would look:

Example 1.3: Creating observer for a select list of events

```
let resultObserver = Signal<Int, AnyError>.Observer(
    value: { print($0 * $0) },
    failed: { print("error: \($0)") }
)
```

Now observing signals becomes as easy as calling one of the observation methods on a signal:

Example 1.4: Observing a signal

```
let signal = Signal<Int, AnyError>({ /* ... */ })

/*
 * Add an observer that looks
 * out for `completed` events.
 */
signal.observe(completionObserver)

/*
 * Add an observer that observes
 * all types of events.
 */
 signal.observe(bigBrother)
```

There is no guarantee in which order observers will receive events so I highly suggest on writing the code that does not rely on this (false) assumption.

Implicit observers

There are also a few shortcuts for implicitly creating observers. Implicit observers are created by calling methods on signals using a set of `observe*` methods. ReactiveSwift is flexible and lets you choose what type of events you want to observe:

- `observeResult`[3]: observe `value` and `failed` events;
- `observeValues`[4]: observe only `value` events carrying a value;
- `observeFailed`: observe only `failed` event that carry an error object;
- `observeCompleted`: observe only signal completion event;

You also saw how you can observe all of the events the observer is interested in by passing the observer to `observe` method.

Each of these methods accept a closure that will be called once the appropriate event is emitted by a signal. The `observeValues` closure accepts a value argument of the `value` event. The `observeFailed` closure accepts an error object that carries a cause for the failure. The `observeCompleted` event does not accept any arguments as is a simple `() -> ()` closure.

There is also a certain set of methods that create implicit observers for signal producers that are going to be discussed in later chapter.

Observing a result

`observeResult` is a new method that arrived to signals when ReactiveSwift 1.0 was separated from ReactiveCocoa 5.0. It tries to follow best coding practices from Swift and tries to be as explicit as possible about developers' intentions on method call sites. Before, framework users could use `observeNext` events (the `value` events were named `next` events before version 5.0) which basically meant "I want only `next` events even if you can send me errors". This have been a cause of a lot of headaches (for yours truly too) when signals don't behave as expected and signal observers are not receiving the values you wanted and ignore the errors. For this very purpose ReactiveCocoa developers decided to replace it with `observeResult` method that accepts a `Result` enum instance with `Value` as `Success` case and `Error` as `Failure` case. This method implicitly creates an observer that subscribes to `value` and `failed` events, but delivers them as one of the cases of `Result` enum. Observe (no pun intended):

[3] this is a default method for observing sent values through a signal;

[4] this method is available only on signals that have `NoError` as an Error value type. Read more about why in Observing a result.

Example 1.5: Creating a pipe for making signal manually emit event

```
let (signal, sink) = Signal<Int, AnyError>.pipe()

signal.observeResult({ result in
    switch result {
    case let .success(number):
        print(number * number)
    case let .failed(error):
        print("Error says: \(error.localizedDescription)")
    }
})

sink.send(value: 2) // prints "4"

let userInfo: [String: Any] = [LocalizedDescriptionKey: "Hi, I am an error!"]
let error = NSError(domain: "com.mycompany.app.errors", code: 0, userInfo: userI\
nfo)

sink.send(error: AnyError(error)) // prints "Hi, I am an error!"
```

As you see this method explicitly requires you to do something about the potential error that can be observed by signal. And the `print` statement there is only for the demo purposes. It's *not* a good way to handle errors.

Sending events

Now you might think that I lied to you that events are sent by the signals. If you look again at the very first code sample in this chapter you will see that an observer is used as an event emitter. Which actually makes signals the transporters of the events rather than emitters. Again - in the first example the object that actually creates the events is implicitly created observer passed into the signal initialisation's closure. As observers that subscribe to signal's events are called subscribers, you can call observers that send events through the signal as *sinks* (which, in my very humble opinion, is very fitting for the tube analogy).

Here is how you emit the events using the observers:

- `.value(Value)`: Emitted by calling `send(value:)` on an instance of an observer
- `.failed(Error)`: Emitted by `send(error:)` method on an instance of an observer;
- `.completed`: Emitted by `sendCompleted()` method.
- `.interrupted`: Emitted by `sendInterrupted()` method.

Pipes

There will be times when you want to manually make signals send events. But the whole setup may require more boilerplate code than necessary (create signals, observers, and then pair them together). That's where pipes come in. Pipes are signal-observer pairs where one is event emitter, and another is event transporter.

Pipes are also called manually controlled signals since you can send events to `observer` and `signal` will instantly emit them. They are useful in making your classes reactive where signal is a public property of a class and observer is a private variable that creates all the events. You will read more about them in chapter "Making things reactive".

You can conveniently create pipes using `Signal` type's `pipe()` method:

Example 1.6: Creating a pipe

```
let (signal, observer) = Signal<Int, NoError>.pipe()
```

This sample shows how to create a pipe that transmits `Int` values and never fails (due to `NoError` error type). Observe (again - no pun intended) that the `pipe()` method call returns a tuple value that contains signal and observer pair. Now you can use `observer` to manually control which events are delivered by the `signal`.

Example 1.7: Observing piped signal

```
1  /*
2   * let's observe signal created above
3   * by printing the sent values
4   */
5
6  signal.observeValues({ print($0)})
7
8  observer.send(value: 0) // prints "0"
9  observer.send(value: 1) // prints "1"
```

I will use this pipe pattern through-out the book to demonstrate miscellaneous signals' features of ReactiveSwift and ReactiveCocoa frameworks. You will also notice that the very same pattern is used in documentation and playgrounds of these frameworks.

Side effects

Side effects are *secondary actions* that can be hooked up to signals on different events or their *aggregates*. The events are the fantastic four that you already know: `value`, `completed`, `failed` and

`interrupted` as well as previously undiscussed `disposed`. The aggregates are *event* (for all of the four events) and *terminated* (for `completed`, `failed` and `interrupted`).

> Note: The `disposed` side effect is a closure that will be executed after the signal completes. These closures are used to clean up after all the work done by the signal (e.g. closing a socket, deleting temporary files, cancelling operations, etc.).

It is important to note that side effects do not affect the event delivered by the signal. Actually, I would rather write this not as a warning, but as an advantage: side-effects let you perform work that does not affect the sent event. Also the side effects will not get triggered when the new observer is added to the signal (as opposed to signal producers' side effects that you will read about later in the book).

The paragraph above is so important that I would rather have you read it thrice.

The side effects are hooked to the signal's events using the `on` operator. Its parameters are named optional closures (accepting values as appropriate). The full method signature is rather long (because you can define all 7 side effects):

```
public func on(
                event: ((Event<Value, Error>) -> Void)? = nil,
                failed: ((Error) -> Void)? = nil,
                completed: (() -> Void)? = nil,
                interrupted: (() -> Void)? = nil,
                terminated: (() -> Void)? = nil,
                disposed: (() -> Void)? = nil,
                value: ((Value) -> Void)? = nil
        ) -> Signal<Value, Error>
```

The `on` method returns another signal that will not only deliver the values as the receiver (the signal where method `on` is being called), but also will execute all the side effects you defined. Again - the side effects will be invoked only when appropriate event is delivered, not when observer is added to the signal.

Let's have a look at how we can use the side effects. The most common/simple scenario is event logging. Let's setup a pipe, send a couple of events and see what gets logged:

Example 1.8: Logging events with side-effects

```
let (signal, sink) = Signal<Int, NoError>.pipe()

let loggedSignal = signal.on(
    completed: { print("Done!") },
    value: { print($0) }
)

sink.send(value: 5)     // prints "5"
sink.sendCompleted()    // prints "Done!"
```

This way you decouple the logic between main and secondary signal's functionality.

Signals are hot!

If you are keen observer (again, no pun intended) you may have noticed that to see a signal do its work they always send the values *after* having observers attached to them. This is because of the nature of signals. Observers can see only values that are sent *after* they decided to observe any type of signal's events. Think of it as wire-tapping on the phone line to listen in on the conversation. You can't hear the conversation if it has already ended and you're only hooking up your headphones now.

This guarantees that all the events sent by the signal will arrive on somewhat the same time to all observers, however the order – which observer will receive the event first - is not known or guaranteed. So don't count on `observer1` receiving events before `observer2` even if the `observer1` has been added before the `observer2`.

Dipping toes in ReactiveSwift & ReactiveCocoa

Now that you know the basic concept of ReactiveSwift framework it's time to put this knowledge into use. Let's start small. As I mentioned in the overview chapter, ReactiveCocoa is a framework built on top of ReactiveSwift. Thus it uses signals (among other things) to provide reactivity for UIKit and AppKit classes. So let's observe signals that are provided by ReactiveCocoa out of the box!

I will skip the boilerplate code with project setup (things like how to add ReactiveSwift and ReactiveCocoa frameworks to the project) and will only provide relevant code samples.

Create a single view application. Then import ReactiveCocoa framework in the view controller created in the application's template:

Header of `ViewController` filelang=swift

```swift
import UIKit
import ReactiveCocoa

class ViewController: UIViewController {
...
```

and add a single text field to the top part of the view controller's view and connect it to the code:

```swift
@IBOutlet var textField: UITextField!
```

Where is reactivity hidden?

The reactivity for existing UIKit and AppKit classes is hidden under `reactive` property of many controls. Xcode's autocompletion can help you find which properties are provided as signals with their values as events. For now there's only that much to know. You will learn more about `reactive` property in dedicated chapter on bindings.

The most used property for text field is its text. ReactiveCocoa provides the signal for text values under a few signals depending on your needs. We are going to utilise `continuousTextValues` signal. This signal emits every value as you edit (type, delete, cut or paste) the text. As I stated before - the signals are provided under the `reactive` property. So to start observing textfield's values you'd write a code similar to this:

Start observing textfield's emitted valueslang=swift

```swift
override func viewDidLoad() {
                super.viewDidLoad()
                let textValuesSignal = textField.reactive.continuousTextValues
                textValuesSignal.observeValues({ (maybeText) in
                                if let text = maybeText {
                                                print(text)
                                }
                })
}
```

Let's walk through the code:

1. We first obtain a reference to the signal that emits values continuously as text field is edited.
2. Then we implicitly create an observer by passing the closure to `observeValues` method of the retrieved signal.

3. The observer closure receives `String?` parameter. We then check whether there's any value in it and if there is - we print it.

   Note: ReactiveSwift framework matches the type of signal's value events with control's attribute. This is why the `continuousTextValues` signal emits `String?` and not `String` even if it always emits a text value when it's present.

   Note: This exact signal will not emit any events when you are trying to delete a character in empty text field (i.e. tapping backspace when no text is present).

This piece of code covers two main components of ReactiveSwift:

1. Signals - we utilise a signal provided by the `continuousTextValues` property.
2. Observers - as noted - observers are functions that do something with a value. We pass a closure that accepts textfield's text as a `String?` typed argument.

Let's flex our observation muscles and observe the signal in a different way. Let's create explicit observer and use it to observe the text values. First we will need to update our import statements. In previous example we didn't use any explicitly types from ReactiveSwift. Now that we are going to explicitly create an observer we need to know where to get it, thus let's import ReactiveSwift:

```
1 import UIKit
2 import ReactiveCocoa
3 import ReactiveSwift
```

If you alt-click on the `textValuesSignal` variable you'll notice that that signal emits errors of `NoError` type (this means that this signal will never emit an error). I mentioned that observers must match the signal's, they are going to observe, events' value types. Thus we'll need to declare an `Observer` with `String?` as value type and `NoError` as error type. The `NoError` type is declared in a dependency of ReactiveSwift framework: the Result framework. This is a utility framework that bears a lot of useful functionality with it. Let's import it too:

```
1 import UIKit
2 import ReactiveCocoa
3 import ReactiveSwift
4 import Result
```

Finally we can start writing the code now that view controller knows about available types. First some refactoring. Let's extract the implicit observer function into separate function inside the VC:

```
private func printFieldValueIfPresent(_ maybeValue: String?) {
    guard let text = maybeText else { return }
    print(text)
}
```

Now let's create our observer and attach it to the signal:

```
override func viewDidLoad() {
    super.viewDidLoad()

    let valueObserver = Observer<String?, NoError>(value: printFieldValueIfPrese\
nt)

    let textValueSignal = textField.reactive.continuousTextValues
    textValueSignal.observe(valueObserver)
}
```

Relaunch the app. The functionality (as much as there is of it) did not change. You still see the console printing the text field text as you edit it. You can also stop here and appreciate how clean this code looks. Only 3 lines of code and they do so much!

Conclusion

There was a lot covered in this very first chapter of the book, but it's only the beginning. I understand that it's mostly theory, bit it's a necessary evil for you to understand the true power provided by ReactiveSwift. All of the concepts introduced here are reused all over the book in later chapters when introducing new material.

I have also found that starting to learn ReactiveSwift and ReactiveCocoa it's easier to start with signals that are already provided to you. This way you don't have to worry about emitting events. You only need to keep the reference to a signal and attach observers. The example in this chapter illustrates the simplest example of how you can take an existing signal for a text field and observe the changes to it without writing a lot of code.

Operators

You can't expect ReactiveCocoa to provide the signals for all possible occasions. You can't also be expected to create signals and their producers for every possible occasion too. Sometimes it's just easier to change the events that are being emitted by the signals you already have. This change is possible by using the *operators*. Operators are functions that can do a lot of interesting stuff with the signals or their emitted values. Some of the popular ones that come bundled with ReactiveCocoa are:

- `take` - takes up to N values sent by the signal;
- `throttle` - sends new signal's values no more often than provided time interval (and it's companion `debounce`);
- `combineLatest` - a static operator that combines values from variable number of signals (or producers) and forwards those values as tuple on returned signal (or producer). Note that the returned signal (producer) will emit a value only when *all* combined signals (producers) have emitted at least one value. They also must have the same error type;

There are a lot more operators and they will not be covered in this book, but I encourage you to check out the interface of the `Signal` class and its extensions.

Flattening operators & strategies

Special attention must be given to flattening operators: `flatten` and its more popular counterpart - `flatMap` (which is a combination of `map` followed by `flatten`). Just like the function in Swift's standard library, `flatten` "reduces the dimensionality" of the signal it is being called on. Thus it is available only for the signals that emit other signals as their events' values.

> Note: Even though the `flatMap` operator and all its variations is available on both signals and signal producers I will mention only signals as to not be repetitive.

I would like to return to the analogy of the tubes. Flatten operators would be tubes sending another tubes that send stuff. And when sent tube arrives at the end of the sending tube, it's being attached to that tube instantly.

There are three ways (strategies in ReactiveSwift terms) to flatten the signals: merge, concatenate and switch to the latest.

Each of the strategies differ in a way they process sent signals. To make it easier to understand how they work let's get acquainted with a few terms:

- Receiver: a signal that `flatMap` is being called on;
- Base signal: a signal that is returned from the receiver's `flatMap` call;
- Inner signal: a signal that is sent by the "Base signal" (first it's the receiver and then it's anything that is sent by the receiver itself);

All of the strategies handle the failure event the same way - `failed` event from the inner signal is instantly forwarded to the base signal that is then terminated.

Merging

Merging strategy (`.merge`) subscribes to the inner signal as soon as it's being sent by the base signal's `value` event. When next signal is sent down the pipe - base signal subscribes to it too without terminating the previous subscription.

This strategy is useful for branching the signals into different layers of your application's architecture.

Concatenation

Concatenation strategy (`.concat`) subscribes to the new inner signal as soon as the previously sent inner signal completes. Thus sending the next signal (before the current inner signal has terminated) puts it into the queue to be started after the current inner signal terminates.

This strategy is useful to perform the job in serial manner. I.e. do not start a new job (i.e. processing newly sent signal's events) until the current one has done its part.

Switch to the latest

"Switch to the latest" strategy (`.latest`) resubscribes to inner signal as soon as the base signal sends a new value down the pipe. The previously sent signal doesn't have to complete for re-subscription to happen. Also when new inner signal is sent, the subscription to the previous one is terminated.

Creating operators

The selection of the operators is not limited to the existing ones in the ReactiveCocoa framework. If you find yourself reusing an operator or combination of operators for some specific task - feel free to create a separate operator that carries a better name.

Swift's extensions provide a very powerful mechanism to create new operators for signals that emit certain type of events carrying selective types of values or errors.

Operators can be unary, binary, tertiary or of any other *arity*. ReactiveCocoa supports implementation of unary and binary operators. Unary operators are operating on a single signal or their sent values.

Creating unary operators

Let's create a few simple unary operators.

The most common unary operator that you can think of is logical negation (or "NOT") operator. Let's reimplement this operator for signals that send boolean values. ReactiveSwift employs Swift's protocol extension for most (if not all) of its operators. We are going to be following that example too:

lang=swift

```swift
extension Signal where Value == Bool {
    public func negate() -> Signal<Value, Error> {
        return map({ bool in
            return !bool
        })
    }
}
```

This is a simple wrapper around a `map` operator that clearly states the intention about the operation it is making. Also, since we're using the Swift extension with `where` clause, this operator won't be accessible to other signals that emit different type of values!

I think this would look way nicer if we use the shorthand syntax for the `map` function:

lang=swift

```swift
extension SignalProtocol where Value == Bool {
    public func negate() -> Signal<Value, Error> {
        return map(!)
    }
}
```

That's it! No more work is needed for this operator. Now you can create negated versions of bool-sending signal by applying the `not()` operator:

lang=swift

```swift
let (boolSignal, boolSink) = Signal<Bool, NoError>.pipe()
let negatedBoolSignal = boolSignal.negate()
negatedBoolSignal.observeValues({ print($0) })

boolSink.send(value: true)  // prints false
boolSink.send(value: false) // prints true
```

But when you look at line 2 it does not read quite well. You as a developer are used to using exclamation mark as a negation operator (I hope), so this `not()` call feels unnatural. Let's add some syntax sugar to this operator.

Operators in Swift are defined using the `operator` keyword with respective modifiers for unary and binary versions and then implemented with public functions that bear the same name as the operator itself.

Let's define a function for the negation operator that will take a bool-sending signal as an argument:

lang=swift

```swift
prefix public func !<E: Error>(a: Signal<Bool, E>) -> Signal<Bool, E> {
        return a.negate()
}
```

Now we can create signals that send negated values of any bool-sending signals using very well known exclamation mark:

lang=swift

```swift
let (boolSignal, boolSink) = Signal<Bool, NoError>.pipe()
let negatedBoolSignal = !boolSignal // much better!
negatedBoolSignal.observeValues({ print($0) })

boolSink.send(value: true)  // prints false
boolSink.send(value: false) // prints true
```

Another simple operator that we can implement is a string length operator. Given the complex nature of string length calculation in Swift we will use most common length calculation approach and implement the `UTF8Length` operator. It will operate on signals of value `String` and return value's length in UTF8 encoding:

lang=swift

```swift
extension SignalProtocol where Value == String {
    func toUTF8Length() -> Signal<Int, Error> {
        return map { $0.lengthOfBytes(using: .utf8) }
    }
}
```

Sadly there is no common symbolic operator to reflect calculation of string's length, so there's not much more what we can do about this operator.

Both of the unary operators above use a single call to `map` function and returns signal with values that are results of these function calls.

Let's add some variety and implement an operator that passes only non-zero length strings. Let's call it `skipEmptyStrings()`. First let's try to lay this process out not in the body of the operator's function:

lang=swift

```swift
let (stringSignal, stringSink) = Signal<String, NoError>.pipe()

let lengthSignal = stringSignal.toUTF8Length()

let tupleSignal = lengthSignal.zipWith(stringSignal)

let nonZeroLengthTupleSignal = zippedSignal.filter {  $0.0 > 0 }

let nonZeroLengthStringSignal = nonZeroLengthTupleSignal.map { $0.1 }
```

This is a bit more complex operator so let's dissect this implementation line by line:

1. create manually controlled string signal that we will use to test the resulting signal;
2. whitespace for readability;
3. create a signal that sends string's length by reusing our previously defined `toUTF8Length()` operator;
4. whitespace for readability;
5. create the signal that sends tuples of (or "zipped" otherwise) values. The `zip(with:)` operator ties the receiver's values (sent by `lengthSignal`) with argument's values (sent by `stringSignal`) in synchronous manner. The Nth value from `lengthSignal` will be paired in a tuple with Nth value from `stringSignal`.
6. whitespace for readability;

7. create the signal that throws away values that have 0 length as tuple argument (i.e. pass only values satisfying the condition `(0, "").0 > 0`;
8. whitespace for readability;
9. create the final signal that extracts only the string value from the tuple and returns it;

> Note: One constraint about the `zip(with:)` operator - zipping signals (receiver and argument) must emit same type of failed events.

Now let's wrap it into the function for protocol extension. Some intermediate steps were omitted for brevity:

Signal operator as an extension to a SignalProtocollang=swift

```
extension SignalProtocol where Value == String {
    public func ignoreEmptyStrings() -> Signal<Value, Error> {
        return lengthSignal
            .zip(with: toUTF8Length())
            .filter {  $0.0 > 0 }
            .map { $0.1 }
    }
}
```

Creating a binary operator

Now let's look at the binary operators. Binary operators in ReactiveCocoa are operators that take other signals (or producers) as their arguments. The `zip(with:)` operator is an example of such binary operator.

For the demo purposes of this mini tutorial, let's create a simple `and` binary operator that performs conjunction with boolean values sent by two different signals.

Again, we will start with a function in `SignalProtocol` extension constrained to a `Value` of `Bool` type:

A binary operator for a signallang=swift

```
extension SignalProtocol where Value == Bool {
    public func and(otherSignal: Signal<Value, Error>) -> Signal<Value, Error> {
        return combineLatest(with: otherSignal).map { (b1, b2) in
            return b1 && b2
        }
    }
}
```

I also want to take this moment and make use of shorthand parameter names in Swift's closures and make `and()` operator even clearer:

lang=swift

```
extension SignalProtocol where Value == Bool {
    public func and(otherSignal: Signal<Value, Error>) -> Signal<Value, Error> {
        return combineLatest(with: otherSignal)
                .map { $0.0 && $0.1 }
    }
}
```

> Note: We are using `combineLatest(with:)` signal operator to combine the values of the receiver and the argument into a tuple. The main difference from `zipWith` operator (which also combines values into tuples) is that the resulting signal will send a value when *any* one of the receiver or argument sends a value. `combineLatest(with:)` operator has the same constraint as the `zipWith` operator: signals must send the same type of failed events.

And for the syntactic sugar — let's define a function that handles the variant of `&&` operator for the boolean signals:

lang=swift

```
public func &&<E: Error>(lhs: Signal<Bool, E>, rhs: Signal<Bool, E>) -> Signal<B\
ool, E> {
    return lhs.and(rhs)
}
```

Let's apply this operator in a login form scenario. Consider two signals that send `Bool` value:

lang=swift

```
let usernameSignal = <#Get Signal<String, Error> emitting username#>
let passwordSignal = <#Get Signal<String, Error> emitting password#>
```

We have previously created an operator that returns lengths of the strings emitted by signals. We can create new signals that send lengths and see if both of these lengths are greater than zero.

lang=swift

```
// you'd probably want
// this as an operator too
let isNonZeroLength: (Int) -> Bool = { $0 > 0 }

let didFillUsername = usernameSignal.toUTF8Length()
                        .map(isNonZeroLength)
let didFillPassword = passwordSignal.toUTF8Length()
                        .map(isNonZeroLength)
```

Now let's combine those signals using our new `and` operator and observe a resulting value:

lang=swift

```
didFillLoginForm = (didFillUsername && didFillPassword)

didFillLoginForm.observeValues({ filled in
    if filled {
        /*
         * form is filled, enable the login
         * button or change some field colors
         */
    } else {
        /*
         * disable login button, show notice asking
         * to enter both username and password
         */
    }
})
```

Lifting

Most operators in ReactiveSwift are implemented in `SignalProtocol` extensions. However framework's design guidelines advise you to be a good citizen and provide the same variant for signal producers and properties. This is where operator lifting comes in. Every operator that you create for the signal can be implemented with minimum code for signal producers and properties too! This is done through `lift` function that takes unary or binary operators from `SignalProtocol` and applies them to `SignalProducerProtocol` and `PropertyProtocol`.

Here's how you'd lift a `negate()`:

Lifting `negate()` operator to signal producers and properties

```
extension SignalProducerProtocol where Value == Bool {
                func negate() -> SignalProducer<Value,Error> {
                                return self.lift { $0.negate() }
                }
}

extension PropertyProtocol where Value == Bool {
                func negate() -> Property<Value> {
                                return self.lift { $0.negate() }
                }
}
```

As you can see these two code snippets (apart from the extension declaration and return value) are identical. `lift` function works by accepting a function of unary or binary operator and creating a signal producer operator out of a signal operator and property operator out of signal producer operator. This means that you can't lift a signal operator straight to a property space (unless you implement the `lift` function that accepts signal operator and returns transformed property).

Conclusion

This chapter showed you how you can use a myriad of operators provided by ReactiveCocoa and create many more for your own use! Swift's protocol extensions provide an easy way to scope the availability of those operators so that operators are available only based on the context of the code.