Up to date for iOS 13, Xcode 11 & Swift 5.1

# Combine

## Asynchronous Programming with Swift

FIRST EDITION

By the raywenderlich.com Tutorial Team

Scott Gardner, Shai Mishali, Florent Pillet & Marin Todorov

# Combine: Asynchronous Programming with Swift

By Scott Gardner, Shai Mishali, Florent Pillet & Marin Todorov



## Notice of Rights



## Notice of Liability



## Trademarks

# Dedications

"To Jenn, for being so supportive and encouraging. To Charlotte, keep up the great work in school — you motivate me! To Betty, my best l'il friend for all her 18 years. And to you, the reader — you make this work meaningful and fulfilling."

*— Scott Gardner*

"For my wife Elia and Baby Ethan—my love, inspiration, and rock ❤️. To my family and friends for their support: Dad, Mom, Ziv, Adam, and everyone else, you're the best!"

*— Shai Mishali*

"To Fabienne and Alexandra ❤️."

*— Florent Pillet*

"To my father. To my mom. To Mirjam and our beautiful daughter."

*— Marin Todorov*

# About the Authors

**Scott Gardner** is an author and the technical editor for this book. Combined, he's authored over a dozen books, video courses, tutorials, and articles on Swift and iOS app development — with a focus on reactive programming. He's also presented at numerous conferences. Additionally, Scott teaches app development and is an Apple Certified Trainer for Swift and iOS. Scott has been developing iOS apps since 2010, ranging from personal apps that have won awards to working on enterprise teams developing apps that serve millions of users. You can find Scott on Twitter or GitHub as @scotteg or connect with him on LinkedIn at scotteg.com.

**Shai Mishali** is an author and the final pass editor on this book. He's the iOS Tech Lead for Gett, the global on-demand mobility company; as well as an international speaker, and a highly active open-source contributor and maintainer on several high-profile projects - namely, the RxSwift Community and RxSwift projects, but also releases many open-source endeavors around Combine such as CombineCocoa, RxCombine and more. As an avid enthusiast of hackathons, Shai took 1st place at BattleHack Tel-Aviv 2014, BattleHack World Finals San Jose 2014, and Ford's Developer Challenge Tel-Aviv 2015. You can find him on GitHub and Twitter as @freak4pc.

**Florent Pillet** is an author of this book. He has been developing for mobile platforms since the last century and moved to iOS on day 1. He adopted reactive programming before Swift was announced, using it in production since 2015. A freelance developer, Florent also uses reactive programming on the server side as well as on Android and likes working on tools for developers like the popular NSLogger when he's not contracting, training or reviewing code for clients worldwide. Say hello to Florent on Twitter and GitHub at @fpillet.

**Marin Todorov** is an author of this book. Marin is one of the founding members of the raywenderlich.com team and has worked on eight of the team's books. He's an independent contractor and has worked for clients like Roche, Realm, and others. Besides crafting code, Marin also enjoys blogging, teaching and speaking at conferences. He happily open-sources code. You can find out more about Marin at www.underplot.com.

## About the Artist

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

# Table of Contents: Overview

# Table of Contents: Extended

# Early Access Edition

You're reading an early access edition of *Combine: Asynchronous Programming with Swift*. This edition contains a sample of the chapters that will be contained in the final release.

We hope you enjoy the preview of this book, and that you'll come back to help us celebrate the full launch of *Combine: Asynchronous Programming with Swift* later in 2019!

The best way to get update notifications is to sign up for our monthly newsletter. This includes a list of the tutorials that came out on raywenderlich.com that month, any important news like book updates or new books, and a list of our favorite development links for that month. You can sign up here:

- www.raywenderlich.com/newsletter

# What You Need

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

- A Mac running **macOS Mojave** (10.14) or later. Earlier versions might work, but they're untested.
- **Xcode 11 or later**. Xcode is the main development tool for iOS. You'll need Xcode 11 or later for the tasks in this book, since Combine was introduced with the iOS 13 SDK. You can download the latest version of Xcode from Apple's developer site here: apple.co/2asi58y
- **An intermediate level knowledge of Swift**. This book teaches you how to write declarative and reactive iOS applications using Apple's Combine framework. Combine uses a multitude of advanced Swift features such as generics, so you should have at least an intermediate-level knowledge of Swift.

If you want to try things out on a physical iOS device, you'll need a developer account with Apple, which you can obtain for free. However, all the sample projects in this book will work just fine in the iOS Simulator bundled with Xcode, so a paid developer account is completely optional.

# Book License

By purchasing *Combine: Asynchronous Programming with Swift*, you have the following license:

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

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

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

# B Book Source Code & Forums

## If you bought the digital edition

This book comes with the source code for the starter and completed projects for each chapter. These resources are shipped with the digital edition you downloaded here:

- https://store.raywenderlich.com/products/combine-asynchronous-programming-with-swift.

## If you bought the print version

You can get the source code for the print edition of the book here: https://store.raywenderlich.com/products/combine-asynchronous-programming-with-swift-source-code.

And if you purchased the print version of this book, you're eligible to upgrade to the digital editions at a significant discount! Simply email support@razeware.com with your receipt for the physical copy and we'll get you set up with the discounted digital edition version of the book.

## Forums

We've also set up an official forum for the book here:

- https://forums.raywenderlich.com.

This is a great place to ask questions about the book or to submit any errors you may find.

# Digital book editions

We have a digital edition of this book available in both ePUB and PDF, which can be handy if you want a soft copy to take with you, or you want to quickly search for a specific term within the book.

Buying the digital edition version of the book also has a few extra benefits: free updates each time we update the book, access to older versions of the book, and you can download the digital editions from anywhere, at anytime.

Visit our book store page here:

- https://store.raywenderlich.com/products/combine-asynchronous-programming-with-swift.

# Section I: Introduction to Combine

In this part of the book, you're going to ramp up over the basics of Combine and learn about some of the building blocks it comprises. You'll learn what Combine aims to solve and what are some of the abstractions it provides to help you solve them: Publisher, Subscriber, Subscription, Subject and much more.

Specifically, you'll cover:

**Chapter 1: Hello Combine**: A gentle introduction to what kind of problems Combine solves, a back story of the roots of reactive programming on Apple's platforms, and a crash course into the basic moving pieces of the framework.

**Chapter 2: Publishers & Subscribers**: The essence of Combine is that publishers send values to subscribers. In this chapter you'll learn all about what that means and how to work with publishers and subscribers, and how to manage the subscriptions that are created between the two of them.

# Chapter 1: Hello, Combine!

By Marin Todorov

This book aims to introduce you to the Combine framework and to writing declarative and reactive apps with Swift for Apple platforms.

In Apple's own words: "*The Combine framework provides a declarative approach for how your app processes events. Rather than potentially implementing multiple delegate callbacks or completion handler closures, you can create a single processing chain for a given event source. Each part of the chain is a Combine operator that performs a distinct action on the elements received from the previous step*."

Although very accurate and to the point, this delightful definition might sound a little too abstract at first. That's why, before delving into coding exercises and working on projects in the following chapters, you'll take a little time to learn a bit about the problems Combine solves and the tools it uses to do so.

Once you've built up the relevant vocabulary and some understanding of the framework in general, you'll move on to covering the basics while coding.

Gradually, as you progress in the book, you'll learn about more advanced topics and eventually work through several projects.

When you've covered everything else you will, in the last chapter, work on a complete app built with Combine.

# Asynchronous programming

In a simple, single-threaded language, a program executes sequentially line-by-line. For example, in pseudocode:

```
begin
  var name = "Tom"
  print(name)
  name += " Harding"
  print(name)
end
```

Synchronous code is easy to understand and makes it especially easy to argue about the state of your data. With a single thread of execution, you can always be sure what the current state of your data is. In the example above, you know that the first `print` will always print "Tom" and the second will always print "Tom Harding".

Now, imagine you wrote the program in a multi-threaded language that is running an asynchronous event-driven UI framework, like an iOS app running on Swift and UIKit.

Consider what could potentially happen:

```
--- Thread 1 ---
begin
  var name = "Tom"
  print(name)

--- Thread 2 ---
name = "Billy Bob"

--- Thread 1 ---
  name += " Harding"
  print(name)
end
```

Here, the code sets `name`'s value to `"Tom"` and then adds `"Harding"` to it, just like before. But because another thread could execute at the same time, it's possible that some other part of your program could run between the two mutations of `name` and set it to another value like `"Billy Bob"`.

When the code is running concurrently on different cores, it's difficult to say which part of the code is going to modify the shared state first.

The code running on "Thread 2" in the example above might be:

- executing at exactly the same time on a different CPU core as your original code.

- executing just before `name += " Harding"`, so instead of the original value `"Tom"`, it gets `"Billy Bob"` instead.

What exactly happens when you run this code depends on the system load, and you might see different results each time you run the program.

Managing mutable state in your app becomes a loaded task once you run asynchronous concurrent code.

## Foundation and UIKit/AppKit

Apple has been improving asynchronous programming for their platforms over the years. They've created several mechanisms you can use, on different system levels, to create and execute asynchronous code. You've probably used these in your projects without giving them a second thought because they are so fundamental to writing mobile apps.

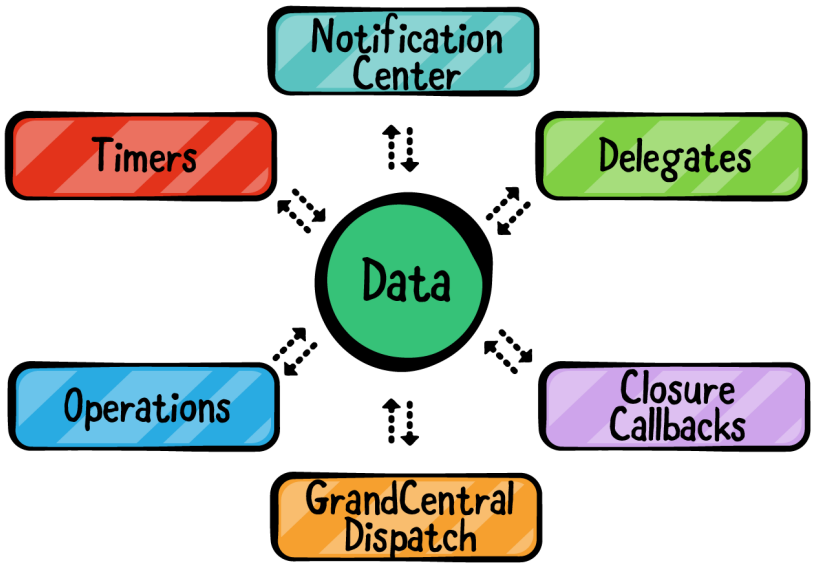
You've probably used most of the following:

- **`NotificationCenter`**: Executes a piece of code any time an event of interest happens, such as when the user changes the orientation of the device or when the software keyboard shows or hides on the screen.
- **The delegate pattern**: Lets you define an object that acts on behalf of, or in coordination with, another object. For example, in your app delegate, you define what should happen when a new remote notification arrives, but you have no idea when this piece of code will be executed or how many times it will execute.
- **Grand Central Dispatch** and **Operations**: Helps you abstract the execution of pieces of work. You can use them to schedule code to be executed sequentially in a serial queue or to run a multitude of tasks concurrently in different queues with different priorities.
- **Closures**: Create detached pieces of code that you can pass around in your code, so other objects can decide whether to execute it, how many times, and in what context.

Since most typical code performs some work asynchronously, and all UI events are inherently asynchronous, it's impossible to make assumptions about which order the **entirety** of your app code will be executed.

And yet, writing good asynchronous programs is possible. It's just more complex than… well, we'd like it to be. Unfortunately, asynchronous code and resource sharing can produce issues which are difficult to reproduce, track down and ultimately fix.

Certainly, one of the causes for these issues is the fact that a solid, real-life app most likely uses all the different kinds of asynchronous APIs, each with its own interface, like so:

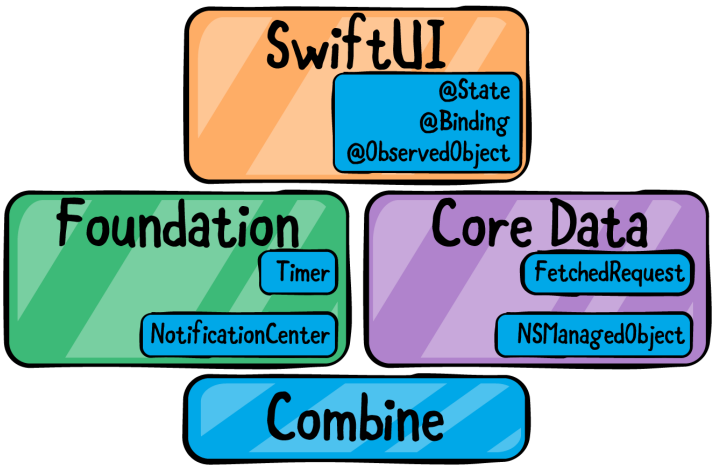



Combine aims to introduce a new language to the Swift ecosystem that helps you bring more order into the chaos of the asynchronous programming world.

Apple has integrated Combine's API deep into the Foundation framework, so `Timer`, `NotificationCenter` and core frameworks like **Core Data** already speak its language. Luckily, Combine is also very easy to integrate into your own code.

Finally, last but definitely not least, Apple designed their amazing new UI framework, **SwiftUI**, to integrate easily with Combine as well.

To give you an idea of how committed Apple is to reactive programming with Combine, here's a simple diagram showing where Combine sits in the system hierarchy:




Various system frameworks, from Foundation all the way up to SwiftUI, depend on Combine and offer Combine integration as an alternative to their "traditional" APIs.

Since Combine is an Apple framework, it doesn't aim to take away the role of well-tested, solid APIs like `Timer` or `NotificationCenter`. Those Foundation types are still present and doing their part. Instead, Combine integrates with them and allows all the types in your app that want to talk asynchronously to each other do so via a new, universal language.

So if the idea of using the same asynchronous tools to connect all the parts of your app, from the data model to the networking layer and the user interface, sounds interesting — you're in the right place, keep reading!

## Foundation of Combine

Declarative, reactive programming isn't a new concept. It's been around for quite a while, but it's made a fairly noticeable comeback in the last decade.

The first "modern-day" reactive solution came in a big way in 2009 when a team at Microsoft launched a library called **Reactive Extensions** for .NET (Rx.NET).

Microsoft made that Rx.NET implementation open source in 2012, and since then, many different languages have started to use its concepts. Currently, there are many ports of the Rx standard like RxJS, RxKotlin, RxScala, RxPHP and more.

For Apple's platforms, there have been several third-party reactive frameworks like RxSwift, which implements the Rx standard; ReactiveSwift, which was inspired by

Rx; Interstellar, which is a custom implementation and others.

Combine implements a standard that is different but similar to Rx, called Reactive Streams. Reactive Streams has a few key differences from Rx, but they both agree on most of the core concepts.

If you haven't previously used one or another of the frameworks mentioned above — don't worry. So far, reactive programming has been a rather niche concept for Apple's platforms, and especially with Swift.

In iOS 13/macOS Catalina, however, Apple brought reactive programming support to its ecosystem via the built-in system framework, **Combine**.

As with any new technology from Apple, its application is at first slightly limited: You can use Combine only for apps that support iOS 13/macOS Catalina or later. But as with any technology that Apple bets on, its support will quickly become widespread and the demand for Combine skills will surge.

With that said, start by learning some of Combine's basics to see how it can help you write safe and solid asynchronous code.

# Combine basics

In broad strokes, the three key moving pieces in Combine are publishers, operators and subscribers. There are, of course, more players in the team, but without those three you can't achieve much.

You'll learn in detail about publishers and subscribers in Chapter 2, "Publishers and Subscribers," and the complete second section of the book is devoted to acquainting you with as many operators as humanly possible.

In this introductory chapter, however, you're going to get a simple crash course to give you a general idea of the purpose those types have in the code and what their responsibilities are.

## Publishers

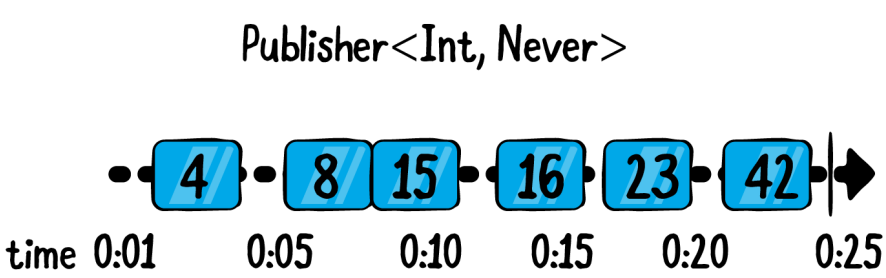
Publishers are types that can emit values over time to one or more interested parties, such as subscribers. Regardless of the internal logic of the publisher, which can be pretty much anything including math calculations, networking or handling user events, every publisher can emit multiple events of these three types:

1. An output value of the publisher's generic `Output` type.

2. A successful completion.
3. A completion with an error of the publisher's `Failure` type.

A publisher can emit zero or more output values, and if it ever completes, either successfully or due to a failure, it will not emit any other events.

Here's how a publisher emitting `Int` values could look like visualized on a timeline:




The blue boxes represent values that are emitted at a given time on the timeline, and the numbers represent the emitted values. A vertical line, like the one you see on the right-hand side of the diagram, represents a successful stream completion.

The simple contract of three possible events is so universal that it could represent any kind of dynamic data in your program. That's why you can address any task in your app using Combine publishers — regardless of whether it's about crunching numbers, making network calls, reacting to user gestures or displaying data on-screen.

Instead of always looking in your toolbox for the right tool to grab for the task at hand, be it adding a delegate or injecting a completion callback — you can just use a publisher instead.

One of the best features of publishers is that they come with error handling built in; error handling isn't something you add optionally at the end, if you feel like it.

The `Publisher` protocol is generic over two types, as you might have noticed in the diagram earlier:

- `Publisher.Output` is the type of the output values of the publisher. If the publisher is specialized as an `Int`, it can never emit a `String` or a `Date` value.

- `Publisher.Failure` is the type of error the publisher can throw if it fails. If the publisher can never fail, you specify that by using a `Never` failure type.

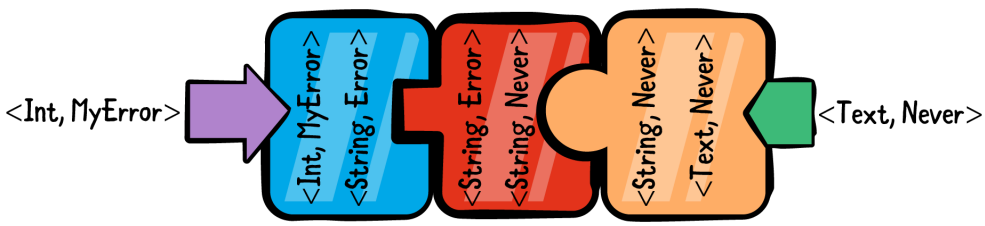
When you subscribe to a given publisher, you know what values to expect from it and which errors it could fail with.

## Operators

Operators are methods declared on the `Publisher` protocol that return either the same or a new publisher. That's very useful because you can call a bunch of operators one after the other, effectively chaining them together.

Because these methods, called "operators", are highly decoupled and composable, they can be combined (aha!) to implement very complex logic over the execution of a single subscription.

It's fascinating how operators fit tightly together like puzzle pieces. They cannot be mistakenly put in the wrong order or fit together if one's output doesn't match the next one's input type:

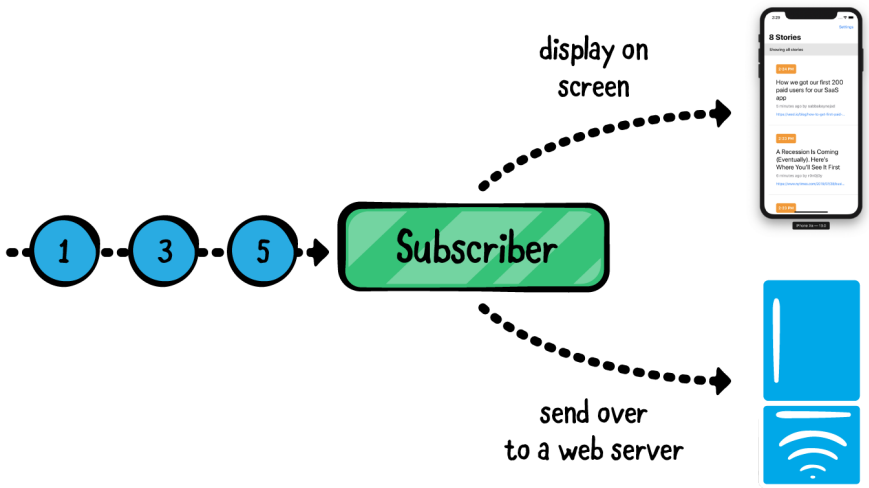



In a clear deterministic way, you can define the order of each of those asynchronous abstracted pieces of work alongside with the correct input/output types and built-in error handling. It's almost too good to be true!

As an added bonus, operators always have input and output, commonly referred to as **upstream** and **downstream** — this allows them to avoid shared state (one of the core issues we discussed earlier). Operators focus on working with the data they receive from the previous operator and provide their output to the next one in the chain. This means that no other asynchronously-running piece of code can "jump in" and change the data you're working on.

## Subscribers

Finally, you arrive at the end of the subscription chain: Every subscription ends with a subscriber. Subscribers generally do "something" with the emitted output or completion events.




Currently, Combine provides two built-in subscribers, which make working with data streams straightforward:

- The **sink** subscriber allows you to provide closures with your code that will receive output values and completions. From there, you can do anything your heart desires with the received events.
- The **assign** subscriber allows you to, without the need of custom code, *bind* the resulting output to some property on your data model or on a UI control to display the data directly on-screen via a key path.

Should you have other needs for your data, creating custom subscribers is even easier than creating publishers. Combine uses a set of very simple protocols that allow you to be able to build your own custom tools whenever the workshop doesn't offer the right one for your task.

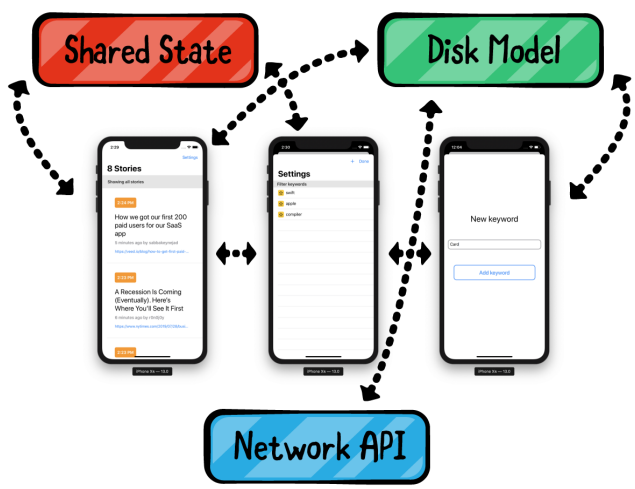
# Subscriptions

> **Note**: This book uses the term **subscription** to describe both Combine's `Subscription` protocol and its conforming objects, as well as the complete chain of a publisher, operators and a subscriber.

When you add a subscriber at the end of a subscription, it "activates" the publisher all the way at the beginning of the chain. This is a curious but important detail to remember — publishers do not emit any values if there are no subscribers to potentially receive the output.

Subscriptions are a wonderful concept in that they allow you to *declare* a chain of asynchronous events with their own custom code and error handling **only once**, and then you never have to think about it again.

If you go full-Combine, you could describe your whole app's logic via subscriptions and once done, just let the system run everything without the need to push or pull data or call back this or that other object:




Once the subscription code compiles successfully and there are no logic issues in your custom code — you're done! The subscriptions, as designed, will asynchronously "fire" each time some event like a user gesture, a timer going off or something else awakes one of your publishers.

Even better, you don't need to specifically memory manage a subscription, thanks to a protocol provided by Combine called `Cancellable`.

Both system-provided subscribers conform to `Cancellable`, which means that your subscription code (e.g. the whole publisher, operators and subscriber call chain) returns a `Cancellable` object. Whenever you release that object from memory, it cancels the whole subscription and releases its resources from memory.

This means you can easily "bind" the lifespan of a subscription by storing it in a property on your view controller, for example. This way, any time the user dismisses the view controller from the view stack, that will deinitialize its properties and will also cancel your subscription.

Or to automate this process, you can just have an `[AnyCancellable]` collection property on your type and throw as many subscriptions inside it as you want. They'll all be automatically canceled and released when the property is released from memory.

As you see, there's plenty to learn, but it's all logical when explained in detail. And that's exactly what the plan is for the next chapters — to bring you slowly but steadily from zero to Combine hero by the end of this book.

## What's the benefit of Combine code over "standard" code?

You can, by all means, never use Combine and still create the best apps out there. There's no argument about that. You can also create the best apps without Core Data, `URLSession`, or even UIKit. But using those frameworks is more convenient, safe and efficient than building those abstractions yourself.

Combine (and other system frameworks) aims to add another abstraction in your async code. Another level of abstraction on the system level means tighter integration that's well tested and a safe-bet technology for long-lasting support.

It's up to you to decide whether Combine is a great fit for your project or not, but here are just a few "pro" reasons you might not have considered yet:

- Combine is integrated on the system level. That means Combine itself uses language features that are not publicly available, offering you APIs that you couldn't build yourself.
- The "old" style async code via delegates, `IBAction` or closures pushes you towards writing custom code for each case of a button or a gesture you need to handle. That's a lot of custom code to write tests for. Combine abstracts all async operations in your code as "operators", which are already well tested.

- When all of your asynchronous pieces of work use the same interface — `Publisher` — composition and reusability become extremely powerful.
- Combine's operators are highly composable. If you need to create a new one, that new operator will instantly plug-and-play with the rest of Combine.
- Testing asynchronous code is usually more complex than testing synchronous code. With Combine, however, the asynchronous operators are already tested, and all that's left for you to do is test your business logic — that is, provide some input and test if your subscription outputs the expected result.

As you see, most of the benefits revolve around safety and convenience. Combined with the fact that the framework comes from Apple, investing in writing Combine code looks promising.

## App architecture

As this question is most likely already sounding alarms in your head, take a look at how using Combine will change your pre-existing code and app architecture.

Combine is not a framework that affects how you structure your apps. Combine deals with asynchronous data events and unified communication contract — it does not alter, for example, how you would separate responsibilities in your project.

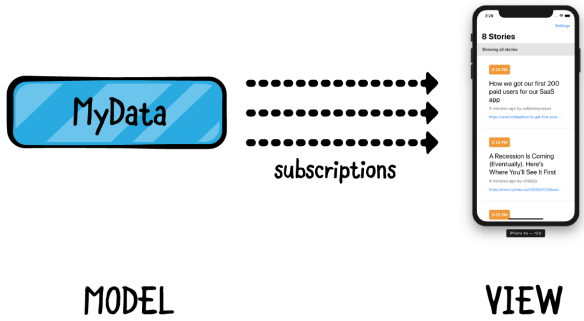
You can use Combine in your MVC (Model-View-Controller) apps, you can use it in your MVVM (Model-View-ViewModel) code, in VIPER and so forth and so on.

This is one of the key aspects of adopting Combine that is important to understand early — you can add Combine code iteratively and selectively, using it only in the parts you wish to improve in your codebase. It's not an "all or nothing" choice you need to make.

You could start by converting your data models, or adapting your networking layer, or simply using Combine only in new code that you add to your app while keeping your existing functionality as-is.

It's a slightly different story if you're adopting Combine and SwiftUI at the same time. In that case, it really does make sense to drop the C from an MVC architecture. But that's thanks to using Combine and SwiftUI in tandem — those two are simply on fire when in the same room.

View controllers just don't have any chance against a Combine/SwiftUI team. When you use reactive programming all the way from your data model to your views, you don't need to have a special controller just to control your views:



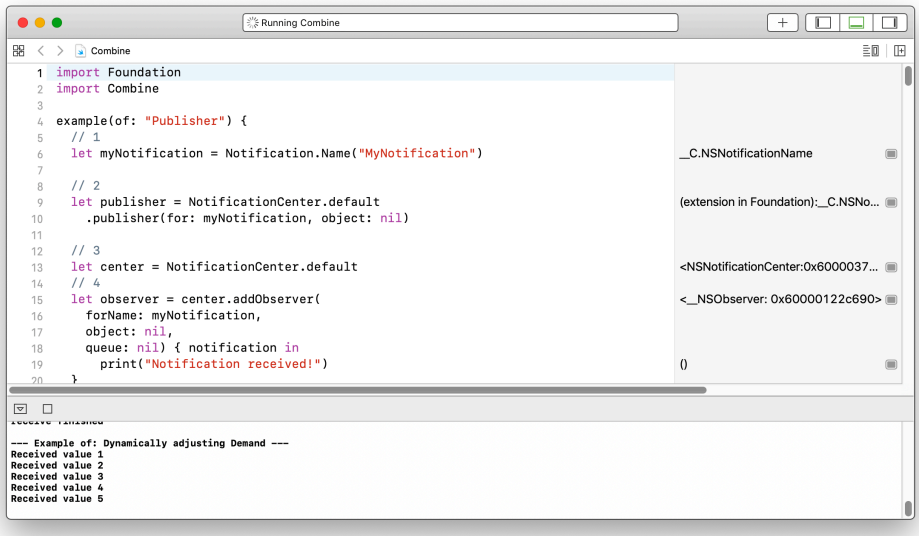

If that sounds interesting, you're in for a treat, as this book includes a solid introduction to using the two frameworks together in Chapter 15, "Combine with SwiftUI".

# Book projects

In this book, you'll start with the concepts first and move on to learning and trying out a multitude of operators.

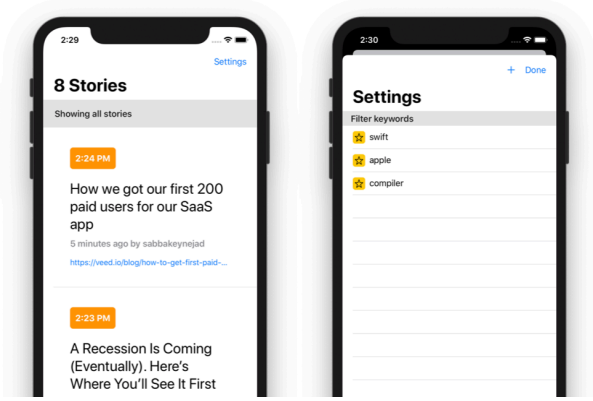
Unlike other system frameworks, you can work pretty successfully with Combine in the isolated context of a playground.

Learning in an Xcode playground makes it easy to move forward and quickly experiment as you progress through a given chapter and to see instantly the results in Xcode's Console:




Combine does not require any third-party dependencies, so usually, a few simple helper files included with the starter playground code for each chapter will suffice to get you running. If Xcode ever gets stuck while you experiment in the playground, a quick restart will likely solve the issue.

Once you move to more complex concepts than playing with a single operator, you'll alternate between working in playgrounds and real Xcode projects like the Hacker News app, which is a newsreader that displays news in real time:




It's important that, for each chapter, you begin with the provided starter playground or project, as they might include some custom helper code which isn't relevant to learning Combine. These tidbits are pre-written so you don't distract yourself from

the focus of that chapter.

In the last chapter, you'll work through building a complete iOS app from scratch with Combine. This will make use of all the skills you learned throughout the book and will give you a final push on your road to building real-life applications with Combine!


# Key points

- Combine is a declarative, reactive framework for processing asynchronous events over time.
- It aims to solve existing problems, like unifying tools for asynchronous programming, dealing with mutable state and making error handling a starting team player.
- Combine revolves around three main types: **publishers** to emit events over time, **operators** to asynchronously process and manipulate upstream events and **subscribers** to consume the results and do something useful with them.

# Where to go from here?

Hopefully, this introductory chapter has been useful and has given you an initial understanding of the issues Combine addresses as well as a look at some of the tools it offers to make your asynchronous code safer and more reliable.

Another important takeaway from this chapter is what to expect from Combine and what is out of its scope. Now, you know what you're in for when we speak of reactive code or asynchronous events over time. And, of course, you don't expect using Combine to magically solve your app's problems with navigation or drawing on-screen.

Finally, having a taste of what's in store for you in the upcoming chapters has hopefully gotten you excited about Combine and reactive programming with Swift. Upwards and onwards, here we go!

# Chapter 2: Publishers & Subscribers

By Scott Gardner

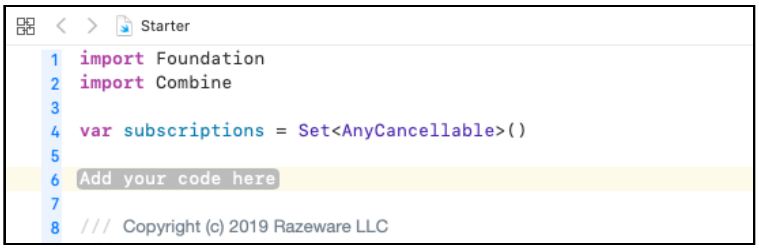
Now that you've learned some of the basic concepts of Combine, it's time to jump in and play with two of Combine's core components — publishers and subscribers.

In this chapter, you'll review several examples of creating publishers and subscribing to those publishers using subscribers. By doing so, you'll acquire important skills that you'll use throughout the rest of this book and beyond.

## Getting started

> **Note**: There are starter and final versions of the playgrounds and projects you'll use in each chapter throughout the book. The starter will be prepared and ready for you to enter the code specified for each example and challenge. You can compare your work with the final version at the end or along the way if you get stuck.

For this chapter, you'll use an Xcode playground with Combine imported. Open **Starter.playground** in the **projects** folder and you'll see the following:




Open **Sources** in the **Project navigator** (**View ▸ Navigators ▸ Show Project Navigator** and twist down the **Combine** playground page), 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'll use this function to encapsulate some examples you'll use throughout this book.

However, before you begin playing with those examples, you first need to learn about publishers, subscribers and subscriptions. They form the foundation of Combine and enable you to *send* and *receive* data, typically asynchronously.

# Hello Publisher

At the heart of Combine is the `Publisher` protocol. This protocol defines the requirements for a type to be able to transmit a sequence of values over time to one or more subscribers. In other words, a publisher publishes or emits events that can include values of interest.

If you've developed on Apple platforms before, you can think of a publisher as a kind of notification center. In fact, `NotificationCenter` now has a method named `publisher(for:object:)` that provides a `Publisher` type that can publish broadcasted notifications.

To check this out, go back to the starter playground and replace the `Add your code here` placeholder with the following code:

```
example(of: "Publisher") {
  // 1
  let myNotification = Notification.Name("MyNotification")
```

```
    // 2
    let publisher = NotificationCenter.default
      .publisher(for: myNotification, object: nil)
}
```

In this code, you:

1. Create a notification name.
2. Access notification center's default center, call its `publisher(for:object:)` method and assign its return value to a local constant.

**Option-click** on `publisher(for:object:)`, and you'll see that it returns a `Publisher` that emits an event when the default notification center broadcasts a notification.

So what's the point of publishing notifications when a notification center is already capable of broadcasting its notifications *without* a publisher? Glad you asked!

You can think of these types of methods as a bridge from the old to the new — a way to *Combine-ify* existing APIs such as `NotificationCenter`.

A publisher emits two kinds of events:

1. Values, also referred to as elements.
2. A completion event.

A publisher can emit zero or more values but only one completion event, which can either be a normal completion event or an error. Once a publisher emits a completion event, it's finished and can no longer emit any more events.

Before diving deeper into publishers and subscribers, you'll first finish the example of using traditional notification center APIs to receive a notification by registering an observer. You'll also unregister that observer when you're no longer interested in receiving that notification.

Add the following code to the end of the previous example:

```
// 3
let center = NotificationCenter.default

// 4
let observer = center.addObserver(
  forName: myNotification,
  object: nil,
  queue: nil) { notification in
    print("Notification received!")
```

```
}

// 5
center.post(name: myNotification, object: nil)

// 6
center.removeObserver(observer)
```

With this code, you:

3. Get a handle to the default notification center.
4. Create an observer to listen for the notification with the name you previously created.
5. Post a notification with that name.
6. Remove the observer from the notification center.

Run the playground. You'll see this output printed to the console:

```
—— Example of: Publisher ——
Notification received!
```

The example title is a little misleading because the output is not actually coming from a publisher. For that to happen, you need a subscriber.

# Hello Subscriber

`Subscriber` is a protocol that defines the requirements for a type to be able to receive input from a publisher. You'll dive deeper into conforming to the `Publisher` and `Subscriber` protocols shortly; for now, you'll focus on the basic flow.

Start by replacing the previous notification handling code with a subscription. Add a new example to the playground that begins like the previous one:

```
example(of: "Subscriber") {
  let myNotification = Notification.Name("MyNotification")

  let publisher = NotificationCenter.default
    .publisher(for: myNotification, object: nil)

  let center = NotificationCenter.default
}
```

If you were to post a notification now, the publisher wouldn't emit it — and that's an

important distinction to remember. A publisher only emits an event when there's at least one subscriber.

## Subscribing with sink(_:_:)

Continuing the previous example, add the following code to the example to create a subscription to the publisher:

```
let subscription = publisher
  .sink { _ in
    print("Notification received from a publisher!")
  }
```

With this code, you create a subscription by calling `sink` on the publisher — but don't let the obscurity of that method name give you a *sinking* feeling. **Option-click** on `sink` and you'll see that it simply provides an easy way to attach a subscriber with closures to handle output from a publisher. In this example, you ignore those closures and instead just print a message to indicate that a notification was received.

The `sink` operator will continue to receive as many values as the publisher emits. This is known as *unlimited demand*, which you'll learn more about shortly. And although you ignored them in the previous example, the `sink` operator actually provides two closures: one handle receiving a completion event, and one to handle received values.

To see how this works, add this new example to your playground:

```
example(of: "Just") {
  // 1
  let just = Just("Hello world!")

  // 2
  _ = just
    .sink(
      receiveCompletion: {
        print("Received completion", $0)
      },
      receiveValue: {
        print("Received value", $0)
    })
}
```

Here, you:

1. Create a publisher using `Just`, which lets you create a publisher from a primitive value type.

2. Create a subscription to the publisher and print a message for each received event.

Run the playground. You'll see the following:

```
—— Example of: Just ——
Received value Hello world!
Received completion finished
```

**Option-click** on `Just` and the Quick Help explains that it's a publisher that emits its output to each subscriber once and then finishes.

Try adding another subscriber by adding the following code to the end of your example:

```
_ = just
  .sink(
    receiveCompletion: {
      print("Received completion (another)", $0)
    },
    receiveValue: {
      print("Received value (another)", $0)
  })
```

Run the playground. True to its word, a `Just` happily emits its output to each new subscriber exactly once and then finishes.

```
Received value (another) Hello world!
Received completion (another) finished
```

## Subscribing with assign(to:on:)

In addition to `sink`, the built-in `assign(to:on:)` operator enables you to assign the received value to a KVO-compliant property of an object.

Add this example to see how this works:

```
example(of: "assign(to:on:)") {
  // 1
  class SomeObject {
    var value: String = "" {
      didSet {
        print(value)
      }
    }
  }
```

```
    // 2
    let object = SomeObject()

    // 3
    let publisher = ["Hello", "world!"].publisher

    // 4
    _ = publisher
      .assign(to: \.value, on: object)
}
```

From the top:

1. Define a class with a property that has a `didSet` property observer that prints the new value.
2. Create an instance of that class.
3. Create a publisher from an array of strings.
4. Subscribe to the publisher, assigning each value received to the `value` property of the object.

Run the playground and you will see printed:

```
—— Example of: assign(to:on:) ——
Hello
world!
```

You'll focus on using the `sink` operator for now — but fear not, you'll get more hands-on practice using `assign` beginning in Chapter 8, "In Practice: Project "Collage"."

# Hello Cancellable

When a subscriber is done and no longer wants to receive values from a publisher, it's a good idea to cancel the subscription to free up resources and stop any corresponding activities from occurring, such as network calls.

Subscriptions return an instance of `AnyCancellable` as a "cancellation token," which makes it possible to cancel the subscription when you're done with it. `AnyCancellable` conforms to the `Cancellable` protocol, which requires the `cancel()` method exactly for that purpose.

Finish the **Subscriber** example from earlier by adding the following code:

```
// 1
center.post(name: myNotification, object: nil)

// 2
subscription.cancel()
```

With this code, you:

1. Post a notification (same as before).
2. Cancel the subscription. You're able to call `cancel()` on the subscription because the `Subscription` protocol inherits from `Cancellable`.

Run the playground. You'll see the following:

```
—— Example of: Subscriber ——
Notification received from a publisher!
```

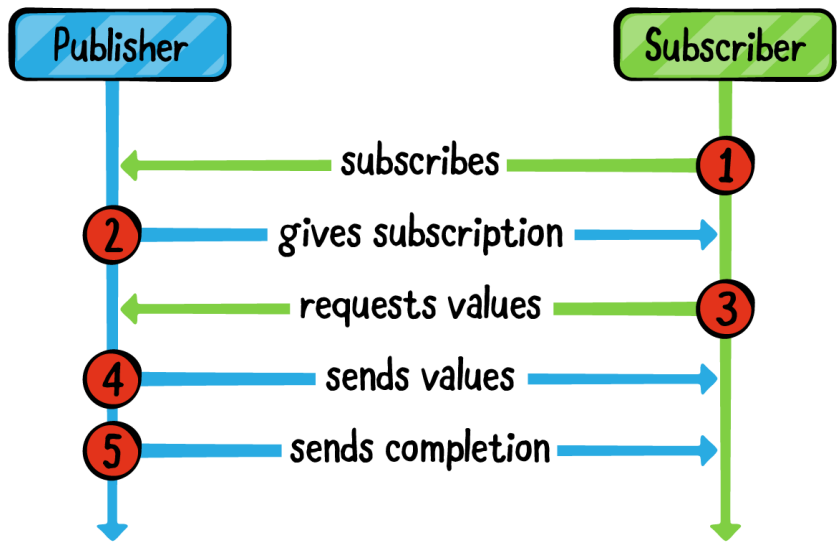
If you don't explicitly call `cancel()` on a subscription, it will continue until the publisher completes, or until normal memory management causes a stored subscription to be deinitialized. At that point it will cancel the subscription for you.

> **Note**: It's also fine to ignore the return value from a subscription in a playground (for example, `_ = just.sink...`). However, one caveat: if you don't store a subscription in full projects, that subscription will cancel as soon as the program flow exits the scope in which it was created!

These are good examples to start with, but there's a lot more going on behind the scenes. It's time to lift the curtain and learn more about the roles of publishers, subscribers and subscriptions in Combine.

# Understanding what's going on

They say a picture is worth a thousand words, so let's kick things off with one to explain the interplay between publishers and subscribers:




Walking through this UML diagram:

1. The subscriber subscribes to the publisher.
2. The publisher creates a subscription and gives it to the subscriber.
3. The subscriber requests values.
4. The publisher sends values.
5. The publisher sends a completion.

> **Note**: The above diagram provides a streamlined overview of what's going on. What you'll learn here is enough to get you started working with Combine. You'll gain a deeper understanding of this process in Chapter 18, "Custom Publishers and Handling Backpressure."

Take a look at the `Publisher` protocol and one of its most crucial extensions:

```
public protocol Publisher {
  // 1
  associatedtype Output
```

```
  // 2
  associatedtype Failure : Error

  // 4
  func receive<S>(subscriber: S)
    where S: Subscriber,
    Self.Failure == S.Failure,
    Self.Output == S.Input
}

extension Publisher {
  // 3
  public func subscribe<S>(_ subscriber: S)
    where S : Subscriber,
    Self.Failure == S.Failure,
    Self.Output == S.Input
}
```

Here's a closer look:

1. The type of values that the publisher can produce.
2. The type of errors that a publisher may produce; or `Never` if the publisher is guaranteed to not produce errors.
3. A subscriber calls `subscribe(_:)` on a publisher to attach to it.
4. The implementation of `subscribe(_:)` will call `receive(subscriber:)` to attach the subscriber to the publisher, i.e., create a subscription.

The associated types are the publisher's interface that a subscriber must match in order to create a subscription.

Now, look at the `Subscriber` protocol:

```
public protocol Subscriber: CustomCombineIdentifierConvertible {
  // 1
  associatedtype Input

  // 2
  associatedtype Failure: Error

  // 3
  func receive(subscription: Subscription)

  // 4
  func receive(_ input: Self.Input) -> Subscribers.Demand

  // 5
  func receive(completion: Subscribers.Completion<Self.Failure>)
```

```
}
```

Here's a closer look:

1. The type of values a subscriber can receive.
2. The type of errors a subscriber can receive; or `Never` if the subscriber won't receive errors.
3. The publisher calls `receive(subscription:)` on the subscriber to give it the subscription.
4. The publisher calls `receive(_:)` on the subscriber to send it a new value that it just published.
5. The publisher calls `receive(completion:)` on the subscriber to tell it that it has finished producing values, either normally or due to an error.

The connection between the publisher and the subscriber is the subscription. Here's the `Subscription` protocol:

```
public protocol Subscription: Cancellable,
CustomCombineIdentifierConvertible {
  func request(_ demand: Subscribers.Demand)
}
```

The subscriber calls `request(_:)` to indicate it is willing to receive more values, up to a max number or unlimited.

> **Note**: The concept of a subscriber stating how many values it's willing to receive is known as **backpressure management**. Without it, or some other strategy, a subscriber could get flooded with more values from the publisher than it can handle, and this can lead to problems. Backpressure is also covered in depth in Chapter 18, "Custom Publishers and Handling Backpressure."

In `Subscriber`, notice that `receive(_:)` returns a `Demand`. Even though the max number of values a subscriber is willing to receive is specified when initially calling `subscription.request(_:)` in `receive(_:)`, you can adjust that max each time a new value is received.

> **Note**: Adjusting `max` in `Subscriber.receive(_:)` is additive, i.e., the new `max`

value is *added* to the current `max`. The `max` value must be positive, and passing a negative value will result in a `fatalError`. This means that you can increase the original `max` each time a new value is received, but you cannot decrease it.

# Creating a custom subscriber

Time to put what you just learned to practice. Add this new example to your playground:

```
example(of: "Custom Subscriber") {
  // 1
  let publisher = (1...6).publisher

  // 2
  final class IntSubscriber: Subscriber {
    // 3
    typealias Input = Int
    typealias Failure = Never

    // 4
    func receive(subscription: Subscription) {
      subscription.request(.max(3))
    }

    // 5
    func receive(_ input: Int) -> Subscribers.Demand {
      print("Received value", input)
      return .none
    }

    // 6
    func receive(completion: Subscribers.Completion<Never>) {
      print("Received completion", completion)
    }
  }
}
```

What you do here is:

1. Create a publisher of integers via the range's `publisher` property.
2. Define a custom subscriber, `IntSubscriber`.
3. Implement the type aliases to specify that this subscriber can receive integer inputs and will never receive errors.

4. Implement the required methods, beginning with `receive(subscription:)`, which is called by the publisher; and in that method, call `.request(_:)` on the subscription specifying that the subscriber is willing to receive up to three values upon subscription.
5. Print each value as it's received and return `.none`, indicating that the subscriber will not adjust its demand; `.none` is equivalent to `.max(0)`.
6. Print the completion event.

For the publisher to publish anything, it needs a subscriber. Add the following at the end of the example:

```
let subscriber = IntSubscriber()

publisher.subscribe(subscriber)
```

In this code, you create a subscriber that matches the `Output` and `Failure` types of the publisher. You then tell the publisher to subscribe, or attach, the subscriber.

Run the playground. You'll see the following printed to the console:

```
—— Example of: Custom Subscriber ——
Received value 1
Received value 2
Received value 3
```

You did not receive a completion event. This is because the publisher has a finite number of values, and you specified a demand of `.max(3)`.

In your custom subscriber's `receive(_:)`, try changing `.none` to `.unlimited`, so your `receive(_:)` method looks like this:

```
func receive(_ input: Int) -> Subscribers.Demand {
  print("Received value", input)
  return .unlimited
}
```

Run the playground again. This time you'll see that all of the values are received and printed, along with the completion event:

```
—— Example of: Custom Subscriber ——
Received value 1
Received value 2
Received value 3
Received value 4
Received value 5
```

```
Received value 6
Received completion finished
```

Try changing `.unlimited` to `.max(1)` and run the playground again.

You'll see the same output as when you returned `.unlimited`, because each time you receive an event, you specify that you want to increase the `max` by `1`.

Change `.max(1)` back to `.none`, and change the definition of `publisher` to an array of strings instead. Replace:

```
let publisher = (1...6).publisher
```

With:

```
let publisher = ["A", "B", "C", "D", "E", "F"].publisher
```

Run the playground. You get an error that the `subscribe` method requires types `String` and `IntSubscriber.Input` (i.e., `Int`) to be equivalent. You get this error because the `Output` and `Failure` associated types of a publisher must match the `Input` and `Failure` types of a subscriber in order for a subscription between the two to be created.

Change the `publisher` definition back to its original range of integers to resolve the error.

# Hello Future

Much like you can use `Just` to create a publisher that emits a single value to a subscriber and then complete, a `Future` can be used to *asynchronously* produce a single result and then complete. Add this new example to your playground:

```
example(of: "Future") {
  func futureIncrement(
    integer: Int,
    afterDelay delay: TimeInterval) -> Future<Int, Never> {

  }
}
```

Here, you create a factory function that returns a future of type `Int` and `Never`; meaning, it will emit an integer and never fail.

You also add a `subscriptions` set in which you'll store the subscriptions to the

future in the example. For long-running asynchronous operations, not storing the subscription will result in the cancelation of the subscription as soon as the current code scope ends. In the case of a Playground, that would be immediately.

Next, fill the function's body to create the future:

```
Future<Int, Never> { promise in
  DispatchQueue.global().asyncAfter(deadline: .now() + delay) {
    promise(.success(integer + 1))
  }
}
```

This code defines the future, which creates a promise that you then execute using the values specified by the caller of the function to increment the `integer` after the `delay`.

A `Future` is a publisher that will eventually produce a single value and finish, or it will fail. It does this by invoking a closure when a value or error is made available, and that closure is referred to as a promise. **Command-click** on `Future` and choose **Jump to Definition**. You'll see the following:

```
final public class Future<Output, Failure> : Publisher
  where Failure: Error {
  public typealias Promise = (Result<Output, Failure>) -> Void
  ...
}
```

`Promise` is a type alias to a closure that receives a `Result` containing either a single value published by the `Future`, or an error.

Head back to the main playground page, and add the following code after the definition of `futureIncrement`:

```
// 1
let future = futureIncrement(integer: 1, afterDelay: 3)

// 2
future
  .sink(receiveCompletion: { print($0) },
        receiveValue: { print($0) })
  .store(in: &subscriptions)
```

Here, you:

1. Create a future using the factory function you created earlier, specifying to increment the integer you passed after a three-second delay.
2. Subscribe to and print the received value and completion event, and store the

resulting subscription in the `subscriptions` set. You'll learn more about storing subscriptions in a collection later in this chapter, so don't worry if you don't entirely understand that portion of the example.

Run the playground. You'll see the example title printed, followed by the output of the future after a three-second delay:

```
—— Example of: Future ——
2
finished
```

Add a second subscription to the future by entering the following code in the playground:

```
future
  .sink(receiveCompletion: { print("Second", $0) },
        receiveValue: { print("Second", $0) })
  .store(in: &subscriptions)
```

Before running the playground, insert the following print statement immediately before the `DispatchQueue` block in the `futureIncrement` function:

```
print("Original")
```

Run the playground. After the specified delay, the second subscription receives the same value. The future does not re-execute its promise; instead, it shares or replays its output.

```
—— Example of: Future ——
Original
2
finished
Second 2
Second finished
```

Also, `Original` is printed right away before the subscriptions occur. This happens because a future executes as soon as it is created. It does not require a subscriber like regular publishers.

In the last few examples, you've been working with publishers that have a finite number of values to publish, which are sequentially and synchronously published.

The notification center example you started with is an example of a publisher that can keep on publishing values indefinitely and asynchronously, provided:

1. The underlying notification sender emits notifications.

2. There are subscribers to the specified notification.

What if there was a way that you could do the same thing in your own code? Well, it turns out, there is! Before moving on, comment out the entire `"Future"` example, so the future isn't invoked every time you run the playground — otherwise its delayed output will be printed after the last example.

# Hello Subject

You've already learned how to work with publishers and subscribers, and even how to create your own custom subscribers. Later in the book, you'll learn how to create custom publishers. For now, though, there's just a couple more things standing between you and a well-deserved `<insert your favorite beverage>` break. First is a **subject**.

A subject acts as a go-between to enable non-Combine imperative code to send values to Combine subscribers. That `<favorite beverage>` isn't going to drink itself, so it's time to get to work!

Add this new example to your playground:

```
example(of: "PassthroughSubject") {
  // 1
  enum MyError: Error {
    case test
  }

  // 2
  final class StringSubscriber: Subscriber {
    typealias Input = String
    typealias Failure = MyError

    func receive(subscription: Subscription) {
      subscription.request(.max(2))
    }

    func receive(_ input: String) -> Subscribers.Demand {
      print("Received value", input)
      // 3
      return input == "World" ? .max(1) : .none
    }

    func receive(completion: Subscribers.Completion<MyError>) {
      print("Received completion", completion)
    }
  }
```

```
    // 4
    let subscriber = StringSubscriber()
}
```

With this code, you:

1. Define a custom error type.
2. Define a custom subscriber that receives strings and `MyError` errors.
3. Adjust the demand based on the received value.
4. Create an instance of the custom subscriber.

Returning `.max(1)` in `receive(_:)` when the input is `"World"` results in the new max being set to `3` (the original max plus `1`).

Other than defining a custom error type and pivoting on the received value to adjust demand, there's nothing new here. Here comes the more interesting part.

Add this code to the example:

```
// 5
let subject = PassthroughSubject<String, MyError>()

// 6
subject.subscribe(subscriber)

// 7
let subscription = subject
  .sink(
    receiveCompletion: { completion in
      print("Received completion (sink)", completion)
    },
    receiveValue: { value in
      print("Received value (sink)", value)
    }
  )
```

This code:

5. Creates an instance of a `PassthroughSubject` of type `String` and the custom error type you defined.
6. Subscribes the subscriber to the subject.
7. Creates another subscription using `sink`.

Passthrough subjects enable you to publish new values on demand. They will happily pass along those values and a completion event. As with any publisher, you must

declare the type of values and errors it can emit in advance; subscribers must match those types to its input and failure types in order to subscribe to that passthrough subject.

Now that you've created a passthrough subject that can send values and subscriptions to receive them, it's time to send some values. Add the following code to your example:

```
subject.send("Hello")
subject.send("World")
```

This sends two values (one at a time) using the subject's `send` method.

Run the playground. You'll see:

```
—— Example of: PassthroughSubject ——
Received value Hello
Received value (sink) Hello
Received value World
Received value (sink) World
```

Each subscriber receives the values as they're published.

Add the following code:

```
// 8
subscription.cancel()

// 9
subject.send("Still there?")
```

Here, you:

8. Cancel the second subscription.

9. Send another value.

Run the playground. As you might have expected, only the first subscriber receives the value. This happens because you previously canceled the second subscriber's subscription:

```
—— Example of: PassthroughSubject ——
Received value Hello
Received value (sink) Hello
Received value World
Received value (sink) World
Received value Still there?
```

Add this code to the example:

```
subject.send(completion: .finished)
subject.send("How about another one?")
```

Run the playground. The second subscription does not receive the `"How about another one?"` value. This happens because the subject previously sent a completion event that the second subscriber *did receive* but the first subscriber *did not* because it was no longer subscribed:

```
—— Example of: PassthroughSubject ——
Received value Hello
Received value (sink) Hello
Received value World
Received value (sink) World
Received value Still there?
Received completion finished
```

Add the following code immediately before the line that sends the completion event.

```
subject.send(completion: .failure(MyError.test))
```

Run the playground, again. You'll see the following printed to the console:

```
—— Example of: PassthroughSubject ——
Received value Hello
Received value (sink) Hello
Received value World
Received value (sink) World
Received value Still there?
Received completion failure(...MyError.test)
```

> **Note:** The error type is abbreviated.

The error is received by the first subscriber, but the completion event that was sent *after* the error is not. This demonstrates that once a publisher sends a *single* completion event — whether it's a normal completion or an error — it's done, as in fini, kaput!

Passing through values with a `PassthroughSubject` is one way to bridge imperative code to the declarative world of Combine. Sometimes, however, you may also want to look at the current value of a publisher in your imperative code — for that, you have an aptly named subject: `CurrentValueSubject`.

In the next example you'll also learn a more convenient way to manage

subscriptions. Instead of storing each subscription as a value, you can store multiple subscriptions in a collection of `AnyCancellable`. The collection will then automatically cancel each subscription added to it when the collection is about to be deinitialized.

Add this new example to your playground:

```
example(of: "CurrentValueSubject") {
  // 1
  var subscriptions = Set<AnyCancellable>()

  // 2
  let subject = CurrentValueSubject<Int, Never>(0)

  // 3
  subject
    .sink(receiveValue: { print($0) })
    .store(in: &subscriptions) // 4
}
```

Here's what's happening:

1. Initialize an empty `subscriptions` set of type `AnyCancellable`.
2. Create a `CurrentValueSubject` of type `Int` and `Never`. This will publish integers and never publish an error, with an initial value of `0`.
3. Create a subscription to the subject and print values received from it.
4. Store the subscription in the `subscriptions` set, which is passed as an `inout` parameter so that the same set is updated instead of a copy.

You must initialize current value subjects with an initial value; new subscribers immediately get that value or the latest value published by that subject. Run the playground to see this in action:

```
—— Example of: CurrentValueSubject ——
0
```

Now, add this code to send two new values:

```
subject.send(1)
subject.send(2)
```

Run the playground again. Those values are also received and printed to the console:

```
1
2
```

Unlike a passthrough subject, you can ask a current value subject for its value at any time. Add the following code to print out the subject's current value:

```
print(subject.value)
```

As you might have inferred by the subject's type name, you can get its current value by accessing its `value` property. Run the playground, and you'll see `2` printed a second time.

Calling `send(_:)` on a current value subject is one way to send a new value. Another way is to assign a new value to its `value` property. Whoah, did we just go all imperative here or what? Add this code:

```
subject.value = 3
print(subject.value)
```

Run the playground. You'll see `2` and `3` each printed twice — once by the receiving subscriber and once from printing the subject's `value` after adding that value to the subject.

Next, at the end of this example, create a new subscription to the current value subject:

```
subject
  .sink(receiveValue: { print("Second subscription:", $0) })
  .store(in: &subscriptions)
```

Here, you create a subscription and print the received values. You also store that subscription in the `subscriptions` set.

You read a moment ago that the `subscriptions` set will automatically cancel the subscriptions added to it, but how can you verify this? You can use the `print()` operator, which will log all publishing events to the console.

Insert the `print()` operator in both subscriptions, between `subject` and `sink`. The beginning of each subscription should look like this:

```
subject
  .print()
  .sink...
```

Run the playground again and you'll see the following output for the entire example:

```
—— Example of: CurrentValueSubject ——
receive subscription: (CurrentValueSubject)
request unlimited
```

```
receive value: (0)
0
receive value: (1)
1
receive value: (2)
2
2
receive value: (3)
3
3
receive subscription: (CurrentValueSubject)
request unlimited
receive value: (3)
Second subscription: 3
receive cancel
receive cancel
```

Each event is printed, along with the values printed in the subscription handlers, and when you printed the subject's `values`.

So, you may be wondering, can you also assign a completion event to the `value` property? Try it out by adding this code:

```
subject.value = .finished
```

Nope! That produces an error. A `CurrentValueSubject`'s `value` property is meant for just that: values. Completion events must still be sent using `send(_:)`. Change the erroneous line of code to the following:

```
subject.send(completion: .finished)
```

Run the playground again. This time you'll see the following output at the bottom:

```
receive finished
receive finished
```

Both subscriptions receive the completion event instead of the cancel event. They are finished and no longer need to be canceled.

# Dynamically adjusting demand

You learned earlier that adjusting demand in `Subscriber.receive(_:)` is additive. You're now ready to take a closer look at how that works in a more elaborate example. Add this new example to the playground:

```
example(of: "Dynamically adjusting Demand") {
  final class IntSubscriber: Subscriber {
    typealias Input = Int
    typealias Failure = Never

    func receive(subscription: Subscription) {
      subscription.request(.max(2))
    }

    func receive(_ input: Int) -> Subscribers.Demand {
      print("Received value", input)

      switch input {
      case 1:
        return .max(2) // 1
      case 3:
        return .max(1) // 2
      default:
        return .none // 3
      }
    }

    func receive(completion: Subscribers.Completion<Never>) {
      print("Received completion", completion)
    }
  }

  let subscriber = IntSubscriber()

  let subject = PassthroughSubject<Int, Never>()

  subject.subscribe(subscriber)

  subject.send(1)
  subject.send(2)
  subject.send(3)
  subject.send(4)
  subject.send(5)
  subject.send(6)
}
```

Most of this code is similar to example you've previously worked on in this chapter, so instead you'll focus on the `receive(_:)` method. You continually adjust the demand from within your custom subscriber:

1. The new `max` is 4 (original `max` of 2 + new `max` of 2).
2. The new `max` is 5 (previous 4 + new 1).
3. `max` remains 5 (previous 4 + new `0`).

Run the playground and you'll see the following:

```
—— Example of: Dynamically adjusting Demand ——
Received value 1
Received value 2
Received value 3
Received value 4
Received value 5
```

As expected, five values are emitted but the sixth is not printed out.

There is one more important thing you'll want to know about before moving on: hiding details about a publisher from subscribers.

# Type erasure

There will be times when you want to let subscribers subscribe to receive events from a publisher without being able to access additional details about that publisher.

This would be best demonstrated with an example, so add this new one to your playground:

```
example(of: "Type erasure") {
  // 1
  let subject = PassthroughSubject<Int, Never>()

  // 2
  let publisher = subject.eraseToAnyPublisher()

  // 3
  publisher
    .sink(receiveValue: { print($0) })
    .store(in: &subscriptions)

  // 4
  subject.send(0)
}
```

With this code you:

1. Create a passthrough subject.
2. Create a type-erased publisher from that subject.
3. Subscribe to the type-erased publisher.
4. Send a new value through the passthrough subject.

**Option-click** on `publisher` and you'll see that it is of type `AnyPublisher<Int,`

`Never>`.

`AnyPublisher` is a type-erased struct that conforms the `Publisher` protocol. Type erasure allows you to hide details about the publisher that you may not want to expose to subscribers — or downstream publishers, which you'll learn about in the next section.

Are you are experiencing a little *déjà vu* right now? If so, that's because you saw another case of type erasure earlier. `AnyCancellable` is a type-erased class that conforms to `Cancellable`, which lets callers cancel the subscription without being able to access the underlying subscription to do things like request more items.

One example of when you would want to use type erasure for a *publisher* is when you want to use a pair of public and private properties, to allow the owner of those properties to send values on the private publisher, and let outside callers only access the public publisher for subscribing but not be able to send values. `AnyPublisher` does not have a `send(_:)` operator, so new values cannot be added to that publisher.

The `eraseToAnyPublisher()` operator wraps the provided publisher in an instance of `AnyPublisher`, hiding the fact that the publisher is actually a `PassthroughSubject`. This is also necessary because you cannot specialize the `Publisher` protocol, e.g., you cannot define the type as `Publisher<UIImage, Never>`.

To prove that `publisher` is type-erased and cannot be used to send new values, add this code to the example.

```
publisher.send(1)
```

You get the error `Value of type 'AnyPublisher<Int, Never>' has no member 'send'`. Comment out that line of code before moving on.

Fantastic job! You've learned a lot in this chapter, and you'll put these new skills to work throughout the rest of this book and beyond. But not so fast! It's time to practice what you just learned. So, grab yourself a `<insert your favorite beverage>` to enjoy while you work through the challenges for this chapter.

# Challenge

Completing challenges helps drive home what you learned in the chapter. There are starter and final versions of the challenge in the exercise files download.

# Challenge: Create a Blackjack card dealer

Open **Starter.playground** in the **challenge** folder, and twist down the playground page and **Sources** in the **Project navigator**. Select **SupportCode.swift**.

Review the helper code for this challenge, including

- A `cards` array that contains 52 tuples representing a standard deck of cards.
- Two type aliases: `Card` is a tuple of `String` and `Int`, and `Hand` is an array of `Cards`.
- Two helper properties on `Hand`: `cardString` and `points`.
- A `HandError` error enumeration.

In the main playground page, add code immediately below the comment `// Add code to update dealtHand here` that evaluates the result returned from the hand's `points` property. If the result is greater than `21`, send the `HandError.busted` on the `dealtHand` subject. Otherwise, send the `hand` value.

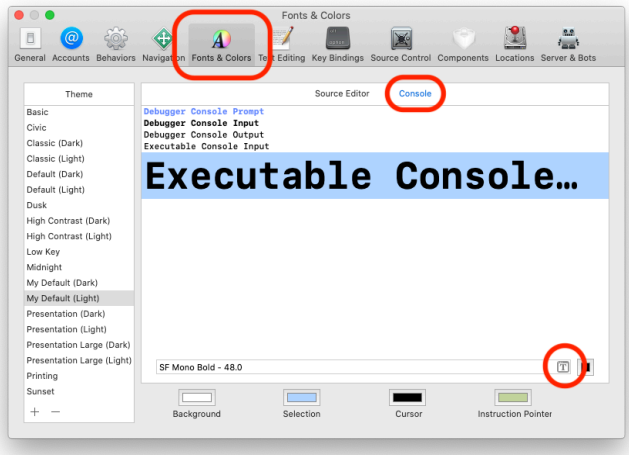
Also in the main playground page, add code immediately after the comment `// Add subscription to dealtHand here` to subscribe to `dealtHand` and handle receiving both values and an error. For received values, print a string containing the results of the hand's `cardString` and `points` properties.

For an error, print it out. A tip though: You can receive either a `.finished` or a `.failure` in the `receivedCompletion` block, so you'll want to distinguish whether that completion is a failure or not, and only print failures. `HandError` conforms to `CustomStringConvertible` so printing it will result in a user-friendly error message. You can use it like this:

```
if case let .failure(error) = $0 {
  print(error)
}
```

The call to `deal(_:)` currently passes `3`, so three cards are dealt each time you run the playground. See how many times you go bust versus how many times you stay in the game. Are the odds stacked up against you in Vegas or what?

The card emoji characters are small when printed in the console. You can temporarily increase the font size of the **Executable Console Output** for this challenge. To do so, select **Xcode ▸ Preferences... ▸ Fonts & Colors/Console**. Then, select **Executable Console Output**, and click the **T** button in the bottom right to change it to a larger font, such as 48.




## Solution

How'd you do?

There were two things you needed to add to complete this challenge. The first was to update the `dealtHand` publisher in the `deal` function, checking the hand's points and sending an error if it's over 21:

```
// Add code to update dealtHand here
if hand.points > 21 {
  dealtHand.send(completion: .failure(.busted))
} else {
  dealtHand.send(hand)
}
```

Next, you needed to subscribe to `dealtHand` and print out the value received or the completion event if it was an error:

```
_ = dealtHand
  .sink(receiveCompletion: {
    if case let .failure(error) = $0 {
      print(error)
    }
  }, receiveValue: { hand in
    print(hand.cardString, "for", hand.points, "points")
  })
```

Each time you run the playground, you'll get a new hand and output similar to the following:

```
—— Example of: Create a Blackjack card dealer ——
🎲🎲🎲 for 21 points
```

Blackjack!

# Key points

- Publishers transmit a sequence of values over time to one or more subscribers, either synchronously or asynchronously.
- A subscriber can subscribe to a publisher to receive values; however, the subscriber's input and failure types must match the publisher's output and failure types.
- There are two built-in operators you can use to subscribe to publishers: `sink(_:_:)` and `assign(to:on:)`.
- A subscriber may increase the demand for values each time it receives a value, but it cannot decrease demand.
- To free up resources and prevent unwanted side effects, cancel each subscription when you're done.
- You can also store a subscription in an instance or collection of `AnyCancellable` to receive automatic cancelation upon deinitialization.
- A future can be used to receive a single value asynchronously at a later time.
- Subjects are publishers that enable outside callers to send multiple values asynchronously to subscribers, with or without a starting value.
- Type erasure enables prevents callers from being able to access additional details of the underlying type.
- Use the `print()` operator to log all publishing events to the console and see what's going on.

# Where to go from here?

Congratulations! You've taken a huge step forward by completing this chapter. You learned how to work with publishers to send values and completion events, and how to use subscribers to receive those values and events. Up next, you'll learn how to

manipulate the values coming from a publisher to help filter, transform or combine them.

# Section II: Operators

If you think of Combine as a language, such as the English language, operators are its words. The more operators you know, the better you can articulate your intention and the logic of your app. Operators are a huge part of the Combine ecosystem which lets you manipulate values emitted from upstream publishers in meaningful and logical ways.

In this section you'll learn the majority of operators Combine has to offer, divided into useful groups: transforming, filtering, combining, time-based and sequence. Once you've got your operator chops down, you'll wrap up this section with a hands-on project to practice your newly-acquired knowledge.

Specifically, you'll cover:

**Chapter 3: Transforming Operators**: Before a subscriber receives values from a publisher, you'll often want to manipulate those values in some way. One of the most common things you'll want to do is transform those values into some form that is ideal for use by the subscriber. By the end of this chapter you'll be transforming all the things.

**Chapter 4: Filtering Operators**: In this chapter you'll learn about filtering values from Combine publishers, so you can easily control the values published by the upstream and only deal with the ones you care about.

**Chapter 5: Combining Operators**: Publishers are extremely powerful, but they're even more powerful when composed together! This chapter will teach you about Combine's combining operators which let you take multiple publishers and create meaningful logical relationships between them.

**Chapter 6: Time Manipulation Operators**: A large part of asynchronous programming relating to processing values over time, this chapter goes into the details of performing complex time-based tasks that would be hard to do without Combine.

**Chapter 7: Sequence Operators**: When you think about it, Publishers are merely

sequences. As such, there are many useful operators that let you target specific values, or gather information about the sequence as a whole, which you'll learn about in this chapter.

**Chapter 8: In Practice: Building a Collage App**: It's time to try your new Combine skills in a real project and learn how to make Foundation and UIKit play along with your reactive code.

# Chapter 3: Transforming Operators

By Scott Gardner

Having completed section 1, you've already learned a lot. You should feel pretty good about that accomplishment! You've laid a solid foundation on the fundamentals of Combine, and now you're ready to build upon it.

In this chapter, you're going to learn about one of the essential categories of operators in Combine: Transforming operators. You'll use transforming operators all the time, to manipulate values coming from publishers into a format that is usable for your subscribers. As you'll see, there are parallels between transforming operators in Combine and regular operators in the Swift standard library, such as `map` and `flatMap`.

By the end of this chapter, you'll be transforming all the things!

## Getting started

Open the starter playground for this chapter, which already has Combine imported and will be ready to go.

## Operators are publishers

In Combine, methods that perform an operation on values coming from a publisher are called *operators*.

Each Combine operator actually returns a publisher. Generally speaking, that publisher receives the upstream values, manipulates the data, and then sends that data downstream. To streamline things conceptually, the focus will be on using the

operator and working with its output. Unless an operator's purpose is to handle errors, if it receives an error from an upstream publisher, it will just publish that error downstream.

> **Note**: You'll focus on transforming operators in this chapter, so error handling will not appear in each operator example. You'll learn all about error handling in Chapter 16, "Error Handling."

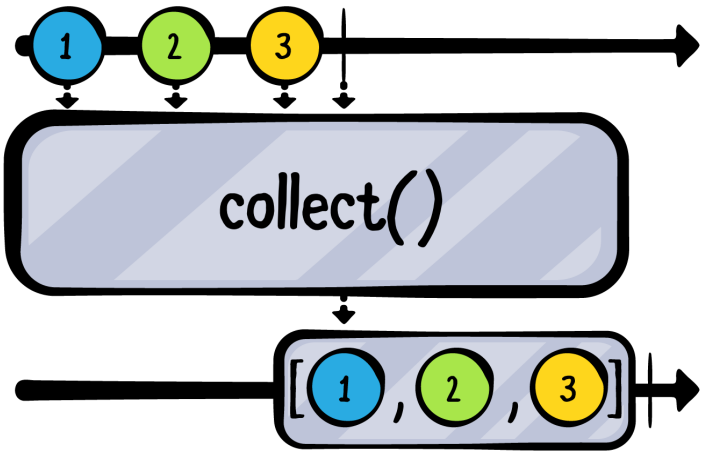
# Collecting values

Publishers can emit individual values or collections of values. You'll frequently want to work with collections, such as when you want to populate a list of views. You'll learn how to do this later in the book.

## collect()

The `collect` operator provides a convenient way to transform a stream of individual values from a publisher into an array of those values. To help understand how this and all other operators you'll learn about in this book, you'll use **marble diagrams**.

Marble diagrams help to visualize how operators work. The top line is the upstream publisher. The box represents the operator. And the bottom line is the subscriber, or more specifically, what the subscriber will *receive* after the operator manipulates the values coming from the upstream publisher. The bottom line could also be another operator that receives the output from the upstream publisher, performs its operation, and sends those values downstream.




As depicted in this marble diagram, `collect` will buffer a stream of individual values into an array of those values once the upstream publisher completes. It will then emit that array downstream.

Add this new example to your playground:

```
example(of: "collect") {
  ["A", "B", "C", "D", "E"].publisher
    .sink(receiveCompletion: { print($0) },
          receiveValue: { print($0) })
    .store(in: &subscriptions)
}
```

This is not using the `collect` operator yet. Run the playground, and you'll see each value is emitted and printed individually followed by the completion:

```
—— Example of: collect ——
A
B
C
D
E
finished
```

Now insert the use of `collect` before the `sink`. Your code should look like this:

```
["A", "B", "C", "D", "E"].publisher
  .collect()
  .sink(receiveCompletion: { print($0) },
        receiveValue: { print($0) })
  .store(in: &subscriptions)
```

Run the playground, and now the sink receives one emitted collection followed by the completion event:

```
—— Example of: collect ——
["A", "B", "C", "D", "E"]
finished
```

> **Note**: Be careful when working with `collect()` and other buffering operators that do not require specifying a count or limit. They will use an unbounded amount of memory to store received values.

There are a few variations of the `collect` operator. For example, you can specify that you only want to receive up to a certain number of values.

Replace the following line:

```
.collect()
```

With:

```
.collect(2)
```

Run the playground, and you'll see the following output:

```
—— Example of: collect ——
["A", "B"]
["C", "D"]
["E"]
finished
```

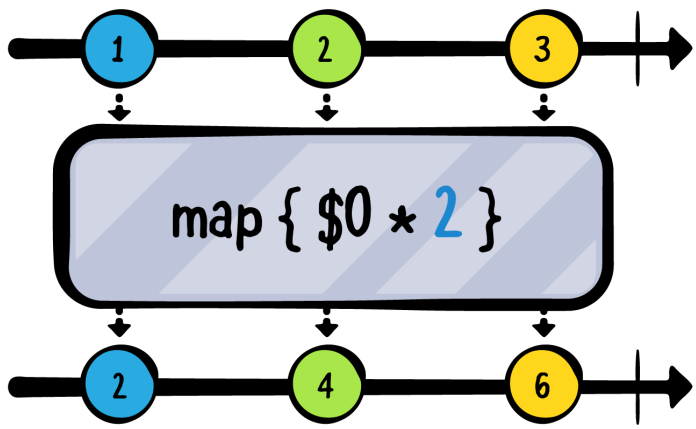
The last value, `E`, is emitted as an array. That's because the upstream publisher completed before `collect` filled its prescribed buffer, so it sent whatever it had left as an array.

# Mapping values

In addition to collecting values, you'll often want to transform those values in some way. Combine offers several mapping operators for that purpose.

## map(_:)

The first you'll learn about is `map`, which works just like Swift's standard `map`, except that it operates on values emitted from a publisher. In the marble diagram, `map` takes a closure that multiplies each value by 2.




Add this new example to your playground:

```
example(of: "map") {
  // 1
  let formatter = NumberFormatter()
  formatter.numberStyle = .spellOut

  // 2
  [123, 4, 56].publisher
    // 3
    .map {
      formatter.string(for: NSNumber(integerLiteral: $0)) ?? ""
    }
    .sink(receiveValue: { print($0) })
    .store(in: &subscriptions)
}
```

Here's the play-by-play:

1. Create a number formatter to spell out each number.
2. Create a publisher of integers.
3. Use `map`, passing a closure that gets upstream values and returns the result of using the formatter to return the number's spelled out string.

Run the playground, and you will see this output:

```
——— Example of: map ———
one hundred twenty-three
four
fifty-six
```

# Map key paths

The `map` family of operators also includes three versions that can map into one, two, or three properties of a value using key paths. Their signatures are as follows:

- `map<T>(_:)`
- `map<T0, T1>(_:_:)`
- `map<T0, T1, T2>(_:_:_:)`

The `T` represents the type of values found at the given key paths.

In the next example, you'll use the `Coordinate` type and `quadrantOf(x:y:)` method defined in **Sources/SupportCode.swift**. `Coordinate` has two properties: `x` and `y`. `quadrantOf(x:y:)` takes x and y values as parameters and returns a string indicating the quadrant for the x and y values.

> **Note**: Quadrants are part of coordinate geometry. For more information you can visit mathworld.wolfram.com/Quadrant.html.

Feel free to review these definitions if you're interested, and then add the following example to your playground:

Add this example to see how you'd use `map(_:_:)` to map into two key paths:

```
example(of: "map key paths") {
  // 1
  let publisher = PassthroughSubject<Coordinate, Never>()

  // 2
```

```
    publisher
      // 3
      .map(\.x, \.y)
      .sink(receiveValue: { x, y in
        // 4
        print(
          "The coordinate at (\(x), \(y)) is in quadrant",
          quadrantOf(x: x, y: y)
        )
      })
      .store(in: &subscriptions)

    // 5
    publisher.send(Coordinate(x: 10, y: -8))
  }
```

In this example you're using the version of `map` that maps into two properties via key paths.

Step-by-step, you:

1. Create a publisher of `Coordinates` that will never emit an error.
2. Begin a subscription to the publisher.
3. Map into the `x` and `y` properties of `Coordinate` using their key paths.
4. Print a statement that indicates the quadrant of the provide `x` and `y` values.
5. Send some coordinates on the publisher.

Run the playground and the output from this subscription will be the following:

```
—— Example of: map key paths ——
The coordinate at (10, -8) is in quadrant 4
The coordinate at (0, 5) is in quadrant boundary
```

## tryMap(_:)

Several operators, including `map`, have a counterpart `try` operator that will take a closure that can throw an error. If you throw an error, it will emit that error downstream. Add this example to the playground:

```
example(of: "tryMap") {
  // 1
  Just("Directory name that does not exist")
    // 2
    .tryMap { try
```

```
    FileManager.default.contentsOfDirectory(atPath: $0) }
        // 3
        .sink(receiveCompletion: { print($0) },
              receiveValue: { print($0) })
        .store(in: &subscriptions)
}
```

Here's what you just did, or at least tried to do!

1. Create a publisher of a string representing a directory name that does not exist.
2. Use `tryMap` to attempt to get the contents of that nonexistent directory.
3. Receive and print out any values or completion events.

Notice that you still need to use the `try` keyword when calling a throwing method.

Run the playground and observe that `tryMap` outputs a failure completion event with the appropriate "folder doesn't exist" error (output abbreviated):

```
—— Example of: tryMap ——
failure(..."The folder “Directory name that does not exist”
doesn't exist."...)
```

# Flattening publishers

This section's title probably won't shed any light on what you're about to learn, unless you have some prior experience with reactive programming. However, even for those *with* previous experience, the following operator has left many developers bewildered about what it's doing.

## flatMap(maxPublishers:_:)

The operator in question here is called `flatMap`. You'll start with a fairly textbook definition, followed by a practical example that really demonstrates how `flatMap` works. By the end of this section, everything should be illuminated for you.

The flatMap operator can be used to flatten multiple upstream publishers into a single downstream publisher — or more specifically, flatten the emissions from those publisher. The publisher returned by flatMap does not — and often will not — be of the same type as the upstream publishers it receives. But, before exploring how flatMap can flatten multiple publisher inputs, first you'll focus on what flatMap outputs.

Do you remember `flatMap` from the Swift standard library, which was renamed `compactMap` in Swift 4.1? If so, you might be thinking: "Woah, this is not at all the same." And you'd be right!

A common use case for `flatMap` in Combine is when you want to subscribe to properties of values emitted by a publisher that are *themselves* publishers. You'll check out an example to see this in action in a moment.

Start by taking a look at the code in **Sources/SupportCode.swift**. It includes the definition of a `Chatter` structure with two properties:

1. `name` is a regular string.
2. `message` is a `CurrentValueSubject` subject that is initialized with the `message` string passed in.

```
public struct Chatter {
  public let name: String
  public let message: CurrentValueSubject<String, Never>

  public init(name: String, message: String) {
    self.name = name
    self.message = CurrentValueSubject(message)
  }
}
```

Switch back to the main playground page and add this new example:

```
example(of: "flatMap") {
  // 1
  let charlotte = Chatter(name: "Charlotte", message: "Hi, I'm
Charlotte!")
  let james = Chatter(name: "James", message: "Hi, I'm James!")

  // 2
  let chat = CurrentValueSubject<Chatter, Never>(charlotte)

  // 3
  chat
    .sink(receiveValue: { print($0.message.value) })
    .store(in: &subscriptions)
}
```

From the top, you:

1. Create two instances of `Chatter`: `charlotte` and `james`.
2. Create a `chat` publisher initialized with `charlotte`.

3. Subscribe to `chat` and print out the message of any received `Chatter` structure.

Run the playground, and as you might expect, Charlotte's message is printed out:

```
—— Example of: flatMap ——
Charlotte wrote: Hi, I'm Charlotte!
```

Now add this code to the example:

```
// 4
charlotte.message.value = "Charlotte: How's it going?"

// 5
chat.value = james
```

With this code you:

4. Change Charlotte's message.

5. Change the `chat` publisher's current value to `james`.

Run the playground again, and you'll see the following:

```
Charlotte wrote: Hi, I'm Charlotte!
James wrote: Hi, I'm James!
```

You *don't* see Charlotte's new message, but you do see James' initial message. That's because you're subscribed to `chat`, which is a `Chatter` publisher. You are not subscribed to the `message` publisher property of each emitted `Chatter`. What if you wanted to subscribe to the `message` of every chat? You can use `flatMap`.

Find the following code:

```
chat
  .sink(receiveValue: { print($0.message.value) })
  .store(in: &subscriptions)
```

And replace it with:

```
chat
  // 6
  .flatMap { $0.message }
  // 7
  .sink(receiveValue: { print($0) })
  .store(in: &subscriptions)
```

Here you:

6. Flat map into the `Chatter` structure `message` publisher.
7. Change the handler to print the value received, which is now a string, not a `Chatter` instance.

Run the playground again, and now you'll see Charlotte's new message printed.

```
Hi, I'm Charlotte!
Charlotte: How's it going?
Hi, I'm James!
```

Now add the following code to the example, which changes each `Chatter`'s message value:

```
james.message.value = "James: Doing great. You?"
charlotte.message.value = "Charlotte: I'm doing fine thanks."
```

Run the playground, and you may be surprised by what you see:

```
James: Doing great. You?
Charlotte: I'm doing fine thanks.
```

Charlotte's new message was printed, even though you changed `chat`'s value to `james`. Herein lies the "flatten" part — huzzah!

Recall the definition from earlier: `flatMap` flattens the output from all received publishers into a single publisher. This poses a memory concern, because it will buffer as many publishers as you send it to update the single publisher it emits downstream.

To help you manage `flatMap`'s memory footprint, you can optionally specify how many publishers `flatMap` will receive and buffer using its `maxPublishers` parameter.

Find the following line:

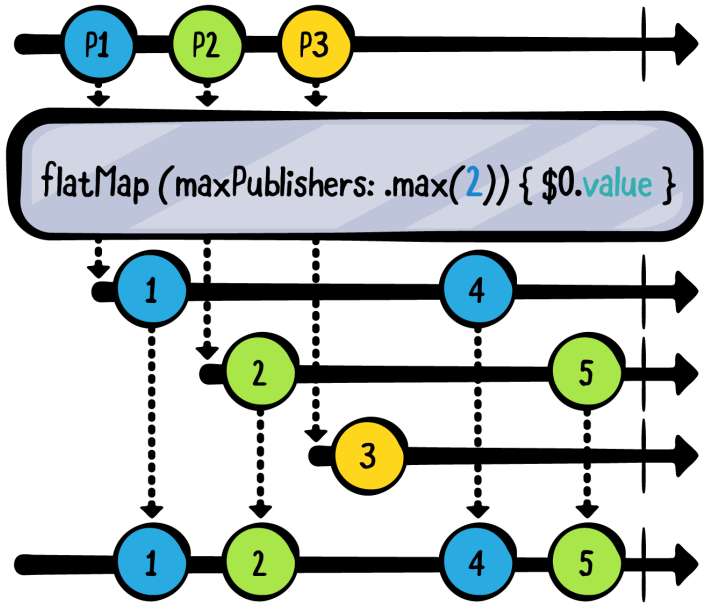
```
.flatMap { $0.message }
```

And replace it with:

```
.flatMap(maxPublishers: .max(2)) { $0.message }
```

You specify that `flatMap` will receive a maximum of two upstream publishers. It will ignore any additional publishers. If not specified, `maxPublishers` defaults to `.unlimited`.

Let's pause on this example for a moment to check out `flatMap` in a marble diagram, now that you have a better understanding of what it's doing:




In the diagram, `flatMap` receives three publishers: `P1`, `P2`, and `P3`. Each of these publishers has a `value` property that is also a publisher. `flatMap` emits the `value` publishers' values from `P1` and `P2`, but ignores `P3` because `maxPublishers` is set to `2`.

Back to the example. You previously set `flatMap`'s `maxPublishers` parameter to `2`. Now add the following code to the bottom of the example to see how this affects the output:

```
// 8
let morgan = Chatter(name: "Morgan",
                     message: "Hey guys, what are you up to?")

// 9
chat.value = morgan

// 10
charlotte.message.value = "Did you hear something?"
```

What you did here is:

8. Create a third `Chatter` instance.

9. Add that `Chatter` onto the `chat` publisher.

10. Change Charlotte's message.

What do you think will be printed out? Run the playground and see if you were right:

```
—— Example of: flatMap ——
Hi, I'm Charlotte!
Charlotte: How's it going?
Hi, I'm James!
James: Doing great. You?
Charlotte: I'm doing fine thanks.
Did you hear something?
```

Morgan's message is not printed, because `flatMap` will only receive up to a max of two publishers.

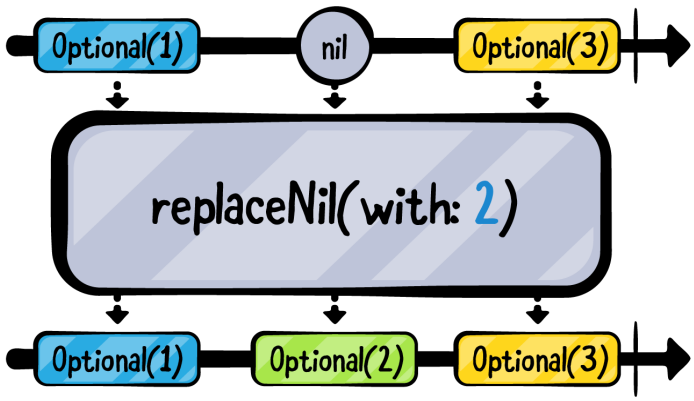
You now have a handle on one of the most powerful operators in Combine. However, `flatMap` is not the only way to swap input with a different output. So, before wrapping up this chapter, you'll learn a couple more useful operating for doing the ol' switcheroo.

# Replacing upstream output

Earlier in the `map` example, you worked with `Foundation`'s `Formatter.string(for:)` method. It produces an optional string, and you used the nil-coalescing operator (`??`) to replace a `nil` value with a non-`nil` value. Combine also includes an operator that you can use when you want to *always* deliver a value.

## replaceNil(with:)

As depicted in the following marble diagram, `replaceNil` will receive optional values and replace `nil`s with the value you specify:




Add this new example to your playground:

```
example(of: "replaceNil") {
  // 1
  ["A", nil, "C"].publisher
    .replaceNil(with: "–") // 2
    .sink(receiveValue: { print($0) }) // 3
    .store(in: &subscriptions)
}
```

What you just did:

1. Create a publisher from an array of optional strings.
2. Use `replaceNil(with:)` to replace `nil` values received from the upstream publisher with a new non-`nil` value.
3. Print out the value.

Run the playground, and you will see the following:

```
—— Example of: replaceNil ——
Optional("A")
Optional("–")
Optional("C")
```

The optional values were not converted to non-optional ones. True to its name, `replaceNil(with:)` replaced a `nil` with a non-`nil` value. One way that you could convert the output from `replaceNil` to a non-optional value is by inserting the use of `map` to force-unwrap it. Do so by changing your code to the following:

```
["A", nil, "C"].publisher
  .replaceNil(with: "–")
  .map { $0! }
  .sink(receiveValue: { print($0) })
  .store(in: &subscriptions)
```

Now, run the playground, and you'll see each unwrapped value printed instead:

```
A
–
C
```

There is a subtle but important difference between using `??` and `replaceNil`. The `??` operator can return another optional, while `replaceNil` cannot. Change the usage of `replaceNil` to the following, and you will get an error that the optional must be unwrapped:

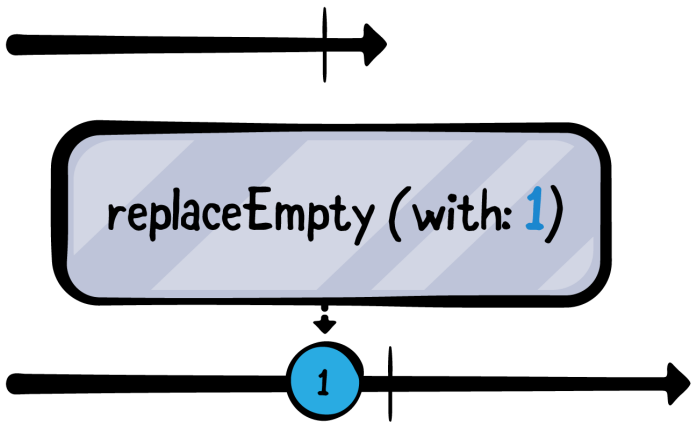
```
.replaceNil(with: "–" as String?)
```

Revert that change before moving on. This example also demonstrates how you can chain together multiple operators in a compositional way. This allows you to manipulate the values coming from the origin publisher to the subscriber in a wide variety of ways.

## replaceEmpty(with:)

You can use the `replaceEmpty(with:)` operator to replace — or really, *insert* — a value if a publisher completes without emitting a value.

In the following marble diagram, the publisher completes without emitting anything, and at that point the `replaceEmpty(with:)` operator inserts a value and publishes it downstream:




Add this new example to see it in action:

```
example(of: "replaceEmpty(with:)") {
  // 1
  let empty = Empty<Int, Never>()

  // 2
  empty
    .sink(receiveCompletion: { print($0) },
          receiveValue: { print($0) })
    .store(in: &subscriptions)
}
```

What you're doing here:

1. Create an empty publisher that immediately emits a completion event.
2. Subscribe to it, and print received events.

The `Empty` publisher type can be used to create a publisher that immediately emits a `.finished` completion event. It can also be configured to *never* emit anything by passing `false` to its `completeImmediately` parameter, which is `true` by default. This publisher is useful for demo or testing purposes, or when all you want to do is signal completion of some task to a subscriber. Run the playground and its completion event is printed:

```
—— Example of: replaceEmpty ——
finished
```

Now, insert this line of code before calling `sink`:

```
  .replaceEmpty(with: 1)
```

Run the playground again, and this time you get a 1 before the completion:

```
1
finished
```

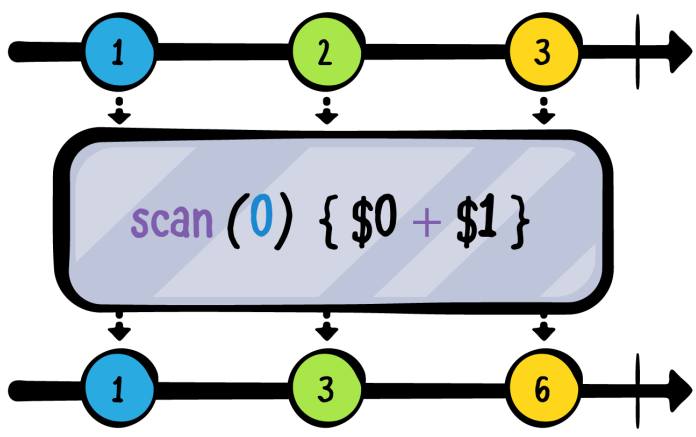
# Incrementally transforming output

You've seen how Combine includes operators such as `map` that correspond and work similarly to higher-order functions found in the Swift standard library. However, Combine has a few more tricks up its sleeve that let you manipulate values received from an upstream publisher.

## scan(_:_:)

A great example of this in the transforming category is `scan`. It will provide the current value emitted by an upstream publisher to a closure, along with the last value returned by that closure.

In the following marble diagram, `scan` begins by storing a starting value of `0`. As it receives each value from the publisher, it adds it to the previously stored value, and then stores *and* emits the result:



> **Note**: If you are using the full project to enter and run this code, there's no straightforward way to plot the output — as is possible in a playground. Instead, you can print the output by changing the `sink` code in the example below to `.sink(receiveValue: { print($0) }).`

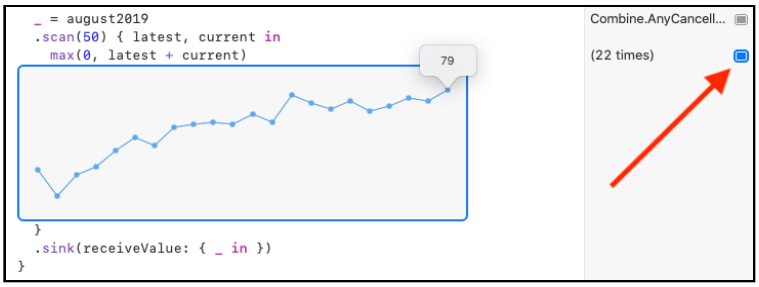
For a practical example of how to use `scan`, add this new example to your playground:

```
example(of: "scan") {
  // 1
  var dailyGainLoss: Int { .random(in: -10...10) }

  // 2
  let august2019 = (0..<22)
    .map { _ in dailyGainLoss }
    .publisher

  // 3
  august2019
    .scan(50) { latest, current in
      max(0, latest + current)
    }
    .sink(receiveValue: { _ in })
    .store(in: &subscriptions)
}
```

In this example, you:

1. Create a computed property that generates a random integer between `-10` and `10`.
2. Use that generator to create a publisher from an array of random integers representing fictitious daily stock price changes for a month.
3. Use `scan` with a starting value of `50`, and then add each daily change to the running stock price. The use of `max` keeps the price non-negative — thankfully stock prices can't fall below zero!

This time, you did not print anything in the subscription. Run the playground, and then click the square **Show Results** button in the right results sidebar.




Talk about a bull run! How'd your stock do?

There's also an error-throwing `tryScan` operator that works similarly. If the closure throws an error, `tryScan` fails with that error.

# Challenge

Practice makes *permanent*. Complete this challenge to ensure you're good to go with transforming operators before moving on.

## Challenge: Create a phone number lookup using transforming operators

Your goal for this challenge is to create a publisher that does two things:

1. Receives a string of ten numbers or letters.
2. Looks up that number in a contacts data structure.

The starter playground, which can be found in the **challenge** folder, includes a `contacts` dictionary and three functions. You'll need to create a subscription to the `input` publisher using transforming operators and those functions. Insert your code right below the `Add your code here` placeholder, before the `forEach` blocks that will test your implementation.

> **Tip**: A function or closure can be passed directly to an operator as a parameter if the function signature matches. For example, `map(convert)`.

Breaking down this challenge, you'll need to:

1. Convert the input to numbers — use the `convert` function, which will return `nil`

if it cannot convert the input to an integer.

2. If `nil` was returned from the previous operator, replace it with a `0`.
3. Collect ten values at a time, which correspond to the three-digit area code and seven-digit phone number format used in the United States.
4. Format the collected string value to match the format of the phone numbers in the contacts dictionary — use the provided `format` function.
5. "Dial" the input received from the previous operator — use the provided `dial` function.

## Solution

Did your code produce the expected results? Starting with a subscription to `input`, first you needed to convert the string input one character at a time into integers:

```
input
  .map(convert)
```

Next you needed to replace `nil` values returned from `convert` with `0`s:

```
.replaceNil(with: 0)
```

To look up the result of the previous operations, you needed to collect those values, and then format them to match the phone number format used in the `contacts` dictionary:

```
.collect(10)
.map(format)
```

Finally, you needed to use the `dial` function to look up the formatted string input, and then subscribe:

```
.map(dial)
.sink(receiveValue: { print($0) })
```

Running the playground will produce the following:

```
——— Example of: Create a phone number lookup ———
Contact not found for 000-123-4567
Dialing Marin (408-555-4321)...
Dialing Shai (212-555-3434)...
```

Bonus points if you hook this up to a VoIP service!

# Key points

- Methods that perform operations on output from publishers are called operators.
- Operators are also publishers.
- Transforming operators convert input from an upstream publisher into output that is suitable for use downstream.
- Marble diagrams are a great way to visualize how each Combine operators work.
- Be careful when using any operators that buffer values such as `collect` or `flatMap` to avoid memory problems.
- Be mindful when applying existing knowledge of functions from Swift standard library. Some similarly-named Combine operators work the same while others work entirely differently.
- Multiple operators can be chained together in a subscription.

# Where to go from here?

Way to go! You just transformed yourself into a transforming titan.

Now it's time to learn how to use another essential collection of operators to filter what you get from an upstream publisher.

# Chapter 4: Filtering Operators

By Shai Mishali

As you might have realized by now, operators are basically the vocabulary that you use to communicate with Combine publishers. The more "words" you know, the better your control of your data will be.

In the previous chapter, you learned how to consume values and transform them into different values — definitely one of the most useful operator categories for your daily work.

But what happens when you want to limit the values or events emitted by the publisher, and only consume some of them? This chapter is all about how to do this with a special group of operators: Filtering operators!

Luckily, many of these operators have parallels with the same names in the Swift standard library, so don't be surprised if you're able to *filter* some of this chapter's content. :]

It's time to dive right in.

## Getting started

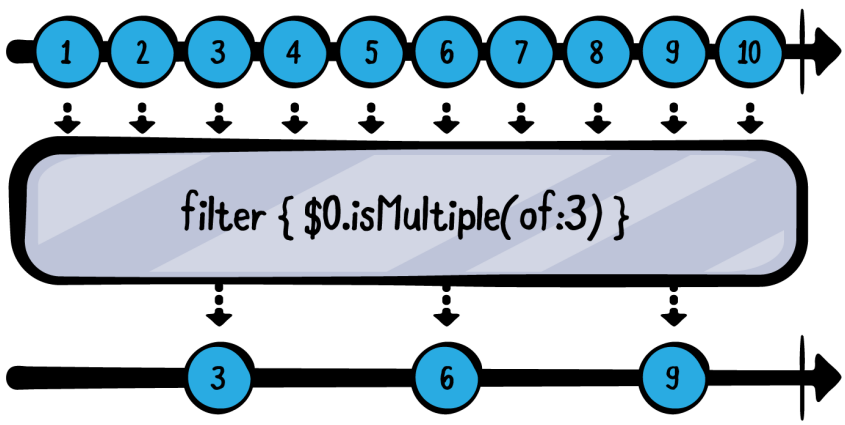
You can find the starter playground for this chapter, **Starter.playground**, in the **projects** folder. As you progress through this chapter, you'll write code in the playground and then run the playground. This will help you understand how different operators manipulate events emitted by your publisher.

**Note**: Most operators in this chapter have parallels with a `try` prefix, for example, `filter` vs. `tryFilter`. The only difference between them is that the latter provides a *throwing* closure. Any error you throw from within the closure will terminate the publisher with the thrown error. For brevity's sake, this chapter will only cover the non-throwing variations, since they are virtually identical.

# Filtering basics

This first section will deal with the basics of filtering — consuming a publisher of values and conditionally deciding which of them to pass to the consumer.

The easiest way to do this is the aptly-named operator — `filter`, which takes a closure returning a `Bool` and only passes down values that match the provided predicate:




Add this new example to your playground:

```
example(of: "filter") {
  // 1
  let numbers = (1...10).publisher

  // 2
  numbers
```

```
    .filter { $0.isMultiple(of: 3) }
    .sink(receiveValue: { n in
      print("\(n) is a multiple of 3!")
    })
    .store(in: &subscriptions)
}
```

In the above example, you:

1. Create a new publisher, which will emit a finite number of values — `1` through `10`, and then complete, using the `publisher` property on `Sequence` types.
2. Use the `filter` operator, passing in a predicate where you'll only allow through numbers that are multiples of three.

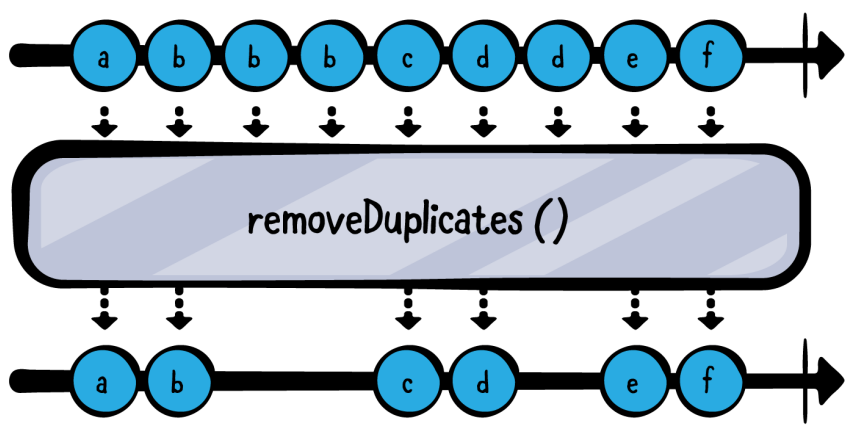
Build and run your playground. You should see the following in your console:

```
—— Example of: filter ——
3 is a multiple of 3!
6 is a multiple of 3!
9 is a multiple of 3!
```

Such an elegant way to cheat on your math homework, isn't it? :]

Many times in the lifetime of your app, you have publishers that emit identical values in a row that you might want to ignore. For example, if a user types "a" five times in a row and then types "b", you might want to disregard the excessive "a"s.

Combine provides the perfect operator for the task: `removeDuplicates`:



Notice how you don't have to provide any arguments to this operator! `removeDuplicate` automatically works for any values conforming to `Equatable`, including `String`.

Add the following example of `removeDuplicates()` to your playground — and be sure to include a space before the `?`:

```
example(of: "removeDuplicates") {
  // 1
  let words = "hey hey there! want to listen to mister mister ?"
                .components(separatedBy: " ")
                .publisher
  // 2
  words
    .removeDuplicates()
    .sink(receiveValue: {
      print($0)
    })
    .store(in: &subscriptions)
}
```

This code isn't too different from the last one. You:

1. Separate a sentence into an array of words (e.g., `Array<String>`) and then create a new publisher to emit these words.
2. Apply `removeDuplicates()` to your `words` publisher.

Build and run your playground and take a look at the debug console:

```
—— Example of: removeDuplicates ——
hey
there!
want
to
listen
to
mister
?
```

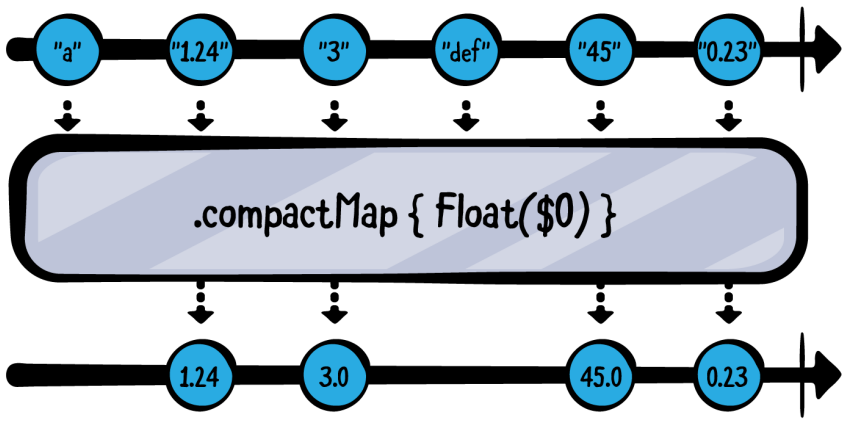
As you can see, you've skipped the second "hey" and the second "mister". Awesome!

> **Note**: What about values that don't conform to `Equatable`? Well, `removeDuplicates` has another overload that takes a closure with two values, and returns a `Bool` to indicate whether the values are equal or not.

# Compacting and ignoring

Quite often, you'll find yourself dealing with a publisher emitting `Optional` values. Or even more commonly, you'll want to perform some operation on your values that might return `nil`, but who wants to handle all those `nil`s ?!

If your spidey sense is tingling, thinking of a very well-known method on `Sequence` from the Swift standard library called `compactMap` that does that job, good news – there's also an operator with the same name!




Add the following to your playground:

```
example(of: "compactMap") {
  // 1
  let strings = ["a", "1.24", "3", "def", "45",
"0.23"].publisher

  // 2
  strings
    .compactMap { Float($0) }
    .sink(receiveValue: {
      // 3
      print($0)
    })
    .store(in: &subscriptions)
}
```

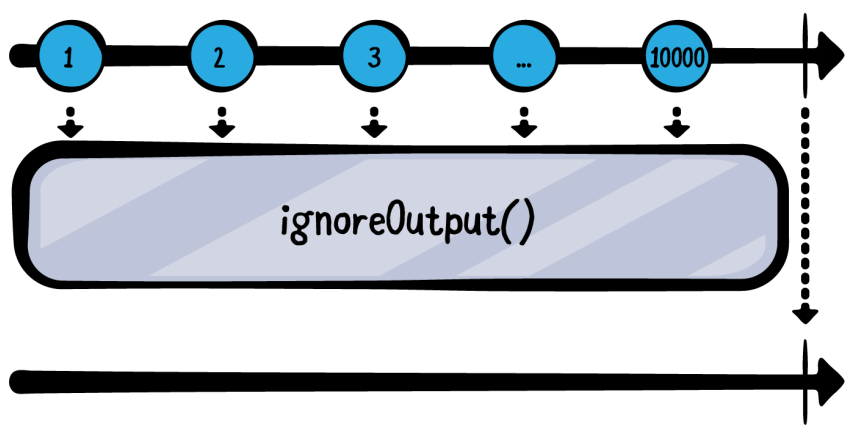
Just as the diagram outlines, you:

1. Create a publisher that emits a finite list of strings.
2. Use `compactMap` to attempt to initialize a `Float` from each individual string. If `Float`'s initializer doesn't know how to convert the provided string, it returns `nil`.
3. Only print strings that have been successfully converted to `Floats`.

Build and run the above example in your playground and you should see output similar to the diagram above:

```
—— Example of: compactMap ——
1.24
3.0
45.0
0.23
```

All right, why don't you take a quick break from all these values... who cares about those, right? Sometimes, all you want to know is that the publisher has finished emitting values, disregarding the actual values. When such a scenario occurs, you can use the `ignoreOutput` operator:




As the diagram above shows, it doesn't matter which values are emitted or how many of them, as they're all ignored; you only push completion events through to the consumer.

Experiment with this example by adding the following code to your playground:

```
example(of: "ignoreOutput") {
  // 1
  let numbers = (1...10_000).publisher

  // 2
  numbers
    .ignoreOutput()
    .sink(receiveCompletion: { print("Completed with: \($0)") },
          receiveValue: { print($0) })
    .store(in: &subscriptions)
}
```

In the above example, you:

1. Create a publisher emitting 10,000 values from `1` through `10,000`.
2. Add the `ignoreOutput` operator to your chain, which omits all values and emits only the completion event to the consumer.

Can you guess what the output of this code will be?

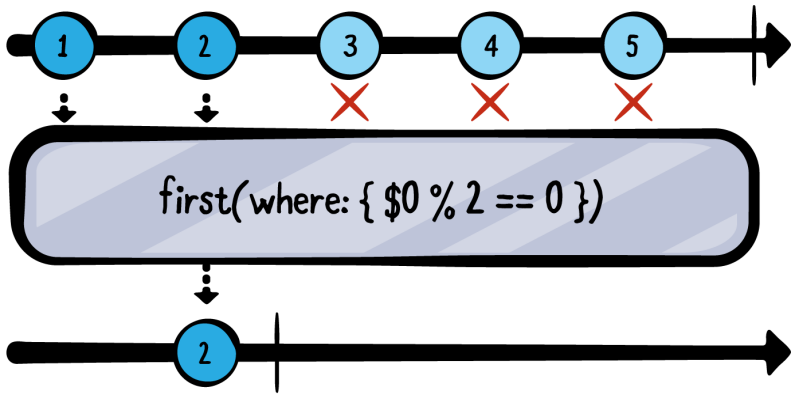
If you guessed that no values will be printed, you're right! Build and run your playground and check out the debug console:

```
—— Example of: ignoreOutput ——
Completed with: finished
```

# Finding values

In this section, you'll learn about two operators that also have their origins in the Swift standard library: `first(where:)` and `last(where:)`. As their names imply, you use them to find and emit *only* the first *or* the last value matching the provided predicate, respectively.

Time to check out a few examples, starting with `first(where:)`.




This operator is interesting because it's *lazy*, meaning: It only takes as many values as it needs until it finds one matching the predicate you provided. As soon as it finds a match, it cancels and completes the entire stream.

Add the following piece of code to your playground to see how this works:

```
example(of: "first(where:)") {
  // 1
  let numbers = (1...9).publisher

  // 2
  numbers
    .first(where: { $0 % 2 == 0 })
    .sink(receiveCompletion: { print("Completed with: \($0)") },
          receiveValue: { print($0) })
    .store(in: &subscriptions)
}
```

Here's what the code you've just added does:

1. Creates a new publisher emitting numbers from `1` through `9`.
2. Uses the `first(where:)` operator to find the first emitted *even* value.

Build and run this example in your playground and look at the console output:

```
—— Example of: first(where:) ——
2
Completed with: finished
```

It works exactly like you probably guessed it would. But wait, what about the

upstream, meaning the `numbers` publisher? Does it keep emitting its values even after it finds a matching even number? Test this theory by finding the following line:

```
numbers
```

Then add the `print("numbers")` operator immediately after that line, so it looks as follows:

```
numbers
  .print("numbers")
```

> **Note**: You can use the `print` operator anywhere in your chain to see exactly what's going from that point onward in the chain.

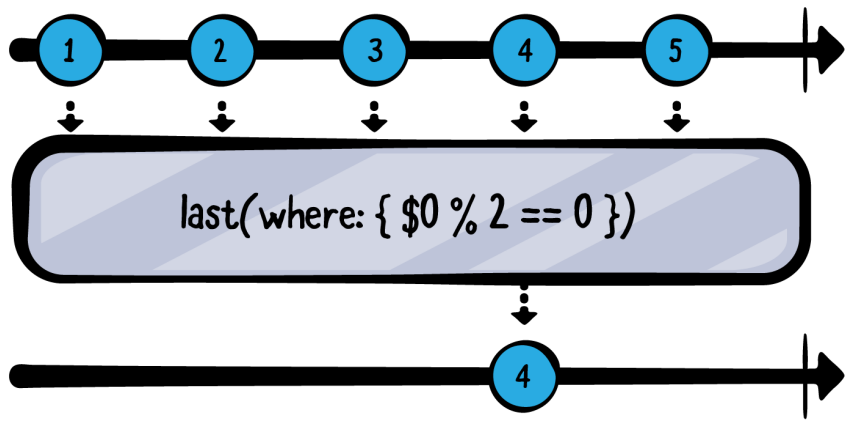
Build and run your playground again, and take a look at it. Your output should like similar to the following:

```
—— Example of: first(where:) ——
numbers: receive subscription: (1...9)
numbers: request unlimited
numbers: receive value: (1)
numbers: receive value: (2)
numbers: receive cancel
2
Completed with: finished
```

This is very interesting!

As you can see, as soon as `first(where:)` finds a matching value, it sends a cancellation through the chain, causing the upstream to stop emitting values. Very handy!

Now, you can move on to the opposite of this operator — `last(where:)`, whose purpose is to find the *last* value matching a provided predicate.




As opposed to `first(where:)`, this operator is *greedy* since it must wait for all values to emit to know whether a matching value has been found. For that reason, the upstream must be a publisher that completes at some point.

Add the following code to your playground:

```
example(of: "last(where:)") {
  // 1
  let numbers = (1...9).publisher

  // 2
  numbers
    .last(where: { $0 % 2 == 0 })
    .sink(receiveCompletion: { print("Completed with: \($0)") },
          receiveValue: { print($0) })
    .store(in: &subscriptions)
}
```

Much like the previous code example, you:

1. Create a publisher that emits numbers between 1 and 9.
2. Use the `last(where:)` operator to find the *last* emitted odd value.

Did you guess what the output will be? Build and run your playground and find out:

```
—— Example of: last(where:) ——
8
Completed with: finished
```

Remember I said earlier that the publisher must complete for this operator to work? Why is that?

Well, that's because there's no way for the operator to know if the publisher will emit a value that matches the criteria down the line, so the operator must know the full scope of the publisher before it can determine the last item matching the predicate.

To see this in action, replace the entire example with the following:

```
example(of: "last(where:)") {
  let numbers = PassthroughSubject<Int, Never>()

  numbers
    .last(where: { $0 % 2 == 0 })
    .sink(receiveCompletion: { print("Completed with: \($0)") },
          receiveValue: { print($0) })
    .store(in: &subscriptions)

  numbers.send(1)
  numbers.send(2)
  numbers.send(3)
  numbers.send(4)
  numbers.send(5)
}
```

In this example, you use a `PassthroughSubject` and manually send events to it.

Build and run your playground again, and you should see... absolutely nothing:

```
—— Example of: last(where:) ——
```

As expected, since the publisher never completes, there's no way to determine the last value matching the criteria.

Finally, add the following as the last line of the example to send a completion to the subject:

```
numbers.send(completion: .finished)
```

Run your playground again, and everything should now work as expected:

```
—— Example of: last(where:) ——
4
Completed with: finished
```

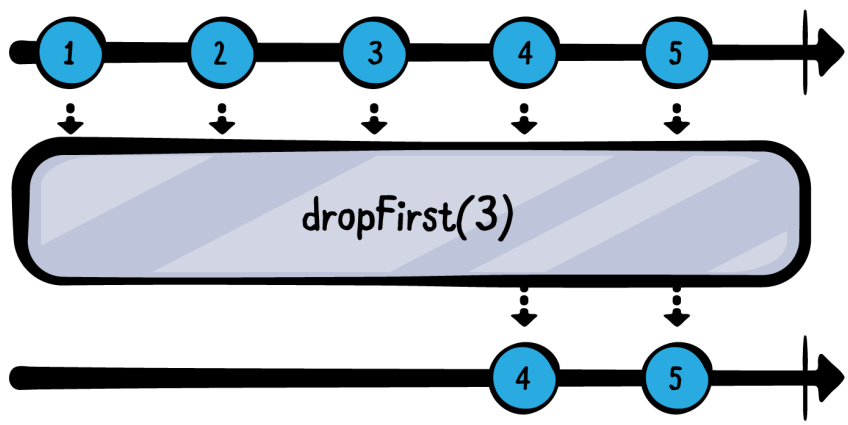
I guess that everything must come to an end... or completion, in this case.

# Dropping values

Dropping values is a useful capability you'll often need to leverage when working with publishers. For example, you can use it when you want to ignore values from one publisher until a second one starts publishing or if you want to ignore a specific amount of values at the start of the stream.

Three operators fall into this category, and you'll start by learning about the simplest one first — `dropFirst`.

The `dropFirst` operator takes a `count` parameter — defaulting to `1` if omitted — and ignores the first `count` values emitted by the publisher. Only values emitted after `count` values have been emitted will be allowed through.




Add the following code to the end of your playground to try this operator:

```
example(of: "dropFirst") {
  // 1
  let numbers = (1...10).publisher

  // 2
  numbers
    .dropFirst(8)
    .sink(receiveValue: {
      print($0)
    })
    .store(in: &subscriptions)
}
```

As in the above diagram, you:

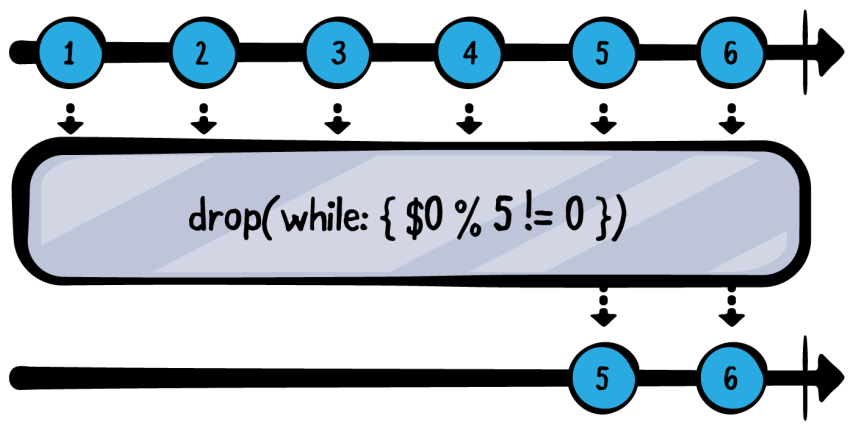
1. Create a publisher that emits 10 numbers between `1` and `10`.
2. Use `dropFirst(8)` to drop the first eight values, printing only `9` and `10`.

Build and run your playground and you should see the following output:

```
—— Example of: dropFirst ——
9
10
```

Simple, right? Often, the most useful operators are!

Moving on to the next operators in the value dropping family – `drop(while:)`. This is another extremely useful variation that takes a predicate closure and ignores any values emitted by the publisher until the *first time* that predicate is met. As soon as the predicate is met, values begin to flow through the operator:




Add the following example to your playground:

```
example(of: "drop(while:)") {
  // 1
  let numbers = (1...10).publisher

  // 2
  numbers
    .drop(while: { $0 % 5 != 0 })
    .sink(receiveValue: {
```

```
      print($0)
    })
    .store(in: &subscriptions)
}
```

In the following code, you:

1. Create a publisher that emits numbers between `1` and `10`.
2. Use `drop(while:)` to wait for the first value that is divisible by five. As soon as the condition is met, values will start flowing through the operator and won't be dropped anymore.

Build and run your playground and look at the debug console:

```
—— Example of: drop(while:) ——
5
6
7
8
9
10
```

Excellent! As you can see, you've dropped the first four values. As soon as `5` arrives, the question "is this divisible by five?" is finally `true`, so it now emits `5` and all future values.

You might ask yourself – how is this operator different from `filter`? Both of them take a closure that controls which values are emitted based on the result of that closure.

The first difference is that `filter` lets values through if you return `true` in the closure, while `drop(while:)` *skips* values as long you return `true` from the closure.

The second, and more important difference is that `filter` never stops evaluating its condition for all values published by the upstream publisher. Even after the condition of `filter` evaluates to `true`, further values are still "questioned" and your closure must answer the question: "Do you want to let this value through?".

On the contrary, `drop(while:)`'s predicate closure will *never* be executed again after the condition is met. To confirm this, replace the following line:

```
.drop(while: { $0 % 5 != 0 })
```

With this piece of code:

```
.drop(while: {
```

```
    print("x")
    return $0 % 5 != 0
  })
```

You added a `print` statement to print `x` to the debug console every time the closure is invoked. Build and run the playground and you should see the following output:

```
——— Example of: drop(while:) ———
x
x
x
x
x
5
6
7
8
9
10
```

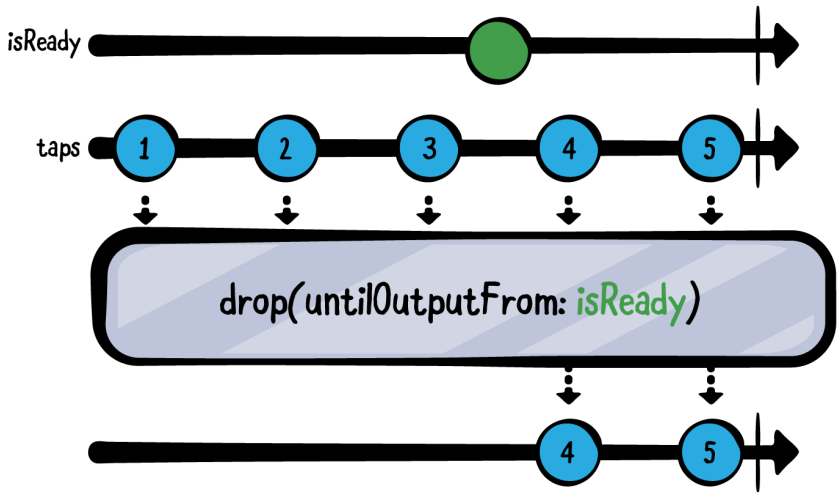
As you might have noticed, `x` prints exactly five times. As soon as the condition is met (when `5` is emitted), the closure is never evaluated again.

Alrighty then. Two down, one more to go.

The final and most elaborate operator of the filtering category is `drop(untilOutputFrom:)`.

Imagine a scenario where you have a user tapping a button, but you want to ignore all taps until your `isReady` publisher emits some result. This operator is perfect for this sort of condition.

It skips any values emitted by a publisher until a second publisher starts emitting values, creating a relationship between them:




The top line represents the `isReady` stream and the second line represents taps by the user passing through `drop(untilOutputFrom:)`, which takes `isReady` as an argument.

At the end of your playground, add the following code that reproduces this diagram:

```
example(of: "drop(untilOutputFrom:)") {
  // 1
  let isReady = PassthroughSubject<Void, Never>()
  let taps = PassthroughSubject<Int, Never>()

  // 2
  taps
    .drop(untilOutputFrom: isReady)
    .sink(receiveValue: {
      print($0)
    })
    .store(in: &subscriptions)

  // 3
  (1...5).forEach { n in
    taps.send(n)

    if n == 3 {
      isReady.send()
    }
  }
}
```

In this code, you:

1. Create two `PassthroughSubjects` that you can manually send values to. The first is `isReady` while the second represents taps by the user.
2. Use `drop(untilOutputFrom: isReady)` to ignore any taps from the user until `isReady` emits at least one value.
3. Send five "taps" to the subject, just like in the diagram above. After the third tap, you send `isReady` a value.

Build and run your playground, then take a look at your debug console. You will see the following output:

```
—— Example of: drop(untilOutputFrom:) ——
4
5
```

This output is the same as the diagram above:

- There are five taps from the user. The first three are ignored.
- After the third tap, `isReady` emits a value.
- All future taps by the user are passed through.

You've gained quite a mastery of getting rid of unwanted values! Now, it's time for the completing operator group: Limiting values.

# Limiting values

In the previous section, you've learned how to drop — or skip — values until a certain condition is met. That condition could be either a static number, a predicate closure, or a value that a different publisher emits.

This section tackles the opposite need: receiving values up to some condition is met, and then forcing the publisher to complete. For example, consider a request that may emit an unknown amount of values, but you only want a single emission and don't care about the rest of them.

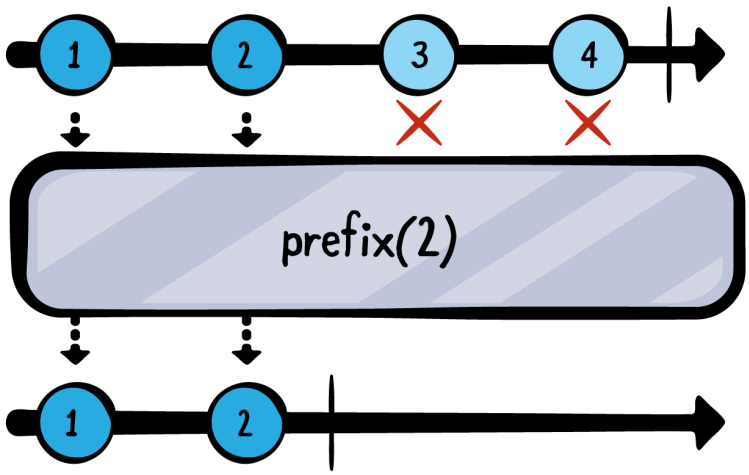
Combine solves this set of problems with the `prefix` family of operators. Even though the name isn't entirely intuitive, the abilities these operators provide are useful for many real-life situations.

The `prefix` family of operators is similar to the `drop` family: `prefix(_:)`, `prefix(while:)` and `prefix(untilOutputFrom:)`. However, instead of dropping

values until some condition is met, the `prefix` operators *take* values until that condition is met.

Now, it's time for you to dive into the final set of operators for this chapter, starting with `prefix(_:)`.

As the opposite of `dropFirst`, `prefix(_:)` will take values **only** up to the provided amount and then complete:




Add the following code to your playground to demonstrate this:

```
example(of: "prefix") {
  // 1
  let numbers = (1...10).publisher

  // 2
  numbers
    .prefix(2)
    .sink(receiveCompletion: { print("Completed with: \($0)") },
          receiveValue: { print($0) })
    .store(in: &subscriptions)
}
```

This code is quite similar to the `drop` code you used in the previous section. You:

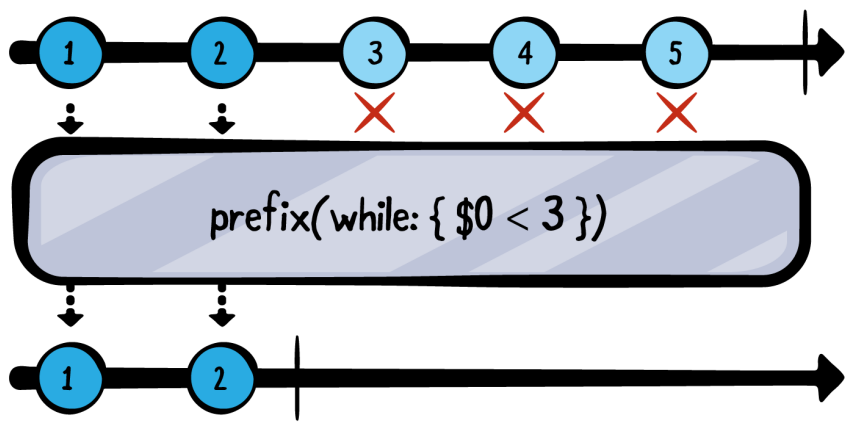
1. Create a publisher that emits numbers from `1` through `10`.
2. Use `prefix(2)` to allow the emission of only the first two values. As soon as two values are emitted, the publisher completes.

Build and run your playground and you'll see the following output:

```
—— Example of: prefix ——
1
2
Completed with: finished
```

Just like `first(where:)`, this operator is *lazy*, meaning it only takes up as many values as it needs and then terminates. This also prevents `numbers` from producing additional values beyond 1 and 2, since it also completes.

Next up is `prefix(while:)`, which takes a predicate closure and lets values from the upstream publisher through as long as the result of that closure is `true`. As soon as the result is `false`, the publisher will complete:




Add the following example to your playground:

```
example(of: "prefix(while:)") {
  // 1
  let numbers = (1...10).publisher

  // 2
  numbers
    .prefix(while: { $0 < 3 })
    .sink(receiveCompletion: { print("Completed with: \($0)") },
          receiveValue: { print($0) })
    .store(in: &subscriptions)
}
```

This example is mostly identical to the previous one, aside from using a closure to evaluate the prefixing condition. You:

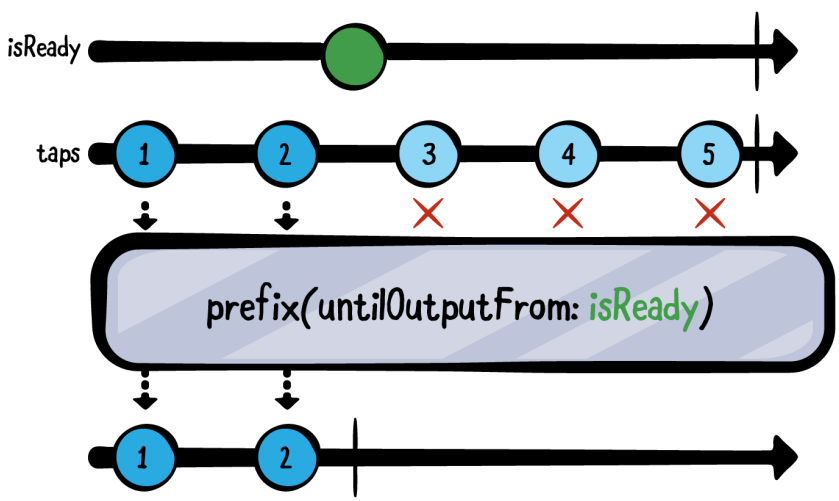
1. Create a publisher that emits values between `1` and `10`.
2. Use `prefix(while:)` to let values through as long as they're smaller than `3`. As soon as a value equal to or larger than `3` is emitted, the publisher completes.

Build and run the playground and check out the debug console; the output should be identical to the one from the previous operator:

```
—— Example of: prefix(while:) ——
1
2
Completed with: finished
```

With the first two `prefix` operators behind us, it's time for the most complex one: `prefix(untilOutputFrom:)`. Once again, as opposed to `drop(untilOutputFrom:)` which *skips* values until a second publisher emits, `prefix(untilOutputFrom:)` *takes* values until a second publisher emits.

Imagine a scenario where you have a button that the user can only tap twice. As soon as two taps occur, further tap events on the button should be omitted:




Add the final example for this chapter to the end of your playground:

```
example(of: "prefix(untilOutputFrom:)") {
  // 1
```

```
    let isReady = PassthroughSubject<Void, Never>()
    let taps = PassthroughSubject<Int, Never>()

    // 2
    taps
      .prefix(untilOutputFrom: isReady)
      .sink(receiveCompletion: { print("Completed with: \($0)") },
            receiveValue: { print($0) })
      .store(in: &subscriptions)

    // 3
    (1...5).forEach { n in
      taps.send(n)

      if n == 2 {
        isReady.send()
      }
    }
  }
```

If you think back to the `drop(untilOutputFrom:)` example, you should find this easy to understand. You:

1. Create two `PassthroughSubjects` that you can manually send values to. The first is `isReady` while the second represents taps by the user.
2. Use `prefix(untilOutputFrom: isReady)` to let tap events through until `isReady` emits at least one value.
3. Send five "taps" to the subject, exactly as in the diagram above. After the second tap, you send `isReady` a value.

Looking at the console, you should see the following:

```
—— Example of: prefix(untilOutputFrom:) ——
1
2
Completed with: finished
```

# Challenge

You have quite a lot of filtering knowledge at your disposal now. Why not try a short challenge?

## Challenge: Filter all the things

Create an example that publishes a collection of numbers from `1` through `100`, and use filtering operators to:

1. Skip the first 50 values emitted by the upstream publisher.
2. Take the next 20 values after those first 50 values.
3. Only take even numbers.

The output of your example should produce the following numbers, one per line:

```
52 54 56 58 60 62 64 66 68 70
```

> **Note**: In this challenge, you'll need to chain multiple operators together to produce the desired values.

You can find the full solution to this challenge in **projects/challenge/Final.playground**.

# Key points

In this chapter, you learned that:

- Filtering operators let you control which values emitted by the upstream publisher are sent downstream, to another operator or to the subscriber.
- When you don't care about the values themselves, and only want a completion event, `ignoreOutput` is your friend.
- Finding values is another sort of filtering, where you can find the first or last values to match a provided predicate using `first(where:)` and `last(where:)`, respectively.
- First-style operators are *lazy*; they take only as many values as needed and then

complete. Last-style operators are *greedy* and must know the full scope of the values before deciding which of the values is the last to fulfill the condition.

- You can control how many values emitted by the upstream publisher are ignored before sending values downstream by using the `drop` family of operators
- Similarly, you can control how many values the upstream publisher may emit before completing by using the `prefix` family of operators.

## Where to go from here?

Wow, what a ride this chapter has been! You should rightfully feel like a master of filtering, ready to channel these upstream values in any way you desire.

With the knowledge of transforming and filtering operators already in your tool belt, it's time for you to move to the next chapter and learn another extremely useful group of operators: Combining operators.

# Chapter 5: Combining Operators

By Shai Mishali

Now that the **transforming** and **filtering** operator categories are in your tool belt, you have a substantial amount of knowledge. You've learned how operators work, how they manipulate the upstream and how to use them to construct logical publisher chains from your data.

In this chapter, you'll learn about one of the more complex, yet useful, category of operators: **Combining operators**. This set of operators lets you combine events emitted by different publishers and create meaningful combinations of data in your Combine code.

Why is combining useful? Think about a form with multiple inputs from the user — a username, a password and a checkbox. You'll need to *combine* this data to compose meaningful relationships.

As you learn more about how each operator functions and how to select the right one for your needs, your code will become more coherent.

## Getting started

You can find the starter playground for this chapter, **Combine.playground**, in the **projects** folder. Throughout this chapter, you'll add code to your playground and run it to see how various operators create different combinations of publishers and their events.

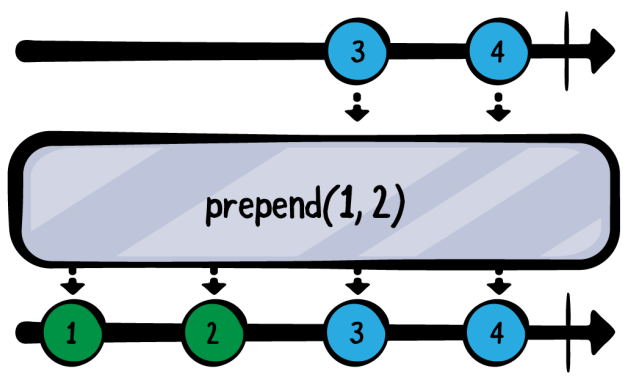
# Prepending

You'll start slowly here with a group of operators that are all about prepending elements at the *beginning* of your publisher. In other words, you'll use them to add elements that emit *before* any elements from your original publisher.

In this section, you'll learn about `prepend(Output...)`, `prepend(Sequence)` and `prepend(Publisher)`.

## prepend(Output...)

This variation of `prepend` takes a *variadic* list of elements using the variadic `...` syntax. This means it can take **any number of elements**, as long as they're of the same `Output` type as the original publisher.




Add the following code to your playground to experiment with the above example:

```
example(of: "prepend(Output...)") {
  // 1
  let publisher = [3, 4].publisher

  // 2
  publisher
    .prepend(1, 2)
    .sink(receiveValue: { print($0) })
    .store(in: &subscriptions)
}
```

In the previous code, you:

1. Create a publisher that emits the numbers 3 and 4.
2. Use `prepend` to add the numbers 1 and 2 before the publisher's own elements.

Run your playground. You should see the following in your debug console:

```
—— Example of: prepend(Output...) ——
1
2
3
4
```

Pretty straightforward!

Hang on, do you remember how operators are chainable? That means you can easily add more than a single `prepend`, if you'd like.

Below the following line:

```
.prepend(1, 2)
```

Add the following:

```
.prepend(-1, 0)
```

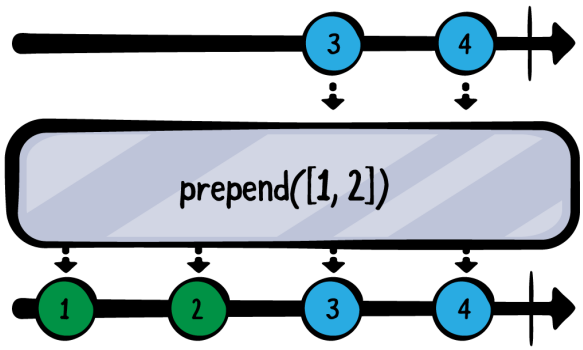
Run your playground again. you should see the following output:

```
—— Example of: prepend(Output...) ——
-1
0
1
2
3
4
```

Notice that the order of operations is crucial here. The last prepend affects the upstream first, meaning –1 and `0` prepend, then 1 and 2, and finally the original publisher's elements.

## prepend(Sequence)

This variation of prepend is similar to the previous one, with the difference that it takes any `Sequence`-conforming object as an input. For example, it could take an `Array` or a `Set`.




Add the following code to your playground to experiment with this operator:

```
example(of: "prepend(Sequence)") {
  // 1
  let publisher = [5, 6, 7].publisher

  // 2
  publisher
    .prepend([3, 4])
    .prepend(Set(1...2))
    .sink(receiveValue: { print($0) })
    .store(in: &subscriptions)
}
```

In the previous code, you:

1. Create a publisher that emits numbers between 5 and 7.
2. Chain `prepend(Sequence)` twice to the original publisher. Once to prepend elements from an `Array` and a second time to prepend elements from a `Set`.

Run the playground. Your output should be similar to the following:

```
—— Example of: prepend(Sequence) ——
1
2
3
4
5
6
7
```

> **Note**: An important fact to remember about `Sets`, as opposed to `Arrays`, is that they are **unordered**, so the order in which the items emit is not guaranteed. This means the first two elements in the above example could be either `1` and `2`, or `2` and `1`.

But wait, there's more! Many types conform to `Sequence` in Swift, which lets you do some interesting things.

After the second prepend:

```
.prepend(Set(1...2))
```

Add the following line:

```
.prepend(stride(from: 6, to: 11, by: 2))
```

In this line of code, you create a `Strideable` which lets you **stride** between `6` and `11` in steps of `2`. Since `Strideable` conforms to `Sequence`, you can use it in `prepend(Sequence)`.

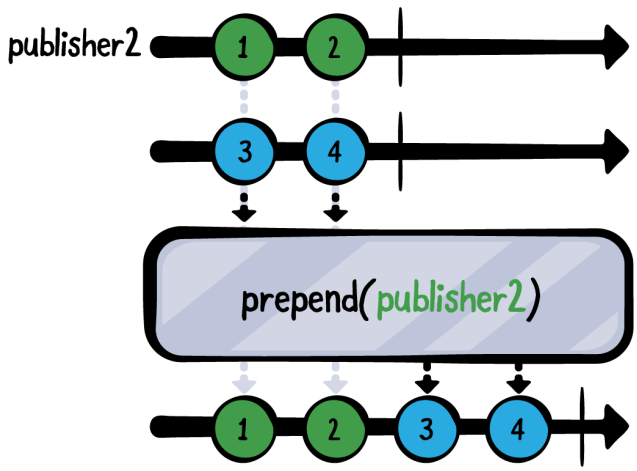
Run your playground one more time and take a look at the debug console:

```
—— Example of: prepend(Sequence) ——
6
8
10
1
2
3
4
5
6
7
```

As you can see, three new elements are now prepended to the publisher before the previous output – `6`, `8` and `10`, the result of striding between `6` and `11` in steps of `2`.

## prepend(Publisher)

The two previous operators prepended lists of elements to an existing publisher. But what if you have two different publishers and you want to glue their elements together? You can use `prepend(Publisher)` to add elements emitted by a second publisher *before* the original publisher's elements.




Try out the above example by adding the following to your playground:

```
example(of: "prepend(Publisher)") {
  // 1
  let publisher1 = [3, 4].publisher
  let publisher2 = [1, 2].publisher

  // 2
  publisher1
    .prepend(publisher2)
    .sink(receiveValue: { print($0) })
    .store(in: &subscriptions)
}
```

In this code, you:

1. Create two publishers. One emitting the numbers 3 and 4, and a second one emitting 1 and 2.
2. Prepend `publisher2` at the beginning of `publisher1`. Elements from `publisher1` emit *only* after elements from `publisher2`.

If you run your playground, your debug console should present the following output:

```
—— Example of: prepend(Publisher) ——
1
2
3
4
```

As expected, elements 1 and 2 are first emitted from `publisher2`; only then are 3 and 4 emitted by `publisher1`.

There's one more detail about this operator that you should be aware of, and it would be easiest to show it with an example.

Add the following to the end of your playground:

```
example(of: "prepend(Publisher) #2") {
  // 1
  let publisher1 = [3, 4].publisher
  let publisher2 = PassthroughSubject<Int, Never>()

  // 2
  publisher1
    .prepend(publisher2)
    .sink(receiveValue: { print($0) })
    .store(in: &subscriptions)

  // 3
  publisher2.send(1)
  publisher2.send(2)
}
```

This example is similar to the previous one, except that `publisher2` is now a `PassthroughSubject` that you can push values to manually.

In the following example, you:

1. Create two publishers. The first emits elements 3, and 4 while the second is a `PassthroughSubject` that can accept values dynamically.
2. Prepend the subject at the beginning of `publisher1`.
3. Send the values 1 and 2 to the subject `publisher2`.

Take a second and run this code inside your head. What do you expect the output to be?

Now, run the playground again and take a look at the debug console. You should see the following:

```
—— Example of: prepend(Publisher) #2 ——
1
2
```

Wait, what? Why are there only two numbers emitted here from `publisher2`? You must be thinking… hey, there, Shai, didn't you just say elements prepend to the existing publisher?

Well, think about it — how can Combine know the prepended publisher, `publisher2`, has finished emitting values? It doesn't, because it has emitted values but no completion event. For that reason, a prepended publisher **must complete** so Combine knows it has finished prepending and can continue to the primary publisher.

After the following line:

```
publisher2.send(2)
```

Add this one:

```
publisher2.send(completion: .finished)
```

Combine now knows it can handle emissions from `publisher1` because `publisher2` has finished its work.

Run your playground again; you should see the expected output this time around:

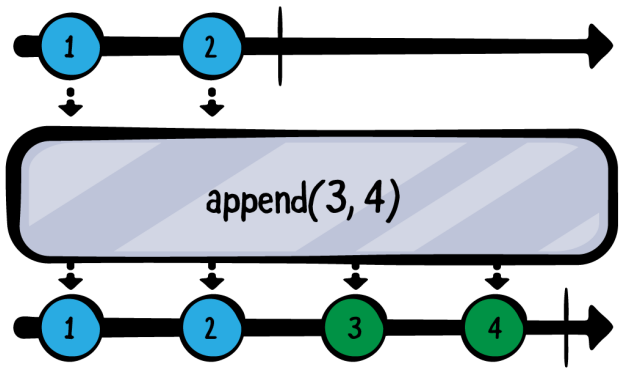
```
—— Example of: prepend(Publisher) #2 ——
1
2
3
4
```

# Appending

This next set of operators deals with concatenating events emitted by publishers with other elements. But in this case, you'll deal with *appending* instead of prepending, using `append(Output...)`, `append(Sequence)` and `append(Publisher)`. These operators work in a similar way as their `prepend` counterparts.

## append(Output...)

`append(Output...)` works similarly to its `prepend` counterpart: It also takes a variadic list of type `Output` but then *appends* its items after the *original* publisher has completed with a `.finished` event.




Add the following code to your playground to experiment with this operator:

```
example(of: "append(Output...)") {
  // 1
  let publisher = [1].publisher

  // 2
  publisher
    .append(2, 3)
    .append(4)
    .sink(receiveValue: { print($0) })
    .store(in: &subscriptions)
}
```

In the previous code, you:

1. Create a publisher emitting only a single element: `1`.
2. Use `append` twice, first to append `2` and `3` and then to append `4`.

Think about this code for a minute — what do you think the output will be?

Run the playground and check out the output:

```
—— Example of: append(Output...) ——
1
2
3
4
```

Appending works exactly like you'd expect, where each `append` waits for the upstream to complete before adding its own work to it.

This means that the upstream **must complete** or appending would never occur since Combine couldn't know the previous publisher has finished emitting all of its elements.

To verify this behavior, add the following example:

```
example(of: "append(Output...) #2") {
  // 1
  let publisher = PassthroughSubject<Int, Never>()

  publisher
    .append(3, 4)
    .append(5)
    .sink(receiveValue: { print($0) })
    .store(in: &subscriptions)

  // 2
  publisher.send(1)
  publisher.send(2)
}
```

This example is identical to the previous one, with two differences:

1. `publisher` is now a `PassthroughSubject`, which lets you manually send values to it.
2. You send `1` and `2` to the `PassthroughSubject`.

Run your playground again and you'll see that only the values sent to `publisher` are emitted:

```
—— Example of: append(Output...) #2 ——
1
2
```

Both `append` operators have no effect since they can't possibly work until `publisher` completes.

Add the following line at the very end of the example:

```
publisher.send(completion: .finished)
```

Run your playground again and you should see all elements, as expected:

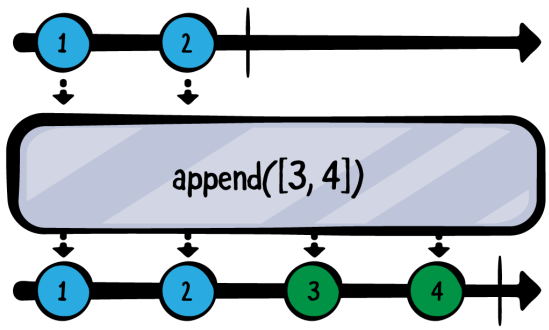
```
—— Example of: append(Output...) #2 ——
```

```
1
2
3
4
5
```

This behavior applies to the entire family of append operators; no appending occurs unless the original publisher sends a completion event.

## append(Sequence)

This variation of append takes any Sequence-conforming object and appends its elements after all elements from the original publisher have emitted.




Add the following to your playground to experiment with this operator:

```
example(of: "append(Sequence)") {
  // 1
  let publisher = [1, 2, 3].publisher

  publisher
    .append([4, 5]) // 2
    .append(Set([6, 7])) // 3
    .append(stride(from: 8, to: 11, by: 2)) // 4
    .sink(receiveValue: { print($0) })
    .store(in: &subscriptions)
}
```

This code is similar to the `prepend(Sequence)` example from the previous section. You:

1. Create a publisher that emits `1`, `2` and `3`.
2. Append an `Array` with the elements `4` and `5` (ordered).
3. Append a `Set` with the elements `6` and `7` (unordered).
4. Append a `Strideable` that strides between `8` and `11` by steps of `2`.

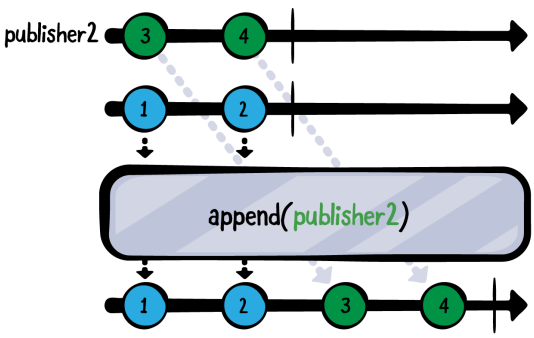
Run your playground and you should see the following output:

```
—— Example of: append(Sequence) ——
1
2
3
4
5
7
6
8
10
```

As you can see, the execution of the `append`s is sequential because the previous publisher must complete before the next `append` performs. Note that the set of `6` and `7` may be in a different order for you, as sets are unordered.

## append(Publisher)

The last member of the `append` operators group is the variation that takes a `Publisher` and appends any elements emitted by it to the end of the original publisher.



To try this example, add the following to your playground:

```
example(of: "append(Publisher)") {
  // 1
  let publisher1 = [1, 2].publisher
  let publisher2 = [3, 4].publisher

  // 2
  publisher1
    .append(publisher2)
    .sink(receiveValue: { print($0) })
    .store(in: &subscriptions)
}
```

In this code, you:

1. Create two publishers, where the first emits `1` and `2`, and the second emits `3` and `4`.
2. Append `publisher2` to `publisher1`, so all elements from `publisher2` are appended at the end of `publisher1` once it completes.

Run the playground and you should see the following output:

```
—— Example of: append(Publisher) ——
1
2
3
4
```

# Advanced combining

At this point, you know everything about appending and prepending elements, sequences and even entire publishers.

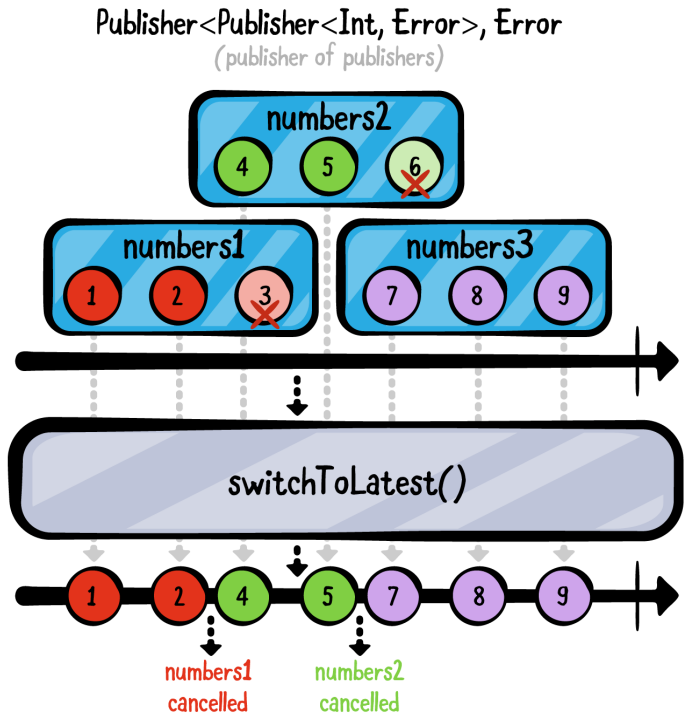
This next section will dive into some of the more complex operators related to combining different publishers. Even though they're relatively complex, they're also some of the most useful operators for publisher composition. It's worth taking the time to get comfortable with how they work.

## switchToLatest

Since this section includes some of the more complex combining operators in Combine, why not start with the most complex one of the bunch?!

Joking aside, `switchToLatest` is complex but highly useful. It lets you switch entire publisher chains on the fly while canceling the pending publisher, thus *switching* to the latest one.

You can only use it on publishers that *themselves* emit publishers.




Add the following code to your playground to experiment with the example you see the above diagram:

```
example(of: "switchToLatest") {
  // 1
  let publisher1 = PassthroughSubject<Int, Never>()
  let publisher2 = PassthroughSubject<Int, Never>()
  let publisher3 = PassthroughSubject<Int, Never>()

  // 2
  let publishers = PassthroughSubject<PassthroughSubject<Int,
Never>, Never>()

  // 3
  publishers
    .switchToLatest()
```

```
      .sink(receiveCompletion: { _ in print("Completed!") },
            receiveValue: { print($0) })
      .store(in: &subscriptions)

    // 4
    publishers.send(publisher1)
    publisher1.send(1)
    publisher1.send(2)

    // 5
    publishers.send(publisher2)
    publisher1.send(3)
    publisher2.send(4)
    publisher2.send(5)

    // 6
    publishers.send(publisher3)
    publisher2.send(6)
    publisher3.send(7)
    publisher3.send(8)
    publisher3.send(9)

    // 7
    publisher3.send(completion: .finished)
    publishers.send(completion: .finished)
  }
```

Yikes, that's a lot of code! But don't worry, it's simpler than it looks. Breaking it down, you:

1. Create three `PassthroughSubjects` that accept integers and no errors.
2. Create a second `PassthroughSubject` that accepts *other* `PassthroughSubjects`. For example, you can send `publisher1`, `publisher2` or `publisher3` to it.
3. Use `switchToLatest` on your `publishers`. Now, every time you send a different publisher to the `publishers` subject, you switch to the new one and cancel the previous subscription.
4. Send `publisher1` to `publishers` and then send `1` and `2` to `publisher1`.
5. Send `publisher2`, which cancels the subscription to `publisher1`. You then send `3` to `publisher1`, but it's ignored, and send `4` and `5` to `publisher2`, which are pushed through because `publisher2` is the current subscription.
6. Send `publisher3`, which cancels the subscription to `publisher2`. As before, you send `6` to `publisher2` and it's ignored, and then send `7`, `8` and `9`, which are pushed through the subscription.

7. Finally, you send a completion event to the current publisher, `publisher3`, and another completion event to `publishers`. This completes all active subscriptions.

If you followed the above diagram, you might have already guessed the output of this example.

Run the playground and look at the debug console:

```
—— Example of: switchToLatest ——
1
2
4
5
7
8
9
Completed!
```

If you're not sure why this is useful in a real-life app, consider the following scenario: Your user taps a button that triggers a network request. Immediately afterward, the user taps the button again, which triggers a second network request. But how do you get rid of the pending request, and only use the *latest* request? `switchToLatest` to the rescue!

Instead of just theorizing, why don't you try out this example?

Add the following code to your playground, and don't forget the `subscription` var above the `example`:

```
var subscription: AnyCancellable?

example(of: "switchToLatest - Network Request") {
  let url = URL(string: "https://source.unsplash.com/random")!

  // 1
  func getImage() -> AnyPublisher<UIImage?, Never> {
      return URLSession
        .shared
        .dataTaskPublisher(for: url)
        .map { data, _ in UIImage(data: data) }
        .print("image")
        .replaceError(with: nil)
        .eraseToAnyPublisher()
  }

  // 2
  let taps = PassthroughSubject<Void, Never>()

  subscription = taps
```

```
      .map { _ in getImage() } // 3
      .switchToLatest() // 4
      .sink(receiveValue: { _ in })

    // 5
    taps.send()

    DispatchQueue.main.asyncAfter(deadline: .now() + 3) {
      taps.send()
    }
    DispatchQueue.main.asyncAfter(deadline: .now() + 3.1) {
      taps.send()
    }
  }
```

As in the previous example, this might look like a long and complicated piece of code, but it's simple once you break it down.

In this code, you:

1. Define a function, `getImage()`, which performs a network request to fetch a random image from Unsplash's public API. This uses `URLSession.dataTaskPublisher`, one of the many Combine extensions for Foundation. You'll learn much more about this and others in Section 3, "Combine in Practice."
2. Create a `PassthroughSubject` to simulate user taps on a button.
3. Upon a button tap, map the tap to a new network request for a random image by calling `getImage()`. This essentially transforms `Publisher<Void, Never>` into `Publisher<Publisher<UIImage?, Never>, Never>` — a publisher of publishers.
4. Use `switchToLatest()` exactly like in the previous example, since you have a publisher of publishers. This guarantees only one publisher will emit elements, and cancel any previous subscriptions.
5. Simulate three delayed button taps using a `DispatchQueue`. The first tap is immediate, the second tap comes after three seconds, and the last tap comes just a tenth of a second after the second tap.

Run the playground and take a look at the output below:

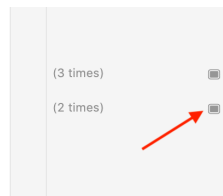
```
——— Example of: switchToLatest - Network Request ———
image: receive subscription: (DataTaskPublisher)
image: request unlimited
image: receive value: (Optional(<UIImage:0x600000364120
anonymous {1080, 720}>))
```

```
image: receive finished
image: receive subscription: (DataTaskPublisher)
image: request unlimited
image: receive cancel
image: receive subscription: (DataTaskPublisher)
image: request unlimited
image: receive value: (Optional(<UIImage:0x600000378d80
anonymous {1080, 1620}>))
image: receive finished
```

You might notice that only two images are actually fetched; that's because only a tenth of a second passes between the second and third taps. The third tap switches to a new request before the second fetch returns, canceling the second subscription – hence the line that says `image: receive cancel`.

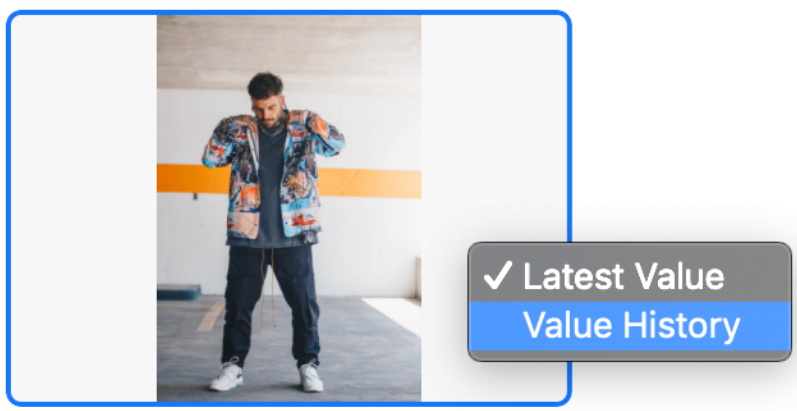
If you want to see a better visualization of this, tap the following button:

```
example(of: "switchToLatest - Network Request") {
  // 1
  func getImage() -> AnyPublisher<UIImage?, Never> {
      return URLSession.shared
                       .dataTaskPublisher(for: URL(string: "https://source.unsplash.com/random")!)
                       .map { data, _ in UIImage(data: data) }
                       .print("image")
                       .replaceError(with: nil)
                       .eraseToAnyPublisher()
  }
```



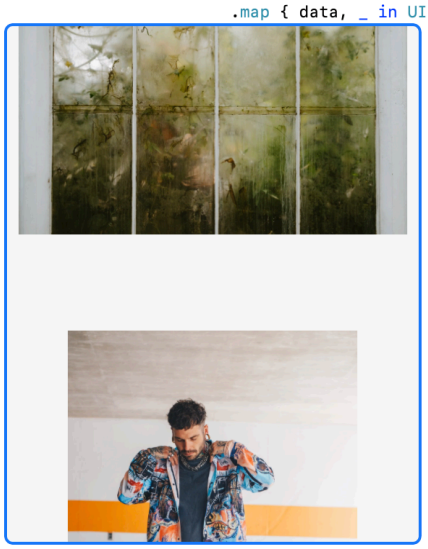

Then run the playground again and wait a few seconds. You should see the last image loaded.

Right-click the image and select **Value History**:



You should see both loaded images — you may have to scroll to see both of them:

```
example(of: "switchToLatest — Network Request") {
  // 1
  func getImage() -> AnyPublisher<UIImage?, Never> {
      return URLSession.shared
                       .dataTaskPublisher(for: URL(string: "https://source.unsplash.com/random")!)
                       .map { data, _ in UIImage(data: data) }
```



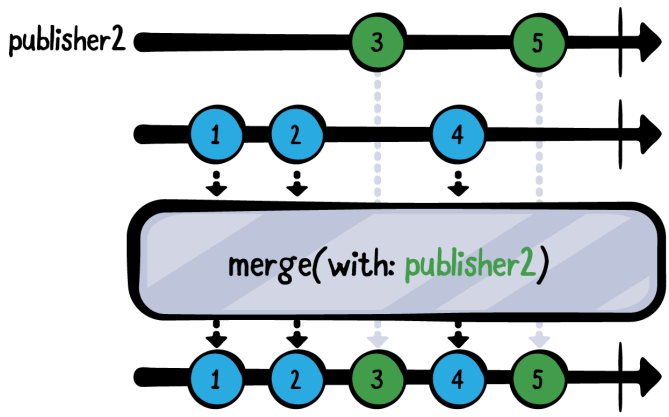
```
                       .print("image")
                       .replaceError(with: nil)
                       .eraseToAnyPublisher()
  }
```

Before moving to the next operator, be sure to comment out this entire example to avoid running the asynchronous network requests every time you run your playground.

## merge(with:)

Before you reach the end of this chapter, you'll wrap up with three operators that focus on combining the emissions of different publishers. You'll start with `merge(with:)`.

This operator **interleaves** emissions from different publishers of the same type, like so:




To try out this example, add the following code to your playground:

```
example(of: "merge(with:)") {
  // 1
  let publisher1 = PassthroughSubject<Int, Never>()
  let publisher2 = PassthroughSubject<Int, Never>()

  // 2
  publisher1
    .merge(with: publisher2)
    .sink(receiveCompletion: { _ in print("Completed") },
          receiveValue: { print($0) })
    .store(in: &subscriptions)

  // 3
  publisher1.send(1)
  publisher1.send(2)

  publisher2.send(3)

  publisher1.send(4)

  publisher2.send(5)

  // 4
  publisher1.send(completion: .finished)
  publisher2.send(completion: .finished)
}
```

In this code, which correlates with the above diagram, you:

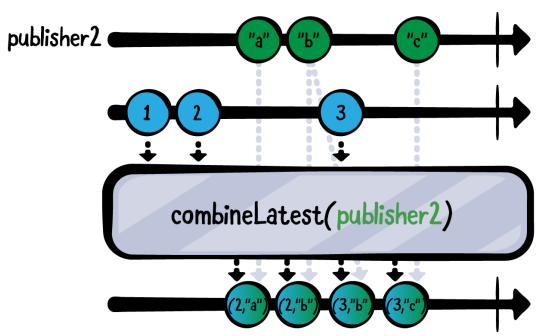
1. Create two `PassthroughSubjects` that accept and emit integer values and which will not emit an error.
2. Merge `publisher1` with `publisher2`, interleaving the emitted elements from both. Combine offers overloads that let you merge up to eight different publishers.
3. You add `1` and `2` to `publisher1`, then add `3` to `publisher2`, then add `4` to `publisher1` again and finally add `5` to `publisher2`.
4. You send a completion event to both `publisher1` and `publisher2`.

Run your playground and you should see the following output, as expected:

```
—— Example of: merge(with:) ——
1
2
3
4
5
Completed
```

## combineLatest

`combineLatest` is another operator that lets you combine different publishers. It also lets you combine publishers of different value types, which can be extremely useful. However, instead of interleaving the emissions of all publishers, it emits **a tuple** with the latest values of *all* publishers whenever *any* of them emit a value. One catch though: The origin publisher and every publisher passed to `combineLatest` must emit *at least* one value before `combineLatest` will emit anything.



Add the following code to your playground to try out this operator:

```
example(of: "combineLatest") {
  // 1
  let publisher1 = PassthroughSubject<Int, Never>()
  let publisher2 = PassthroughSubject<String, Never>()

  // 2
  publisher1
    .combineLatest(publisher2)
    .sink(receiveCompletion: { _ in print("Completed") },
          receiveValue: { print("P1: \($0), P2: \($1)") })
    .store(in: &subscriptions)

  // 3
  publisher1.send(1)
  publisher1.send(2)

  publisher2.send("a")
  publisher2.send("b")

  publisher1.send(3)

  publisher2.send("c")

  // 4
  publisher1.send(completion: .finished)
  publisher2.send(completion: .finished)
}
```

This code reproduces the above diagram. You:

1. Create two `PassthroughSubjects`. The first accepts integers with no errors, while the second accepts strings with no errors.
2. Combine the latest emissions of `publisher2` with `publisher1`. You may combine up to four different publishers using different overloads of `combineLatest`.
3. Send `1` and `2` to `publisher1`, then `"a"` and `"b"` to `publisher2`, then `3` to `publisher1` and finally `"c"` to `publisher2`.
4. Send a completion event to both `publisher1` and `publisher2`.

Run the playground and take a look at the output in your console:

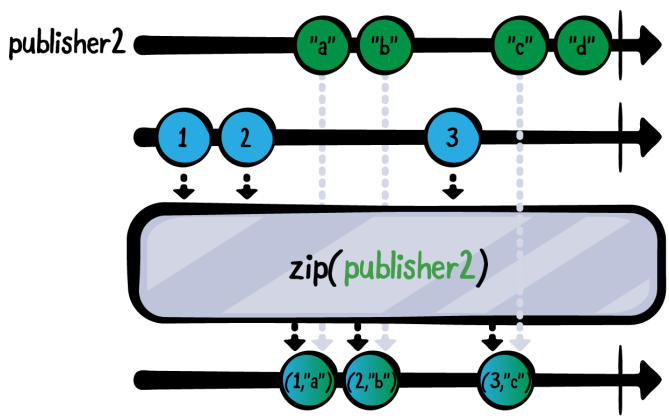
```
—— Example of: combineLatest ——
P1: 2, P2: a
P1: 2, P2: b
P1: 3, P2: b
P1: 3, P2: c
Completed
```

You might notice that the 1 emitted from `publisher1` is never pushed through `combineLatest`. That's because `combineLatest` only combines once every publisher emits *at least* one element. Here, that condition is true only after `"a"` emits, at which point the latest emitted value from `publisher1` is 2. That's why the first emission is `(2, "a")`.

## zip

You'll finish with one final operator for this chapter: `zip`. You might recognize this one from the Swift standard library method with the same name on `Sequence` types.

This operator works similarly, emitting tuples of paired values in the same indexes. It waits for each publisher to emit an item, then emits a single tuple of items after all publishers have emitted an element at the current index. This means that if you are zipping two publishers, you'll get a single tuple emitted every time both publishers emit a value.




Add the following code to your playground to try this example:

```
example(of: "zip") {
  // 1
  let publisher1 = PassthroughSubject<Int, Never>()
  let publisher2 = PassthroughSubject<String, Never>()

  // 2
  publisher1
      .zip(publisher2)
      .sink(receiveCompletion: { _ in print("Completed") },
            receiveValue: { print("P1: \($0), P2: \($1)") })
```

```
        .store(in: &subscriptions)

    // 3
    publisher1.send(1)
    publisher1.send(2)
    publisher2.send("a")
    publisher2.send("b")
    publisher1.send(3)
    publisher2.send("c")
    publisher2.send("d")

    // 4
    publisher1.send(completion: .finished)
    publisher2.send(completion: .finished)
}
```

In this final example, you:

1. Create two `PassthroughSubjects`, where the first accepts integers and the second accepts strings.
2. Zip `publisher1` with `publisher2`, pairing their emissions once they each emit a new value.
3. Send `1` and `2` to `publisher1`, then `"a"` and `"b"` to `publisher2`, then `3` to `publisher1` again, and finally `"c"` and `"d"` to `publisher2`.
4. Complete both `publisher1` and `publisher2`.

Run your playground a final time and take a look at the debug console:

```
—— Example of: zip ——
P1: 1, P2: a
P1: 2, P2: b
P1: 3, P2: c
Completed
```

Notice how each emitted element "waits" for the other zipped publisher to emit an element. `1` waits for the first emission from the second publisher, so you get `(1, "a")`. Likewise, `2` waits for the next emission from the second publisher, so you get `(2, "b")`. The last emitted element from the second publisher, `"d"`, is ignored since there is no corresponding emission from the first publisher to pair with.

# Key points

In this chapter, you learned how to take different publishers and create meaningful combinations with them. More specifically, you learned that:

- You can use the `prepend` and `append` families of operators to add emissions from one publisher before or after the original publisher.
- While `switchToLatest` is relatively complex, it's extremely useful. It takes a publisher that emits publishers, switches to the latest publisher and cancels the subscription to the previous publisher.
- `merge(with:)` lets you *interleave* elements from multiple publishers.
- `combineLatest` emits the latest elements of all combined publishers whenever *any* of them emit a value, once all of the combined publishers have emitted at least one value.
- `zip` pairs emissions from different publishers, emitting a tuple of pairs after all publishers have emitted an element.
- You can mix combination operators to create interesting and complex relationships between publishers and their emissions.

# Where to go from here?

This has been quite a long chapter, but it includes some of the most useful and involved operators Combine has to offer. Kudos to you for making it this far!

No challenges this time. Try to experiment with all of the operators you've learned thus far, there are plenty of use cases to play with.

You have two more groups of operators to learn about in the next two chapters: "Time Manipulation Operators" and "Sequence Operators," so move on to the next chapter!

# Chapter 6: Time Manipulation Operators

By Florent Pillet

Timing is everything. The core idea behind reactive programming is to model asynchronous event flow *over time*. In this respect, the Combine framework provides a range of operators that allow you to deal with time. In particular, how sequences react to and transform values over time.

As you'll see throughout this chapter, managing the time dimension of your sequence of values is easy and straightforward. It's one of the great benefits of using a framework like Combine.

## Getting started

To learn about time manipulation operators, you'll practice with an animated Xcode Playground that visualizes how data flows over time. This chapter comes with a starter playground you'll find in the **projects** folder.

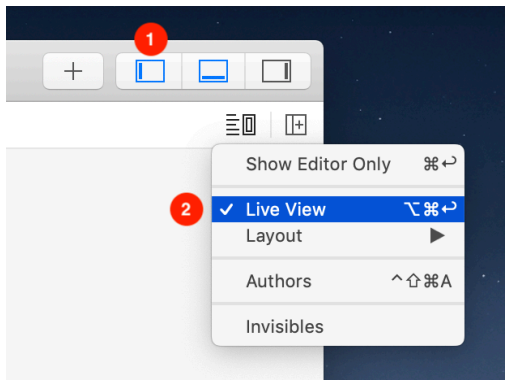
The playground is divided into several pages. You'll use each page to exercise one or more related operators. It also includes some ready-made classes, functions and sample data that'll come in handy to build the examples.

If you have the playground set to show rendered markup, at the bottom of each page there will be a **Next** link that you can click to go to the next page.

> **Note**: To toggle showing rendered markup on and off, select **Editor ▸ Show Rendered/Raw Markup** from the menu.

You can also select the page you want from the Project navigator in the left sidebar or even the jump bar at the top of the page. There are lots of ways to get around in Xcode!

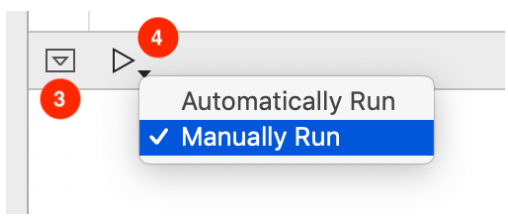
Look at Xcode, you can see controls at top-right of the window:




1. Make sure the left sidebar button is enabled so you can see the list of Playground pages.

2. Show the editor with **Live View**. This will display a live view of the sequences you build in code. This is where the real action will happen! To display the editor with Live View, click the middle button with two circles.

Also, remember that playgrounds can run manually or automatically. Showing the Debug area and configuring the playground for manual / automatic run is done using the controls at bottom left of the editor:

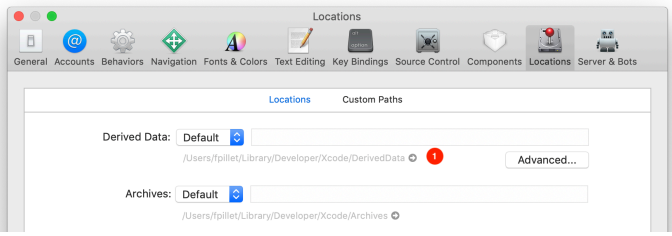



3. Click the vertical arrow at left to show/hide the Debug area. This is where the debug prints go.

4. Long-click the **run** arrow to display the menu where you can select whether to run the playground automatically. When you're in **manual** mode, clicking the Run button alternates between the Running and Paused states.

# Playground not working?

From time to time Xcode may "act up" and not run properly your playground. If this happens to you, open the **Preferences** dialog in Xcode and select the **Locations** tab. Click the arrow next to the Derived Data location, depicted by the red circled **1** in the screenshot below. It shows the `DerivedData` folder in the Finder.




Quit Xcode, move the `DerivedData` folder to trash then launch Xcode again. Your playground should now work properly!

# Shifting time

Every now and again you need time traveling. While Combine can't help with fixing your past relationship mistakes, it can freeze time for a little while to let you wait until self-cloning is available.

The most basic time manipulation operator delays values emitted by a publisher so that you see them later than they actually occur.

The `delay(for:tolerance:scheduler:options)` operator time-shifts a whole sequence of values: Every time the upstream publisher emits a value, `delay` keeps it for a while then emits it after the delay you asked for, on the Scheduler you specified.

Open the **Delay** playground page to get started. The first thing you'll see is that you're not only importing the **Combine** framework but also **SwiftUI**! This animated playground is built with SwiftUI and Combine. When you feel in an adventurous mood, it'll be a good idea to peruse through the code in the **Sources** folder.

But first things first. Start by defining a couple of constants you'll be able to tweak later:

```
let valuesPerSecond = 1.0
let delayInSeconds = 1.5
```

You're going to create a publisher that emits one value every second, then delay it by

1.5 seconds and display both timelines simultaneously to compare them. Once you complete the code on this page, you'll be able to adjust the constants and watch results in the timelines.

Next, create the publishers you need:

```
// 1
let sourcePublisher = PassthroughSubject<Date, Never>()

// 2
let delayedPublisher =
sourcePublisher.delay(for: .seconds(delayInSeconds), scheduler:
DispatchQueue.main)

// 3
let subscription = Timer
  .publish(every: 1.0 / valuesPerSecond, on: .main, in: .common)
  .autoconnect()
  .subscribe(sourcePublisher)
```

Breaking this code down:

1. `sourcePublisher` is a simple `Subject` which you'll feed dates emitted by a `Timer`. The type of values is of little importance here. You only care about imaging **when** a value is emitted by a publisher, and when the delayed value shows up.
2. `delayedPublisher` will delay values emitted by `sourcePublisher` and emit them on the main scheduler. You'll learn all about schedulers in Chapter 17, "Combine Schedulers." For now, specify that values must end up on the main queue, ready for display to consume them.
3. Create a timer that delivers one value per second on the main thread. Start it immediately with `autoconnect()` and feed the values it emits to the `sourcePublisher` subject.

> **Note**: This particular timer is a Combine extension on the Foundation `Timer` class. It takes a `RunLoop` and `RunLoop.Mode`, and not a `DispatchQueue` as you may expect. You'll learn all about timers in Chapter 11, "Combine Timers." Also, timers are part of a class of publishers that are **connectable**. This means they need to be connected to before they start emitting values. You use `autoconnect()` which immediately connects upon the first subscription.

You're getting to the part where you create the two views that will let you visualize

events. Add this code to your playground:

```
// 4
let sourceTimeline = TimelineView(title: "Emitted values (\
(valuesPerSecond) per sec.):")

// 5
let delayedTimeline = TimelineView(title: "Delayed values (with
a \(delayInSeconds)s delay):")

// 6
let view = VStack(spacing: 50) {
  sourceTimeline
  delayedTimeline
}

// 7
PlaygroundPage.current.liveView = UIHostingController(rootView:
view)
```

In this code, you:

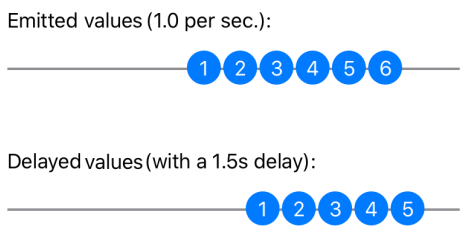
4. Create a `TimelineView` that will display values emitted by the timer. `TimelineView` is a SwiftUI view, its code can be found at **Sources/Views.swift**.
5. Create another `TimelineView` to display delayed values.
6. Create a simple SwiftUI vertical stack to display both timelines one above the other.
7. Set up the live view for this playground page.

At this stage, you see two empty timelines on the screen. You now need to feed them with the values emitted by each publisher! Add this final code to the playground:

```
sourcePublisher.displayEvents(in: sourceTimeline)
delayedPublisher.displayEvents(in: delayedTimeline)
```

In this last piece of code, you connect the source and delayed publishers to their respective timelines to display events.

Once you save these source changes, Xcode will recompile the playground code and... look at the Live View pane! Finally!




You'll see two timelines. The top timeline shows values emitted by the timer. The bottom timeline shows the same values, delayed. The numbers inside the circles reflect the count of emitted values, not their actual value.

> **Note**: As exciting as it is to see a live observable diagram, it might confuse at first. Static timelines usually have their values aligned to the left. But, if you think twice about it, they also have the most recent ones on the right side just as the animated diagrams you observe right now.

# Collecting values

In certain situations, you may need to collect values emitted by a publisher at specified intervals. This is a form of buffering that can be useful. For example, when you want to average a group of values over short periods of time and output the average.

Switch to the **Collect** page by clicking the **Next** link at the bottom, or by selecting it in the Project navigator or jump bar.

As in the previous example, you'll begin with some constants:

```
let valuesPerSecond = 1.0
let collectTimeStride = 4
```

Of course, reading these constants gives you an idea of where this is all going. Create your publishers now:

```
// 1
let sourcePublisher = PassthroughSubject<Date, Never>()

// 2
let collectedPublisher = sourcePublisher
  .collect(.byTime(DispatchQueue.main, .seconds(collectTimeStrid
e)))
```

Like in the previous example, you:

1. Set up a source publisher — a subject that emits values published by a timer.
2. Create a `collectedPublisher` which collects values emitted during strides of `collectTimeStride` using the `collect` operator. The operator emits these groups of values as arrays on the specified scheduler: `DispatchQueue.main`.

> **Note**: You might remember learning about the `collect` operator in Chapter 3: "Transforming Operators." where you used a simple number to define how to group values together. The overload of `collect` you just used accepts a strategy for grouping values; in this case, by time.

You'll use a `Timer` again to emit values at regular intervals as you did for the `delay` operator:

```
let subscription = Timer
  .publish(every: 1.0 / valuesPerSecond, on: .main, in: .common)
  .autoconnect()
  .subscribe(sourcePublisher)
```

Next, create the timeline views like in the previous example. Then, set the playground's live view to a vertical stack showing the source timeline and the timeline of collected values:

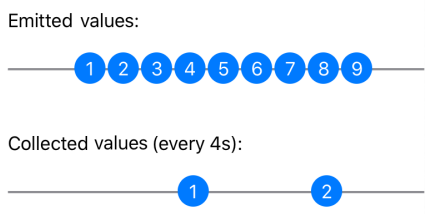
```
let sourceTimeline = TimelineView(title: "Emitted values:")
let collectedTimeline = TimelineView(title: "Collected values
(every \(collectTimeStride)s):")

let view = VStack(spacing: 40) {
  sourceTimeline
  collectedTimeline
}
```

```
PlaygroundPage.current.liveView = UIHostingController(rootView:
view)
```

Finally, feed the timelines with events from both publishers:

```
sourcePublisher.displayEvents(in: sourceTimeline)
collectedPublisher.displayEvents(in: collectedTimeline)
```

You're done! Now look at the live view for a while:



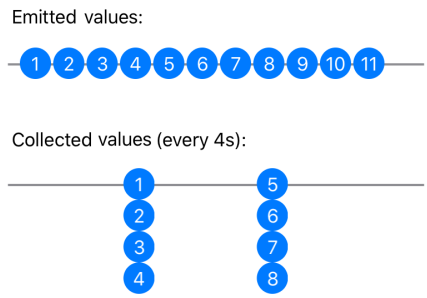

You see values emitted at regular intervals on the **Emitted values** timeline. Below it, you see that every four seconds the **Collected values** timeline displays a single value. But what is it?

You may have guessed that the value is an array of values received during the last four seconds. You can improve the display to see what's actually in it! Go back to the line where you created the `collectedPublisher` object. Add the use of the `flatMap` operator just below it, so it looks like this:

```
let collectedPublisher = sourcePublisher
  .collect(.byTime(DispatchQueue.main, .seconds(collectTimeStrid
e)))
  .flatMap { dates in dates.publisher }
```

Do you remember your friendly `flatMap` you learned about in Chapter 3? You're putting it to good use here: Every time `collect` emits a group of values it collected, `flatMap` breaks it down again to individual values but emitted **all at the same time**. To this end, it uses the `publisher` extension of `Collection` that turns a sequence of values into a Publisher, emitting immediately all values in the sequence as individual values.

Now, look at the effect it has on the timeline:




You can now see that, every four seconds, `collect` emits an array of values collected during the last time interval.

# Collecting values (part 2)

The second option offered by the `collect(_:options:)` operator allows you to keep collecting values at regular intervals. It also allows you to limit the number of collected values.

Staying on the same **Collect** page, and add a new constant right below `collectTimeStride` at the top:

```
let collectMaxCount = 2
```

Next, create a new publisher after `collectedPublisher`:

```
let collectedPublisher2 = sourcePublisher
  .collect(.byTimeOrCount(DispatchQueue.main,
                          .seconds(collectTimeStride),
                          collectMaxCount))
  .flatMap { dates in dates.publisher }
```

This time, you are using the `.byTimeOrCount(Context, Context.SchedulerTimeType.Stride, Int)` variant to collect up to `collectMaxCount` values at a time. What does this mean? Keep adding code and you'll find out!

Add a new `TimelineView` for the second collect publisher in between `collectedTimeline` and `let view = VStack...`:

```
let collectedTimeline2 = TimelineView(title: "Collected values
```

```
  (at most \(collectMaxCount) every \(collectTimeStride)s):")
```

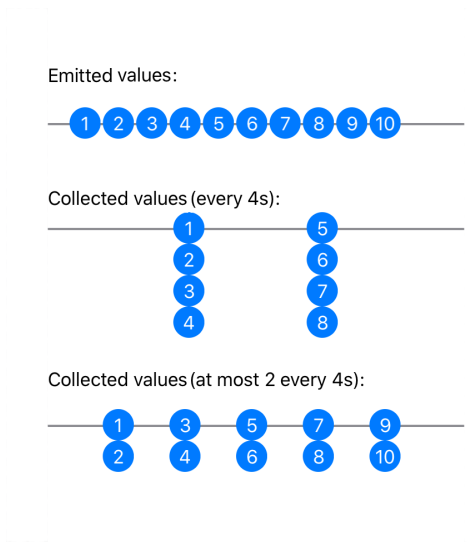
And of course add it to the list of stacked views, so `view` looks like this:

```
let view = VStack(spacing: 40) {
  sourceTimeline
  collectedTimeline
  collectedTimeline2
}
```

Finally, make sure it displays the events it emits in the timeline by adding the following at the end of your playground:

```
collectedPublisher2.displayEvents(in: collectedTimeline2)
```

Now, let this timeline run for a while so you can witness the difference:




You can see here that the second timeline is limiting its collection to two values at a time, as required by the `collectMaxCount` constant. It's a useful tool to know about!

# Holding off on events

When coding user interfaces, you frequently deal with text fields. Wiring up text field contents to an action using Combine is a common task. For example, you may want to send a search URL request that returns a list of items matching what's typed in the text field.

But of course, you don't want to send a request every time your user types a single letter! You need some kind of mechanism to help pick up on typed text only when the user is done typing for a while.

Combine offers two operators that can help you here: `debounce` and `throttle`. Let's explore them!

## Debounce

Switch to the playground page named **Debounce**. Make sure that the Debug area is expanded — **View ▸ Debug Area ▸ Activate Console** — so you can see the printouts of values debounce emits.

Start by creating a couple of publishers:

```
// 1
let subject = PassthroughSubject<String, Never>()

// 2
let debounced = subject
  .debounce(for: .seconds(1.0), scheduler: DispatchQueue.main)
  // 3
  .share()
```

In this code, you:

1. Create a source publisher which will emit strings.
2. Use `debounce` to wait for one second on emissions from `subject`. Then, it will send the last value sent in that one-second interval, if any. This has the effect of allowing a max of one value per second to be sent.
3. You are going to subscribe multiple times to `debounced`. To guarantee consistency of the results, you use `share()` to create a single subscription point to `debounce` that will show the same results at the same time to all subscribers.

> **Note:** Diving into the `share()` operator is out of the scope of this chapter. Just remember that it is helpful when a single subscription to a publisher is required to deliver the same results to multiple subscribers. You'll learn more about `share()` in Chapter 13, "Resources in Combine."

For these next few examples, you will use a set of data to simulate a user typing text in a text field. Don't type this in — it's already been implemented in **Sources/**

**Data.swift** for you:

```
public let typingHelloWorld: [(TimeInterval, String)] = [
  (0.0, "H"),
  (0.1, "He"),
  (0.2, "Hel"),
  (0.3, "Hell"),
  (0.5, "Hello"),
  (0.6, "Hello "),
  (2.0, "Hello W"),
  (2.1, "Hello Wo"),
  (2.2, "Hello Wor"),
  (2.4, "Hello Worl"),
  (2.5, "Hello World")
]
```

The simulated user starts typing at `0.0` seconds, pauses after `0.6` seconds, and resumes typing at `2.0` seconds.

In the playground's **Debounce** page, create a couple of timelines to visualize events, and wire them up to the two publishers:

```
let subjectTimeline = TimelineView(title: "Emitted values")
let debouncedTimeline = TimelineView(title: "Debounced values")

let view = VStack(spacing: 100) {
    subjectTimeline
    debouncedTimeline
}

PlaygroundPage.current.liveView = UIHostingController(rootView:
view)

subject.displayEvents(in: subjectTimeline)
debounced.displayEvents(in: debouncedTimeline)
```

You are now familiar with this playground structure where you stack timelines on the screen and connect them to the publishers for event display.

This time, you're going to do something more: Print values emitted by each publisher, along with the time (since start) at which they show up. This will help you figure out what's happening.

Add this code:

```
let subscription1 = subject
  .sink { string in
    print("+\(deltaTime)s: Subject emitted: \(string)")
  }
```

```
let subscription2 = debounced
  .sink { string in
    print("+\(deltaTime)s: Debounced emitted: \(string)")
  }
```

Each subscription prints the values it receives, along with the time since start. `deltaTime` is a dynamic global variable defined in **Sources/DeltaTime.swift** which formats the time difference since the playground started running.

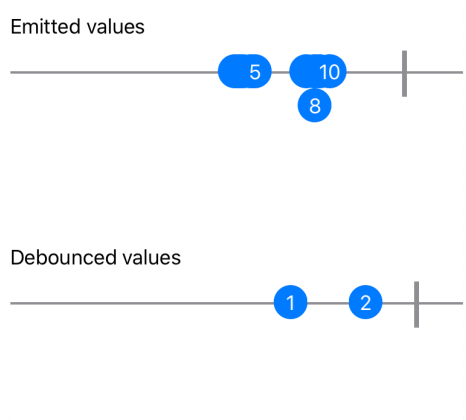
Now you need to feed your subject with data. This time you're going to use a pre-made data source that simulates a user typing text. It's all defined in **Sources/Data.swift** and you can modify it at will. Take a look, you'll see that it's a simulation of a user typing the words "Hello World".

Add this code to the end of the playground page:

```
subject.feed(with: typingHelloWorld)
```

The `feed(with:)` method takes a data set and sends data to the given `subject` at pre-defined time intervals. A handy tool for simulations and mocking data input! You may want to keep this around when you write tests for your code because you **will** write tests, won't you?

Now look at the result:




You see the emitted values at the top, there are 11 strings total being pushed to the `sourcePublisher`. You can see that the user *paused* between the two words. This is the time where `debounce` emitted the captured input.

You can confirm this by looking at the debug area where the prints show up:

```
+0.0s: Subject emitted: H
+0.1s: Subject emitted: He
+0.2s: Subject emitted: Hel
+0.3s: Subject emitted: Hell
+0.5s: Subject emitted: Hello
+0.6s: Subject emitted: Hello
+1.6s: Debounced emitted: Hello
+2.1s: Subject emitted: Hello W
+2.1s: Subject emitted: Hello Wo
+2.4s: Subject emitted: Hello Wor
+2.4s: Subject emitted: Hello Worl
+2.7s: Subject emitted: Hello World
+3.7s: Debounced emitted: Hello World
```

As you can see, at `0.6` seconds the user pauses and resumes typing only at `2.2` seconds. Meanwhile, you configured `debounce` to wait for a one-second pause. It obliges (at `1.6` seconds) and emits the latest received value.

Same around the end where typing ends at `2.7` seconds and `debounce` kicks in one second later at `3.7` seconds. Cool!

> **Note**: One thing to watch out for is the publisher's completion. If your publisher completes right after the last value was emitted, but before the time configured for `debounce` elapses, you will never see the last value in the debounced publisher!

## Throttle

The kind of holding-off pattern that `debounce` allows is so useful that Combine provides a close relative: `throttle(for:scheduler:latest:)`. It's very close to `debounce`, but the differences justify the need for two operators.

Switch to the **Throttle** page in the playground and get coding. First, you need a constant, as usual:

```
let throttleDelay = 1.0

// 1
let subject = PassthroughSubject<String, Never>()

// 2
let throttled = subject
  .throttle(for: .seconds(throttleDelay), scheduler:
DispatchQueue.main, latest: false)
  // 3
```

```
    .share()
```

Breaking down this code:

1. The source publisher will emit strings.
2. Your throttled subject will now only emit the first value received from `subject` during each one-second interval because you set `latest` to `false`.
3. Like in the previous operator, `debounce`, adding the `share()` operator here guarantees that all subscribers see the same output at the same time from the throttled subject.

Create a couple timelines to visualize events, and wire them up to the two publishers:

```
let subjectTimeline = TimelineView(title: "Emitted values")
let throttledTimeline = TimelineView(title: "Throttled values")

let view = VStack(spacing: 100) {
  subjectTimeline
  throttledTimeline
}

PlaygroundPage.current.liveView = UIHostingController(rootView:
view)

subject.displayEvents(in: subjectTimeline)
throttled.displayEvents(in: throttledTimeline)
```

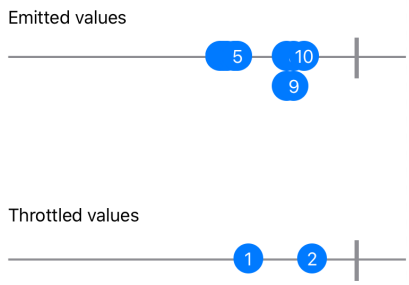
Now you also want to print the values each publisher emits, to better understand what's going on. Add this code:

```
let subscription1 = subject
  .sink { string in
    print("+\(deltaTime)s: Subject emitted: \(string)")
  }

let subscription2 = throttled
  .sink { string in
    print("+\(deltaTime)s: Throttled emitted: \(string)")
  }
```

Again, you are going to feed your source publisher with a simulated "Hello World" user input. Add this final line to your playground page:

```
subject.feed(with: typingHelloWorld)
```

Your playground is ready! You can now see what's happening in the live view:




Isn't this puzzling? It doesn't look *that much* different from the previous `debounce` output! Well, it actually is.

First, look closely at both. You can see that the values emitted by `throttle` have slightly different timing.

Second, to get a better picture of what's happening, look at the debug console:

```
+0.0s: Subject emitted: H
+0.1s: Subject emitted: He
+0.2s: Subject emitted: Hel
+0.3s: Subject emitted: Hell
+0.5s: Subject emitted: Hello
+0.6s: Subject emitted: Hello
+1.0s: Throttled emitted: H
+2.2s: Subject emitted: Hello W
+2.2s: Subject emitted: Hello Wo
+2.2s: Subject emitted: Hello Wor
+2.4s: Subject emitted: Hello Worl
+2.7s: Subject emitted: Hello World
+3.0s: Throttled emitted: Hello W
```

This is clearly different! You can see a few interesting things here:

- At `1.0` second, `throttle` kicks in and emits "`H`". Remember you asked it to send you the first value after one second.
- At `2.0` seconds, typing resumes. You can see that at this time, `throttle` didn't emit anything. This is because no new value had been received from the source publisher.
- At `3.0` seconds, after typing completes, `throttle` kicks in again and outputs the first value again, i.e., the value at `2.0` seconds.

There you have the fundamental difference between `debounce` and `throttle`:

- `debounce` waits for a pause in values it receives, then emits the latest one after the specified interval.
- `throttle` waits for the specified interval, then emits either the first or the latest of the values it received during that interval. It doesn't care about pauses.

To see what happens when you change `latest` to `true`, change your implementation of `throttled` to the following:

```
let throttled = subject
  .throttle(for: .seconds(throttleDelay), scheduler:
DispatchQueue.main, latest: true)
  .share()
```

Now, observe the resulting output in the debug area:

```
+0.0s: Subject emitted: H
+0.1s: Subject emitted: He
+0.2s: Subject emitted: Hel
+0.3s: Subject emitted: Hell
+0.5s: Subject emitted: Hello
+0.6s: Subject emitted: Hello
+1.0s: Throttled emitted: Hello
+2.0s: Subject emitted: Hello W
+2.3s: Subject emitted: Hello Wo
+2.3s: Subject emitted: Hello Wor
+2.6s: Subject emitted: Hello Worl
+2.6s: Subject emitted: Hello World
+3.0s: Throttled emitted: Hello World
```

The throttled output occurs at precisely `1.0` second and `3.0` seconds. Compare this with the output from `debounce` from the earlier example:

```
...
+1.6s: Debounced emitted: Hello
...
+3.7s: Debounced emitted: Hello World
```

The output is the same, but `debounce` is delayed from the pause.

# Timing out

Next, in this roundup of time manipulation operators is a special one: **timeout**. Its primary purpose is to semantically distinguish an actual timer from a timeout

condition. Therefore, when a timeout operator fires, it either completes the publisher or emits an error you specify. In both cases, the publisher terminates.

Switch to the **Timeout** playground page. Begin by adding this code:

```
let subject = PassthroughSubject<Void, Never>()

// 1
let timedOutSubject = subject.timeout(.seconds(5), scheduler:
DispatchQueue.main)
```

1. The `timedOutSubject` publisher will time-out after five seconds without the upstream publisher emitting any value. This form of `timeout` forces a publisher completion without any failure.

You now need to add your timeline, as well as a button to let you trigger events:

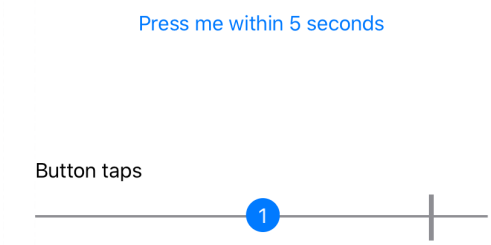
```
let timeline = TimelineView(title: "Button taps")

let view = VStack(spacing: 100) {
  // 1
  Button(action: { subject.send() }) {
    Text("Press me within 5 seconds")
  }
  timeline
}

PlaygroundPage.current.liveView = UIHostingController(rootView:
view)

timedOutSubject.displayEvents(in: timeline)
```

1. This is a new one! You add a button above the timeline, which sends a new value to the source subject when pressed. The `action` closure will execute every time you press the button.

> **Note**: Have you noticed you're using a subject that emits `Void` values? Yes, this is totally legitimate! It signals that *something* happened. But, there is no particular value to carry. So, you simply use `Void` as the value type. This is such a common case that `Subject` has an extension with a `send()` function that takes no parameter in case the `Output` type is `Void`. This saves you from writing the awkward `subject.send(())` statement!

Your playground page is now complete. Watch it run and do nothing: the `timeout` will trigger after five seconds and complete the publisher.

Now run it again. This time, keep pressing the button at less-than-five-seconds intervals. The publisher never completes because `timeout` doesn't kick in.




Of course, the simple completion of a publisher is not what you want in many cases. Instead, you need the `timeout` publisher to send a failure so you can accurately take action in this case.

Go to the top of the playground page and define the error type you want:

```
enum TimeoutError: Error {
  case timedOut
}
```

Next, modify the definition of `subject` to change the error type from `Never` to `TimeoutError`. Your code should look like this:

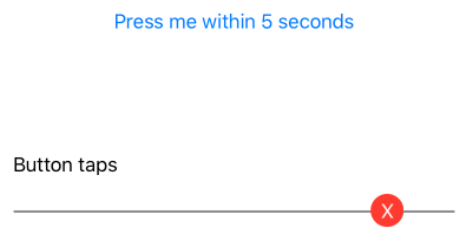
```
let subject = PassthroughSubject<Void, TimeoutError>()
```

Now you need to modify the call to `timeout`. The complete signature for this operator is `timeout(_:scheduler:options:customError:)`. Here is your chance to provide your custom error type!

Modify the line that creates the `timedOutSubject`to this:

```
let timedOutSubject = subject.timeout(.seconds(5),
                                      scheduler:
DispatchQueue.main,
                                      customError:
{ .timedOut })
```

Now when you run the playground and don't press the button for five seconds, you can see that the `timedOutSubject` emits a failure.




Now that the time allocated to this operator ran out, let's move to the last one in this section.

# Measuring time

To complete this roundup of time manipulation operators, you'll look at one particular operator which doesn't manipulate time but just *measures* it. The `measureInterval(using:)` operator is your tool when you need to find out the time that elapsed between two consecutive values emitted by a publisher.

Switch to the **MeasureInterval** playground page. Begin by creating a couple of publishers:

```
let subject = PassthroughSubject<String, Never>()

// 1
let measureSubject = subject.measureInterval(using:
DispatchQueue.main)
```

The `measureSubject` will emit measurements on the scheduler you specify. Here, the main queue.

Now as usual, add a couple of timelines:

```
let subjectTimeline = TimelineView(title: "Emitted values")
let measureTimeline = TimelineView(title: "Measured values")

let view = VStack(spacing: 100) {
  subjectTimeline
  measureTimeline
}

PlaygroundPage.current.liveView = UIHostingController(rootView:
view)
```

```
subject.displayEvents(in: subjectTimeline)
measureSubject.displayEvents(in: measureTimeline)
```

Finally, here comes the interesting part. Print out values emitted by both publishers and then feed the subject:

```
let subscription1 = subject.sink {
  print("+\(deltaTime)s: Subject emitted: \($0)")
}

let subscription2 = measureSubject.sink {
  print("+\(deltaTime)s: Measure emitted: \($0)")
}

subject.feed(with: typingHelloWorld)
```

Run your playground and have a look at the debug area! This is where you see what `measureInterval(using:)` emits:

```
+0.0s: Subject emitted: H
+0.0s: Measure emitted: Stride(magnitude: 16818353)
+0.1s: Subject emitted: He
+0.1s: Measure emitted: Stride(magnitude: 87377323)
+0.2s: Subject emitted: Hel
+0.2s: Measure emitted: Stride(magnitude: 111515697)
+0.3s: Subject emitted: Hell
+0.3s: Measure emitted: Stride(magnitude: 105128640)
+0.5s: Subject emitted: Hello
+0.5s: Measure emitted: Stride(magnitude: 228804831)
+0.6s: Subject emitted: Hello
+0.6s: Measure emitted: Stride(magnitude: 104349343)
+2.2s: Subject emitted: Hello W
+2.2s: Measure emitted: Stride(magnitude: 1533804859)
+2.2s: Subject emitted: Hello Wo
+2.2s: Measure emitted: Stride(magnitude: 154602)
+2.4s: Subject emitted: Hello Wor
+2.4s: Measure emitted: Stride(magnitude: 228888306)
+2.4s: Subject emitted: Hello Worl
+2.4s: Measure emitted: Stride(magnitude: 138241)
+2.7s: Subject emitted: Hello World
+2.7s: Measure emitted: Stride(magnitude: 333195273)
```

The values are a bit puzzling, aren't they? It turns out that, as per the documentation, the type of the value `measureInterval` emits is "*the time interval of the provided scheduler*". In the case of `DispatchQueue`, the `TimeInterval` is defined as "*A* DispatchTimeInterval *created with the value of this type in nanoseconds.*".

What you are seeing here is a count, in nanoseconds, **between each consecutive value** received from the source subject. You can now fix the display to show more

readable values. Modify the code that prints values from `measureSubject` like so:

```
let subscription2 = measureSubject.sink {
  print("+\(deltaTime)s: Measure emitted: \(Double($0.magnitude)
/ 1_000_000_000.0)")
}
```

Now, you'll see values in seconds.

But what happens if you use a different scheduler? You can try it using a `RunLoop` instead of a `DispatchQueue`!

> **Note**: You will explore the `RunLoop` and `DispatchQueue` schedulers in depth in Chapter 17, "Combine Schedulers."

Back to the top of the file, create a second subject that uses a `RunLoop`:

```
let measureSubject2 = subject.measureInterval(using:
RunLoop.main)
```

You don't need to bother wiring up a new timeline view, because what's interesting is the debug output. Add this third subscription to your code:

```
let subscription3 = measureSubject2.sink {
  print("+\(deltaTime)s: Measure2 emitted: \($0)")
}
```

Now, you'll see the output from the `RunLoop` scheduler as well, with magnitudes directly expressed in seconds:

```
+0.0s: Subject emitted: H
+0.0s: Measure emitted: 0.016503769
+0.0s: Measure2 emitted: Stride(magnitude: 0.015684008598327637)
+0.1s: Subject emitted: He
+0.1s: Measure emitted: 0.087991755
+0.1s: Measure2 emitted: Stride(magnitude: 0.08793699741363525)
+0.2s: Subject emitted: Hel
+0.2s: Measure emitted: 0.115842671
+0.2s: Measure2 emitted: Stride(magnitude: 0.11583995819091797)
...
```

The scheduler you use for measurement is really up to your personal taste. It is generally a good idea to stick with `DispatchQueue` for everything. But that's your personal choice!

# Challenge

If time allows, you may want to try a little challenge to put this new knowledge to good use!

Open the starter challenge playground in the **projects/challenge** folder. You see some code waiting for you:

- A subject that emits integers.
- A function call that feeds the subject with mysterious data.

In between those parts, your challenge is to:

- Group data by batches of `0.5` seconds.
- Turn the grouped data into a string.
- If there is a pause longer than `0.9` seconds in the feed, print the 👏 emoji. Hint: Create a second publisher for this step and merge it with the first publisher in your subscription.
- Print it.

> **Note**: To convert an `Int` to a `Character`, you can do something like `Character(Unicode.Scalar(value)!)`.

If you code this challenge correctly, you'll see a sentence printed in the Debug area. What is it?

## Solution

You'll find the solution to this challenge in the **challenge/final** folder.

Here's the solution code:

```
// 1
let strings = subject
  // 2
  .collect(.byTime(DispatchQueue.main, .seconds(0.5)))
  // 3
  .map { array in
    String(array.map { Character(Unicode.Scalar($0)!) })
  }
```

```
// 4
let spaces = subject.measureInterval(using: DispatchQueue.main)
  .map { interval in
    // 5
    interval > 0.9 ? "👏" : ""
  }

// 6
let subscription = strings
  .merge(with: spaces)
  // 7
  .filter { !$0.isEmpty }
  .sink {
    // 8
    print($0)
  }
```

From the top, you:

1. Create a first publisher derived from the subject which emits the strings.
2. Use `collect()` using the `.byTime` strategy to group data in `0.5` seconds batches.
3. Map each integer value to a Unicode scalar, then to a character and then turn the whole lot into a string using `map`.
4. Create a second publisher derived from the subject, which measures the intervals between each character.
5. If the interval is greater than `0.9` seconds, map the value to the 👏 emoji. Otherwise, map it to an empty string.
6. The final publisher is a merge of both strings and the 👏 emoji.
7. Filter out empty strings for better display.
8. Print the result!

Your solution might have been subtly different, and that's OK. As long as you met the requirements, you get the W!

Running the playground with this solution will print the following output to the console:

```
Combine
👏
is
```

👋
cool!

# Key points

In this chapter, you looked at time from a different angle. In particular, you learned that:

- Combine's handling of asynchronous events extends to manipulating time itself.
- Even though it doesn't provide time-traveling options, the framework has operators that let you abstract work over long periods of time, rather than just handling discrete events.
- Time can be shifted using the `delay` operator.
- You can manage the flow of values over time like a dam and release them by chunks using `collect`.
- Picking individual values over time is easy with `debounce` and `throttle`.
- Not letting time run out is the job of `timeout`.
- Time can be measured with `measureInterval`.

# Where to go from here?

This was a lot to learn. To put events in their right order, move along to the next chapter and learn about sequence operators!

# Chapter 7: Sequence Operators

By Shai Mishali

At this point, you know most of the operators that Combine has to offer! How great is that? There's one more category for you to dig into: **Sequence Operators**.

Sequence operators are easiest to understand when you realize that publishers are just sequences themselves. Sequence operators work with the collection of a publisher's values, more like an array or set — which, of course, are just finite sequences!

With that in mind, sequence operators mostly deal with the sequence as a whole and not with individual values, as other operator categories do.

Many of the operators in this category have nearly identical names and behaviors as their counterparts in the Swift standard library.

## Getting started

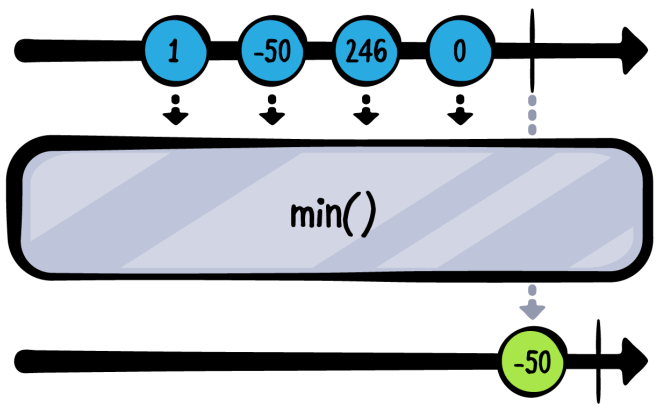
You can find the starter playground for this chapter, **Starter.playground**, in the **projects** folder. Throughout this chapter, you'll add code to your playground and run it to see how these different sequence operators manipulate your publisher. You'll use the `print` operator to log all publishing events.

# Finding values

The first section of this chapter consists of operators that locate specific values the publisher emits based on different criteria. These are similar to the collection methods in the Swift standard library.

## min

The `min` operator lets you find the minimum value emitted by a publisher. It's *greedy*, which means it must wait for the publisher to send a `.finished` completion event. Once the publisher completes, only the minimum value is emitted by the operator:




Find the placeholder comment `// Add your code here` and replace it with the following example:

```
example(of: "min") {
  // 1
  let publisher = [1, -50, 246, 0].publisher

  // 2
  publisher
    .print("publisher")
    .min()
    .sink(receiveValue: { print("Lowest value is \($0)") })
    .store(in: &subscriptions)
}
```

In the above code, you:

1. Create a publisher emitting four different numbers.
2. Use the `min` operator to find the minimum number emitted by the publisher and print that value.

Run your playground and you'll see the following output in the console:

```
—— Example of: min ——
publisher: receive subscription: ([1, -50, 246, 0])
publisher: request unlimited
publisher: receive value: (1)
publisher: receive value: (-50)
publisher: receive value: (246)
publisher: receive value: (0)
publisher: receive finished
Lowest value is -50
```

As you can see, the publisher emits all its values and finishes, then `min` finds the minimum and sends it downstream to the `sink` to print it out.

But wait, how does Combine know which of these numbers is the minimum? Well, because numbers conform to the `Comparable` protocol. You can use `min()` directly, without any arguments, on publishers that emit `Comparable`-conforming types.

But what happens if your values don't conform to `Comparable`? Luckily, you can provide your own comparator closure using the `min(by:)` operator.

Consider the following example, where your publisher emits many pieces of `Data` and you'd like to find the smallest one.

Add the following code to your playground:

```
example(of: "min non-Comparable") {
  // 1
  let publisher = ["12345",
                   "ab",
                   "hello world"]
    .compactMap { $0.data(using: .utf8) } // [Data]
    .publisher // Publisher<Data, Never>

  // 2
  publisher
    .print("publisher")
    .min(by: { $0.count < $1.count })
    .sink(receiveValue: { data in
      // 3
      let string = String(data: data, encoding: .utf8)!
      print("Smallest data is \(string), \(data.count) bytes")
    })
```

```
    .store(in: &subscriptions)
}
```

In the above code:

1. You create a publisher that emits three `Data` objects created from various strings.
2. Since `Data` doesn't conform to `Comparable`, you use the `min(by:)` operator to find the `Data` object with the smallest number of bytes.
3. You convert the smallest `Data` object back to a string and print it out.

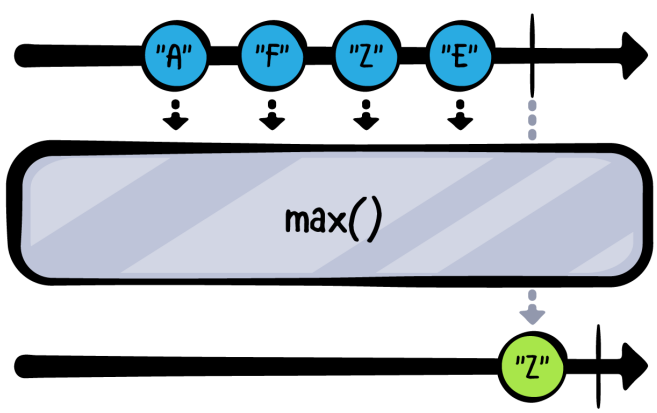
Run your playground and you'll see the following in your console:

```
—— Example of: min non-Comparable ——
publisher: receive subscription: ([5 bytes, 2 bytes, 11 bytes])
publisher: request unlimited
publisher: receive value: (5 bytes)
publisher: receive value: (2 bytes)
publisher: receive value: (11 bytes)
publisher: receive finished
Smallest data is ab, 2 bytes
```

Like the previous example, the publisher emits all its `Data` objects and finishes, then `min(by:)` finds and emits the data with the smallest byte size and `sink` prints it out.

## max

As you'd guess, `max` works exactly like `min`, except that it finds the *maximum* value emitted by a publisher:



Add the following code to your playground to try this example:

```
example(of: "max") {
  // 1
  let publisher = ["A", "F", "Z", "E"].publisher

  // 2
  publisher
    .print("publisher")
    .max()
    .sink(receiveValue: { print("Highest value is \($0)") })
    .store(in: &subscriptions)
}
```

In the following code, you:

1. Create a publisher that emits four different letters.

2. Use the `max` operator to find the letter with the highest value and print it.

Run your playground. You'll see the following output in your playground:

```
—— Example of: max ——
publisher: receive subscription: (["A", "F", "Z", "E"])
publisher: request unlimited
publisher: receive value: (A)
publisher: receive value: (F)
publisher: receive value: (Z)
publisher: receive value: (E)
publisher: receive finished
Highest value is Z
```

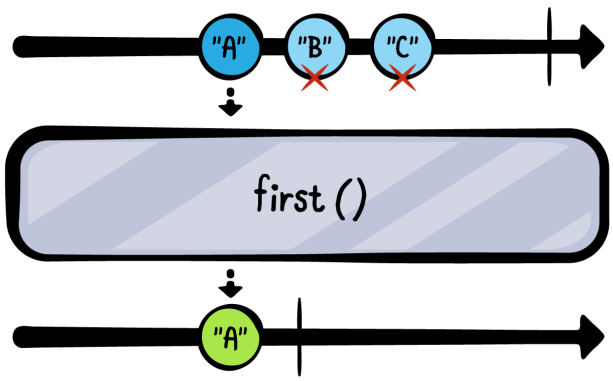
Exactly like `min`, `max` is *greedy* and must wait for the upstream publisher to finish emitting its values before it determines the maximum value. In this case, that value is Z.

> **Note**: Exactly like `min`, `max` also has a companion `max(by:)` operator that takes a predicate to determine the maximum value emitted among non-`Comparable` values.

## first

While the `min` and `max` operators deal with finding a published value at some unknown index, the rest of the operators in this section deal with finding emitted values at *specific* places, starting with the `first` operator.

The `first` operator is similar to Swift's `first` property on collections, except that it lets the first emitted value through and then completes. It's *lazy*, meaning it doesn't wait for the upstream publisher to finish, but instead will cancel the subscription when it receives the first value emitted.




Add the above example to your playground:

```
example(of: "first") {
  // 1
  let publisher = ["A", "B", "C"].publisher

  // 2
  publisher
    .print("publisher")
    .first()
    .sink(receiveValue: { print("First value is \($0)") })
    .store(in: &subscriptions)
}
```

In the above code, you:

1. Create a publisher emitting three letters.
2. Use `first()` to let only the first emitted value through and print it out.

Run your playground and take a look at the console:

```
—— Example of: first ——
publisher: receive subscription: (["A", "B", "C"])
publisher: request unlimited
publisher: receive value: (A)
publisher: receive cancel
First value is A
```

As soon as `first()` lets the first value through, it immediately cancels the subscription to the upstream publisher.

If you're looking for more granular control, you can also use `first(where:)`. Just like its counterpart in the Swift standard library, it will emit the first value that matches a provided predicate — if there is one.

Add the following example to your playground:

```
example(of: "first(where:)") {
  // 1
  let publisher = ["J", "O", "H", "N"].publisher

  // 2
  publisher
    .print("publisher")
    .first(where: { "Hello World".contains($0) })
    .sink(receiveValue: { print("First matching value is \($0)")
})
    .store(in: &subscriptions)
}
```

In this code, you:

1. Create a publisher that emits four letters.
2. Use the `first(where:)` operator to find the first letter contained in `Hello World` and then print it out.

Run the playground and you'll see the following output:

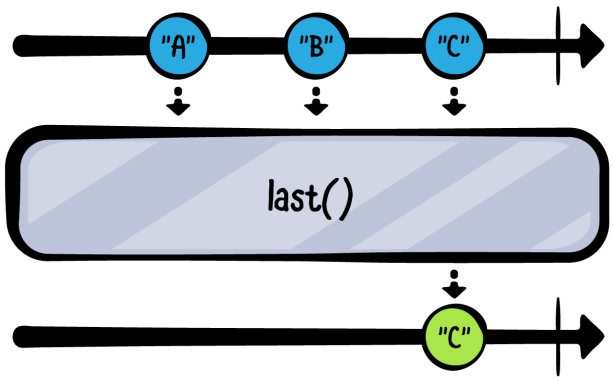
```
—— Example of: first(where:) ——
publisher: receive subscription: (["J", "O", "H", "N"])
publisher: request unlimited
publisher: receive value: (J)
publisher: receive value: (O)
publisher: receive value: (H)
publisher: receive cancel
First matching value is H
```

In the above example, the operator checks if `Hello World` contains the emitted letter until it finds the first match: `H`. Upon finding that much, it cancels the subscription and emits the letter for `sink` to print out.

## last

Just as `min` has an opposite, `max`, `first` also has an opposite: `last`!

`last` works exactly like `first`, except it emits the *last* value that the publisher emits. This means it's also *greedy* and must wait for the upstream publisher to finish:




Add this example to your playground:

```
example(of: "last") {
  // 1
  let publisher = ["A", "B", "C"].publisher

  // 2
  publisher
    .print("publisher")
    .last()
    .sink(receiveValue: { print("Last value is \($0)") })
    .store(in: &subscriptions)
}
```

In this code, you:

1. Create a publisher that will emit three letters and finish.
2. Use the `last` operator to only emit the last value published and print it out.

Run the playground and you'll see the following output:

```
—— Example of: last ——
publisher: receive subscription: (["A", "B", "C"])
publisher: request unlimited
publisher: receive value: (A)
publisher: receive value: (B)
publisher: receive value: (C)
publisher: receive finished
Last value is C
```

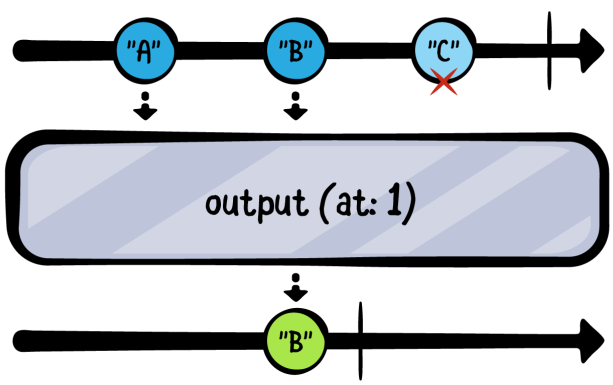
`last` waits for the upstream publisher to send a `.finished` completion event, which it sends downstream to be printed out in `sink`.

> **Note**: Exactly like `first`, `last` also has a companion `last(where:)` operator, which emits the last value emitted by a publisher that matches the specified predicate.

## output(at:)

The last two operators in this section don't have counterparts in the Swift standard library. The `output` operators will only let values through if they're emitted by the upstream publisher at the specified indices.

Start with `output(at:)`, which emits only the value emitted at the specified index:




Add the following code to your playground to try this example:

```
example(of: "output(at:)") {
  // 1
  let publisher = ["A", "B", "C"].publisher

  // 2
  publisher
    .output(at: 1)
    .sink(receiveValue: { print("Value at index 1 is \($0)") })
    .store(in: &subscriptions)
}
```

In the above code, you:

1. Create a publisher which emits three letters.
2. Use `output(at:)` to only let through the value emitted at index `1` — i.e., the second value.

Run the example in your playground and peek at your console:

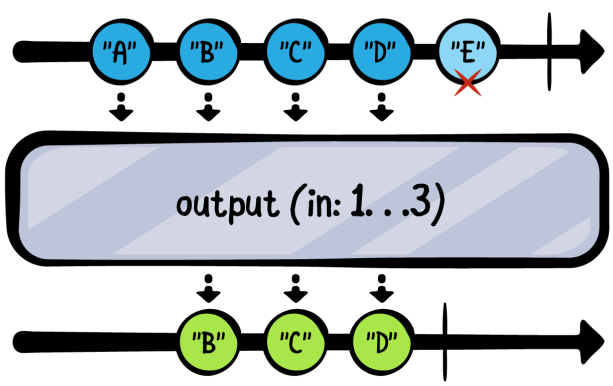
```
—— Example of: output(at:) ——
publisher: receive subscription: (["A", "B", "C"])
publisher: request unlimited
publisher: receive value: (A)
publisher: request max: (1) (synchronous)
publisher: receive value: (B)
Value at index 1 is B
publisher: receive cancel
```

Here, the output indicates the value at index `1` is `B`. However, you might've noticed an additional interesting fact: The operator *demands* one more value after every emitted value, since it knows it's only looking for an item at a specific index. While this is an implementation detail of the specific operator, it provides interesting insight into how Apple designs some of their own built-in Combine operators.

## output(in:)

You'll wrap up this section with the second overload of the `output` operator: `output(in:)`.

While `output(at:)` emits a *single* value emitted in a specified index, `output(in:)` emits values whose indices are within a provided *range*:




To try this out, add the following example to your playground:

```
example(of: "output(in:)") {
  // 1
  let publisher = ["A", "B", "C", "D", "E"].publisher

  // 2
  publisher
    .output(in: 1...3)
    .sink(receiveCompletion: { print($0) },
          receiveValue: { print("Value in range: \($0)") })
    .store(in: &subscriptions)
}
```

In the above code, you:

1. Create a publisher that emits five different letters.
2. Use the `output(in:)` operator to only let through values emitted in indices `1` through `3`, then print out those values.

Can you guess what the output of this example will be? Run your playground and find out:

```
—— Example of: output(in:) ——
Value in range: B
Value in range: C
Value in range: D
finished
```

Well, did you guess correctly? The operator emits **individual values** within the range of indices, not a collection of them.

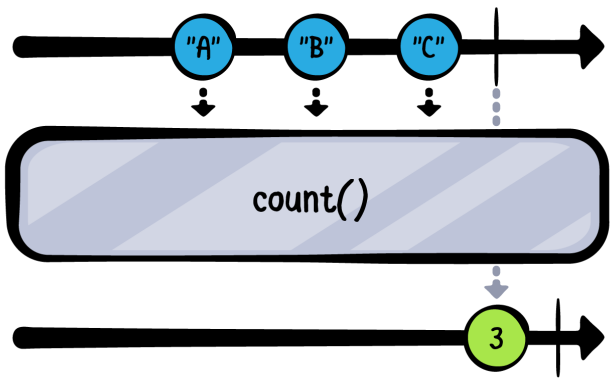
The operator prints the values `B`, `C` and `D` as they're in indices `1`, `2` and `3`, respectively. Then, since all items within the range have been emitted, it cancels the subscription as soon as it receives everything it needs to complete its work.

# Querying the publisher

The following operators also deal with the entire set of values emitted by a publisher, but they don't produce any specific value that it emits. Instead, these operators emit a different value representing some query on the publisher as a whole. A good example of this is the `count` operator.

## count

The `count` operator will emit a single number depicting how many values were emitted by the upstream publisher, once the publisher sends a `.finished` completion event:




Add the following code to try this example:

```
example(of: "count") {
  // 1
  let publisher = ["A", "B", "C"].publisher

  // 2
  publisher
    .print("publisher")
    .count()
    .sink(receiveValue: { print("I have \($0) items") })
    .store(in: &subscriptions)
}
```

In the above code, you:

1. Create a publisher that emits three letters.
2. Use `count()` to emit a single value indicating the number of values emitted by the upstream publisher.

Run your playground and check your console. You'll see the following output:

```
—— Example of: count ——
publisher: receive subscription: (["A", "B", "C"])
publisher: request unlimited
```

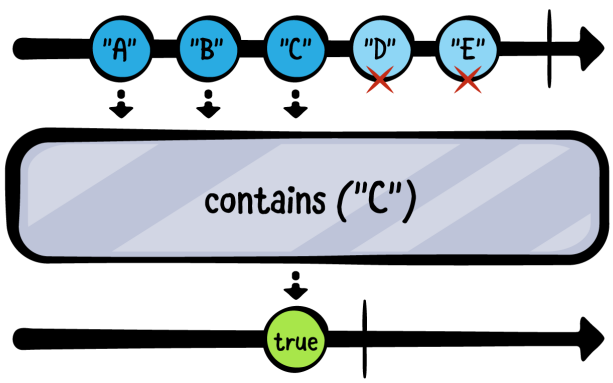
```
publisher: receive value: (A)
publisher: receive value: (B)
publisher: receive value: (C)
publisher: receive finished
I have 3 items
```

As expected, the value 3 is only printed out once the upstream publisher sends a `.finished` completion event.

## contains

Another useful operator is `contains`. You've probably used its counterpart in the Swift standard library more than once.

The `contains` operator will emit `true` and cancel the subscription if the specified value is emitted by the upstream publisher, or `false` if none of the emitted values are equal to the specified one:




Add the following to your playground to try `contains`:

```
example(of: "contains") {
  // 1
  let publisher = ["A", "B", "C", "D", "E"].publisher
  let letter = "C"

  // 2
  publisher
    .print("publisher")
    .contains(letter)
    .sink(receiveValue: { contains in
      // 3
      print(contains ? "Publisher emitted \(letter)!"
```

```
                        : "Publisher never emitted \(letter)!")
    })
    .store(in: &subscriptions)
}
```

In the following code, you:

1. Create a publisher emitting five different letters — `A` through `E` — and create a `letter` value to use with `contains`.
2. Use `contains` to check if the upstream publisher emitted the value of `letter`: `C`.
3. Print an appropriate message based on whether or not the value was emitted.

Run your playground and check the console:

```
—— Example of: contains ——
publisher: receive subscription: (["A", "B", "C", "D", "E"])
publisher: request unlimited
publisher: receive value: (A)
publisher: receive value: (B)
publisher: receive value: (C)
publisher: receive cancel
Publisher emitted C!
```

Huzzah! You got a message indicating `C` was emitted by the publisher. You might have also noticed `contains` is lazy, as it only consumes as many upstream values as it needs to perform its work. Once `C` is found, it cancels the subscription and doesn't produce any further values.

Why don't you try another variation? Replace the following line:

```
let letter = "C"
```

With:

```
let letter = "F"
```

Next, run your playground again. You'll see the following output:

```
—— Example of: contains ——
publisher: receive subscription: (["A", "B", "C", "D", "E"])
publisher: request unlimited
publisher: receive value: (A)
publisher: receive value: (B)
publisher: receive value: (C)
publisher: receive value: (D)
publisher: receive value: (E)
```

```
publisher: receive finished
Publisher never emitted F!
```

In this case, `contains` waits for the publisher to emit `F`. However, the publisher finishes without emitting `F`, so `contains` emits `false` and you see the appropriate message printed out.

Finally, sometimes you want to look for a match for a predicate that you provide or check for the existence of an emitted value that doesn't conform to `Comparable`. For these specific cases, you have `contains(where:)`.

Add the following code to your playground:

```
example(of: "contains(where:)") {
  // 1
  struct Person {
      let id: Int
      let name: String
  }

  // 2
  let people = [
    (456, "Scott Gardner"),
    (123, "Shai Mishali"),
    (777, "Marin Todorov"),
    (214, "Florent Pillet")
  ]
  .map(Person.init)
  .publisher

  // 3
  people
    .contains(where: { $0.id == 800 })
    .sink(receiveValue: { contains in
      // 4
      print(contains ? "Criteria matches!"
                     : "Couldn't find a match for the criteria")
    })
    .store(in: &subscriptions)
}
```

This above code is a bit more complex, but not by much. You:

1. Define a `Person` struct with an `id` and a `name`.
2. Create a publisher that emits four different instances of `People`.
3. Use `contains` to see if the `id` of any of them is `800`.
4. Print an appropriate message based on the emitted result.

Run your playground and you'll see the following output:

```
—— Example of: contains(where:) ——
Couldn't find a match for the criteria
```

It didn't find any matches, as expected, because none of the emitted people have an `id` of `800`.

Next, change the implementation of `contains(where:)`:

```
.contains(where: { $0.id == 800 })
```

To the following:

```
.contains(where: { $0.id == 800 || $0.name == "Marin Todorov" })
```

Run the playground again and look at the console:

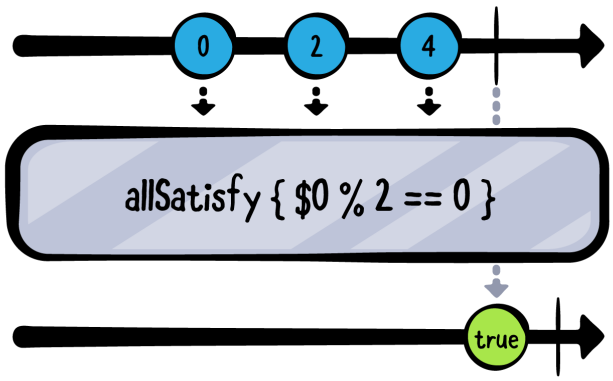
```
—— Example of: contains(where:) ——
Criteria matches!
```

This time it found a value matching the predicate, since Marin is indeed one of the people in your list. Awesome! :]

## allSatisfy

A bunch of operators down, and only two to go! Both of them have counterpart collection methods in the Swift standard library.

You'll start with `allSatisfy`, which takes a closure predicate and emits a Boolean indicating whether *all* values emitted by the upstream publisher match that predicate. It's greedy and will, therefore, wait until the upstream publisher emits a `.finished` completion event:




Add the following example to your playground:

```
example(of: "allSatisfy") {
  // 1
  let publisher = stride(from: 0, to: 5, by: 2).publisher

  // 2
  publisher
    .print("publisher")
    .allSatisfy { $0 % 2 == 0 }
    .sink(receiveValue: { allEven in
      print(allEven ? "All numbers are even"
                    : "Something is odd...")
    })
    .store(in: &subscriptions)
}
```

In the above code, you:

1. Create a publisher that emits numbers between `0` to `5` in steps of `2` (i.e., `0`, `2` and `4`).
2. Use `allSatisfy` to check if all emitted values are even, then print an appropriate message based on the emitted result.

Run the code and check the console output:

```
—— Example of: allSatisfy ——
publisher: receive subscription: (Sequence)
publisher: request unlimited
publisher: receive value: (0)
publisher: receive value: (2)
publisher: receive value: (4)
```

```
publisher: receive finished
All numbers are even
```

Since all values are indeed even, the operator emits `true` after the upstream publisher sends a `.finished` completion, and the appropriate message is printed out.

However, if even a single value doesn't pass the predicate condition, the operator will emit `false` immediately and will cancel the subscription.

Replace the following line:

```
let publisher = stride(from: 0, to: 5, by: 2).publisher
```

With:

```
let publisher = stride(from: 0, to: 5, by: 1).publisher
```

You simply changed the `stride` to step between `0` and `5` by `1`, instead of `2`. Run the playground once again and take a look at the console:

```
—— Example of: allSatisfy ——
publisher: receive subscription: (Sequence)
publisher: request unlimited
publisher: receive value: (0)
publisher: receive value: (1)
publisher: receive cancel
Something is odd...
```

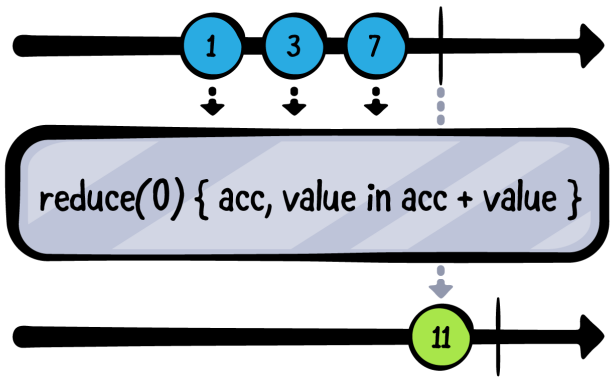
In this case, as soon as `1` is emitted, the predicate doesn't pass anymore, so `allSatisfy` emits `false` and cancels the subscription.

## reduce

Well, here we are! The final operator for this rather packed chapter: `reduce`.

The `reduce` operator is a bit different from the rest of the operators covered in this chapter. It doesn't look for a specific value or query the publisher as a whole. Instead, it lets you iteratively accumulate a *new* value based on the emissions of the upstream publisher.

This might sound confusing at first, but you'll get it in a moment. The easiest way to start is with a diagram:




Combine's `reduce` operator works like its counterparts in the Swift standard library: `reduce(_:_)` and `reduce(into:_:)`. It lets you provide a seed value and an accumulator closure. That closure receives the accumulated value — starting with the seed value — and the current value. From that closure, you return *a new* accumulated value. Once the operator receives a `.finished` completion event, it emits the final accumulated value.

In the case of the above diagram, you can think of it this way :

```
Seed value is 0
Receives 1, 0 + 1 = 1
Receives 3, 1 + 3 = 4
Receives 7, 4 + 7 = 11
Emits 11
```

Time for you to try a quick example to get a better sense of this operator. Add the following to your playground:

```
example(of: "reduce") {
  // 1
  let publisher = ["Hel", "lo", " ", "Wor", "ld", "!"].publisher

  publisher
    .print("publisher")
    .reduce("") { accumulator, value in
      // 2
      accumulator + value
    }
    .sink(receiveValue: { print("Reduced into: \($0)") })
    .store(in: &subscriptions)
}
```

In this code, you:

1. Create a publisher that emits `Strings` this time.
2. Use `reduce` with a seed of an empty string, appending the emitted values to it to create the final string result.

Run the playground and take a look at the console output:

```
—— Example of: reduce ——
publisher: receive subscription: (["Hel", "lo", " ", "Wor",
"ld", "!"])
publisher: request unlimited
publisher: receive value: (Hel)
publisher: receive value: (lo)
publisher: receive value: ( )
publisher: receive value: (Wor)
publisher: receive value: (ld)
publisher: receive value: (!)
publisher: receive finished
Reduced into: Hello World!
```

Notice how the accumulated result — `Hello World!` — is only printed once the upstream publisher sent a `.finished` completion event.

The second argument for `reduce` is a closure that takes two values of some type and returns a value of that same type. In Swift, `+` is an *also* a function that matches that signature.

So as a final neat trick, you can *reduce* the syntax above. Replace the following code:

```
.reduce("") { accumulator, value in
  // 3
  return accumulator + value
}
```

With simply:

```
.reduce("", +)
```

If you run your playground again, it will work exactly the same as before, with a bit of a fancier syntax. ;]

> **Note**: Does this operator feel a bit familiar? Well, that might be because you learned about `scan` in Chapter 2, "Transforming Operators." `scan` and `reduce` have the same functionality, with the main difference being that `scan` emits

the accumulated value for *every* emitted value, while `reduce` emits *a single* accumulated value once the upstream publisher sends a `.finished` completion event. Feel free to change `reduce` to `scan` in the above example and try it out for yourself.

# Key points

- Publishers are actually sequences, as they produce values much as collections and sequences do.
- Use `min` and `max` to emit the minimum or maximum value emitted by a publisher, respectively.
- Use `first`, `last` and `output(at:)` to find a value emitted at a specific index. Use `output(in:)` to find values emitted within a *range* of indices.
- `first(where:)` and `last(where:)` each take a predicate to determine which values it should let through.
- Operators such as `count`, `contains` and `allSatisfy` don't emit values emitted by the publisher. Rather, they emit a different value based on the emitted values.
- `contains(where:)` takes a predicate to determine if the publisher contains the given value.
- Use `reduce` to accumulate emitted values into a single value.

# Where to go from here?

Congrats on completing the last chapter on operators for this book! give yourself a quick pat on the back and high-five yourself while you're at it. :]

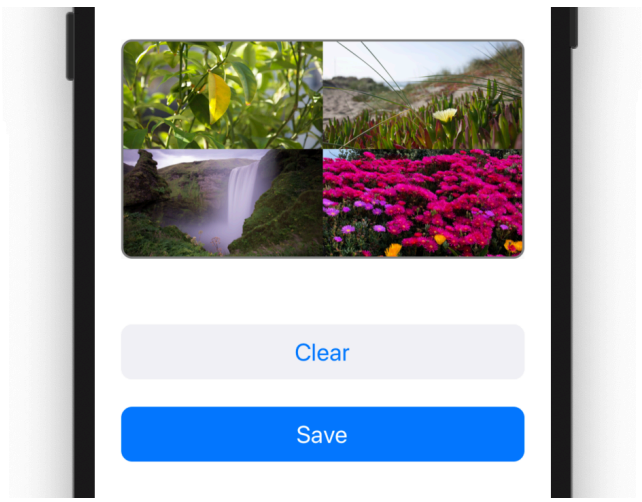
You'll wrap up this section by working on your first practical project, where you'll build a Collage app using Combine and many of the operators you've learned. Take a few deep breaths, grab a cup of coffee, and move on to the next chapter.

# Chapter 8: In Practice: Project "Collage"

By Marin Todorov

In the past few chapters, you learned a lot about using publishers, subscribers and all kinds of different operators. You did that in the "safety" of a Swift playground. But now, it's time to put those new skills to work and get your hands dirty with a real iOS app.

To wrap up this section, you'll work on a project that includes real-life scenarios where you can apply your newly acquired Combine knowledge. The project is called **Collage** and it's an iOS app which allows the user to create simple collages out of their photos, like this:



This project will take you through:

- Using Combine publishers in your UIKit view controllers.
- Handling user events with Combine.
- Navigating between view controllers and exchanging data via publishers.
- Using a variety of operators to create different subscriptions to implement your app's logic.
- Wrapping existing Cocoa APIs so you can conveniently use them in your Combine code.

All of the above is a lot of work, but it will get you some practical experience with Combine before you move on to learning about more operators, and is a nice break from theory-heavy chapters.

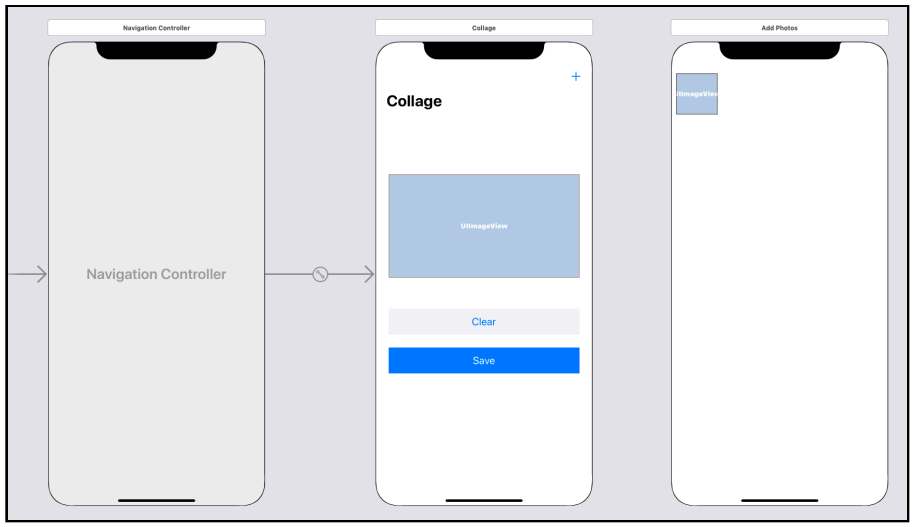
This chapter will guide you, in a tutorial style, through a variety of loosely connected tasks where you will use techniques based on the material you have covered so far in this book.

Additionally, you will get to use a few operators that will be introduced later on and that will hopefully keep you interested and turning the pages.

Without further ado — it's time to get coding!

## Getting started with "Collage"

To get started with Collage, open the starter project provided for this chapter and select **Assets/Main.storyboard** in the project navigator. The app's structure is rather simple — there is a main view controller to create and preview collages and an additional view controller where users select photos to add to their current collage:




> **Note**: In this chapter, you will work on integrating Combine data workflows and UIKit user controls and events. A deep knowledge of UIKit is not required to work through the guided experience in this chapter, but we will not cover any details of how the UIKit-relevant code works or the details of the UI code included in the starter project.

Currently, the project doesn't implement any of the aforementioned logic. But, it does include some code you can leverage so you can focus only on Combine related code. Let's start by fleshing out the user interaction that adds photos to the current collage.

Open **MainViewController.swift** and import the Combine framework at the top of the file:

```
import Combine
```

This will allow you to use Combine types in this file. To get started, add two new private properties to the `MainViewController` class:

```
private var subscriptions = Set<AnyCancellable>()
private let images = CurrentValueSubject<[UIImage], Never>([])
```

`subscriptions` is the collection where you will store any UI subscriptions tied to the lifecycle of the current view controller. When you bind your UI controls tying those subscriptions to the lifecycle of the current view controller is usually what you need.

This way, in case the view controller is popped out of the navigation stack or dismissed otherwise, all UI subscriptions will be canceled right away.

> **Note**: As mentioned in Chapter 1, "Hello Combine," subscribers return a `Cancellable` token to allow manually canceling a subscription. `AnyCancellable` is a type-erased type to allow storing cancelables of different types in the same collection like in your code above.

You will use `images` to emit the user's currently selected photos for the current collage. When you bind data to UI controls, it's most often suitable to use a `CurrentValueSubject` instead of a `PassthroughSubject`. The former always guarantees that upon subscription at least one value will be sent and your UI will never have an undefined state. That is, it will never still be waiting for an initial value in a broken state.

Next, to get some images added to the collage and test your code, add in `actionAdd()`:

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

Whenever the user taps on the + button in the top-right corner of the screen, you will add the **IMG_1907.jpg** to the current `images` array value and send that value to the subject, so all subscribers receive it.

You can find **IMG_1907.jpg** in the project's Asset Catalog — it's a nice photo I took near Barcelona some years ago.

To also be able to clear the currently selected photos, move over to `actionClear()` and add there:

```
images.send([])
```

This line simply sends an empty array to the `images` subject, pushing it to all of its subscribers.

Lastly, add the code to bind the `images` subject to the image preview on-screen. Append at the end of `viewDidLoad()`:

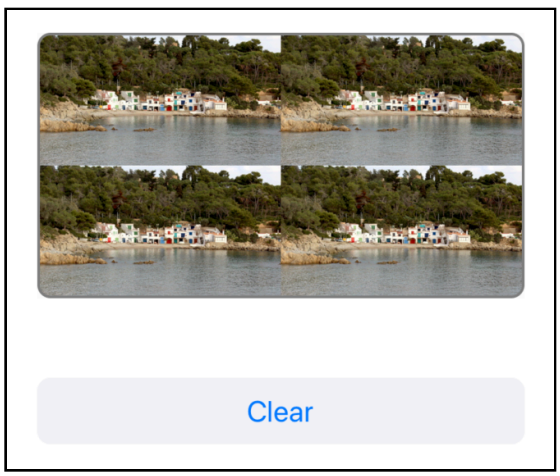
```
// 1
images
  // 2
  .map { photos in
```

```
      UIImage.collage(images: photos, size: collageSize)
    }
    // 3
    .assign(to: \.image, on: imagePreview)
    // 4
    .store(in: &subscriptions)
```

The play-by-play for this subscription is as follows:

1. You begin a subscription to the current collection of photos.
2. You use `map` to convert them to a single collage by calling into `UIImage.collage(images:size:)`, a helper method defined in **UIImage+Collage.swift**.
3. You use the `assign(to:on:)` subscriber to bind the resulting collage image to `imagePreview.image`, which is the center screen image view.
4. Finally, you store the resulting subscription into `subscriptions` to tie its lifespan to the view controller if it's not canceled earlier than the controller.

Time to test that new subscription! Build and run the app and click the + button few times. You should see a collage preview, featuring one more copy of the same photo each time your click +:




Thanks to the simplicity of binding via assign, you can get the photos collection, convert it to a collage and assign it to an image view in a single subscription!

In a typical scenario, however, you will need to update not one UI control but several. Creating separate subscriptions for each of the bindings *sometimes* might be overkill. So, let's see how we can perform a number of updates as a single batch.

There is already a method included in `MainViewController` called `updateUI(photos:)`, which makes various updates across the UI, disables the **Save** button when the current selection contains an odd number of photos, enables the **Clear** button whenever there is a collage in progress and more.

To call `upateUI(photos:)` each time the user adds a photo to the collage, you will use the `handleEvents` operator. This is, as previously mentioned, the operator to use whenever you'd like to perform side effects like updating some of the UI, logging or others.

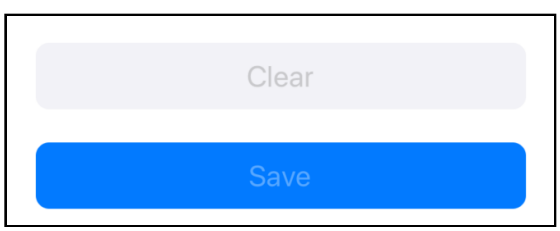
Back in `viewDidLoad()`, insert this operator just before the line where you use `map`:

```
.handleEvents(receiveOutput: { [weak self] photos in
  self?.updateUI(photos: photos)
})
```

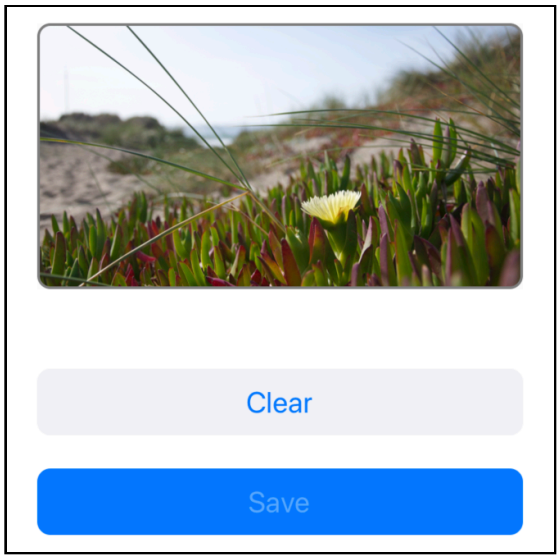
> **Note:** The `handleEvents` operator enables you to perform side effects when a publisher emits an event. You'll learn a lot more about it in Chapter 10, "Debugging Combine."

This will feed the current selection to `updateUI(photos:)` just before they are converted into a single collage image inside the `map` operator.

As soon as you run the project again you will notice the two buttons below the preview are disabled, which is the correct initial state:




The buttons will keep changing state as you add more photos to the current collage. For example, when you select one or three photos the **Save** button will be disabled but **Clear** enabled like so:




# Talking to other view controllers

You saw how easy it is to route your UI's data through a subject and bind it to some controls on-screen. Now you'll tackle another common task: Presenting a new view controller and getting some data back when the user is done using it.

The generic idea of exchanging data between two view controllers is exactly the same as subscribing to a subject in the same view controller. At the end of the day, you just have a publisher that emits some output and you use a subscriber to make something useful with the emitted values.

Scroll back to `actionAdd()`, comment the existing body of that method and add the following code instead:

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

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

This code instantiates a `PhotosViewController` from the project storyboard and pushes it onto the navigation stack. Since accessing the photos library and displaying a list of the available photos isn't specifically a Combine related task, that code is already fleshed out for you.

Open **PhotosViewController.swift** and in `viewDidLoad()` you will see it already contains the code to load photos from the Camera Roll and display them in a collection view.

Your next task is to add a subject to the view controller and emit any images that the user taps in the Camera Roll list.

First and foremost, just like before, add a new import in **PhotosViewController.swift**:

```
import Combine
```

Unlike the `images` subject in the main view controller which you both subscribe to and send values to, in this view controller you'd like to only allow other consumers to subscribe, making it a read-only publisher. In other words, you'd like to be able to send values from within `PhotosViewController` but only allow `MainViewController` to subscribe. To achieve that, you'll use another common pattern: Exposing a publisher publicly by abstracting a private subject.

Add this pair of public and private properties under their respective `MARK` comments:

```
// MARK: - Public properties
var selectedPhotos: AnyPublisher<UIImage, Never> {
  return selectedPhotosSubject.eraseToAnyPublisher()
}

// MARK: - Private properties
private let selectedPhotosSubject =
  PassthroughSubject<UIImage, Never>()
```

This code allows the current type to use `selectedPhotosSubject` to send values while other types can only access the type-erased `selectedPhotos` to subscribe: For example, `selectedPhotos` cannot be used to send new values through the publisher. Speaking of, let's hook up the collection view delegate method to that subject.

Scroll down to `collectionView(_:didSelectItemAt:)`. The code in that method flashes the tapped collection cell and then fetches the photo asset from the device library. Finally, having the photo ready, you should use the subject to send out the image to any subscribers.

Replace the `// Send the selected photo` comment with:

```
self.selectedPhotosSubject.send(image)
```

Well, that was easy! However, since you're exposing the subject to other types, you'd like to explicitly send a completion event in case the view controller is being

dismissed to tear down any external subscriptions. Scroll up to `viewWillDisappear(animated:)` and append:

```
selectedPhotosSubject.send(completion: .finished)
```

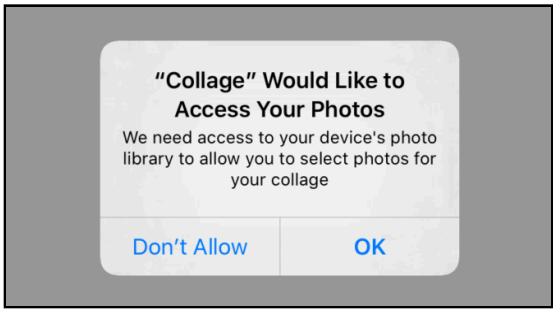
This code will send a `finished` event when you navigate back from the view controller. To wrap up the current task, you still need to subscribe to the selected photos from your main view controller. Open **MainViewController.swift**, find `actionAdd()` and insert **before** the last line presenting `PhotosViewController`:

```
let newPhotos = photos.selectedPhotos

newPhotos
  .map { [unowned self] newImage in
  // 1
    return self.images.value + [newImage]
  }
  // 2
  .subscribe(images)
  // 3
  .store(in: &subscriptions)
```

In this subscription you:

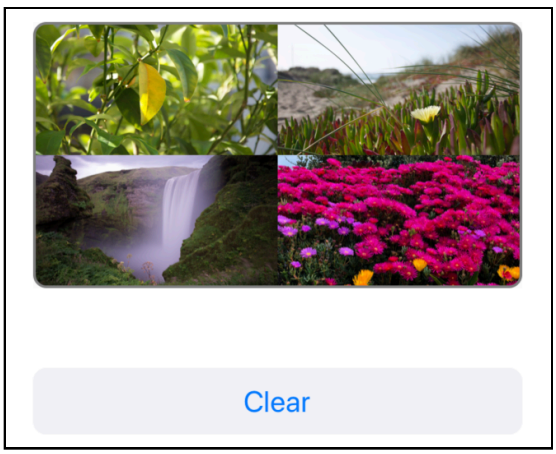
1. Get the current list of selected images and append any new images to it.
2. You use a special `subscribe` overload to pipe the new image list straight to the `images` subject.
3. You store the new subscription in `subscriptions`. However, the subscription will end whenever the user dismisses the presented view controller.

Now, run the app and let's try out the newly added code. Tap on the + button and you will see the system Photos access dialogue pop-up on-screen. Tap **OK**:




This will reload the collection view with the default photos included with the iOS Simulator, or your own photos if you're testing on your device:


Tap a few of those. They'll flash to indicate they've been added to the collage. Then, tap to go back to the main screen you will see your new collage in full glory:




# Wrapping a callback function as a future

In a playground, you might play with subjects and publishers and be able to design everything exactly as you like it, but in day-to-day iOS code, you will interact with various Cocoa APIs, such as accessing the Camera Roll, reading the device's sensors or interacting with some database.

Later in this book, you will learn how to create your own custom publishers. However, in many cases adding a subject to an existing Cocoa class is enough to plug its functionality in your Combine workflow.

In this part of the chapter, you will work on a new custom type called `PhotoWriter`

which will allow you to save the user's collage to disk. You will use the callback-based Photos API to do the saving and a future to allow other types to subscribe to the operation result.

Open **Utility/PhotoWriter.swift**, which contains an empty `PhotoWriter` class, and add the following static function to it:

```
static func save(_ image: UIImage) -> Future<String,
PhotoWriter.Error> {
  return Future { resolve in

  }
}
```

This function will try to asynchronously store the given image on disk and return a future that this API's consumers will subscribe to.

You'll use the closure-based `Future` initializer to return a ready-to-go future which will execute the code in the provided closure once initialized.

Let's start fleshing out the future's logic by inserting the following code inside the closure:

```
do {

} catch {
  resolve(.failure(.generic(error)))
}
```

This is a pretty good start. You will perform the saving inside the `do` and, should it throw an error, you'll resolve the future with a failure.

Since you don't know the exact errors that could be thrown while saving the photo, you just take the thrown error and wrap it as a `PhotoWriter.Error.generic` error.

Now, for the real "meat" of the function: Insert the following inside the `do` body:

```
// 1
try PHPhotoLibrary.shared().performChangesAndWait {
  // 2
  let request =
PHAssetChangeRequest.creationRequestForAsset(from: image)

  // 3
  guard let savedAssetID =
    request.placeholderForCreatedAsset?.localIdentifier else {
    return resolve(.failure(.couldNotSavePhoto))
  }
```

```
    // 4
    resolve(.success(savedAssetID))
  }
```

Here, you use `PHPhotoLibrary.performChangesAndWait(_)` to access the Photos library synchronously. The future's closure is itself executed asynchronously, so don't worry about blocking the main thread. With this, you'll perform the following changes from within the closure:

1. First, you create a request to store `image`.
2. Then, you attempt to get the newly-created asset's identifier via `request.placeholderForCreatedAsset?.localIdentifier`.
3. If the creation has failed and you didn't get an asset id back, you resolve the future with a `PhotoWriter.Error.couldNotSavePhoto` error.
4. Finally, in case you got back a `savedAssetID`, you resolve the future with success.

That's pretty much everything you need to wrap a callback function, resolve with a failure if you get back an error or resolve with success in case you have some result to return!

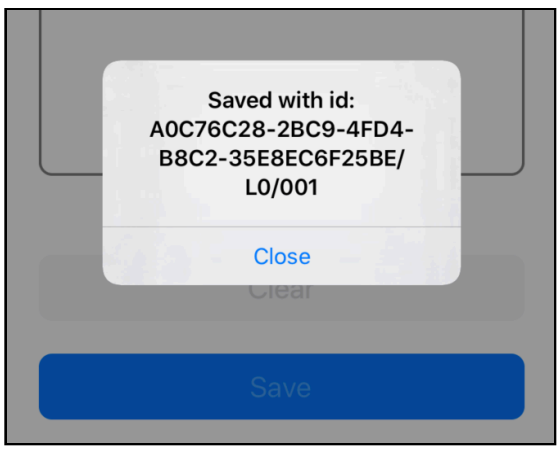
Now, you can use `PhotoWriter.save(_)` to save the current collage when the user taps **Save**. Open **MainViewController.swift** and at the bottom of `actionSave()` append:

```
// 1
PhotoWriter.save(image)
  .sink(receiveCompletion: { [unowned self] completion in
    // 2
    if case .failure(let error) = completion {
      self.showMessage("Error", description:
error.localizedDescription)
    }
    self.actionClear()
  }, receiveValue: { [unowned self] id in
    // 3
    self.showMessage("Saved with id: \(id)")
  })
  .store(in: &subscriptions)
```

1. You subscribe the `PhotoWriter.save(_)` future by using `sink(receiveCompletion:receiveValue:)`.
2. In case of completion with a failure, you call into `showMessage(_:description:)` to display an error alert on-screen.

3. In case you get back a value — the new asset id — you use `showMessage(_:description:)` to let the user know their collage is saved successfully.

Run the app one more time, pick a couple of photos and tap **Save**. This will call into your shiny, new publisher and, upon saving the collage, will display an alert like so:




## A note on memory management

Here is a good place for a quick side-note on memory management with Combine. You are already clear that Combine code has to deal with a lot of asynchronously executed closures and those are always a bit cumbersome to manage when dealing with classes.

When you write your own custom Combine code, you might be dealing predominantly with structs, so you won't need to explicitly specify capturing semantics in the closures you use with `map`, `flatMap`, `filter`, etc.

However, when you're dealing with UI code, and specifically with UIKit/AppKit related code, you will always need to work with classes like `UIViewController`, `UICollectionController`, `NSFetchedController`, etc.

When writing Combine code, standard rules apply, so you should use the same Swift capture semantics as always:

- If you're capturing an object that could be released from memory, like the presented photos view controller earlier, you should use `[weak self]` or another variable than `self` if you capture another object.
- If you're capturing an object that could not be released, like the main view

controller in that Collage app, you can safely use `[unowned self]`. For example, one that you never pop-out of the navigation stack and is therefore always present.

With that said, let's continue with the next task: Presenting a view controller and fetching back the result via a future.

# Presenting a view controller as a future

This task builds upon two of the tasks you've already completed. Previously you:

- Wrapped a UI-less callback function as a `Future`.
- Manually presented a view controller and subscribed to one of its exposed publishers.

This time, you will use a future to present a new view controller on-screen, wait until the user is done with it and then complete the future — all in one go!

In a new extension on `ViewController`, you will recreate the logic you have in `MainViewController.showMessage(title:description:)` but build it using a Combine future.

To do that, open the placeholder file **UIViewController+Combine.swift** and add the initial skeleton of the new method inside the provided extension:

```
func alert(title: String, text: String?) -> AnyPublisher<Void,
Never> {
  let alertVC = UIAlertController(title: title,
                                  message: text,
                                  preferredStyle: .alert)

}
```

The method returns an `AnyPublisher<Void, Never>` as you are not interested in returning any values but simply in completing the publisher when the user taps **Close**.

You begin by creating an alert controller called `alertVC`. Next, you will present it on-screen and dismiss it when the future completes.

Right after the line `let alertVC = ...`, append:

```
return Future { resolve in
  alertVC.addAction(UIAlertAction(title: "Close",
```

```
                                    style: .default) { _ in
      resolve(.success(()))
    })

    self.present(alertVC, animated: true, completion: nil)
  }
  .handleEvents(receiveCancel: {
    self.dismiss(animated: true)
  })
  .eraseToAnyPublisher()
```

Here, you create a `Future`, and upon subscription you add a **Close** button to the alert and present it on-screen. If the user taps the button, you resolve the future with success.

In case the subscription gets canceled, you dismiss the alert automatically from within `handleEvents(receiveCancel:)`. This code handles the case when you tie the alert subscription to the currently-presented view controller and that controller gets dismissed itself. This will cancel the alert subscription and dismiss that alert as well.

To test this code, replace the code in the existing `showMessage` method in `MainViewController` with:

```
  alert(title: title, text: description)
    .sink(receiveValue: { _ in })
    .store(in: &subscriptions)
```

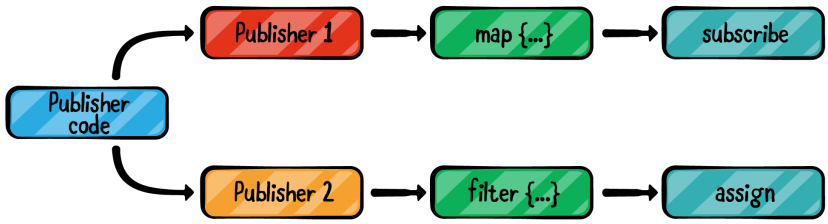
Build and run the app another time and save some collages. You will see the alerts behaving just like before, only this time with your new Combine-ified code.

# Sharing subscriptions

Looking back to the code in `actionAdd()`, you could do a few more things with the images being selected by the user in the presented `PhotosViewController`.

This poses an uneasy question: Should you subscribe multiple times to the same `photos.selectedPhotos` publisher, or do something else?

Turns out, subscribing to the same publisher might have unwanted side effects. If you think about it, you don't know what the publisher is doing upon subscription, do you? It might be creating new resources, making network requests or something else.

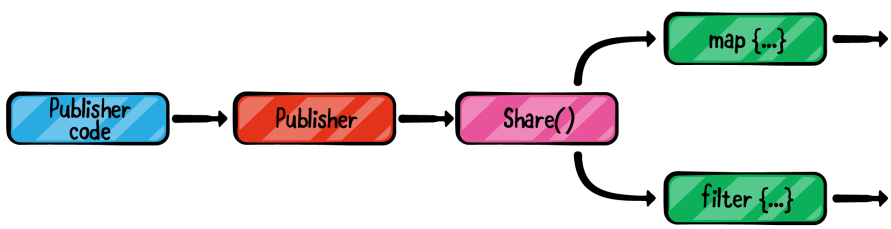



The correct way to go when creating multiple subscriptions to the same publisher is to share the original publisher via the `share()` operator. This wraps the publisher in a class and therefore it can safely emit to multiple subscribers.

Find the line `let newPhotos = photos.selectedPhotos` and replace it with:

```
let newPhotos = photos.selectedPhotos.share()
```

Now, it's safe to create multiple subscriptions to `newPhotos` without being afraid that the publisher is performing side effects multiple times upon each of the subscriptions:




A caveat to keep in mind is that `share()` does not re-emit any values from the shared subscription. For example, if you have two subscriptions on a `share()` and the source publisher emits synchronously upon subscribing that will send the initial output value only to the first subscriber before the second one has the chance to subscribe. (If the source publisher emits asynchronously, that's obviously not an issue.)

A reliable solution to that problem is building your own sharing operator which re-

emits, or *replays*, past values when a new subscriber subscribes. Building your own operators is not complicated at all - you will, in fact, build one called `shareReplay()` in Chapter 18, "Custom Publishers & Handling Backpressure," which will allow you to use share in the way described above.

# Publishing properties with @Published

The Combine framework offers a few property wrappers, a new feature introduced in Swift 5.1. Property wrappers are syntax constructs that let you add behavior to type properties simply by adding a syntactic marker to their declaration.

Combine offers two property wrappers: `@Published` and `@ObservedObject`. In this chapter, you will get to try `@Published` and will cover `@ObservedObject` in a later one.

The `@Published` property wrapper allows you to automatically add a publisher to back your property that will emit a new output value every time you change the original property's value.

The syntax looks like this:

```
struct Person {
  @Published var age: Int = 0
}
```

The property `age` behaves just like any normal property. You can get and set its value imperatively as usual. The compiler will, however, generate another property automatically in your type, with the same accessibility level (private or public), called `$age`.

`$age` is a publisher that can never error out and its output is of the same type as the `age` property. Whenever you modify the value of `age`, `$age` will emit that new value.

> **Note**: `@Published` requires an initial value. You either need to provide a default value for the original property or initialize it when instantiating your type.

This automation of creating publishers for your types allows you to super easily provide consumers of your APIs the ability to subscribe for data changes in real-time.

To try out `@Published`, you will add a new property to `PhotosViewController` to

expose how many photos the user has selected. Open **PhotosViewController.swift** and add the new property below `selectedPhotos`:

```
var selectedPhotosCount = 0
```

Every time the user taps a photo, you will increase the value of the new property to keep track of how many the user has selected. Scroll down to `collectionView(_:didSelectItemAt:)` and find this line: `self.selectedPhotosSubject.send(image)`.

Below that line, add:

```
self.selectedPhotosCount += 1
```

So far, `selectedPhotosCount` is a vanilla `Int` property. You can get and set its value, but you *cannot* subscribe to it.

Go back to the property declaration and add `@Published` like so:

```
@Published var selectedPhotosCount = 0
```

This makes the compiler generate behind the scenes a publisher for the property called `$selectedPhotosCount`.

You can now subscribe to this publisher from your main view controller and display the info about how many photos were selected on the main screen.

Subscribing works like any other publisher, just don't forget to add the `$` prefix to the property name. Open **MainViewController.swift** and scroll to `actionAdd()`. Here, towards the top of the method just after you create `photos`, add:

```
photos.$selectedPhotosCount
  .filter { $0 > 0 }
  .map { "Selected \($0) photos" }
  .assign(to: \.title, on: self)
  .store(in: &subscriptions)
```

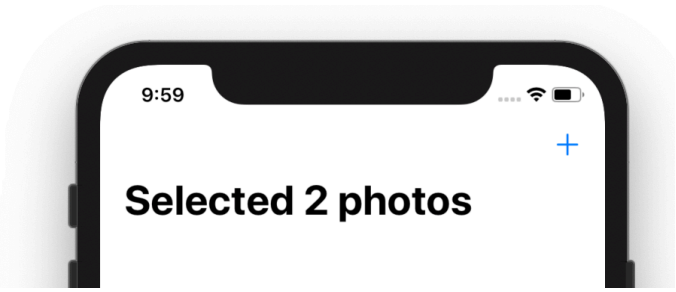
You subscribe `photos.$selectedPhotosCount` and bind it to the view controller's `title` property.

What we call a "binding" here, and also later in this book, is a subscription that "ends" on an `assign(to:on:)` subscriber. The noun "binding" describes very well the nature of such a subscription, or at least better than "assigning". You bind the output of a publisher to a specific instance property on the receiving end.

You provide `assign(to:on:)` a value and a key path, and it updates that key path

with any values it receives. It's a practical subscriber in common use-cases, like binding your model to properties on your views, binding network requests to inputs on your view models and more.

Run the project again and tap a few photos. Then, navigate back and check the main view controller title:




# Operators in practice

Now that you learned about a few useful reactive patterns, it's time to practice some of the operators you covered in previous chapters and see them in action.

## Updating the UI after the publisher completes

Right now, when you tap some photos, you change the main view controller title to display how many photos were selected. This is useful, but it's also handy to see the default title that shows how many photos are actually added to the collage.

Scroll to `actionAdd()` and add another subscription to `newPhotos` just after the existing one:

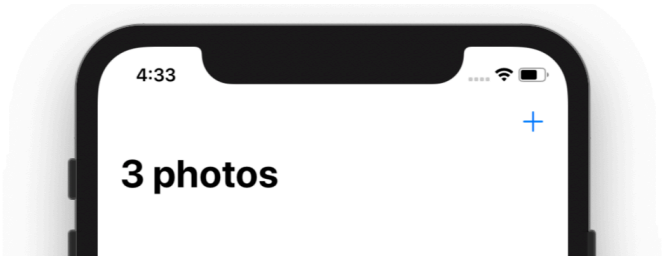
```
newPhotos
  .ignoreOutput()
  .delay(for: 2.0, scheduler: DispatchQueue.main)
  .sink(receiveCompletion: { [unowned self] _ in
    self.updateUI(photos: self.images.value)
  }, receiveValue: { _ in })
  .store(in: &subscriptions)
```

This time, you perform the following:

1. `ignoreOutput()` ignores emitted values, only providing a completion event to the subscriber.

2. `delay(for:scheduler:)` waits a given amount of seconds. This will give few seconds of the previous message saying "X photos selected" to tell the user how many they selected in one go before switching to the total amount of selected photos.
3. `sink(receiveCompletion:)` calls `updateUI(photos:)` to update the UI with the default controller title.

The subscription lets the custom title on-screen (that you display in the previous subscription) for 2 seconds and then invokes `updateUI(photos:)` to reset the title to its default value showing the total number of selected photos:




## Accepting values while a condition is met

Change the code where you share the `selectedPhotos` subscription to the following:

```
let newPhotos = photos.selectedPhotos
  .prefix(while: { [unowned self] _ in
    return self.images.value.count < 6
  })
  .share()
```

You already learned about `prefix(while:)` as one of the powerful Combine filtering operators and here you get to use it in practice. The code above will keep the subscription to `selectedPhotos` alive as long as the total count of images selected is less than six. (I.e. will effectively allow the user to select up to six photos for their collage.)

Adding `prefix(while:)` just before the call to `share()` allows you to filter the incoming values, not only on one subscription, but on all subscriptions that consequently subscribe to `newPhotos`.

Run the app and try adding more than six photos. You will see that after the first six that the main view controller doesn't accept more. In this chapter's challenges, you will refine this by popping out the photos controller automatically when the user reaches the photos limit.

And that's a wrap for this chapter! You did well and deserve a nice pat on the shoulder!

# Challenges

Congratulations on working through this tutorial-style chapter! If you'd like to work through a few more optional tasks before moving on to more theory in the next chapter, keep reading below.

## Challenge 1: Try more operators

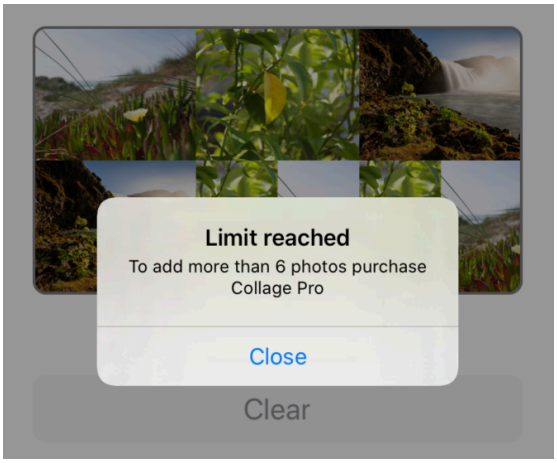
Start by adding yet another filter. Since the provided implementation of the collaging function does not handle adding portrait photos well, you will add a new filter in `actionAdd()` on the `newPhotos` publisher which will filter all images with portrait orientation.

> **Tip**: You can check the image's `size` property and compare the `width` and `height` values to determine if the orientation is a landscape or a portrait.

Once you're finished with your first task, create a new subscription to `newPhotos` which:

1. Ignore all values and only pass a completion event. You can use `ignoreOutput()`, just like earlier in the chapter.
2. Use a `filter` operator to pass the emitted value only in case the current count of images in `images.value` is equal to `6` — the maximum amount of photos in a collage.
3. Use a `flatMap` to display an alert letting the user know they reached the maximum photos and wait until they tap the **Close** button.
4. Use a `sink(receiveCompletion:receiveValue:)` to pop the photos view controller out of the navigation stack.

This subscription should, when the maximum number of photos for a collage is selected, pop the photos view controller automatically and take the user back to the main view controller:




**Note**: When testing this last functionality, pay attention, because any portrait photos you select won't be counted towards the maximum of six. This got me a few times while I was working on this chapter because the system photo selector crops all photos as square thumbnails and you can't really tell if they are in portrait or landscape orientation.

## Challenge 2: PHPhotoLibrary authorization publisher

Open **Utility/PHPhotoLibrary+Combine.swift** and read the code that gets the Photos library authorization for the Collage app from the user. You will certainly notice that the logic is quite straightforward and is based on a "standard" callback API.

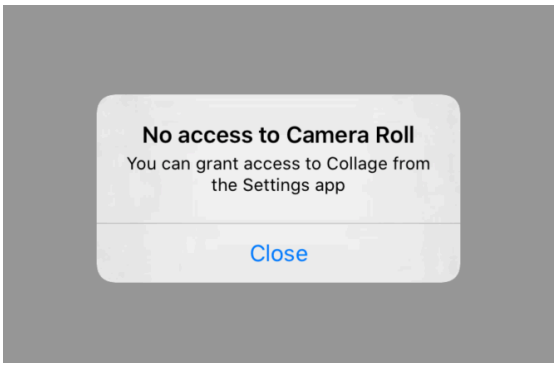
This provides you with a great opportunity to wrap a Cocoa API as a future on your own. For this challenge, add a new static property to `PHPhotoLibrary` called `isAuthorized`, which is of type `Future<Bool, Never>` and allows other types to subscribe to the Photos library authorization status.

You've already done this a couple of times in this chapter and the existing `fetchAuthorizationStatus(callback:)` function should be pretty straight forward to use. Good luck! Should you experience any difficulties along the way, don't forget that you can always peak into the **challenge** folder provided for this chapter and have a look at the example solution.

Finally, don't forget to use the new `isAuthorized` publisher in

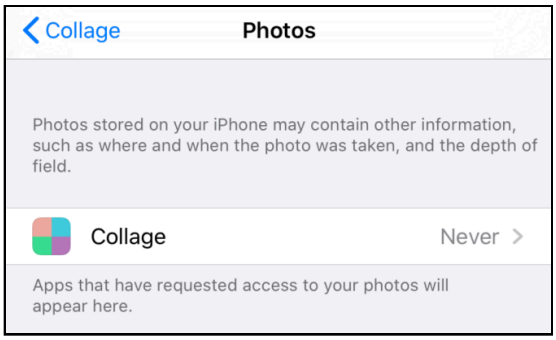
`PhotosViewController` code! That's a good opportunity to exercise receiving values in the main queue as well.

For bonus points, display an error message via your custom alert publisher in case the user doesn't grant access to their photos and navigate back to the main view controller when they tap **Close**.




To play with different authorization states and test your code, open the Settings app on your Simulator or device and navigate to Settings/Collage/Photos/Collage.

Change the authorization status to either "Never" or "Read and Write" to test how your code behaves in those states:




If you made it successfully on your own so far into the challenges, you really deserve an extra round of applause! Either way, one possible solution you can consult with at any time is provided in the challenges folder for this chapter.

# Key points

- In your day-to-day tasks, you'll most likely have to deal with callback or delegate-based APIs. Luckily, those are easily wrapped as futures or publishers by using a

subject.

- Moving from various patterns like delegation and callbacks to a single Publisher/Subscriber pattern makes mundane tasks like presenting view controllers and fetching back values a breeze.
- To avoid unwanted side-effects when subscribing a publisher multiple times, use a shared publisher via the `share()` operator.

# Where to go from here?

That's a wrap for Section II: "Operators!" Starting with the next chapter, you will start looking into various ways Combine integrates with the existing Foundation and UIKit/AppKit APIs and experiment with these integrations in real-life scenarios.

# Section III: Combine in Practice

You now know most of Combine's basics. You learned how publishers, subscribers and subscriptions work and the intertwined relationship between these pieces, as well as how you can use operators to manipulate publishers and their emissions.

While theory is great and definitely useful, practical real-life knowledge is king!

This section is divided into five mini-chapters, each with its own focus on practical approaches for leveraging Combine for specific use-cases. You'll learn how to leverage Combine for networking, how to debug your combine publishers, how to use timers and observe KVO-compliant objects, as well as learn how resources work in Combine.

To wrap up this section, you'll get your hands dirty and build an entire Combine-backed network layer — how exciting!

Specifically, you'll cover:

**Chapter 9: Combine for Networking**: This chapter teaches you how to seamlessly integrate network requests in Combine publisher chains and make your networking easier and more resilient.

**Chapter 10: Debugging Combine**: Debugging asynchronous code being notoriously harder when compared to linear code, Combine helps with a few tools that will greatly help you improve your understanding of the processes at play.

**Chapter 11: Combine Timers**: A niche part of your code, timers are always useful to have around and easy to create in Combine, as you'll learn in this chapter.

**Chapter 12: Key-Value Observing (KVO)**: This chapter shows you how Key-Value Observing, which was historically the best way to get notified when a value changed, now integrates nicely with Combine publishers.

**Chapter 13: Resources in Combine**: Precisely controlling resource usage is a touchy subject, but this chapter will shed light on sharing operators which let you do just that.

**Chapter 14: In Practice: Networking**: In this chapter you will work on a networking API that talks in real-time to the Hacker News servers and along the way exercise using multiple operators and publishers in Foundation.

# Chapter 9: Combine for Networking

By Florent Pillet

As programmers, a lot of what we do revolves around networking. Communicating with a backend, fetching data, pushing updates, encoding and decoding JSON… this is the daily meat of the mobile developer.

Combine offers a few select APIs to help perform common tasks declaratively. These APIs revolve around two key components of modern applications:

- `URLSession`.
- JSON encoding and decoding through the `Codable` protocol.

## URLSession extensions

`URLSession` is the recommended way to perform network data transfer tasks. It offers a modern asynchronous API with powerful configuration options and fully transparent backgrounding support. It supports a variety of operations such as:

- Data transfer tasks to retrieve the content of a URL.
- Download tasks to retrieve the content of a URL and save it to a file.
- Upload tasks to upload files and data to a URL.
- Stream tasks to stream data between two parties.
- Websocket tasks to connect to websockets.

p

Out of these, only the first one, data transfer tasks, exposes a Combine publisher. Combine handles these tasks using a single API with two variants, taking a `URLRequest` or just a `URL`.

Here's a look at how you can use this API:

```
guard let url = URL(string: "https://mysite.com/mydata.json")
else {
  return
}

// 1
let subscription = URLSession.shared
  // 2
  .dataTaskPublisher(for: url)
  .sink(receiveCompletion: { completion in
    // 3
    if case .failure(let err) = completion {
      print("Retrieving data failed with error \(err)")
    }
  }, receiveValue: { data, response in
    // 4
    print("Retrieved data of size \(data.count), response = \
(response)")
  })
```

Here's what's happening with this code:

1. It's crucial that you keep the resulting subscription; otherwise, it gets immediately canceled and the request never executes.
2. You're using the overload of `dataTaskPublisher(for:)` that takes a URL as a parameter.
3. Make sure you always handle errors! Network connections are prone to failure.
4. The result is a tuple with both a `Data` object and a `URLResponse`.

As you can see, Combine provides a transparent bare-bones publisher abstraction on top of `URLSession.dataTask`, only exposing a publisher instead of a closure.

# Codable support

The `Codable` protocol is a modern, powerful and Swift-only encoding and decoding mechanism that you absolutely should know about. If you don't, please do yourself a favor and learn about it from Apple's documentation and tutorials on raywenderlich.com!

Foundation supports encoding to and decoding from JSON through `JSONEncoder` and `JSONDecoder`. You can also use `PropertyListEncoder` and `PropertyListDecoder`, but these are less useful in the context of network requests.

In the previous example, you downloaded some JSON. Of course, you could decode it with a `JSONDecoder`:

```
let subscription = URLSession.shared
  .dataTaskPublisher(for: url)
  .tryMap { data, _ in
    try JSONDecoder().decode(MyType.self, from: data)
  }
  .sink(receiveCompletion: { completion in
    if case .failure(let err) = completion {
      print("Retrieving data failed with error \(err)")
    }
  }, receiveValue: { object in
    print("Retrieved object \(object)")
  })
```

You decode the JSON inside a `tryMap`, which works, but Combine provides an operator to help reduce the boilerplate: `decode(type:decoder:)`.

In the example above, replace the `tryMap` operator with the following lines:

```
.map(\.data)
.decode(type: MyType.self, decoder: JSONDecoder())
```

Unfortunately, since `dataTaskPublisher(for:)` emits a tuple, you can't directly use `decode(type:decoder:)` without first using a `map(_:)` that only emits the `Data` part of the result.

The only advantage is that you instantiate the `JSONDecoder` only once, when setting up the publisher, versus creating it every time in the `tryMap(_:)` closure.

# Publishing network data to multiple subscribers

Every time you subscribe to a publisher, it starts doing work. In the case of network requests, this means sending the same request multiple times if multiple subscribers need the result.

Combine, surprisingly, lacks operators to make this easy, as other frameworks have. You could use the `share()` operator, but that's tricky because you need to subscribe all your subscribers before the result comes back.

Besides using a caching mechanism, one solution is to use the `multicast()` operator, which creates a `ConnectablePublisher` that publishes values to a `Subject`. It allows you to subscribe multiple times to the subject, then call the publisher's `connect()` method when you're ready:

```
let url = URL(string: "https://www.raywenderlich.com")!
let publisher = URLSession.shared
// 1
  .dataTaskPublisher(for: url)
  .map(\.data)
  .multicast { PassthroughSubject<Data, URLError>() }

// 2
let subscription1 = publisher
  .sink(receiveCompletion: { completion in
    if case .failure(let err) = completion {
      print("Sink1 Retrieving data failed with error \(err)")
    }
  }, receiveValue: { object in
    print("Sink1 Retrieved object \(object)")
  })

// 3
let subscription2 = publisher
  .sink(receiveCompletion: { completion in
    if case .failure(let err) = completion {
      print("Sink2 Retrieving data failed with error \(err)")
    }
  }, receiveValue: { object in
    print("Sink2 Retrieved object \(object)")
  })

// 4
let subscription = publisher.connect()
```

In this code, you:

1. Create your `DataTaskPublisher`, `map` to its data and then `multicast` it. The closure you pass must return a subject of the appropriate type. Alternately, you can pass an existing subject to `multicast(subject:)`. You'll learn more about `multicast` in Chapter 13: “Resources in Combine.”
2. Subscribe a first time to the publisher. Since it's a `ConnectablePublisher` it won't start working right away.

3. Subscribe a second time.
4. Connect the publisher, when you're ready. It will start working and pushing values to all of its subscribers.

With this code, you send the request one time and share the outcome to the two subscribers.

> **Note**: Make sure to store all of your `Cancellables`; otherwise, they would be deallocated and canceled when leaving the current code scope, which would be immediate in this specific case.

This process remains a bit convoluted, as Combine does not offer operators for this kind of scenario like other reactive frameworks do. In Chapter 18, "Custom Publishers and Handling Backpressure," you'll explore crafting a better solution.

# Key points

- Combine offers a publisher-based abstraction for its `dataTask(with:completionHandler:)` method called `dataTaskPublisher(for:)`.
- You can decode `Codable`-conforming models using the built-in `decode` operator on a publisher that emits `Data` values.
- While there's no operator to share a replay of a subscription with multiple subscribers, you can recreate this behavior using a `ConnectablePublisher` and the `multicast` operator.

# Where to go from here?

Great job on going through this chapter!

If you want to learn more about using `Codable`, you can check out the following resources:

- "*Encoding and Decoding in Swift*" at raywenderlich.com: https://www.raywenderlich.com/3418439-encoding-and-decoding-in-swift
- "*Encoding and Decoding Custom Types*" on Apple's official documentation: https://developer.apple.com/documentation/foundation/archives_and_serialization/encoding_and_decoding_custom_types

# Chapter 10: Debugging Combine

By Florent Pillet

Understanding the event flow in asynchronous programs has always been a challenge. It is particularly the case in the context of Combine, as chains of operators in a publisher may not all emit events at the same time. For example, operators like `throttle(for:scheduler:latest:)` will not emit all events they receive, so you need to understand what's going on.

Combine provides a few operators to help with debugging your reactive flows. Knowing them will help you troubleshoot puzzling situations.

## Printing events

The `print(_:to:)` operator is the first one you should use when you're unsure whether anything is going through your publishers. It's a passthrough publisher which prints a lot of information about what's happening.

Even with simple cases like this one:

```
let subscription = (1...3).publisher
  .print("publisher")
  .sink { _ in }
```

The output is very detailed:

```
publisher: receive subscription: (1...3)
publisher: request unlimited
publisher: receive value: (1)
publisher: receive value: (2)
publisher: receive value: (3)
```

```
publisher: receive finished
```

Here you see that the `print(_:to:)` operators shows a lot of information, as it:

- Prints when it receives a subscription and shows the description of its upstream publisher.
- Prints the subscriber's demand requests so you can see how many items are being requested.
- Prints every value emitted by the upstream publisher.
- Finally, prints the completion event.

There is an additional parameter that takes a `TextOutputStream` object. You can use this to redirect strings to print to a logger. You can also add information to the log, like the current date and time, etc. The possibilities are endless!

For example, you can create a simple logger that displays the time interval between each string so you can get a sense of how fast your publisher emits values:

```
class TimeLogger: TextOutputStream {
  private var previous = Date()
  private let formatter = NumberFormatter()

  init() {
    formatter.maximumFractionDigits = 5
    formatter.minimumFractionDigits = 5
  }

  func write(_ string: String) {
    let trimmed =
string.trimmingCharacters(in: .whitespacesAndNewlines)
    guard !trimmed.isEmpty else { return }
    let now = Date()
    print("+\(formatter.string(for:
now.timeIntervalSince(previous))!)s: \(string)")
    previous = now
  }
}
```

It's very simple to use in your code:

```
let subscription = (1...3).publisher
  .print("publisher", to: TimeLogger())
  .sink { _ in }
```

And the result displays the time between each printed line:

```
+0.00111s: publisher: receive subscription: (1...3)
+0.03485s: publisher: request unlimited
+0.00035s: publisher: receive value: (1)
+0.00025s: publisher: receive value: (2)
+0.00027s: publisher: receive value: (3)
+0.00024s: publisher: receive finished
```

As mentioned above, the possibilities are quite endless here.

# Acting on events — performing side effects

Besides printing out information, it is often useful to perform actions upon specific events. We call this **performing side effects**, as actions you take "on the side" don't directly impact further publishers down the stream, but can have an effect like modifying an external variable.

The `handleEvents(receiveSubscription:receiveOutput:receiveCompletion:receiveCancel:receiveRequest:)` (wow, what a signature!) lets you intercept any and all events in the lifecycle of a publisher and then take action at each step.

Imagine you are tracking an issue where a publisher must perform a network request, then emit some data. When you run it, it never receives any data. What's happening? Is the request really working? Do you even listen to what comes back?

Consider this code:

```
let request = URLSession.shared
  .dataTaskPublisher(for: URL(string: "https://
www.raywenderlich.com/")!)

request
  .sink(receiveCompletion: { completion in
    print("Sink received completion: \(completion)")
  }) { (data, _) in
    print("Sink received data: \(data)")
  }
```

You run it and never see anything print. Can you see the issue by looking at the code?

If not, use `handleEvents` to track what's happening. You can insert this operator between the publisher and `sink`:

```
.handleEvents(receiveSubscription: { _ in
  print("Network request will start")
}, receiveOutput: { _ in
  print("Network request data received")
}, receiveCancel: {
  print("Network request cancelled")
})
```

Then, run the code again. This time you see some debugging output:

```
Network request will start
Network request cancelled
```

There! You found it: You forgot to keep the cancellable around. So, the subscription starts but gets canceled immediately. Now, you can fix your code by retaining the cancellable:

```
let subscription = request
  .handleEvents...
```

Then, running your code again, you'll now see it behaving correctly:

```
Network request will start
Network request data received
Sink received data: 153253 bytes
Sink received completion: finished
```

# Using the debugger as a last resort

The last resort operator is one you pull in situations where you really need to introspect things at certain times in the debugger, because nothing else helped you figure out what's wrong.

The first simple operator is `breakpointOnError()`. As the name suggests, when you use this operator, if any of the upstream publishers emits an error, Xcode will break in the debugger to let you look at the stack and, hopefully, find why and where your publisher errors out.

A more complete variant is `breakpoint(receiveSubscription:receiveOutput:receiveCompletion:)`. It allows you to intercept various events and decide on a case-by-case basis whether you want to pause the debugger.

For example, you could break only if certain values pass through the publisher:

```
.breakpoint(receiveOutput: { value in
  return value > 10 && value < 15
})
```

Assuming the upstream publisher emits integer values, but values `11` to `14` should never happen, you can configure `breakpoint` to break only in this case and let you investigate!

You can also conditionally break subscription and completion times, but cannot intercept cancelations like the `handleEvents` operator.

> **Note**: None of the breakpoint publishers will work in playgrounds. You will see an error that execution was interrupted, but it won't drop into the debugger.

# Key points

- Track the lifecycle of a publisher with the `print` operator,
- Create your own `TextOutputStream` to customize the output strings,
- Use the `handleEvents` operator to intercept lifecycle events and perform actions,
- Use the `breakpointOnError` and `breakpoint` operators to break on specific events.

# Where to go from here?

You found out how to track what your publishers are doing, now it's time... for timers! Move on to the next chapter to learn how to trigger events at regular intervals with Combine.

# Chapter 11: Combine Timers

By Florent Pillet

Repeating and non-repeating timers are always useful when coding. Besides executing code asynchronously, you often need to control *when* and *how often* a task should repeat.

Before the `Dispatch` framework was available, developers relied on `RunLoop` to asynchronously perform tasks and implement concurrency. `Timer` (`NSTimer` in Objective-C) could be used to create repeating and non-repeating timers. Then `Dispatch` arrived and with it, `DispatchSourceTimer`.

Although all of the above are capable of creating timers, not all timers are equal in Combine. Read on!

## Using RunLoop

The main thread and any thread you create, preferably using the `Thread` class, can have its own `RunLoop`. Just invoke `RunLoop.current` from the current thread: Foundation would create one for you if needed. Beware, unless you understand how run loops operate — in particular, that you need a loop that runs the run loop — you'll be better off simply using the main `RunLoop` that runs the main thread of your application.

> **Note**: One important note and a red light warning in Apple's documentation is that the `RunLoop` class is not thread-safe. You should only call `RunLoop` methods for the run loop of the current thread.

RunLoop implements the Scheduler protocol you'll learn about in Chapter 17, "Combine Schedulers." It defines several methods which are relatively low-level, and the only one that lets you create cancellable timers:

```
let runLoop = RunLoop.main

let subscription = runLoop.schedule(
  after: runLoop.now,
  interval: .seconds(1),
  tolerance: .milliseconds(100)
) {
  print("Timer fired")
}
```

This timer does not pass any value and does not create a publisher. It starts at the date specified in the `after:` parameter with the specified `interval` and `tolerance`, and that's about it. Its only usefulness in relation to Combine is that the `Cancellable` it returns lets you stop the timer after a while.

An example of this could be:

```
runLoop.schedule(after: .init(Date(timeIntervalSinceNow: 3.0)))
{
  cancellable.cancel()
}
```

But all things considered, RunLoop is not the best way to create a timer. You'll be better off using the `Timer` class!

# Using the Timer class

`Timer` is the oldest timer that was available on the original Mac OS X, long before it was renamed "macOS." It has always been tricky to use because of its delegation pattern and tight relationship with RunLoop. Combine brings a modern variant you can directly use as a publisher without all the setup boilerplate.

You can create a repeating timer publisher this way:

```
let publisher = Timer.publish(every: 1.0, on: .main,
in: .common)
```

The two parameters `on` and `in` determine:

- Which RunLoop your timer attaches to. Here, the main thread's RunLoop.

- Which run loop mode(s) the timer runs in. Here, the default run loop mode.

Unless you understand how a run loop operates, you should stick with these default values. Run loops are the basic mechanism for asynchronous event source processing in macOS, but their API is a bit cumbersome. You can get a `RunLoop` for any `Thread` that you create yourself or obtain from Foundation by calling `RunLoop.current`, so you could write the following as well:

```
let publisher = Timer.publish(every: 1.0, on: .current,
in: .common)
```

> **Note**: Running this code on a Dispatch queue other than `DispatchQueue.main` may lead to unpredictable results. The Dispatch framework manages its threads without using run loops. Since a run loop requires one of its `run` methods to be called to process events, you would never see the timer fire on any queue other than the main one. Stay safe and target `RunLoop.main` for your Timers.

The publisher the timer returns is a `ConnectablePublisher`. It's a special variant of `Publisher` that won't start firing upon subscription until you explicitly call its `connect()` method. You can also use the `autoconnect()` operator which automatically connects when the first subscriber subscribes.

> **Note**: You'll learn more about connectable publishers in Chapter 13: "Resources in Combine."

Therefore, the best way to create a publisher that will start a timer upon subscription is to write:

```
let publisher = Timer
  .publish(every: 1.0, on: .main, in: .common)
  .autoconnect()
```

The timer repeatedly emits the current date, its `Publisher.Output` type being a `Date`. You can make a timer that emits increasing values by using the `scan` operator. For example:

```
let subscription = Timer
  .publish(every: 1.0, on: .main, in: .common)
  .autoconnect()
```

```
    .scan(0) { counter, _ in counter + 1 }
    .sink { counter in
      print("Counter is \(counter)")
    }
```

There is an additional `Timer.publish()` parameter you didn't see here: `tolerance`. It specifies the acceptable deviation from the duration you asked for, as a `TimeInterval`. But note that using a value lower than your `RunLoop`'s `minimumTolerance` may not produce the expected results.

# Using DispatchQueue

You can use a dispatch queue to generate timer events. While the Dispatch framework has a `DispatchTimerSource` event source, Combine surprisingly doesn't provide a timer interface to it. Instead, you're going to use an alternative method to generate timer events in your queue. This can be a bit convoluted, though:

```
let queue = DispatchQueue.main

// 1
let source = PassthroughSubject<Int, Never>()

// 2
var counter = 0

// 3
let cancellable = queue.schedule(
  after: queue.now,
  interval: .seconds(1)
) {
  source.send(counter)
  counter += 1
}

// 4
let subscription = source.sink {
  print("Timer emitted \($0)")
}
```

In the above code, you:

1. Create a Subject you will send timer values to.
2. Prepare a counter. You'll increment it every time the timer fires.
3. Schedule a repeating action on the selected queue every second. The action starts immediately.

4. Subscribe to the Subject to get the timer values.

As you can see, this is not pretty. It would help to move this code to a function and pass both the interval and the start time.

# Key points

- Create timers using good old `RunLoop` class if you have Objective-C code nostalgia,
- Use `Timer.publish` to obtain a publisher which generates values at given intervals on the specified `RunLoop`,
- Use `DispatchQueue.schedule` for modern timers emitting events on a dispatch queue.

# Where to go from here?

In Chapter 18, "Custom Publishers & Handling Backpressure," you'll learn how to write your own publishers, and you'll create an alternative timer publisher using `DispatchSourceTimer`.

But don't hurry! There is plenty to learn before that, starting with Key-Value Observing in the next chapter.

# Chapter 12: Key-Value Observing

By Florent Pillet

Dealing with change is at the core of Combine. Publishers let you subscribe to them to handle asynchronous events. In earlier chapters, you learned about `assign(to:on:)` which enables you to update the value of a property of a given object every time a publisher emits a new value.

But, what about a mechanism to observe changes to single variables?

Combine ships with a few options around this:

- It provides a publisher for any property of an object that is **KVO (Key-Value Observing)**-compliant.
- The `ObservableObject` protocol handles cases where multiple variables could change.

## Introducing publisher(for:options:)

KVO has always been an essential component of Objective-C. A large number of properties from Foundation, UIKit and AppKit classes are KVO-compliant. Therefore, you can observe their changes using the KVO machinery.

It's easy to observe KVO-compliant properties. Here is an example using an `OperationQueue` (a class from Foundation):

```
let queue = OperationQueue()

let subscription = queue.publisher(for: \.operationCount)
  .sink {
```

```
    print("Outstanding operations in queue: \($0)")
  }
```

Every time you add a new operation to the queue, its `operationCount` increments, and your sink receives the new count. When the queue has consumed an operation, the count decrements and again, your sink receives the updated count.

There are many other framework classes exposing KVO-compliant properties. Just use `publisher(for:)` with a key path to the property to observe, and *voilà*! You get a publisher capable of emitting value changes. You'll learn more about this and available options later in this chapter.

> **Note**: There is no central list of KVO-compliant properties throughout the frameworks. The documentation for each class usually indicates which properties are KVO-compliant. But sometimes the documentation can be sparse, and you'll only find a quick note in the documentation for some of the properties.

# Preparing and subscribing to your own KVO-compliant properties

You can also use Key-Value Observing in your own code, provided that:

- Your objects are classes (not structs) and inherit from `NSObject`,
- You mark the properties to make observable with the `@objc dynamic` attributes.

Once you do this, your object and the properties you marked become KVO-compliant and can be observed with Combine!

> **Note**: While the Swift language doesn't directly support KVO, marking your properties `@objc dynamic` forces the compiler to generate hidden methods that trigger the KVO machinery. Describing this machinery is out of the scope of this book. Suffice to say the machinery heavily relies on specific methods from the `NSObject` protocol, which explains why your objects need to inherit from it.

Try an example in a playground:

```
// 1
class TestObject: NSObject {
  // 2
  @objc dynamic var integerProperty: Int = 0
}

let obj = TestObject()

// 3
let subscription = obj.publisher(for: \.integerProperty)
  .sink {
    print("integerProperty changes to \($0)")
  }

// 4
obj.integerProperty = 100
obj.integerProperty = 200
```

In the above code, you:

1. Create a class that inherits the `NSObject` protocol. This is required for KVO.
2. Mark any property you want to make observable as `@objc dynamic`.
3. Create and subscribe to a publisher observing the `integerProperty` property of `obj`.
4. Make a couple of updates.

When running this code in a playground, can you guess what the debug console displays?

You may be surprised, but here is the display you obtain:

```
integerProperty changes to 0
integerProperty changes to 100
integerProperty changes to 200
```

You first get the initial value of `integerProperty`, which is `0`, then you receive the two changes. You can avoid this initial value if you're not interested in it — read on to find out how!

Did you notice that in `TestObject` you are using a plain Swift type (`Int`) and that KVO, which is an Objective-C feature, still works? KVO will work fine with any Objective-C type and with any Swift type *bridged to Objective-C*. This includes all the native Swift types as well as arrays and dictionaries, provided their values are all bridgeable to Objective-C.

Try it! Add a couple more properties to `TestObject`:

```
@objc dynamic var stringProperty: String = ""
@objc dynamic var arrayProperty: [Float] = []
```

As well as subscriptions to their publishers:

```
let subscription2 = obj.publisher(for: \.stringProperty)
  .sink {
    print("stringProperty changes to \($0)")
  }

let subscription3 = obj.publisher(for: \.arrayProperty)
  .sink {
    print("arrayProperty changes to \($0)")
  }
```

And finally, some property changes:

```
obj.stringProperty = "Hello"
obj.arrayProperty = [1.0]
obj.stringProperty = "World"
obj.arrayProperty = [1.0, 2.0]
```

You'll see both initial values and changes appear in your debug area. Nice!

If you ever use a pure-Swift type that isn't bridged to Objective-C though, you'll start running into trouble:

```
struct PureSwift {
  let a: (Int, Bool)
}
```

Then, add a property to `TestObject`:

```
@objc dynamic var structProperty: PureSwift = .init(a:
(0,false))
```

You'll immediately see an error in Xcode, stating that "Property cannot be marked `@objc` because its type cannot be represented in Objective-C." Here, you reached the limits of Key-Value Observing.

> **Note**: Be careful when observing changes on system frameworks objects. Make sure the documentation mentions the property is observable because you can't have a clue by just looking at a system object's property list. This is true for

> Foundation, UIKit, AppKit, etc. Historically, properties had to be made "KVO-aware" to be observable.

## Observation options

The full signature of the method you are calling to observe changes is `publisher(for:options:)`. The `options` parameter is an option set with four values: `.initial`, `.prior`, `.old` and `.new`. The default is `[.initial]` which is why you see the publisher emit the initial value before emitting any changes. Here is a breakdown of the options:

- `.initial` emits the initial value.
- `.prior` emits both the *previous* and the *new* value when a change occurs.
- `.old` and `.new` are unused in this publisher, they both do nothing (just let the new value through).

If you don't want the initial value, you can simply write:

```
obj.publisher(for: \.stringProperty, options: [])
```

If you specify `.prior`, you'll get two separate values every time a change occurs. Modifying the `integerProperty` example:

```
let subscription = obj.publisher(for: \.integerProperty,
options: [.prior])
```

You would now see the following in the debug console for the `integerProperty` subscription:

```
integerProperty changes to 0
integerProperty changes to 100
integerProperty changes to 100
integerProperty changes to 200
```

The property first changes from `0` to `100`, so you get two values: `0` and `100`. Then, it changes from `100` to `200` so you again get two values: `100` and `200`.

# ObservableObject

Combine's `ObservableObject` protocol works on Swift objects, not just on objects derived from `NSObject`. It teams up with the `@Published` property wrapper to help you create classes with a compiler-generated `objectWillChange` publisher.

It saves you from writing a lot of boilerplate and allows creating objects which self-monitor their own properties and notify when any of them will change.

Here is an example:

```
class MonitorObject: ObservableObject {
  @Published var someProperty = false
  @Published var someOtherProperty = ""
}

let object = MonitorObject()
let subscription = object.objectWillChange.sink {
  print("object will change")
}

object.someProperty = true
object.someOtherProperty = "Hello world"
```

The `ObservableObject` protocol conformance makes the compiler automatically generate the `objectWillChange` property. It is an `ObservableObjectPublisher` which emits `Void` items and `Never` fails.

You'll get `objectWillChange` firing every time one of the object's `@Published` variables change. Unfortunately, you can't know *which* property actually changed. This is designed to work very well with SwiftUI which coalesces events to streamline screen updates.

# Key points

- Key-Value Observing mostly relies on the Objective-C runtime and methods of the `NSObject` protocol.
- Many Objective-C classes in Apple frameworks offer some KVO-compliant properties.
- You can make your own properties observable provided they are in a `class` inheriting `NSObject`, and marked with the `@objc dynamic` attributes.

- You can also inherit from `ObservableObject` and use `@Published` for your properties. The compiler-generated `objectWillChange` publisher triggers every time one of the `@Published` properties changes (but doesn't tell you which one changed).

## Where to go from here?

Observing is a lot of fun, but sharing is caring! Keep reading to learn about Resources in Combine, and how you can save them by sharing them!

# Chapter 13: Resources in Combine

By Florent Pillet

In previous chapters, you discovered that rather than duplicate your efforts, you sometimes want to share resources like network requests, image processing and file decoding. Anything resource-intensive that you can avoid repeating multiple times is worth looking into. In other words, you should *share* the outcome of a single resource – the values a publisher's work produces – between multiple subscribers rather than duplicate that outcome.

Combine offers two operators for you to manage resources: The `share()` operator and the `multicast(_:)` operator.

## The share() operator

The purpose of this operator is to let you obtain a publisher by **reference** rather than by **value**. Publishers are usually structs: When you pass a publisher to a function or store it in several properties, Swift copies it several times. When you subscribe to each of the copies, the publisher can only do one thing: Start the work it's designed to do and deliver the values.

The `share()` operator returns an instance of the `Publishers.Share` class. This new publisher "shares" the upstream publisher. It will subscribe to the upstream publisher once, with the first incoming subscriber. It will then relay the values it receives from the upstream publisher to this subscriber and to all those that subscribe after it.

> **Note**: New subscribers will only receive values the upstream publisher emits *after* they subscribe. There's no buffering or replay involved. If a subscriber

> subscribes to a shared publisher after the upstream publisher has completed, that new subscriber only receives the completion event.

To put this concept into practice, imagine you're performing a network request, like you learned how to do in Chapter 9, "Combine for Networking." You want multiple subscribers to receive the result without requesting multiple times. Your code would look something like this:

```
let shared = URLSession.shared
  .dataTaskPublisher(for: URL(string: "https://
www.raywenderlich.com")!)
  .map(\.data)
  .print("shared")
  .share()

print("subscribing first")

let subscription1 = shared.sink(
  receiveCompletion: { _ in },
  receiveValue: { print("subscription1 received: '\($0)'") }
)

print("subscribing second")

let subscription2 = shared.sink(
  receiveCompletion: { _ in },
  receiveValue: { print("subscription2 received: '\($0)'") }
)
```

The first subscriber triggers the "work" (in this case, performing the network request) of `share()`'s upstream publisher. The second subscriber will simply "connect" to it and receive values at the same time as the first.

Running this code in a playground, you'd see an output similar to:

```
subscribing first
shared: receive subscription: (DataTaskPublisher)
shared: request unlimited
subscribing second
shared: receive value: (153217 bytes)
subscription1 received: '153217 bytes'
subscription2 received: '153217 bytes'
shared: receive finished
```

Using the `print` operator's output, you can see that:

- The first subscription triggers a subscription to the `DataTaskPublisher`.

- The second subscription doesn't change anything: The publisher keeps running. No second request goes out.
- When the request completes, the publisher emits the resulting data to both subscribers then completes.

To verify that the request is only made once, you could comment out the `share()` line and the output would look similar to this:

```
subscribing first
shared: receive subscription: (DataTaskPublisher)
shared: request unlimited
subscribing second
shared: receive subscription: (DataTaskPublisher)
shared: request unlimited
shared: receive value: (153217 bytes)
subscription1 received: '153217 bytes'
shared: receive finished
shared: receive value: (153217 bytes)
subscription2 received: '153217 bytes'
shared: receive finished
```

You can clearly see that when the `DataTaskPublisher` is not shared, it receives two subscriptions! And in this case, performs the request twice.

But there's a problem: What if the second subscriber comes after the request has completed? You could simulate this case by delaying the second subscription. Don't forget to uncomment `share()` if you're following along in a playground:

```
var subscription2: AnyCancellable? = nil

DispatchQueue.main.asyncAfter(deadline: .now() + 5) {
  print("subscribing second")

  subscription2 = shared.sink(
    receiveCompletion: { print("subscription2 completion \($0)")
},
    receiveValue: { print("subscription2 received: '\($0)'") }
  )
}
```

Running this, you'd see that subscription2 receives nothing if the delay is longer than the time it takes for the request to complete:

```
subscribing first
shared: receive subscription: (DataTaskPublisher)
shared: request unlimited
shared: receive value: (153217 bytes)
subscription1 received: '153217 bytes'
```

```
shared: receive finished
subscribing second
subscription2 completion finished
```

By the time `subscription2` is created, the request has already completed and the resulting data has been emitted. How can you make sure *both* subscriptions receive the request result?

# The multicast(_:) operator

To share a single subscription to a publisher and *replay* the values to new subscribers even after the upstream publisher has completed, you need something like a `shareReplay()` operator. Unfortunately, this operator is not part of Combine. However, you'll learn how to create one in Chapter 18, "Custom Publishers and Handling Backpressure."

In Chapter 9, "Combine for Networking," you used `multicast(_:)`. This operator builds on `share()` and uses a `Subject` of your choice to publish values to subscribers.

The unique characteristic of `multicast(_:)` is that the publisher it returns is a `ConnectablePublisher`. What this means is it won't subscribe to the upstream publisher until you call its `connect()`. This leaves you ample time to set up all the subscribers you need before letting it connect to the upstream publisher and start the work.

To adjust the previous example to use `multicast(_:)` you could write:

```
// 1
let subject = PassthroughSubject<Data, URLError>() // 2

// 2
let multicasted = URLSession.shared
  .dataTaskPublisher(for: URL(string: "https://
www.raywenderlich.com")!)
  .map(\.data)
  .print("shared")
  .multicast(subject: subject)

// 3
let subscription1 = multicasted
  .sink(
    receiveCompletion: { _ in },
    receiveValue: { print("subscription1 received: '\($0)'") }
  )
```

```
let subscription2 = multicasted
  .sink(
    receiveCompletion: { _ in },
    receiveValue: { print("subscription2 received: '\($0)'") }
  )

// 4
multicasted.connect()

// 5
subject.send(Data())
```

Here's what this code does:

1. Prepares a subject, which relays the values and completion event the upstream publisher emits.
2. Prepares the multicasted publisher, using the above `subject`.
3. Subscribes to the shared — i.e., *multicasted* — publisher, like earlier in this chapter.
4. Instructs the publisher to **connect** to the upstream publisher.
5. Sends empty data just to test that both subscriptions receive the data.

This effectively starts the work, but only after you've had time to set up all your subscriptions. This way, you make sure no subscriber will miss the downloaded data.

The resulting output, if you run this in a playground, would be:

```
subscribing first
shared: receive subscription: (DataTaskPublisher)
shared: request unlimited
subscription1 received: '0 bytes'
subscription2 received: '0 bytes'
shared: receive cancel
```

> **Note**: A multicast publisher, like all `ConnectablePublishers`, also provides an `autoconnect()` method, which makes it work like `share()`: The first time you subscribe to it, it connects to the upstream publisher and starts the work immediately. This is useful in scenarios where the upstream publisher emits a single value and you can use a `CurrentValueSubject` to share it with subscribers.

Sharing subscription work, specifically for resource-heavy processes such as

networking, is a must for most modern apps. Not keeping an eye on this could result not only in memory issues, but also possibly bombarding your server with a ton of unnecessary network requests.

# Future

While `share()` and `multicast(_:)` give you full-blown publishers, Combine comes with one more way to let you share the result of a computation: `Future`, which you learned about in Chapter 2, "Publishers & Subscribers."

You create a `Future` by handing it a closure which receives a `Promise` argument. You further *fulfill* the promise whenever you have a result available, either successful or failed. Look at an example to refresh your memory:

```
let future = Future<Int, Error> { fulfill in
  do {
    let result = try performSomeWork()
    fulfill(.success(result))
  } catch {
    fulfill(.failure(error))
  }
}
```

This simple synchronous example tries to perform some work and immediately fulfills the promise with the result of a computation, or with the thrown error in case of a failure. What's interesting from a resource perspective is that:

- `Future` is a class.
- Upon creation, it immediately invokes your closure to start computing the result and fulfill the promise as soon as possible.
- It stores the result of the fulfilled `Promise` and delivers it to **current and future** subscribers.

In practice, it means that `Future` is a convenient way to immediately start performing some work (without waiting for subscriptions) while performing work only once and delivering the result to any amount of subscribers.

It's a good candidate to use for when you need to share the single result a network request produces!

> **Note**: Even if you never subscribe to a `Future`, creating it will call your closure

and perform the work. You cannot rely on `Deferred` to defer closure execution until a subscriber comes in, because `Deferred` is a struct and would cause a new `Future` to be created every time there is a new subscriber!

## Key points

- Sharing subscription work is critical when dealing with resource-heavy processes, such as networking.
- Use `share()` when you simply need to share a publisher with multiple subscribers.
- Use `multicast(_:)` when you need fine control over when the upstream publisher starts to work and how values propagate to subscribers.
- Use `Future` to share the single result of a computation to multiple subscribers.

## Where to go from here?

Congratulations for finishing the last theoretical mini-chapter for this section!

You'll wrap up this section by working on a hands-on project, where you'll build a API client to interact with the Hacker News API. Time to move along!

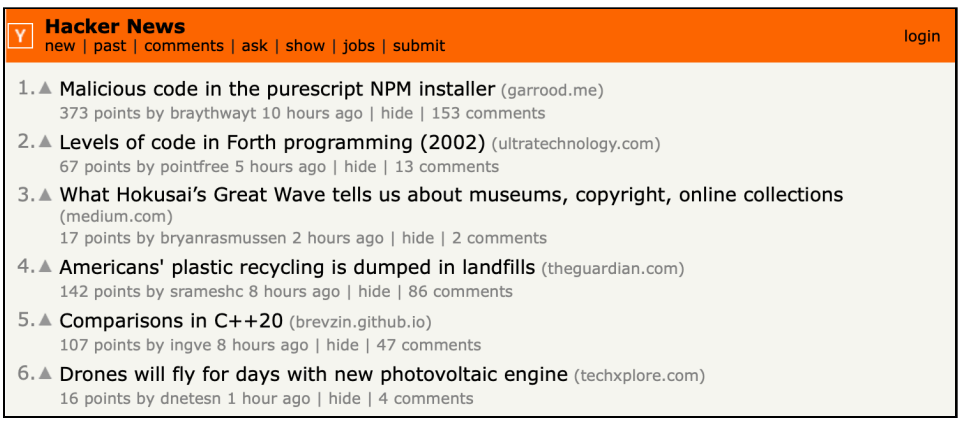
# Chapter 14: In Practice: Project "News"

By Marin Todorov

In the past few chapters, you learned about quite a few practical applications of the Combine integration in Foundation types. You learned how to use `URLSession`'s data task publisher to make network calls, you saw how to observe KVO-compatible objects with Combine and more.

In this chapter, you will *combine* your solid knowledge about operators with some of the Foundation integrations you just discovered and will work through a series of tasks like in the previous "In Practice" chapter. This time around, you will work on building a Hacker News API client.

"Hacker News," whose API you are going to be using in this chapter, is a social news website focused on computers and entrepreneurship. If you haven't already, you can check them out at: https://news.ycombinator.com.



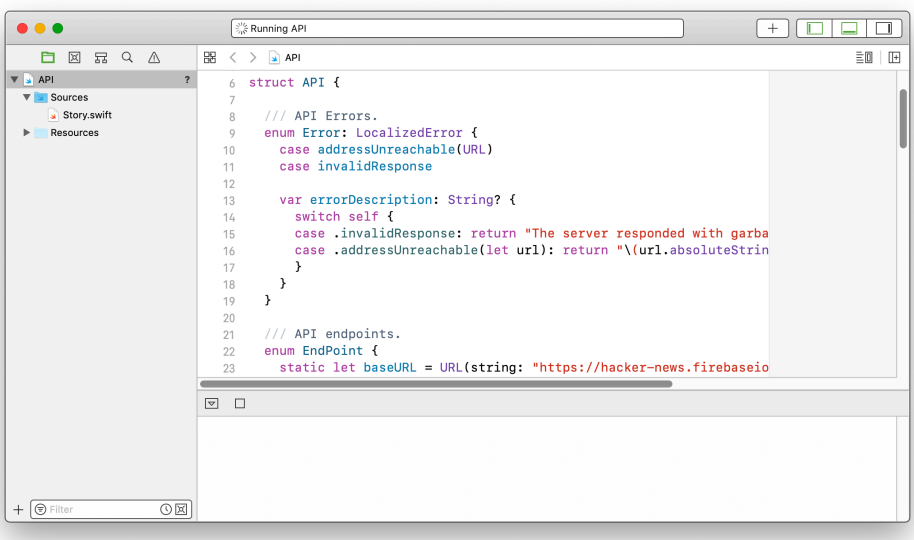

In this chapter, you will work in an Xcode playground focusing only on the API client itself.

In Chapter 15, "Combine with SwiftUI," you will take the completed API and use it to build a real Hacker News reader app by plugging the network layer into a SwiftUI-based user interface. Along the way, you will learn the basics of SwiftUI and how to make your Combine code work with the new declarative Apple framework for building amazing, reactive app UIs.

Without further ado, let's get started!

# Getting started with the Hacker News API

Open the included starter playground **API.playground** in **projects/starter** and peek inside. You will find some simple starter code included to help you hit the ground running and let you focus on Combine code only:




Inside the `API` type, you will find two nested helper types:

- An enum called `Error` which features two custom errors your API will throw in case it cannot reach the server or it cannot decode the server response.
- A second enum called `EndPoint` which contains the URLs of the two API endpoints your type is going to be connecting to.

Further down, you will find the `maxStories` property. You will use this to limit how many of the latest stories your API client will fetch, to help reduce the load on the

Hacker News server, and a `decoder` which you will use to decode JSON data.

Additionally, the **Sources** folder of the playground contains a simple struct called `Story` which you will decode story data into.

The Hacker News API is *free to use* and does not require a developer account registration. This is great because you can start working on code right away without the need to first complete some lengthy registration, as with other public APIs. The Hacker News team wins a ton of karma points!

# Getting a single story

Your first task is to add a method to `API` which will contact the server using the `EndPoint` type to get the correct endpoint URL and will fetch the data about a single story. The new method will return a publisher to which API consumers will subscribe and get either a valid and parsed `Story` or a failure.

Scroll down the playground source code and find the comment saying `// Add your API code here`. Just below that line, insert a new method declaration:

```
func story(id: Int) -> AnyPublisher<Story, Error> {
  return Empty().eraseToAnyPublisher()
}
```

To avoid compilation errors in your playground, you return an `Empty` publisher which completes immediately. As you'll finish building the method body, you'll remove the expression and return your new subscription, instead.

As mentioned, this publisher's output is a `Story` and its failure is the custom `API.Error` type. As you will see later on, in case there are network errors or other mishaps, you will need to convert those into one of the `API.Error` cases to match the expected return type.

Start modeling the subscription by creating a network request to the single-story endpoint of the Hacker News API. Inside the new method, above the `return` statement, insert:

```
URLSession.shared
  .dataTaskPublisher(for: EndPoint.story(id).url)
```

You start by making a request to `Endpoint.story(id).url`. The `url` property of the endpoint contains the complete HTTP URL to request. The single story URL looks like this (with a matching ID): https://hacker-news.firebaseio.com/v0/item/

12345.json (Visit https://bit.ly/2nL2ojS if you'd like to preview the API response.)

Next, to parse JSON on a background thread and keep the rest of the app responsive, let's create a new custom dispatch queue. Add a new property to `API` above the `story(id:)` method like so:

```
private let apiQueue = DispatchQueue(label: "API",
                                     qos: .default,
                                     attributes: .concurrent)
```

You will use this queue to process JSON responses and, therefore, you need to switch your network subscription to that queue. Back in `story(id:)`, add the line **below** calling `dataTaskPublisher(for:)`:

```
.receive(on: apiQueue)
```

Once you've switched to the background queue, you need to fetch the JSON data out of the response. The `dataTaskPublisher(for:)` publisher returns an output of type `(Data, URLResponse)` as a tuple but for your subscription, you need only the data.

Add another line to the method to map the current output to only the data from the resulting tuple:

```
.map(\.data)
```

The output type of this operator is `Data`, which you can feed to a `decode` operator and try converting the response to a `Story`.

Append to the subscription:

```
.decode(type: Story.self, decoder: decoder)
```

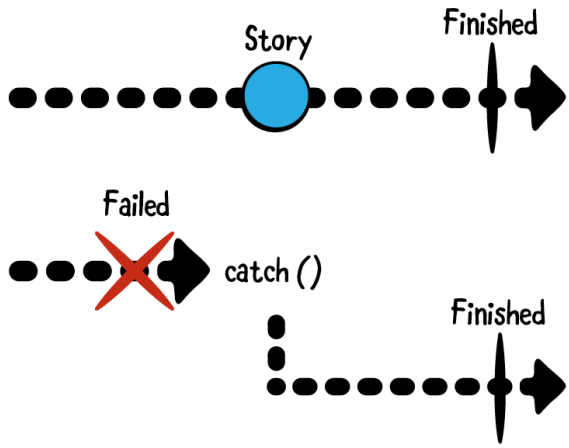
In case anything but a valid story JSON was returned, `decode(...)` will throw an error and the publisher will complete with a failure.

You will learn about error handling in detail in Chapter 16, "Error Handling." In the current chapter, you will use few operators and get a taste of a few different ways to handle errors but you will not go into the nitty-gritty of how things work.

For the current `story(id:)` method, you will return an empty publisher in case things go south for any reason. This is achieved easily by using the `catch` operator. Add to the subscription:

```
.catch { _ in Empty<Story, Error>() }
```

You ignore the thrown error and return `Empty()`. This, as you hopefully still remember, is a publisher that completes immediately without emitting any output values like so:




Handling the upstream errors this way via `catch(_)` allows you to:

- Emit the value and complete if you get back a `Story`.
- Return an `Empty` publisher which completes successfully without emitting any values, in case of a failure.

Next, to wrap up the method code and return your neatly designed publisher, you need to replace the current subscription at the end. Add:

```
.eraseToAnyPublisher()
```

You can now remove the temporary `Empty` you've added before. Find the following line and remove it:

```
return Empty().eraseToAnyPublisher()
```

Your code should now compile with no issues, but just to make sure you haven't missed a step in all the excitement, review your progress so far and make sure your completed code looks like this:

```
func story(id: Int) -> AnyPublisher<Story, Error> {
```

```
    URLSession.shared
      .dataTaskPublisher(for: EndPoint.story(id).url)
      .receive(on: apiQueue)
      .map(\.data)
      .decode(type: Story.self, decoder: decoder)
      .catch { _ in Empty<Story, Error>() }
      .eraseToAnyPublisher()
  }
```

Even though your code compiles, this method still won't produce any output just yet. You're about to take care of that next.

Now you can instantiate `API` and try calling into the Hacker News server.

Scroll down just a bit and find the comment line `// Call the API here`. This is a good spot to make a test API call. Insert the following code:

```
let api = API()
var subscriptions = [AnyCancellable]()
```

Fetch a story by providing a random ID to test with by adding:

```
api.story(id: 1000)
   .sink(receiveCompletion: { print($0) },
         receiveValue: { print($0) })
   .store(in: &subscriptions)
```

You create a new publisher by calling `api.story(id: 1000)` and subscribe to it via `sink(...)` which prints any output values or completion event. To keep the subscription alive until the work is done, you store it in `subscriptions`.

As soon as the playground runs again, it will make a network call to `hacker-news.firebaseio.com` and print the result in the console:

```
61  api.story(id: 1000)
62    .sink(receiveCompletion: { print($0) }) { print($0) }
63    .store(in: &subscriptions)
```

```
How Important is the .com TLD?
by python_kiss
http://www.netbusinessblog.com/2007/02/19/how-important-is-the-dot-com/
-----
finished
```

The returned JSON data from the server is a rather simple structure like this:

```
{
  "by":"python_kiss",
  "descendants":0,
  "id":1000,
  "score":4,
  "time":1172394646,
  "title":"How Important is the .com TLD?",
  "type":"story",
  "url":"http://www.netbusinessblog.com/2007/02/19/how-
important-is-the-dot-com/"
}
```

The `Codable` conformance of `Story` parses and stores the values of the following properties: `by`, `id`, `time`, `title` and `url`.

Once the request completes successfully, you'll see the following output, or a similar output in case you changed the 1000 value in the request, in the console:

```
How Important is the .com TLD?
by python_kiss
http://www.netbusinessblog.com/2007/02/19/how-important-is-the-
dot-com/
-----
finished
```

The `Story` type conforms to `CustomDebugStringConvertible` and it has a custom `debugDescription` that returns the title, author name and story URL neatly ordered, like above.

The output ends with a `finished` completion event. To try what happens in case of an error, replace the id `1000` with `-5` and check the output in the console. You will only see `finished` printed because you caught the error and returned `Empty()`.

Nice work! The first method of the `API` type is completed and you exercised some of the concepts you covered in previous chapters like calling the network and decoding JSON. Additionally you got a gentle introduction to basic dispatch queue switching and some easy error handling. These will be covered in more detail in future chapters.

Despite what an incredibly nice exercise this task was, you're probably hungry for more. So, in the next section, you will dig deeper and lay some serious code down.

# Multiple stories via merging publishers

Getting a single story out of the API server was a relatively straight forward task.

Next, you'll touch on a few more of the concepts you've been learning by creating a custom publisher to fetch multiple stories at the same time.

The new method `mergedStories(ids:)` will get a story publisher for each of the given story `ids` and merge them all together. Add this new method declaration to the `API` type after the `story(id:)` method you implemented earlier:

```
func mergedStories(ids storyIDs: [Int]) -> AnyPublisher<Story,
Error> {

}
```

What this method will essentially do is call `story(id:)` for each of the given `ids` and then flatten the result into a single stream of output values.

First of all, to reduce the number of network calls during development, you will fetch only the first `maxStories` `ids` from the provided list. Start the new method by inserting the following code:

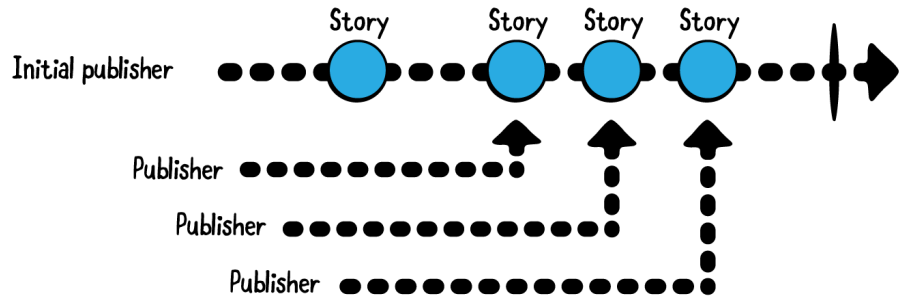
```
let storyIDs = Array(storyIDs.prefix(maxStories))
```

To get started, create the first publisher:

```
precondition(!storyIDs.isEmpty)

let initialPublisher = story(id: storyIDs[0])
let remainder = Array(storyIDs.dropFirst())
```

By using `story(id:)`, you create the `initialPublisher` publisher that fetches the story with the first `id` in the list.

Next, you will use `reduce(_:_:)` from the Swift standard library on the remaining story ids to merge each next story publisher into the initial publisher like so:




To reduce the rest of the stories into the initial publisher add:

```
return remainder.reduce(initialPublisher) { combined, id in

}
```

`reduce(_:_:)` will start with the initial publisher and provide each of the `id`s in the `remainder` array to the closure to process. Insert this code to create a new publisher for the given story `id` in the empty closure, and merge it to the current combined result:

```
return combined
  .merge(with: story(id: id))
  .eraseToAnyPublisher()
```

The final result is a publisher which emits each successfully fetched story and ignores any errors that each of the single-story publishers might encounter.

> **Note**: Congratulations, you just created a custom implementation of the `MergeMany` publisher. Working through the code yourself was not in vain though. You learned about operator composition and how to apply operators like `merge` and `reduce` in a real-world use case.

With the new API method completed, scroll down to this code and comment or delete it to speed up the execution of the playground while testing your newer code:

```
api.story(id: -5)
   .sink(receiveCompletion: { print($0) },
         receiveValue: { print($0) })
```

```
    .store(in: &subscriptions)
```

In place of the just deleted code, insert:

```
api.mergedStories(ids: [1000, 1001, 1002])
   .sink(receiveCompletion: { print($0) },
         receiveValue: { print($0) })
   .store(in: &subscriptions)
```

Let the playground run one more time with your latest code. This time, you should see in the console these three story summaries:

```
How Important is the .com TLD?
by python_kiss
http://www.netbusinessblog.com/2007/02/19/how-important-is-the-
dot-com/
-----

Wireless: India's Hot, China's Not
by python_kiss
http://www.redherring.com/Article.aspx?a=21355
-----

The Battle for Mobile Search
by python_kiss
http://www.businessweek.com/technology/content/feb2007/
tc20070220_828216.htm?campaign_id=rss_daily
-----
finished
```

Another glorious success along your path of learning Combine! In this section, you wrote a method that combines any number of publishers and reduces them to a single one. That's very helpful code to have around, as the built-in `merge` operator can merge only up to 8 publishers. Sometimes, however, you just don't know how many publishers you'll need in advance!

# Getting the latest stories

In this final chapter section, you will work on creating an API method that fetches the list of latest Hacker News stories.

This chapter is following a bit of a pattern. First, you reused your single story method to fetch multiple stories. Now, you are going to reuse the multiple stories method to fetch the list of latest stories.

Add the new empty method declaration to the `API` type as follows:

```
func stories() -> AnyPublisher<[Story], Error> {
  return Empty().eraseToAnyPublisher()
}
```

Like before, you return an `Empty` object to prevent any compilation errors while you construct your method body and publisher.

Unlike before, though, this time your returned publisher's output is a list of stories. You will design the publisher to fetch multiple stories and accumulate them in an array, emitting each intermediary state as the responses come in from the server.

This behavior will allow you to, in the next chapter, bind this new publisher directly to a `List` UI control that will automatically animate the stories live on-screen as they come in from the server.

Begin, as you did previously, by firing off a network request to the Hacker News API. Insert the following in your new method, above the `return` statement:

```
URLSession.shared
  .dataTaskPublisher(for: EndPoint.stories.url)
```

The `stories` endpoint lets you hit the following URL to get the latest story `ids`: https://hacker-news.firebaseio.com/v0/newstories.json.

Again, you need to grab the data component of the emitted result. So, map the output by adding:

```
.map(\.data)
```

The JSON response you will get from the server is a plain list like this:

```
[1000, 1001, 1002, 1003]
```

You need to parse the list as an array of integer numbers and, if that succeeds, you can use the `ids` to fetch the matching stories.

Append to the subscription:

```
.decode(type: [Int].self, decoder: decoder)
```

This will map the current subscription output to an `[Int]` and you will use it to fetch the corresponding stories one-by-one from the server.

Now is the moment, however, to go back to the topic of error handling for a moment. When fetching a single story, you just ignore any errors. But, in `stories()`, let's see how you can do a little more than that.

`API.Error` is the error type to which you will constrain the errors thrown from `stories()`. You have two errors defined as enumeration cases:

- `invalidResponse`: for when you cannot decode the server response into the expected type.
- `addressUnreachable(URL)`: for when you cannot reach the endpoint URL.

Currently, your subscription code in `stories()` can throw two types of errors:

- `dataTaskPublisher(for:)` could throw different variations of a `URLError` when a network problem occurs.
- `decode(type:decoder:)` could throw a decoding error when the JSON doesn't match the expected type.

Your next task is to handle those various errors in a way that would map them to the single `API.Error` type to match the expected failure of the returned publisher.

You will jump the gun yet another time and get a "soft" introduction to another error handling operator. Append this code to your current subscription, after `decode`:

```
.mapError { error -> API.Error in
  switch error {
  case is URLError:
    return Error.addressUnreachable(EndPoint.stories.url)
  default:
    return Error.invalidResponse
  }
}
```

`mapError` handles any errors occurring upstream and allows you to map them into a single error type — similar to how you use `map` to change the type of the output.

In the code above, you switch over any errors and:

- In case `error` is of type `URLError` and therefore occurred while trying to reach the `stories` server endpoint, you return `.addressUnreachable(_)`.
- Otherwise, you return `.invalidResponse` as the only other place where an error could occur. Once successfully fetched, the network response is decoding the JSON data.

With that, you matched the expected failure type in `stories()` and can leave it to the API consumers to handle errors downstream. You will use `stories()` in the next chapter. So, you will do a little more with error handling before you get to Chapter 15, "Error Handling," and dive into the details.

So far, the current subscription fetches a list of `ids` from the JSON API but doesn't do much on top of that. Next, you will use a few operators to filter unwanted content and map the `id` list to the actual stories.

First, filter empty results — in case the API goes bonkers and returns an empty list for its latest stories. Append:

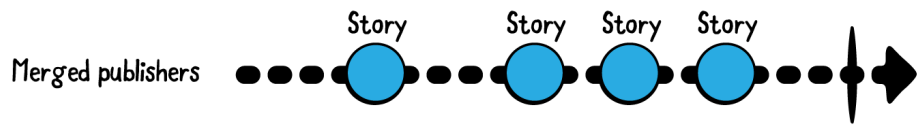
```
.filter { !$0.isEmpty }
```

This will guarantee that downstream operators receive a list of story `ids` with at least one element. This is very handy because, as you remember, `mergedStories(ids:)` has a precondition ensuring that its input parameter is not empty.

To use `mergedStories(ids:)` and fetch the story details, you will flatten all the story publishers by appending a `flatMap` operator:

```
.flatMap { storyIDs in
  return self.mergedStories(ids: storyIDs)
}
```

Merging all the publishers into a single downstream will produce a continuous stream of `Story` values. These are emitted as soon as they are fetched from the network:




You could leave the current subscription as is right now but you'd like to design the API to be easily bindable to a list UI control. This will allow the consumers to simply subscribe `stories()` and assign the result to an `[Story]` property in their view controller or SwiftUI view.

To achieve that, you will need to aggregate the emitted stories and map the subscription to return an ever-growing array — instead of single `Story` values.

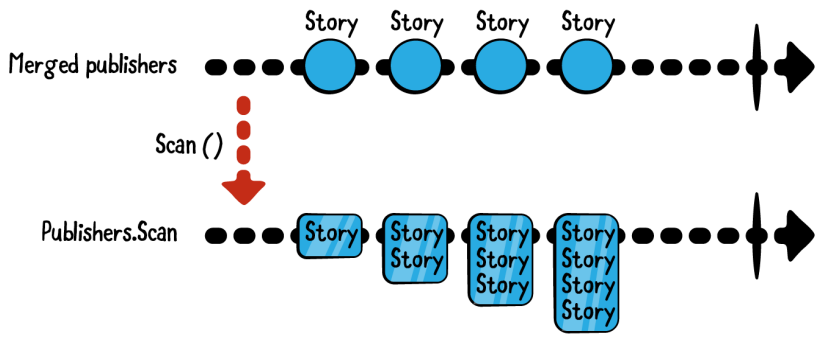
It's time for some serious magic! Remember the `scan` operator from Chapter 3, "Transforming Operators" I know that was some time ago, but, this is the operator that will help you achieve your current task. So, if needed, jump back to that chapter and come back here when refreshed on `scan`.

Append to your current subscription:

```
.scan([]) { stories, story -> [Story] in
  return stories + [story]
}
```

You let `scan(...)` start emitting with an empty array. Each time a new story is being emitted, you append it to the current aggregated result via `stories + [story]`.

This addition to the subscription code changes its behavior so that you get the — sort of — buffered contents each time a new story is fetched from the batch you are working on:




Finally, it can't hurt to sort the stories before emitting output. `Story` conforms to `Comparable` so you don't need to implement any custom sorting. You just need to call `sorted()` on the result. Append:

```
.map { $0.sorted() }
```

Wrap up the current, rather long, subscription by type erasing the returned publisher. Append one last operator:

```
.eraseToAnyPublisher()
```

At this point, you can find the following temporary return statement, and remove it:

```
return Empty().eraseToAnyPublisher()
```

Your playground should now finally compile with no errors. However, it still shows the test data from the previous chapter section. Find and comment out:

```
api.mergedStories(ids: [1000, 1001, 1002])
   .sink(receiveCompletion: { print($0) },
```

```
          receiveValue: { print($0) })
    .store(in: &subscriptions)
```

In its place, insert:

```
api.stories()
    .sink(receiveCompletion: { print($0) },
          receiveValue: { print($0) })
    .store(in: &subscriptions)
```

This code subscribes to `api.stories()` and prints any returned output and completion events.

Once you let the playground run one more time, you should see a dump of the latest Hacker News stories in the console. The list is dumped **iteratively**. Initially, you will see the story fetched first on its own:

```
[
More than 70% of America’s packaged food supply is ultra-
processed
by xbeta
https://news.northwestern.edu/stories/2019/07/us-packaged-food-
supply-is-ultra-processed/
-----]
```

Then, the same one accompanied by a second story:

```
[
More than 70% of America’s packaged food supply is ultra-
processed
by xbeta
https://news.northwestern.edu/stories/2019/07/us-packaged-food-
supply-is-ultra-processed/
-----,
New AI project expects to map all the word’s reefs by end of
next year
by Biba89
https://www.independent.co.uk/news/science/coral-bleaching-ai-
reef-paul-allen-climate-a9022876.html
-----]
```

Then, a list of the same stories plus a third one and so on:

```
[
More than 70% of America’s packaged food supply is ultra-
processed
by xbeta
https://news.northwestern.edu/stories/2019/07/us-packaged-food-
supply-is-ultra-processed/
```

```
-----,
New AI project expects to map all the word's reefs by end of
next year
by Biba89
https://www.independent.co.uk/news/science/coral-bleaching-ai-
reef-paul-allen-climate-a9022876.html
-----,
People forged judges' signatures to trick Google into changing
results
by lnguyen
https://arstechnica.com/tech-policy/2019/07/people-forged-
judges-signatures-to-trick-google-into-changing-results/
-----]
```

Please note, since you're fetching live data from the Hacker News website the stories, what you see in your console **will be different** as more and more stories are added every few minutes.

To see that you are indeed fetching live data, wait a few minutes and re-run the playground. You should see some new stories show up alongside the ones you already saw.

Nice effort working through this somewhat longer section of the chapter! You've completed the development of the Hacker News API client and are ready to move on to the next chapter. There, you will use SwiftUI to build a proper Hacker News reader app.

# Challenges

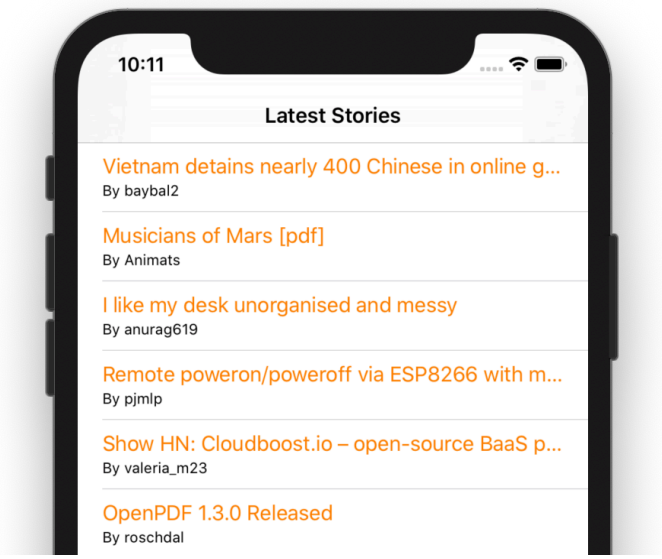
There is nothing to add per se to the API client but you can still play around a little if you'd like to put some more work into this chapter's project.

## Challenge 1: Integrating the API client with UIKit

As already mentioned, in the next chapter, you will learn about SwiftUI and how to integrate it with your Combine code.

In this challenge, try to build an iOS app that uses your completed API client to display the latest stories in a table view. You can develop as many details as you want and add some styling or fun features but the main point to exercise in this challenge is subscribing the `API.stories()` and binding the result to a table view — much like the bindings you worked on in Chapter 8, "In practice - Project 'Collage'."

If you successfully work through the challenge as described, you should see the latest stories "pour in" when you launch the app in the simulator, or on your device:




# Key points

- Foundation includes several publishers that mirror counterpart methods in the Swift standard library and you can even use them interchangeably as you did with `reduce` in this chapter.
- Many of the pre-existing APIs, such as `Decodable`, have also integrated Combine support. This lets you use one standard approach across all of your code.
- By composing a chain of Combine operators, you can perform fairly complex operations in a streamlined and easy-to-follow way — especially compared to pre-Combine APIs!

# Where to go from here?

Congratulations on completing the "Combine in Practice" section! What a ride this was, wasn't it?

You've learned most of what Combine's foundations has to offer, so it's now times to

pull out the big guns in an entire section dedicated to advanced topics in the Combine framework, starting with building an app that uses both SwiftUI *and* Combine.

# Section IV: Advanced Combine

This book is currently in early access release. We will keep you updated as the following chapters become available!

With a huge portion of Combine foundations already in your tool belt, it's time to learn some of the more advanced concepts and topics Combine has to offer on your way to true mastery.

You'll start by learning how to use SwiftUI with Combine to build truly reactive and fluent UI experiences and switch to learn how to properly handle errors in your Combine apps. You'll then learn about schedulers, the core concept behind scheduling work in different execution contexts and follow up with how you can create your own custom publishers and handling the demand of subscribers by understanding backpressure.

Finally, having a slick code base is great, but it doesn't help much if it's not well tested, so you'll wrap up this section by learning how to properly test your new Combine code.

Specifically, you'll cover:

**Chapter 15: In Practice: SwiftUI with Combine**: SwiftUI is the new user interface paradigm from Apple and it's designed for an interplay with Combine on a whole new level. In this chapter you are going to work through a Combine based SwiftUI project.

**Chapter 16: Error Handling**: This chapter will teach about Combine's powerful typed error system, and how you can leverage it to handle errors in your app, and within Combine publishers.

**Chapter 17: Combine Schedulers**: In this chapter you'll learn what Schedulers are, how they relate to RunLoops, Dispatch queues and Threads and how Combine extends Foundation to integrate with them.

**Chapter 18: Custom Publishers & Handling Backpressure**: Creating your own publishers and operators is an advanced topic you'll learn to master in this chapter, while also learning about and experimenting with backpressure management.

**Chapter 19: Testing Combine Code**: This chapter will introduce you to unit testing techniques for use with Combine code. You'll go from testing basics in a playground to applying what you've learned in adding tests to an existing iOS app project.

# Section V: Building a Complete App

This book is currently in early access release. We will keep you updated as the following chapters become available!

Mastery takes practice, and practice you shall!

You've made it through this entire book, an amazing feat by all means. It's time to truly solidify the knowledge you've acquired throughout this chapter and build an entire app using Combine and SwiftUI.

Specifically, you'll cover:

**Chapter 20: In Practice: A Complete Combine App**: You've gained valuable skills throughout this book, and in the last section you picked up some advanced Combine know how too. In this final chapter, the pièce de résistance, you'll build a complete app that applies what you've learned—but the learning is not done yet! Core Data in Combine anyone?

# Conclusion

You're finally here! Congratulations on completing this book, and we hope you enjoyed learning about Combine from the book as much as we've enjoyed making it.

In this book, you've learned about how Combine enables you to write apps in a declarative and expressive way while also making your app reactive to changes as they occur. This makes your app code much more versatile and easier to reason about, along with powerful compositional abilities between different pieces of logic and data.

You started off as a complete Combine beginner, and look at you now; oh, the things you've been through—operators, networking, debugging, error handling, schedulers, custom publishers, testing, and you've even worked with SwiftUI.

This is where we part ways, but we have full confidence in you! We hope you'll continue experimenting with Combine and constantly enhancing your "Combine muscles." As the saying goes—"practice makes perfect."

And like anything new you learn—don't forget to enjoy the ride.

If you have any questions or comments about the projects in this book, please stop by our forums at http://forums.raywenderlich.com.

Thank you again for purchasing this book. Your continued support is what makes the books, tutorials, videos and other things we do at raywenderlich.com possible. We truly appreciate it!

— Florent, Marin, Sandra, Scott, Shai, Tyler and Vicki

The *Combine: Asynchronous Programming with Swift* team