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

# 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.

# 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*

# Table of Contents: Overview

# Table of Contents: Extended

# 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.

# About This Book Sample

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.

We are pleased to offer you this sample from the full *Combine: Asynchronous Programming with Swift* book that will introduce you to these concepts and give you a chance to practice them in our hands-on, step-by-step style.

This sample includes:

**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.

You can get the the complete *Combine: Asynchronous Programming with Swift* book here:

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

The *Combine: Asynchronous Programming with Swift* Team

# 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.

# 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.

# Chapter 2: Publishers & Subscribers

By Scott Gardner

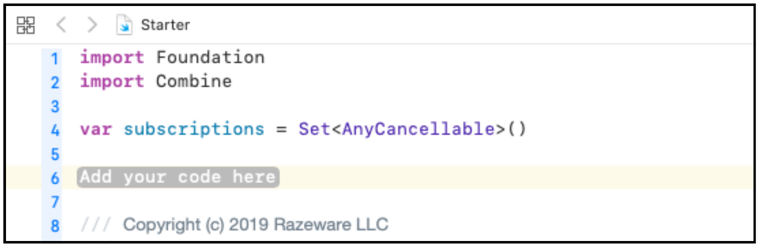
Now that you've learned some of the basic concepts of Combine, it's time to jump in and play with two of Combine's core components — publishers and subscribers.

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:



```
import Foundation
import Combine

var subscriptions = Set<AnyCancellable>()

Add your code here

/// Copyright (c) 2019 Razeware LLC
```

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!
```

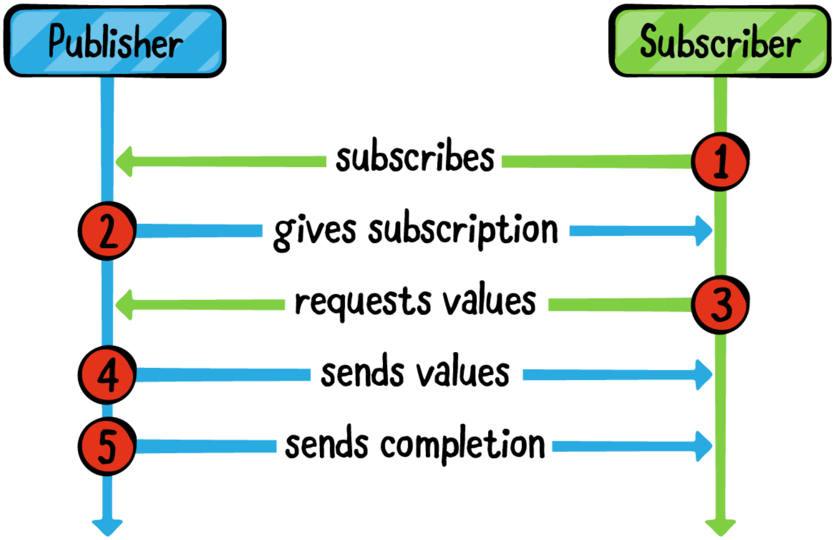
If you don't explicitly call `cancel()` on a subscription, it will continue until the publisher completes, or until normal memory management causes a stored subscription to be deinitialized. At that point it will cancel the subscription for you.

> **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 `Card`s.
- 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.

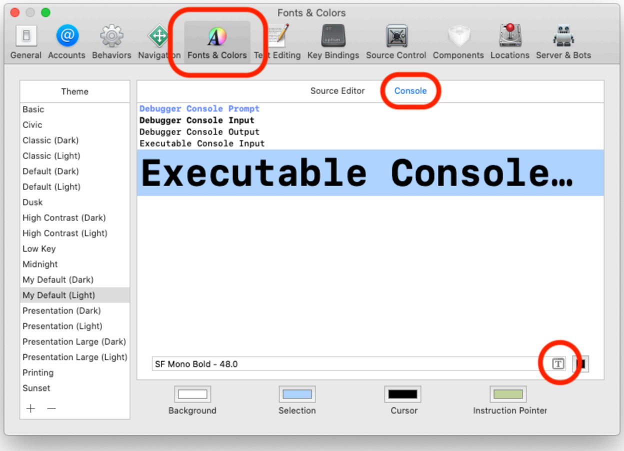
Also in the main playground page, add code immediately after the comment `// Add subscription to dealtHand here` to subscribe to `dealtHand` and handle receiving both values and an error. For received values, print a string containing the results of the hand's `cardString` and `points` properties.

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.

# 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