# Using Combine

Joseph Heck

Version 0.3, 2019-07-07

# Table of Contents

# About This Book

## Supporting this effort

***This is a work in progress.***

The book is being made available at no cost. The content for this book, including sample code and tests is available on GitHub at https://github.com/heckj/swiftui-notes.

If you want to report a problem (typo, grammar, or technical fault), please Open an issue in GitHub. If you are so inclined, feel free to fork the project and send me pull requests with updates or corrections.

I am working through how to make it available to the widest audience while also generating a small amount of money to support the creation of this book, from technical authoring and review through copy editing.

## Author Bio

Joe Heck has broad software engineering development and management experience crossing startups and large companies. He works across all the layers of solutions, from architecture, development, validation, deployment, and operations.

Joe has developed projects ranging from mobile and desktop application development to cloud-based distributed systems. He has established teams, development processes, CI and CD pipelines, and developed validation and operational automation. Joe also builds and mentors people to learn, build, validate, deploy and run software services and infrastructure.

Joe works extensively with and in open source, contributing and collaborating with a wide variety of open source projects. He writes online across a variety of topics at https://rhonabwy.com/.



## Where to get this book

The contents of this book are available as HTML, PDF, and ePub.

There is also an Xcode project (`SwiftUI-Notes.xcodeproj`) available on GitHub. The project includes tests, snippets, and trials used in creating this work.

### Download the project

The project associated with this book requires Xcode v11 (which has been released as beta3 as of this writing) and MacOS 10.14 or later.


# Welcome to Xcode

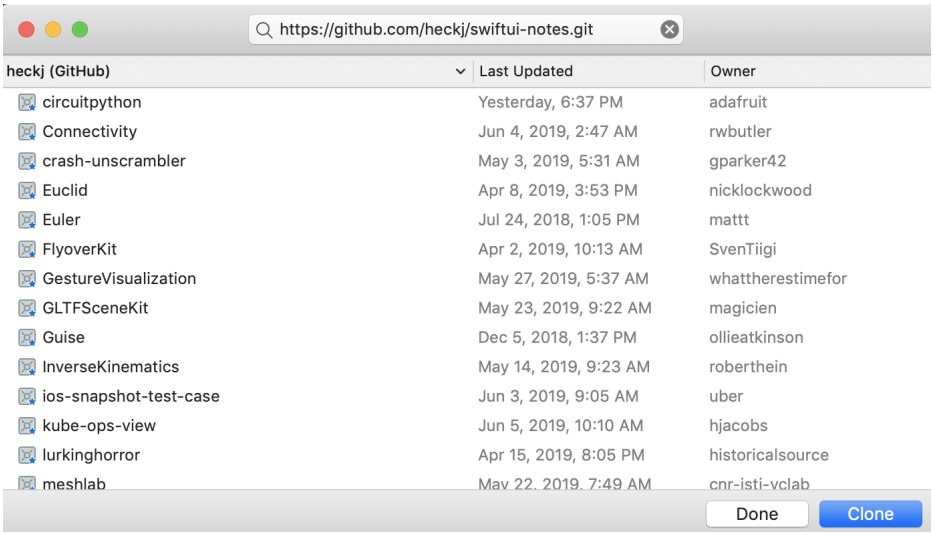
Version 11.0 beta 2 (11M337n)

**Get started with a playground**
Explore new ideas quickly and easily.

**Create a new Xcode project**
Create an app for iPhone, iPad, Mac, Apple Watch, or Apple TV.

**Clone an existing project**
Start working on something from a Git repository.

- From the Welcome to Xcode window, choose **Clone an existing project**
- Enter `https://github.com/heckj/swiftui-notes.git` and click `Clone`




- Choose the `master` branch to check out

# Introduction to Combine

In Apple's words, Combine is:

> a declarative Swift API for processing values over time.

Combine is Apple's take on a functional reactive programming library, akin to RxSwift. RxSwift itself is a port of ReactiveX. Apple's framework uses many of the same functional reactive concepts that can be found in other languages and libraries, applying the staticly-typed nature of Swift to their solution.

If you are already familar with RxSwift there is a pretty good cheat-sheet for translating the specifics between Rx and Combine, built and inspired by the data collected at https://github.com/freak4pc/rxswift-to-combine-cheatsheet.

Another good over is a post Combine: Where's the Beef? by Casey Liss describing how Combine maps back to RxSwift and RxCocoa's concepts, and where it is different.

## Functional reactive programming

Functional reactive programming, also known as data-flow programming, builds on the concepts of functional programming. Where functional programming applies to lists of elements, functional reactive programming is applied to streams of elements. The kinds of functions in functional programming, such as `map`, `filter`, and `reduce` all have analogues that can be applied to streams. In addition, functional reactive programming includes functions to split streams, create pipelines of operations to transform the data within a stream, and merge streams.

There are many parts of the systems we program that can be viewed as asynchronous streams of information - events, objects, or pieces of data. Programming practices defined the Observer pattern for watching a single object, getting notified of changes and updates. If you view this over time, these updates make up a stream of objects. Functional reactive programming, or Combine in this case, allows you to create code that describes what happens when getting data in a stream.

You may want to create logic to watch more than one element that is changing. You may also want to include logic that does additional asynchronous operations, some of which may fail. You may also want to change the content of the streams based on timing, or change the timing of the content. Handling the flow of these event streams, the timing, errors when they happen, and coordinating how a system responds to all those events is at the heart of this kind of programming.

A solution based on functional reactive programming is particularly effective when programming user interfaces. Or more generally for creating pipelines that process data from external sources or rely on asynchronous APIs.

## Combine specifics

Applying these concepts to a strongly typed language like swift is part of what Apple has created in

Combine. Combine embeds the concept of back-pressure, which allows the subscriber to control how much information it gets at once and needs to process. In addition, it supports efficient operation with the notion of streams that are cancellable and driven primarily by the subscriber.

Combine is set up to be composed, and includes affordances to integrate existing code to incrementally support adoption.

Combine is supported by a couple of Apple's other frameworks. SwiftUI is the obvious example that has the most attention, with both subscriber and publisher elements. RealityKit also has publishers that you can use to react to events. And Foundation has a number of Combine specific additions including NotificationCenter, URLSession, and Timer as publishers.

Any asynchronous operation API *can* be leveraged with Combine. For example, you could use some of the APIs in the Vision framework, composing data flowing to it, and from it, by leveraging Combine.

> In this work, I'm going to call a set of composed operations in Combine a **pipeline**. Pipeline is not a term that Apple is (yet?) using in its documentation.

# When to use Combine

Combine fits most naturally when you want to set up a something that is "immediately" reactive to a variety of inputs. User interfaces fit very naturally into this pattern.

The classic examples in functional reactive programming and user interfaces frequently show form validation, where user events such as changing text fields, taps, or mouse-clicks on UI elements make up the data being streamed. Combine takes this quite a bit further, enabling watching of properties, binding to objects, sending and receiving higher level events from UI controls, and supporting integration with almost all of Apple's existing API ecosystem.

Some things you can do with Combine include:

- You can set up pipelines to enable the button for submission only when values entered into the fields are valid.
- A pipeline can also do asynchronous actions (such as checking with a network service) and using the values returned to choose how and what to update within a view.
- Pipelines can also be used to react to a user typing dynamically into a text field and updating the user interface view based on what they're typing.

Combine is not limited to user interfaces. Any sequence of asynchronous operations can be effective as a pipeline, especially when the results of each step flow to the next step. An example of such might be a series of network service requests, followed by decoding the results.

Combine can also be used to define how to handle errors from asynchronous operations. Combine supports doing this by setting up pipelines and merging them together. One of Apple's examples with Combine include a pipeline to fall back to getting a lower-resolution image from a network service when the local network is constrained.

Many of the pipelines you create with Combine will only be a few operations. Even with just a few operations, Combine can still make it much easier to view and understand what's happening when you compose a pipeline.

# Apple's Documentation

The online documentation for Combine can be found at https://developer.apple.com/documentation/combine. Apple's developer documentation is hosted at https://developer.apple.com/documentation/.

## WWDC content

Apple provides video, slides, and some sample code in sessions it's developer conferences. Details on Combine are primarily from WWDC 2019.

A number of these introduce and go into some depth on Combine:

- Introducing Combine
  - PDF of presentation notes
- Combine in Practice
  - PDF of presentation notes

A number of additional WWDC19 sessions mention Combine:

- Modern Swift API Design
- Data Flow Through SwiftUI
- Introducing Combine and Advances in Foundation
- Advances in Networking, Part 1
- Building Collaborative AR Experiences
- Expanding the Sensory Experience with Core Haptics

# Core Concepts

## Publisher, Subscriber

Two key concepts, described in swift with protocols, are **publisher** and **subscriber**.

A publisher provides data. It is described with two associated types: one for Output and one for Failure. A subscriber requests data. It is also described with two associated types, one for Input and one for Failure. When you connect a subscriber to a publisher, both types must match: Output to Input, and Failure to Failure. You can view this as a series of operations on two types in parallel.

```
Publisher source       Subscriber
+--------------+      +--------------+
|        <Output> --> <Input>        |
|       <Failure> --> <Failure>      |
+--------------+      +--------------+
```

Operators are classes that adopt both the Subscriber protocol and Publisher protocol. They support subscribing to a publisher, and sending results to any subscribers.

You can create chains of these together, for processing, reacting, and transforming the data provided by a publisher, and requested by the subscriber.

I'm calling these composed sequences **pipelines**.

```
Publisher source       Operator                          Subscriber
+--------------+      +---------------------------+      +--------------+
|        <Output> --> <Input>      map      <Output> --> <Input>        |
|       <Failure> --> <Failure>  function  <Failure> --> <Failure>      |
+--------------+      +---------------------------+      +--------------+
```

Operators can be used to transform types - both the Output and Failure type. Operators may also split or duplicate streams, or merge streams together. Operators must always be aligned by the combination of Output/Failure types. The compiler will enforce the matching types, so getting it wrong will result in a compiler error (and sometimes a useful *fixit* snippet.)

A simple pipeline, using Combine, might look like:

```
let _ = Just(5) ①
    .map { value -> String in ②
        // do something with the incoming value here
        // and return a string
        return "a string"
    }
    .sink { receivedValue in ③
        // sink is the subscriber and terminates the pipeline
        print("The end result was \(receivedValue)")
    }
```

① The pipeline starts with the publisher `Just`, which responds with the value that its defined with (in this case, the Integer `5`). The output type is <Integer>, and the failure type is <Never>.

② the pipeline then has a `map` operator, which is transforming the value. In this example it is ignoring the published input and returning a string. This is also transforming the output type to <String>, and leaving the failure type still set as <Never>

③ The pipeline then ends with a `sink` subscriber.

When you are viewing a pipeline, or creating one, you can think of it as a sequence of operations linked by the types. This pattern will come in handy when you start constructing your own pipelines. When creating pipelines, you are often selecting operators to help you transform the types, to achieve your end goal. That end goal might be enabling or disabling a user interface element, or it might be retrieving some piece of data to be displayed.

Many combine operators are specifically created to help with these transformations. Some operators require conformance to an input or failure type. Other operators may change either or both the failure and output types. For example, there are a number of operators that have a similar operator prefixed with `try`, which indicates they return an <Error> failure type.

An example of this is map and tryMap. The `map` operator allows for any combination of Output and Failure type and passes them through. `tryMap` accepts any Input, Failure types, and allows any Output type, but will always output an `<Error>` failure type.

Operators like `map` allow you to define the output type being returned by infering the type based on what you return in a closure provided to the operator. In the example above, the `map` operator is returning a String output type since that it what the closure returns.

To illustrate the the example of changing types more concretely, we expand upon the logic to use the values being passed. This example still starts with a publisher providing the types `<Int>`, `<Never>` and end with a subscription taking the types `<String>`, `<Never>`.

```
let _ = Just(5) ①
    .map { value -> String in ②
        switch value {
        case _ where value < 1:
            return "none"
        case _ where value == 1:
            return "one"
        case _ where value == 2:
            return "couple"
        case _ where value == 3:
            return "few"
        case _ where value > 8:
            return "many"
        default:
            return "some"
        }
    }
    .sink { receivedValue in ③
        print("The end result was \(receivedValue)")
    }
```

① Just is a publisher that creates an `<Int>`, `<Never>` type combination, provides a single value and then completes.

② the closure provided to the `.map()` function takes in an `<Int>` and transforms it into a `<String>`. Since the failure type of <Never> is not changed, it is passed through.

③ `sink`, the subscriber, receives the `<String>`, `<Never>` combination.

> When you are creating pipelines in Xcode and don't match the types, the error message from Xcode may include a helpful *fixit*. In some cases, such as the example above, the compiler is unable to infer the return types of closure provided to `map` withpout specifying the return type. Xcode (11 beta 2 and beta 3) displays this as the error message: `Unable to infer complex closure return type; add explicit type to disambiguate`. In the example above, we explicitly specified the type being returned with the line `value -> String in`.

You can view Combine publishers, operators, and subscribers as having two parallel types that both need to be aligned - one for the functional case and one for the error case. Designing your pipeline is frequently choosing how to convert one or both of those types and the associated data with it.

More examples, and some common tasks, are detailed in the section on patterns.

# Lifecycle of Publishers and Subscribers

The data flow in Combine is driven by, and starts from, the subscriber. This is how Combine supports the concept of back pressure.

Internally, Combine supports this with the enumeration Demand. When a subscriber is communicating with a publisher, it requests data based on demand. This request is what drives calling all the closures up the composed pipeline.

Because subscribers drive the closure execution, it also allows Combine to support cancellation. Cancellation can be triggered by the subscriber.

This is all enabled by subscribers and publishers communicating in a well defined sequence, or lifecycle.

1. When the subscriber is attached to a publisher, it starts with a call to `.subscribe(Subscriber)`.
2. The publisher in turn acknowledges the subscription calling `receive(subscription)`.
    - After the subscription has been acknowledged, the subscriber requests *N* values with `request(_ : Demand)`.
    - The publisher may then (as it has values) sending *N* (or fewer) values: `receive(_ : Input)`. A publisher should never send **more** than the demand requested.
    - Also after the subscription has been acknowledged, the subscriber can send a cancellation with `.cancel()`
3. A publisher may optionally send completion: `receive(completion:)` which is also how errors are propogated.

# Publishers

The publisher is the provider of data. The publisher protocol has a strict contract returning values when asked from subscribers, and possibly terminating with an explicit completion enumeration.

Just and Future are extremely common sources to start your own publisher from a value or function. Combine provides a number of additional convenience publishers:

| Just | Future | Published |
|---|---|---|
| Publishers.Once | Publishers.Optional | Publishers.Sequence |
| Publishers.Empty | Publishers.Fail | Publishers.Deferred |

A number of Apple APIs outside of Combine provide publishers as well.

- SwiftUI provides `@ObjectBinding` which can be used to create a publisher.
- `NoticationCenter .publisher`
- `Timer .publish` and `Timer.TimerPublisher`
- `URLSession dataTaskPublisher`

- `RealityKit .Scene .publisher()`

# Operators

Operators are a convenient name for a number of pre-built functions that are included under Publisher in Apple's reference documentation. These functions are all meant to be composed into pipelines. Many will accept one of more closures from the developer to define the business logic of the operator, while maintaining the adherance to the publisher/subscriber lifecycle.

Some operators support bringing together outputs from different pipelines, or splitting to send to multiple subscribers. Operators may also have constraints on the types they will operate on. Operators can also help with error handling and retry logic, buffering and prefetch, controlling timing, and supporting debugging.

| **Mapping elements** | | |
|---|---|---|
| scan | tryScan | setFailureType |
| map | tryMap | flatMap |

| **Filtering elements** | | |
|---|---|---|
| compactMap | tryCompactMap | replaceEmpty |
| filter | tryFilter | replaceError |
| removeDuplicates | tryRemoveDuplicates | |

| **Reducing elements** | | |
|---|---|---|
| collect | collectByCount | collectByTime |
| reduce | tryReduce | ignoreOutput |

| **Mathematic opertions on elements** | | |
|---|---|---|
| comparison | tryComparison | count |

| **Applying matching criteria to elements** | | |
|---|---|---|
| allSatisfy | tryAllSatisfy | contains |
| containsWhere | tryContainsWhere | |

| **Applying sequence operations to elements** | | |
|---|---|---|
| firstWhere | tryFirstWhere | first |
| lastWhere | tryLastWhere | last |
| dropWhile | tryDropWhile | dropUntilOutput |
| concatenate | drop | prefixUntilOutput |
| prefixWhile | tryPrefixWhile | output |

| Combining elements from multiple publishers | | |
|---|---|---|
| combineLatest | tryCombineLatest | merge |
| zip | | |

| Handling errors | | |
|---|---|---|
| catch | tryCatch | assertNoFailure |
| retry | mapError | |

| Adapting publisher types | | |
|---|---|---|
| switchToLatest | eraseToAnyPublisher | |

| Controlling timing | | |
|---|---|---|
| debounce | delay | measureInterval |
| throttle | timeout | |

| Encoding and decoding | | |
|---|---|---|
| encode | decode | |

| Working with multiple subscribers | | |
|---|---|---|
| multicast | | |

| Debugging | | |
|---|---|---|
| breakpoint | handleEvents | print |

# Subjects

Subjects are a special case of publisher that also adhere to the `subject` protocol. This protocol requires subjects to have a `.send()` method to allow the developer to send specific values to a subscriber (or pipeline).

Subjects can be used to "inject" values into a stream, by calling the subject's `.send()` method. This is useful for integrating existing imperative code with Combine.

A subject can also broadcast values to multiple subscribers. If multiple subscribers are connected to a subject, it will fanning out values to the multiple subscribers when `send()` is invoked.

There are two built-in subjects with Combine:

The first is currentValueSubject.

- CurrentValue remembers the current value so that when a subscriber is attached, it immediately receives the current value.

It is created and initialized with an initial value. When a subscriber is connected to it and requests

data, the initial value is sent. Further calls to `.send()` afterwards will then send those values to any subscribers.

The second is PassthroughSubject.

- Passthrough doesn't maintain any state - just passes through provided values.

When it is created, only the types are defined. When a subscriber is connected and requests data, it will not receive any values until a `.send()` call is invoked. Calls to `.send()` will then send values to any subscribers.

PassthroughSubject is extremely useful when writing tests for pipelines, as sending of any requested data (or a failure) is under the test writer's control using the `.send()` function.

Both CurrentValueSubject and PassthroughSubject are also useful for creating publishers from objects conforming to the `BindableObject` protocol within SwiftUI.

# Subscribers

While `subscriber` is the protocol used to receive data throughout a pipeline, *the Subscriber* typically refers to the end of a pipeline.

There are two subscribers built-in to Combine: assign and sink.

Subscribers can support cancellation, which terminates a subscription and shuts down all the stream processing prior to any Completion sent by the publisher. Both `Assign` and `Sink` conform to the cancellable protocol.

`assign` applies values passed down from the publisher to an object defined by a keypath. The keypath is set when the pipeline is created. An example of this in swift might look like:

```
.assign(to: \.isEnabled, on: signupButton)
```

`sink` accepts a closure that receives any resulting values from the publisher. This allows the developer to terminate a pipeline with their own code. This subscriber is also extremely helpful when writing unit tests to validate either publishers or pipelines. An example of this in swift might look like:

```
.sink { receivedValue in
    print("The end result was \(String(describing: receivedValue))")
}
```

Most other subscribers are part of other Apple frameworks. For example, nearly every control in SwiftUI can act as a subscriber. The `.onReceive(publisher)` function is used on SwiftUI views to act as a subscriber, taking a closure akin to `.sink()` that can manipulate `@State` or `@Bindings` within SwiftUI.

An example of that in swift might look like:

```
struct MyView : View {

    @State private var currentStatusValue = "ok"
    var body: some View {
        Text("Current status: \(currentStatusValue)")
    }
    .onReceive(MyPublisher.currentStatusPublisher) { newStatus in
        currentStatusValue = newStatus
    }
}
```

For any type of UI object (UIKit, AppKit, or SwiftUI), `.assign` can be used with pipelines to manipulate properties.

## Swift types and exposing pipelines or subscribers

When you compose pipelines within swift, the chaining is interpretted as nesting generic types to the compiler. If you expose a pipeline as a publisher, subscriber, or subject the exposed type can be exceptionally complex.

For example, if you created a publisher from a PassthroughSubject such as:

```
let x = PassthroughSubject<String, Never>()
    .flatMap { name in
        return Future<String, Error> { promise in
            promise(.success(""))
            }.catch { _ in
                Just("No user found")
            }.map { result in
                return "\(result) foo"
        }
}
```

The resulting type would reflect that composition:

```
Publishers.FlatMap<Publishers.Map<Publishers.Catch<Future<String, Error>, Just<
String>>, String>, PassthroughSubject<String, Never>>
```

When you want to expose the code, all of that composition detail can be very distracting and make your publisher, subject, or subscriber) harder to use. To clean up that interface, and provide a nice API boundary, the three major protocols all support methods that do type erasure. This cleans up the exposed type to a simpler generic form.

These three methods are:

- `.eraseToAnyPublisher()`

- `.eraseToAnySubscriber()`
- `.eraseToAnySubject()`

If you updated the above code to add .eraseToAnyPublisher() at the end of the pipeline:

```
let x = PassthroughSubject<String, Never>()
    .flatMap { name in
        return Future<String, Error> { promise in
            promise(.success(""))
            }.catch { _ in
                Just("No user found")
            }.map { result in
                return "\(result) foo"
        }
}.eraseToAnyPublisher()
```

The resulting type would simplify to:

```
AnyPublisher<String, Never>
```

# Pipelines and threads

Combine is not just a single threaded construct. Combine allows for publishers to specify the scheduler used when either receiving from an upstream publisher (in the case of operators), or when sending to a downstream subscriber. This is critical when working with a subscriber that updates UI elements, as that should always be called on the main thread.

You may see this in code as an operator, for example:

```
    .receive(on: RunLoop.main)
```

# Patterns and Recipes

Included are a series of patterns and examples of Publishers, Subscribers, and pipelines. These examples are meant to illustrate how to use the Combine framework to accomplish various tasks.

*Since this is a work in progress:* if you have a suggestion for a pattern or recipe, I'm happy to consider it.

Please Open an issue in GitHub to request something.

## Pattern 1.1: Creating a subscriber with sink

***Goal***

- To receive the output, and the errors or completion messages, generated from a publisher or through a pipeline, you can create a subscriber with `sink`.

***References***

- sink

***See also***

- Pattern 1.2: Creating a subscriber with assign
- Pattern N.2: Testing a publisher
- Pattern N.1: Testing pipelines

***Code and explanation***

Sink creates an all-purpose subscriber to capture or react the data from a Combine pipeline, while also supporting cancellation and the publisher subscriber lifecycle.

*simple sink*

```
let cancellablePipeline = publishingSource.sink { someValue in ①
    // do what you want with the resulting value passed down
    // be aware that depending on the data type being returned, you may get this
closure invoked
    // multiple times.
    print(".sink() received \(someValue)")
})
```

① The simple version of a sink is very compact, with a single trailing closure that only receives data when presented through the pipeline.

*sink with completions and data*

```
let cancellablePipeline = publishingSource.sink(receiveCompletion: { completion in ①
    switch completion {
    case .finished:
        // no associated data, but you can react to knowing the request has been
completed
        break
    case .failure(let anError):
        // do what you want with the error details, presenting, logging, or hiding as
appropriate
        print("received the error: ", anError)
        break
    }
}, receiveValue: { someValue in
    // do what you want with the resulting value passed down
    // be aware that depending on the data type being returned, you may get this
closure invoked
    // multiple times.
    print(".sink() received \(someValue)")
})

cancellablePipeline.cancel() ②
```

① Sinks are created by chaining the code from a publisher or pipeline, and terminate the pipeline. When the sink is created or invoked on a publisher, it implicitly starts the lifecycle with the `subscribe` and will request unlimited data.

② Creating a sink is cancellable subscriber, so at any time you can take the reference that terminated with sink and invoke `.cancel()` on it to invalidate and shut down the pipeline.

# Pattern 1.2: Creating a subscriber with assign

***Goal***

- To use the results of a pipeline to set a value, often a property on a user interface view or control, but any KVO compliant object can be the target

***References***

- assign
- receive

***See also***

- Pattern 1.1: Creating a subscriber with sink

***Code and explanation***

Assign is a subscriber that's specifically designed to apply data from a publisher or pipeline into a property, updating that property whenever it receives data. Like sink, it activates when created and requests an unlimited data updates. Assign requires the failure type to be specified as `<Never>`, so if your pipeline could fail (such as using an operator like tryMap) you will need to convert or handle the the failure cases before using `.assign`.

*simple sink*

```
let cancellablePipeline = publishingSource ①
    .receive(on: RunLoop.main) ②
    .assign(to: \.isEnabled, on: yourButton) ③

cancellablePipeline.cancel() ④
```

① `.assign` is typically chained onto a publisher when you create it, and the return value is cancellable.

② If `.assign` is being used to update a user interface element, you need to make sure that it is being updated on the main thread. This call makes sure the subscriber is received on the main thread.

③ Assign references the property being updated using a key path, and a reference to the object being updated.

④ At any time you can can to terminate and invalidate pipelines with cancel(). Frequently, you cancel the pipelines when you deactivate the objects (such as a viewController) that are getting updated from the pipeline.

# Pattern 2.1: Making a network request with dataTaskPublisher

***Goal***

- One of the common use cases is requesting JSON data from a URL and decoding it.

***References***

- URLSession.dataTaskPublisher
- map
- decode
- sink
- subscribe

***See also***

- Pattern 2.2: Stricter request processing with dataTaskPublisher
- Pattern 3.2: Using catch to handle errors in a one-shot pipeline
- Pattern 3.3: Retrying in the event of a temporary failure

***Code and explanation***

This can be readily accomplished with Combine using URLSession.dataTaskPublisher followed by a series of operators that process the data. Minimally, this is map and decode before going into your subscriber.

dataTaskPublisher on URLSession.

The simplest case of using this might be:

```
let myURL = URL(string: "https://postman-echo.com/time/valid?timestamp=2016-10-10")
// checks the validity of a timestamp - this one returns {"valid":true}
// matching the data structure returned from https://postman-echo.com/time/valid
fileprivate struct PostmanEchoTimeStampCheckResponse: Decodable, Hashable { ①
    let valid: Bool
}

let remoteDataPublisher = URLSession.shared.dataTaskPublisher(for: myURL!) ②
    // the dataTaskPublisher output combination is (data: Data, response: URLResponse)
    .map { $0.data } ③
    .decode(type: PostmanEchoTimeStampCheckResponse.self, decoder: JSONDecoder()) ④

let cancellableSink = remoteDataPublisher.sink
    .sink(receiveCompletion: { completion in
            print(".sink() received the completion", String(describing: completion))
            switch completion {
                case .finished: ⑤
                    break
                case .failure(let anError): ⑥
                    print("received error: ", anError)
            }
    }, receiveValue: { someValue in ⑦
        print(".sink() received \(someValue)")
    })
```

① Commonly you'll have a struct defined that supports at least Decodable (if not the full Codable protocol). This struct can be defined to only pull the pieces you're interested in from the JSON provided over the network.

② dataTaskPublisher is instantiated from URLSession. You can configure your own options on URLSession, or use the general shared session as you require.

③ The data that is returns down the pipeline is a tuple: `(data: Data, response: URLResponse)`. The map operator is used to get the data and drop the URL response, returning just Data down the pipeline.

④ decode is used to load the data and attempt to transform it into the struct defined. Decode can throw an error itself if the decode fails. If it succeeds, the object passed down the pipeline will be the struct from the JSON data.

⑤ If the decoding happened without errors, the finished completion will be triggered, and the value will also be passed to the receiveValue closure.

⑥ If the a failure happened (either with the original network request or the decoding), the error will be passed into with the `.failure` completion.

⑦ Only if the data succeeded with request and decoding will this closure get invoked, and the data format received with be an instance of the struct `PostmanEchoTimeStampCheckResponse`.

# Pattern 2.2: Stricter request processing with dataTaskPublisher

***Goal***

- When URLSession makes a connection, it only reports an error if the remote server doesn't respond. You may want to consider a number of responses, based on status code, to be errors. To accomplish this, you can use tryMap to inspect the http response and throw an error in the pipeline.

***References***

- URLSession.dataTaskPublisher
- tryMap
- decode
- sink
- subscribe

***See also***

- Pattern 2.1: Making a network request with dataTaskPublisher
- Pattern 3.2: Using catch to handle errors in a one-shot pipeline
- Pattern 3.3: Retrying in the event of a temporary failure

***Code and explanation***

To have more control over what is considered a failure in the URL response, use a `tryMap` operator on the tuple response from dataTaskPublisher. Since dataTaskPublisher returns both the response data and the URLResponse into the pipeline, you can immediately inspect the response and throw an error of your own if desired.

An example of that might look like:

```
let myURL = URL(string: "https://postman-echo.com/time/valid?timestamp=2016-10-10")
// checks the validity of a timestamp - this one returns {"valid":true}
// matching the data structure returned from https://postman-echo.com/time/valid
fileprivate struct PostmanEchoTimeStampCheckResponse: Decodable, Hashable {
    let valid: Bool
}
enum testFailureCondition: Error {
    case invalidServerResponse
}


let remoteDataPublisher = URLSession.shared.dataTaskPublisher(for: myURL!)
    .tryMap { data, response -> Data in ①
                guard let httpResponse = response as? HTTPURLResponse, ②
                    httpResponse.statusCode == 200 else { ③
                        throw testFailureCondition.invalidServerResponse ④
                }
                return data ⑤
    }
    .decode(type: PostmanEchoTimeStampCheckResponse.self, decoder: JSONDecoder())

let cancellableSink = remoteDataPublisher.sink
    .sink(receiveCompletion: { completion in
            print(".sink() received the completion", String(describing: completion))
            switch completion {
                case .finished:
                    break
                case .failure(let anError):
                    print("received error: ", anError)
            }
    }, receiveValue: { someValue in
        print(".sink() received \(someValue)")
    })
```

Where the previous pattern used a map operator, this uses tryMap, which allows us to identify and throw errors in the pipeline based on what was returned.

① tryMap still gets the tuple of `(data: Data, response: URLResponse)`, and is defined here as returning just the type of Data down the pipeline.

② Within the closure for tryMap, we can cast the response to HTTPURLResponse and dig deeper into it, including looking at the specific status code.

③ In this case, we want to consider **anything** other than a 200 response code as a failure. HTTPURLResponse.status_code is an Int type, so you could also have logic such as `httpResponse.statusCode > 300`.

④ If the predicates aren't met, then we can throw an instance of an error of our choosing, `invalidServerResponse` in this case.

⑤ If no error has occured, then we simply pass down Data for further processing.

When an error is triggered on the pipeline, a `.failure` completion is sent with the error encapsulated within it, regardless of where it happened in the pipeline.

# Error Handling

The examples above expected that the subscriber would handle the error conditions, if they occured. However, you are not always able to control the subscriber - as might be the case if you're using SwiftUI view properties as the subscriber, and you're providing the publisher. In these cases, you need to build your pipeline so that the output types match the subscriber types.

For example, if you are working with SwiftUI and the you want to use `.assign` to set the `isEnabled` property on a button, the subscriber will have a few requirements:

1. the subcriber should match the type output of `<Bool>`, `<Never>`
2. the subscriber should be called on the main thread

With a publisher that can throw an error (such as `dataTaskPublisher`), you need to construct a pipeline to convert the output type, but also handle the error within the pipeline to match a failure type of `<Never>`.

How you handle the errors within a pipeline is very dependent on how the pipeline is working. If the pipeline is set up to return a single result and terminate, continue to Pattern 3.2: Using catch to handle errors in a one-shot pipeline. If the pipeline is set up to continually update, the error handling needs to be a little more complex. Jump ahead to Pattern 3.4: Using flatMap with catch to handle errors.

## Pattern 3.1: verifying a failure hasn't happened using assertNoFailure

***Goal***

- Verify no error has occured within a pipeline

***References***

- assertNoFailure

***See also***

- << link to other patterns>>

***Code and explanation***

> Useful in testing invariants in pipelines, the assertNoFailure operator also converts the failure type to `<Never>`. The operator will cause the application to terminate (and tests to crash to a debugger) if the assertion is triggered.

This is useful for verifying the invariant of having dealt with an error. If you are sure you handled the errors and need to map a pipeline which technically can generate a failure type of `<Error>` to a subscriber that requires a failure type of `<Never>`.

It is far more likely that you want to handle the error with and not have the application terminate. Look forward to Pattern 3.2: Using catch to handle errors in a one-shot pipeline and Pattern 3.4: Using flatMap with catch to handle errors for patterns of how to provide logic to handle errors in a pipeline.

## Pattern 3.2: Using catch to handle errors in a one-shot pipeline

### *Goal*

- If you need to handle a failure within a pipeline, for example before using the `assign` operator or another operator that requires the failure type to be `<Never>`, you can use `catch` to provide the appropriate logic.

### *References*

- catch
- Just

### *See also*

- Pattern 3.3: Retrying in the event of a temporary failure
- Pattern 3.4: Using flatMap with catch to handle errors
- Pattern 4: Requesting data from an alternate URL when the network is constrained

### *Code and explanation*

`catch` handles errors by replacing the upstream publisher with another publisher that you provide as a return in a closure.

Be aware that this effectively terminates the earlier portion of the pipeline. If you're using a one-shot publisher (one that doesn't create more than a single event), then this is fine.

For example, `dataTaskPublisher` is a one-shot publisher and you might use catch with it to ensure that you get a response, returning a placeholder in the event of an error. Extending our previous example to provide a default response:

```
struct IPInfo: Codable {
    // matching the data structure returned from ip.jsontest.com
    var ip: String
}
let myURL = URL(string: "http://ip.jsontest.com")
// NOTE(heckj): you'll need to enable insecure downloads in your Info.plist for this
example
// since the URL scheme is 'http'

let remoteDataPublisher = URLSession.shared.dataTaskPublisher(for: myURL!)
    // the dataTaskPublisher output combination is (data: Data, response: URLResponse)
    .map({ (inputTuple) -> Data in
        return inputTuple.data
    })
    .decode(type: IPInfo.self, decoder: JSONDecoder()) ①
    .catch { err in ②
        return Publishers.Just(IPInfo(ip: "8.8.8.8"))③
    }
    .eraseToAnyPublisher()
```

① Often, a catch operator will be placed after several operators that could fail, in order to provide a fallback or placeholder in the event that any of the possible previous operations failed.

② When using catch, you get the error type in and can inspect it to chosoe how you provide a response.

③ The Just publisher is frequently used to either start another one-shot pipeline or to directly provide a placeholder response in the event of failure.

A possible problem with this technique is that the if the original publisher generates more values to which you wish to react, the original pipeline has been ended. If you are creating a pipeline that reacts to a `@Published` property, then after any failed value that activates the catch operator, the pipeline will cease to react further. See catch for more illustration and examples of how this works.

If you want to continue to respond to errors and handle them, see Pattern 3.4: Using flatMap with catch to handle errors for an example of how to do that using `flatMap`

## Pattern 3.3: Retrying in the event of a temporary failure

***Goal***

- The `retry` operator can be included in a pipeline to retry a subscription when a `.failure` completion occurs.

***References***

- catch
- retry

***See also***

- Pattern 3.2: Using catch to handle errors in a one-shot pipeline
- Pattern 3.4: Using flatMap with catch to handle errors

***Code and explanation***

When you specify this operator in a pipeline and it receives a subscription, it first tries to request a subscription from it's upstream publisher. If the response to that subscription fails, then it will retry the subscription to the same publisher.

The retry operator can be specified with a number of retries to attempt. If no number of retries is specified, it will attempt to retry indefinitely until it receives a .finished completion from it's subscriber. If the number of retries is specified and all requests fail, then the `.failure` completion is passed down to the subscriber of this operator.

In practice, this is mostly commonly desired when attempting to request network resources with an unstable connection. If you use a retry operator, you should add a specific number of retries so that the subscription doesn't effectively get into an infinite loop.

An example of the above example using retry in combination with a delay:

```
let remoteDataPublisher = urlSession.dataTaskPublisher(for: self.mockURL!)
    .delay(for: DispatchQueue.SchedulerTimeType.Stride(integerLiteral: Int.random(in:
1..<5)), scheduler: backgroundQueue) ①
    .retry(3) ②
    .tryMap { data, response -> Data in ③
        guard let httpResponse = response as? HTTPURLResponse,
            httpResponse.statusCode == 200 else {
                throw testFailureCondition.invalidServerResponse
        }
        return data
    }
    .decode(type: PostmanEchoTimeStampCheckResponse.self, decoder: JSONDecoder())
    .subscribe(on: backgroundQueue)
    .eraseToAnyPublisher()
```

① the delay operator will delay further processing on the pipeline, in this case for a random selection of 1 to 5 seconds. By adding delay here in the pipeline, it will always occur, even if the original request is successful.

② retry is specified as trying 3 times. If you specify retry without any options, it will retry infinitely, and may cause your pipeline to never resolve any values or completions.

③ tryMap is being used to investigate errors after the retry so that retry will only re-attempt the request when the site didn't respond.

When using the `retry()` operator with `dataTaskPublisher`, verify that the URL you are requesting isn't going to have negative side effects if requested repeatedly or with a retry. Ideally such requests are be expected to be idempotent.

## Pattern 3.4: Using flatMap with catch to handle errors

### *Goal*

- The `flatMap` operator can be used with `catch` to continue to handle errors on new published values.

### *References*

- [reference-flatmap]
- Just
- Publishers.Once
- catch

### *See also*

- Pattern 3.2: Using catch to handle errors in a one-shot pipeline
- Pattern 3.3: Retrying in the event of a temporary failure

### *Code and explanation*

The `flatMap` operator is the operator to use in handling errors on a continual flow of events.

You provide a closure to flatMap that can read in the value that was provided, and creates a one-shot closure that does the possibly failing work. An example of this is requesting data from a network and then decoding the returned data. You can include a catch operator to capture any errors and provide any appropriate value.

This is a perfect mechanism for when you want to maintain updates up an upstream publisher, as it creates one-shot publisher or short pipelines that send a single value and then complete for every incoming value. The completion from the created one-shot publishers terminates in the flatMap and isn't passed to downstream subscribers.

An example of this with a dataTaskPublisher:

```
let remoteDataPublisher = Just(self.testURL!) ①
    .flatMap { url in ②
        URLSession.shared.dataTaskPublisher(for: url) ③
        .tryMap { data, response -> Data in ④
            guard let httpResponse = response as? HTTPURLResponse,
                httpResponse.statusCode == 200 else {
                    throw testFailureCondition.invalidServerResponse
            }
            return data
        }
        .decode(type: PostmanEchoTimeStampCheckResponse.self, decoder: JSONDecoder())
⑤
        .catch {_ in ⑥
            return Just(PostmanEchoTimeStampCheckResponse(valid: false))
        }
    }
    .subscribe(on: self.myBackgroundQueue!)
    .eraseToAnyPublisher()
```

① Just starts this publisher as an example by passing in a URL.

② flatMap takes the URL as input and the closure goes on to create a one-shot publisher chain.

③ dataTaskPublisher uses the input url

④ which flows to tryMap to parse for additional errors

⑤ and finally decode to attempt to refine the returned data into a local type

⑥ if any of these have failed, catch will convert the error into a placeholder sample, in this case an object with a preset `valid = false` property.

# Pattern 4: Requesting data from an alternate URL when the network is constrained

***Goal***

- From Apple's WWDC 19 presentation Advances in Networking, Part 1, a sample pattern was provided using `tryCatch` and `tryMap` operators to react to the specific error of having the network be constrained.

***References***

- URLSession.dataTaskPublisher
- tryCatch
- tryMap

***See also***

- Pattern 3.2: Using catch to handle errors in a one-shot pipeline
- Pattern 3.3: Retrying in the event of a temporary failure

***Code and explanation***

This sample is originally from the WWDC session. The API and example is evolving with the beta releases of Combine since that presentation. `tryCatch` was missing in the beta2 release, and has returned in beta3.

```
// Generalized Publisher for Adaptive URL Loading
func adaptiveLoader(regularURL: URL, lowDataURL: URL) -> AnyPublisher<Data, Error> {
    var request = URLRequest(url: regularURL) ①
    request.allowsConstrainedNetworkAccess = false ②
    return URLSession.shared.dataTaskPublisher(for: request) ③
        .tryCatch { error -> URLSession.DataTaskPublisher in ④
            guard error.networkUnavailableReason == .constrained else {
               throw error
            }
            return URLSession.shared.dataTaskPublisher(for: lowDataURL) ⑤
        .tryMap { data, response -> Data in
            guard let httpResponse = response as? HTTPUrlResponse, ⑥
                   httpResponse.status_code == 200 else {
                       throw MyNetworkingError.invalidServerResponse
            }
            return data
}
.eraseToAnyPublisher() ⑦
```

This example, from Apple's WWDC, provides a function that takes two URLs - a primary and a fallback. It returns a publisher that will request data and fall back requesting a secondary URL when the network is constrained.

① The request starts with an attempt requesting data.

② Setting `request.allowsConstrainedNetworkAccess` will cause the dataTaskPublisher to error if the network is constrained.

③ Invoke the dataTaskPublisher to make the request.

④ tryCatch is used to capture the immediate error condition and check for a specific error (the constrained network).

⑤ If it finds an error, it creates a new one-shot publisher with the fall-back URL.

⑥ The resulting publisher can still fail, and tryMap can map this a failure by throwing an error on HTTP response codes that map to error conditions

⑦ eraseToAnyPublisher will do type erasure on the chain of operators so the resulting signature of the adaptiveLoader function is of type `AnyPublisher<Data, Error>`

In the sample, if the error returned from the original request wasn't an issue of the network being constrained, it passes on the .failure completion down the pipeline. If the error is that the network is constrained, then the tryCatch operator creates a new request to an alternate URL.

# Pattern 5: Update the status of your interface from a network request

***Goal***

- Querying a web based API and returning the data to be displayed in your UI

***References***

- << link to reference pages>>

***See also***

- << link to other patterns>>

***Code and explanation***

Below is a contrived example where you want to make a network to check for the username availability that you are watching with `@Published`. As the property `username` is updated, you want to check to see if the updated username is available.

This contrived example expects that you have a web service that you can query, which will return a structured response in JSON.

```
@Published var username: String = ""

struct UsernameResponse: Codable {
    username: String
    available: Bool
}

var validatedUsername: AnyPublisher<String?, Never> {
    return $username
        .debounce(for: 0.5, scheduler: RunLoop.main)
        .removeDuplicates()
        .flatMap { username in
            let constructedURL = URL(string: "https://yourhost.com/?user=\(username)")
            return remoteDataPublisher = URLSession.shared.dataTaskPublisher(for:
constructedURL!)
                .map({ (inputTuple) -> Data in
                    return inputTuple.data
                })
                .decode(type: UsernameResponse.self, decoder: JSONDecoder())
                .map { response: UsernameResponse in
                    return response.available
                }
                .catch { err in
                    // if the service is down, or the JSON malformed, return a false
response
                    return Publishers.Just(False))
                }
        }
}
```

In the example above, for every update into .flatMap() we are creating a request to check and parse for the availability from the service.

# Pattern 6: Wrapping an asynchronous call with a Future

***Goal***

- Using Future to turn an an asynchronous call into publisher

***References***

- << link to reference pages>>

***See also***

- << link to other patterns>>

***Code and explanation***

```
let myPublisher = Publishers.Future { promise in
    asyncFunctionWithACompletion(inputValue) { outputValue in
        promise(.success(outputValue ? inputValue : nil))
    }
}
.eraseToAnyPublisher()
```

This setup can be used to inline just about anything into a Combine pipeline.

possible example: - take periodic frame updates from the camera and run them into the Vision framework to ask if there's a bar code, and if so, retrieve it and display its info

# Pattern 7: Coordinating a sequence of asynchronous operations

***Goal***

- There are a variety of ways to chain together asynchronous operations.

***References***

- << link to reference pages>>

***See also***

- << link to other patterns>>

***Code and explanation***

Combine adds to this variety, and is effective when you want to use the data from one operation as the input to the next. If you are familiar with using Promises in another language, such as Javascript, this pattern is roughly the equivalent of Promise chaining.

The benefit to using Combine is that the sequencing can be relatively easy to parse visually.

# Pattern 8: Responding to updates in properties with @Published

***Goal***

- @Published with properties and using read-only updates of those properties as publishing sources

***References***

- << link to reference pages>>

***See also***

- << link to other patterns>>

***Code and explanation***

```
@Published var username: String = ""

var validatedUsername: AnyPublisher<String?, Never> {
    return $username
        .debounce(for: 0.5, scheduler: RunLoop.main)
           //  <String?>|<Never>
        .removeDuplicates()
           //  <String?>|<Never>
        .flatMap { username in
            return Future { promise in
                self.usernameAvailable(username) { available in
                   promise(.success(available ? username : nil))
                }
          //    <Result<Output, Failure>>
            }
        }
          // <String?>|<Never>
        .eraseToAnyPublisher()
}
```

validation - listening for changes to validate them together

```
@Published var password: String = ""
@Published var passwordAgain: String = ""

var validatedPassword: AnyPublisher<String?, Never> {
    return CombineLatest($password, $passwordAgain) { password, passwordAgain in
        guard password == passwordAgain, password.count > 8 else { return nil }
        return password
    }
      //  <String?>|<Never>
    .map { $0 == password1 ? nil : $0 }
      //  <String?>|<Never>
    .eraseToAnyPublisher()
      //  <String?>|<Never>
}
```

more complex validation - bringing together substreams

```
var validatedCredentials: AnyPublisher<(String, String)?, Never> {
    return CombineLatest(validatedUsername, validatedPassword) { username, password in
        guard let uname = username, let pwd = password else { return nil }
        return (uname, pwd)
    }
    .eraseToAnyPublisher()
}

@IBOutlet var signupButton: UIButton!

var signupButtonStream: AnyCancellable?

override func viewDidLoad() {
    super.viewDidLoad()
    self.signupButtonStream = self.validatedCredentials
        .map { $0 != nil }
        .receive(on: RunLoop.main)
        .assign(to: \.isEnabled, on: signupButton)
}
```

# Pattern 9: Responding to updates from NotificationCenter

***Goal***

- The big "master bus" of events across a variety of Apple platforms, its where you can listen for updates and changes from controls and events across a variety of frameworks.

***References***

- << link to reference pages>>

***See also***

- << link to other patterns>>

***Code and explanation***

Casey Liss talks about about this (not entirely happily) based on the apple documentation Receiving and Handling Events with Combine.

# Pattern 10: Using BindableObject with SwiftUI models as a publisher source

***Goal***

- SwiftUI includes @Binding and the BindableObject protocol, which provides a publishing source to alerts to model objects changing.

***References***

- << link to reference pages>>

***See also***

- << link to other patterns>>

***Code and explanation***

# Pattern N.1: Testing pipelines

***Goal***

- For testing what's happening in a pipeline

***References***

- << link to reference pages>>

***See also***

- << link to other patterns>>

***Code and explanation***

- use a Subject → pipeline → sink()
- set up a sink to collect value
- drive values through the pipeline with Subject.send()
- assert results after the sink has processed them

# Pattern N.2: Testing a publisher

***Goal***

- For testing a publisher (and any pipeline attached)

***References***

- << link to reference pages>>

***See also***

- << link to other patterns>>

***Code and explanation***

- set up an expectation (`XCTestExpectation`)
- create your publisher & relevant pipeline if so desired
- create a sink to capture the results that works on both completions and values
  - this can be separate, or just chained to the pipeline, depending on what makes most sense to you
- `wait` on the expectation to let the test "do it's thing" in the background

# Pattern N.3: Testing a subscriber

***Goal***

- For testing a subscriber (how it reacts):

***References***

- << link to reference pages>>

***See also***

- << link to other patterns>>

***Code and explanation***

- set up your subscriber and any pipeline leading to it
- use a Subject to "inject" values
- test the results after the value is sent
- if the updates include some async/background work before data is available, use an expectation
    - add a callback to the async pieces where you can trigger the expectation, and do your asserts just prior to fulfilling the expectation

& of course you can always test using UITesting - spinning up your whole app, initializing state, and then driving and verifying the results.

*(Uncertain)* Mocking or faking a publisher (such as dataTaskPublisher) to validate things like using `retry()` operator * might be able to use a `Publishers.Future()`, expose as an AnyPublisher() (do the same with dataTaskPublisher) to make the the same - and in your setup, inject in the one you want to use. ** Instrument the Future() closure to record what gets called, and maybe set it up to return an explicit set of responses.

# Pattern N+1: Debugging pipelines

***Goal***

- For testing a subscriber (how it reacts):

***References***

- << link to reference pages>>

***See also***

- << link to other patterns>>

***Code and explanation***

1. use print() and/or print("prefixValue") to get console output of what's happening in the pipeline lifecycle
    - create a .sink() to capture results, and drive it with a PassthroughSubject for specific control
2. add a handleEvents() operator
    - create closures to do additional poking at values or digging into more structured pieces than get exposed with a print()
    - allows you to ignore some sections you don't care about
    - closures on `receiveSubscription`, `receiveRequest`, `receiveCancel`, `receiveOutput`, and `receiveCompletion`
3. breakPoint
    - if you want to break into a debugger, add in a closure that returns true and you can inspect to your heart's content
        - closure's on `receiveSubscription`, `receiveOutput`, and `receiveCompletion`
    - might also be interesting to use breakpointOnError() which triggers only when a failure completion

# Reference

reference preamble goes here…

This is intended to extend Apple's documentation, not replace it.

- The documentation associated with beta2 is better than beta1, but still fairly anemic.

things to potentially include for each segment

- narrative description of what the function does
  - notes on why you might want to use it, or where you may see it
  - xref back to patterns document where functions are being used
- marble/railroad diagram explaining what the transformation/operator does
- sample code showing it being used and/or tested

## Publishers

### Just

***Summary***

`Just` provides a single result and then terminates, providing a publisher with a failure type of `<Never>`

***docs***

`Just`

***Usage***

- Pattern 3.4: Using flatMap with catch to handle errors

***Details***

Often used within a closure to flatMap in error handling, it can see a one-shot pipeline for use in error handling of continuous values.

### Future

***Summary***

A future is initialized with a closure that eventually resolves to a value.

***docs***

Future.

***Usage***

n/a

***Details***

- you provide a closure that converts a callback/function of your own choosing into a Promise.
- in creating a Future publisher, you need to handle the logic of when you generate the relevant `Result<Output, Error>` with the asynchronous calls.

## Published

***Summary***

A property wrapper that adds a Combine publisher to any property

***docs***

Published

***Usage***

n/a

***Details***

Output type is inferred from the property being wrapped.

publisher → `<SomeType>`, `<Never>`

- extracts a property from an object and returns it
  - ex: `.publisher(for: \.name)`

## Publishers.Empty

***Summary***

`empty` never publishes any values, and optionally finishes immediately.

***docs***

`Publishers.Empty`

***Usage***

- Pattern 3.2: Using catch to handle errors in a one-shot pipeline shows an example of using `catch` to handle errors with a one-shot publisher.
- Pattern 3.4: Using flatMap with catch to handle errors shows an example of using `catch` with `flatMap` to handle errors with a continual publisher.

***Details***

- Empty → `<SomeType>`, `<Error>`
  - `Empty(completeImmediately: false)`

## Publishers.Fail

***Summary***

`fail` immediately terminates publishing with the specified failure.

***docs***

`Publishers.Fail`

***Usage***

n/a

***Details***

n/a

## Publishers.Once

***Summary***

Generates an output to each subscriber exactly once then finishes or fails immediately.

***docs***

`Once`

***Usage***

TBD

***Details***

Used similiarly to Just, it provides a value and then completes. It is often used with flatMap when you want to flow have flatMap return a publisher that returns a publisher with an error type. Where Just returns a failure type of `<Never>`, `once` allows for a failure type of `<Error>`.

## Publishers.Optional

***Summary***

generates a value exactly once for each subscriber, if the optional has a value

***docs***

`Publishers.Optional`

***Usage***

n/a

***Details***

n/a

## Publishers.Sequence

***Summary***

Publishes a provided sequence of elements.

***docs***

`Publishers.Sequence`

***Usage***

n/a

***Details***

n/a

## Publishers.Deferred

***Summary***

Publisher waits for a subscriber before running the provided closure to create values for the subscriber.

***docs***

`Publishers.Deferred`

***Usage***

n/a

***Details***

n/a

## SwiftUI

- @ObjectBinding (swiftUI)
- BindableObject
- often linked with method `didChange` to publish changes to model objects
    - `@ObjectBinding var model: MyModel`

## Foundation

- `NotificationCenter .publisher`
- `Timer .publish` and `Timer.TimerPublisher`
    - * TimerPublisher

## URLSession.dataTaskPublisher

***Summary***

Foundation's `URLSession` has a publisher specifically for requesting data from URLs: dataTaskPublisher

***Constraints on connected publisher***

- *none*

***docs***

`URLSession.DataTaskPublisher`

***Usage***

- Pattern 2.1: Making a network request with dataTaskPublisher
- Pattern 3.2: Using catch to handle errors in a one-shot pipeline
- Pattern 3.3: Retrying in the event of a temporary failure
- Pattern 4: Requesting data from an alternate URL when the network is constrained

***Details***

`dataTaskPublisher`, on URLSession, has two variants for creating a publisher. The first takes an instance of URL, the second URLRequest. The data returned from the publisher is a tuple of `(data: Data, response: URLResponse)`.

```
let request = URLRequest(url: regularURL)
return URLSession.shared.dataTaskPublisher(for: request)
```

## RealityKit

- `RealityKit .Scene .publisher()`

Scene Publisher (from RealityKit)

- Scene.Publisher
  - SceneEvents
  - AnimationEvents
  - AudioEvents
  - CollisionEvents

# Operators

## Mapping elements

### scan

- scan

### tryScan

- tryScan

### map

***Summary***

map is most commonly used to convert one data type into another along a pipeline.

***Constraints on connected publisher***

- *none*

***docs***

https://developer.apple.com/documentation/combine/publishers/map

n/a

***Usage***

- Pattern 2.1: Making a network request with dataTaskPublisher
- Pattern 3.2: Using catch to handle errors in a one-shot pipeline
- Pattern 3.3: Retrying in the event of a temporary failure
- unit tests illustrating using map with dataTaskPublisher: `UsingCombineTests/DataTaskPublisherTests.swift`

***Details***

The map operator doesn't allow for any additional failures to be thrown, and doesn't transform the failure type. If you want to throw an error within your closure, then use the tryMap operator.

map takes a single closure where you provide the logic for the map operation.

For example, the URLSession.dataTaskPublisher provides a tuple of `(data: Data, response: URLResponse)` as its output. You can use map to pass along the data, for example to use with decode.

```
.map { $0.data } ①
```

① the `$0` indicates to grab the first parameter passed in, which is a tuple of `data` and `response`.

In some cases, the closure may not be able to infer what data type you are returning, so you may need to provide a definition to help the compiler. For example, if you have an object getting passed

down that has a boolean property "isValid" on it, and you just want the boolean for your pipeline, you might set that up like:

```
struct myStruct {
    isValid: bool = true
}
//
Just(myStruct())
.map { inValue -> Bool in ①
  inValue.isValid ②
}
```

① inValue is named as the parameter coming in, and the return type is being explicitly specified to `Bool`

② A single line is an implicit return, in this case it's pulling the `isValid` property off the struct and passing it down the pipeline.

#### tryMap

##### *Summary*

tryMap is effectively the similiar to map, except that it also allows you to provide a closure that throws additional errors if your conversion logic is unsuccessful.

##### *Constraints on connected publisher*

- *none*

##### *docs*

https://developer.apple.com/documentation/combine/publishers/trymap

##### *Usage*

- Pattern 2.2: Stricter request processing with dataTaskPublisher
- unit tests illustrating using tryMap with dataTaskPublisher: `UsingCombineTests/DataTaskPublisherTests.swift`

##### *Details*

tryMap is useful when you have more complex business logic around your map and you want to indicate that the data passed in is an error, possibly handling that error later in the pipeline. If you are looking at tryMap to decode JSON, you may want to consider using the decode operator instead, which is set up for that common task.

```
enum myFailure: Error {
    case notBigEnough
}

//
Just(5)
.tryMap {
  if inValue < 5 { ①
      throw myFailure.notBigEnough ②
  }
  return inValue ③
}
```

① You can specify whatever logic is relevant to your use case within tryMap

② and throw an error, although throwing an Error isn't required.

③ If the error condition doesn't occur, you do need to pass down data for any further subscribers.

### flatMap

#### *Summary*

Used with error recovery or async operations that might fail (ex: Future), flatMap will replace any incoming values with another publisher.

#### *Constraints on connected publisher*

- *none*

#### *docs*

`flatMap`

#### *Usage*

- Pattern 3.4: Using flatMap with catch to handle errors

#### *Details*

Most typically used in error handling scenarios, flatMap takes a closure that allows you to read the incoming data value, and provide a publisher that returns a value to the pipeline.

In error handling, this is most frequently used to take the incoming value and create a one-shot pipeline that does some potentially failing operation, and then handling the error condition with a catch operator.

A diagram version of this pipeline construct might be:

```
     one-shot-publisher(value) -> catch ( fallback )      // <- one-shot pipeline
                          ^                         \
                          |                          \
publisher -> flatMap -> ( +                           +  ) -> subscriber
```

In swift, this looks like:

```
.flatMap { data in
    return Just(data)
    .decode(YourType.self, JSONDecoder())
    .catch {
        return Just(YourType.placeholder)
    }
}
```

#### setFailureType

- setFailureType

## Filtering elements

#### compactMap

- compactMap
  - republishes all non-nil results of calling a closure with each received element.
  - there's a variant `tryCompactMap` for use with a provided error-throwing closure.

#### tryCompactMap

- tryCompactMap

#### filter

- filter
  - requires Failure to be `<Never>`
  - takes a closure where you can specify how/what gets filtered
  - there's a variant `tryFilter`for use with a provided error-throwing closure.

#### tryFilter

- tryFilter

#### removeDuplicates

- removeDuplicates
  - `.removeDuplicates()`
  - remembers what was previously sent in the stream, and only passes forward new values
  - there's a variant `tryRemoveDuplicates` for use with a provided error-throwing closure.

**tryRemoveDuplicates**

- tryRemoveDuplicates

**replaceEmpty**

- replaceEmpty
  - requires Failure to be `<Never>`

**replaceError**

- replaceError
  - requires Failure to be `<Never>`

**replaceNil**

- replaceNil
  - requires Failure to be `<Never>`
  - Replaces nil elements in the stream with the proviced element.

## Reducing elements

### collect

- collect
  - multiple variants
    - buffers items
    - `collect()` Collects all received elements, and emits a single array of the collection when the upstream publisher finishes.
    - `collect(Int)` collects N elements and emits as an array
    - `collect(.byTime)` or `collect(.byTimeOrCount)`

### collectByCount

- collectByCount

### collectByTime

- collectByTime

### ignoreOutput

- ignoreOutput

### reduce

- reduce
  - A publisher that applies a closure to all received elements and produces an accumulated value when the upstream publisher finishes.
  - requires Failure to be `<Never>`
  - there's a varient `tryReduce` for use with a provided error-throwing closure.

### tryReduce

- tryReduce

## Mathematic opertions on elements

### max

- max
  - Available when Output conforms to Comparable.
  - Publishes the maximum value received from the upstream publisher, after it finishes.

### min

- Publishes the minimum value received from the upstream publisher, after it finishes.
- Available when Output conforms to Comparable.

### comparison

- comparison
  - republishes items from another publisher only if each new item is in increasing order from the previously-published item.
  - there's a variant `tryComparson` which fails if the ordering logic throws an error

### tryComparison

- tryComparison

### count

- count
  - publishes the number of items received from the upstream publisher

## Applying matching criteria to elements

### allSatisfy

- allSatisfy
  - Publishes a single Boolean value that indicates whether all received elements pass a given predicate.
  - there's a variant `tryAllSatisfy` when the predicate can throw errors

### tryAllSatisfy

- tryAllSatisfy

### contains

- contains
  - emits a Boolean value when a specified element is received from its upstream publisher.
  - variant `containsWhere` when a provided predicate is satisfied
  - variant `tryContainsWhere` when a provided predicate is satisfied but could throw errors

### containsWhere

- containsWhere

### tryContainsWhere

- tryContainsWhere

## Applying sequence operations to elements

### first

- first
  - requires Failure to be `<Never>`
  - publishes the first element to satisfy a provided predicate

### firstWhere

- firstWhere

### tryFirstWhere

- tryFirstWhere

### last

- last
  - requires Failure to be `<Never>`
  - publishes the last element to satisfy a provided predicate

### lastWhere

- lastWhere

### tryLastWhere

- tryLastWhere

### dropUntilOutput

- dropUntilOutput

### dropWhile

- dropWhile

### tryDropWhile

- tryDropWhile

### concatenate

- concatenate

### drop

- drop

- multiple variants
- requires Failure to be `<Never>`
- Ignores elements from the upstream publisher until it receives an element from a second publisher.
- or `drop(while: {})`

**prefixUntilOutput**

- prefixUntilOutput
  - Republishes elements until another publisher emits an element.
  - requires Failure to be `<Never>`

**prefixWhile**

- prefixWhile
  - Republishes elements until another publisher emits an element.
  - requires Failure to be `<Never>`

**tryPrefixWhile**

- tryPrefixWhile
  - Republishes elements until another publisher emits an element.
  - requires Failure to be `<Never>`

**output**

- output

## Combining elements from multiple publishers

### combineLatest

- combineLatest
  - brings inputs from 2 (or more) streams together
  - you provide a closure that gets the values and chooses what to publish

### tryCombineLatest

- tryCombineLatest

### merge

- merge
  - Combines elements from this publisher with those from another publisher of the same type, delivering an interleaved sequence of elements.
  - requires Failure to be `<Never>`
  - multiple variants that will merge between 2 and 8 different streams

### zip

- zip
  - Combine elements from another publisher and deliver pairs of elements as tuples.
  - requires Failure to be `<Never>`

## Handling errors

See Error Handling for more detail on how you can design error handling.

### catch

#### *Summary*

The operator `catch` handles errors (completion messages of type `.failure`) from an upstream publisher by replacing the failed publisher with another publisher. The operator also transforms the Failure type to `<Never>`.

#### *Constraints on connected publisher*

- *none*

#### *Documentation reference*

`Publishers.Catch`

#### *Usage*

- Pattern 3.2: Using catch to handle errors in a one-shot pipeline shows an example of using `catch` to handle errors with a one-shot publisher.
- Pattern 3.4: Using flatMap with catch to handle errors shows an example of using `catch` with `flatMap` to handle errors with a continual publisher.

#### *Details*

Once catch receives a `.failure` completion, it won't send any further incoming values from the original upstream publisher. You can also view catch as a switch that only toggles in one direction: to using a new publisher that you define, but only when the original publisher to which it is subscribed sends an error.

This can be illustrated with the following code snippet:

```
enum testFailureCondition: Error {
    case invalidServerResponse
}

let simplePublisher = PassthroughSubject<String, Error>()

let _ = simplePublisher
    .catch { err in
        // must return a Publisher
        return Just("replacement value")
    }
    .sink(receiveCompletion: { fini in
        print(".sink() received the completion:", String(describing: fini))
    }, receiveValue: { stringValue in
        print(".sink() received \(stringValue)")
    })

simplePublisher.send("oneValue")
simplePublisher.send("twoValue")
simplePublisher.send(completion: Subscribers.Completion.failure(testFailureCondition
.invalidServerResponse))
simplePublisher.send("redValue")
simplePublisher.send("blueValue")
simplePublisher.send(completion: .finished)
```

In this example, we are using a `PassthroughSubject` so that we can control when and what gets sent from the publisher. In the above code, we are sending two good values, then a failure, then attempting to send two more good values. The values you would see printed from our `.sink()` closures are:

```
.sink() received oneValue
.sink() received twoValue
.sink() received replacement value
.sink() received the completion: finished
```

When the failure was sent through the pipeline, catch intercepts it and returns "replacement value" as expected. The replacement publisher it used (`Just`) sends a single value and then sends a completion. If we want the pipeline to remain active, we need to change how we handle the errors.

### tryCatch

***Summary***

A variant of the catch operator that also allows an `<Error>` failure type, and doesn't convert the failure type to `<Never>`.

***Constraints on connected publisher***

- *none*

***docs***

https://developer.apple.com/documentation/combine/publishers/trycatch

***Usage***

- Pattern 4: Requesting data from an alternate URL when the network is constrained

***Details***

`tryCatch` is a variant of catch that has a failure type of `<Error>` rather than catch's failure type of `<Never>`. This allows it to be used where you want to immediately react to an error by creating another publisher that may also produce a failure type.

**assertNoFailure**

***Summary***

Raises a fatal error when its upstream publisher fails, and otherwise republishes all received input and converts failure type to `<Never>`.

***Constraints on connected publisher***

- *none*

***docs***

https://developer.apple.com/documentation/combine/publishers/assertnofailure

***Usage***

- Pattern 3.1: verifying a failure hasn't happened using assertNoFailure

***Details***

If you need to verify that no error has occured (treating the error output as an invariant), this is the operator to use. Like it's namesakes, it will cause the program to terminate if the assert is violated.

Adding it into the pipeline requires no additional parameters, but you can include a string:

```
.assertNoFailure()
// OR
.assertNoFailure("What could possibly go wrong?")
```

I'm not entirely clear on where that string would appear if you did include it.

When trying out this code in unit tests, the tests invariably drop into a debugger at the assertion point when a .failure is processed through the pipeline.

If you want to convert an failure tyoe output of `<Error>` to `<Never>`, you probably want to look at the catch operator.

Apple asserts this function should be primarily used for testing and verifying "internal sanity checks that are active during testing".

**retry**

***Summary***

The retry opeator is used to repeat requests to a previous publisher in the event of an error.

***Constraints on connected publisher***

- failure type must be `<Error>`

***docs***

https://developer.apple.com/documentation/combine/publishers/retry

***Usage***

- Pattern 3.3: Retrying in the event of a temporary failure
- unit tests illustrating using map with dataTaskPublisher: `UsingCombineTests/DataTaskPublisherTests.swift`

***Details***

When you specify this operator in a pipeline and it receives a subscription, it first tries to request a subscription from it's upstream publisher. If the response to that subscription fails, then it will retry the subscription to the same publisher.

The retry operator accepts an optional (but recommended) single parameter that specifies a number of retries to attempt. If no number of retries is specified, it will attempt to retry indefinitely until it receives a .finished completion from it's subscriber.

Using retry without any specific count can result in your pipeline never resolving any data or completions. If you use retry without a count, you may also want to use the timeout operator to force a completion from the pipeline.

If the number of retries is specified and all requests fail, then the `.failure` completion is passed down to the subscriber of this operator.

In practice, this is mostly commonly desired when attempting to request network resources with an unstable connection. If you use a retry operator, you should add a specific number of retries so that the subscription doesn't effectively get into an infinite loop.

```
struct IPInfo: Codable {
    // matching the data structure returned from ip.jsontest.com
    var ip: String
}
let myURL = URL(string: "http://ip.jsontest.com")
// NOTE(heckj): you'll need to enable insecure downloads in your Info.plist for this
example
// since the URL scheme is 'http'

let remoteDataPublisher = URLSession.shared.dataTaskPublisher(for: myURL!)
    // the dataTaskPublisher output combination is (data: Data, response: URLResponse)
    .retry(3)
    // if the URLSession returns a .failure completion, try at most 3 times to get a
successful response
    .map({ (inputTuple) -> Data in
        return inputTuple.data
    })
    .decode(type: IPInfo.self, decoder: JSONDecoder())
    .catch { err in
        return Publishers.Just(IPInfo(ip: "8.8.8.8"))
    }
    .eraseToAnyPublisher()
```

#### mapError

- mapError
    - Converts any failure from the upstream publisher into a new error.

## Adapting publisher types

#### switchToLatest

- switchToLatest?

## Controlling timing

### debounce

- debounce
  - `.debounce(for: 0.5, scheduler: RunLoop.main)`
  - collapses multiple values within a specified time window into a single value
  - often used with `.removeDuplicates()`

### delay

- delay
  - Delays delivery of all output to the downstream receiver by a specified amount of time on a particular scheduler.
  - requires Failure to be `<Never>`

### measureInterval

- measureInterval
  - Measures and emits the time interval between events received from an upstream publisher.
  - requires Failure to be `<Never>`

### throttle

- throttle
  - Publishes either the most-recent or first element published by the upstream publisher in the specified time interval.
  - requires Failure to be `<Never>`

### timeout

***Summary***

Terminates publishing if the upstream publisher exceeds the specified time interval without producing an element.

***Constraints on connected publisher***

- requires Failure to be `<Never>`

***docs***

https://developer.apple.com/documentation/combine/publishers/timeout

***Usage***

- unit tests illustrating using retry and timeout with dataTaskPublisher: `UsingCombineTests/DataTaskPublisherTests.swift`

***Details***

> Timeout will force a resolution to a pipeline after a given amount of time, but does not guarantee either data or errors, only a completion. If a timeout does trigger and force a completion, it will not generate an failure completion with an error.

Timeout is specified with two parameters, a time period and a scheduler.

If you are using a specific background thread (for example, with the subscribe operator), then timeout should likely be using the same scheduler.

The time period specified will take a literal integer, but otherwise needs to conform to the protocol SchedulerTimeIntervalConvertible. If you want to set a number from a Float or Int, you need to create the relevant structure, as Int or Float directly doesn't conform. For example, if you're using a DispatchQueue, you could use DispatchQueue.SchedulerTimeType.Stride.

```
let remoteDataPublisher = urlSession.dataTaskPublisher(for: self.mockURL!)
    .delay(for: 2, scheduler: backgroundQueue)
    .retry(5) // 5 retries, 2 seconds each ~ 10 seconds for this to fall through
    .timeout(5, scheduler: backgroundQueue) // max time of 5 seconds before failing
    .tryMap { data, response -> Data in
        guard let httpResponse = response as? HTTPURLResponse,
            httpResponse.statusCode == 200 else {
                throw testFailureCondition.invalidServerResponse
        }
        return data
    }
    .decode(type: PostmanEchoTimeStampCheckResponse.self, decoder: JSONDecoder())
    .subscribe(on: backgroundQueue)
    .eraseToAnyPublisher()
```

## Encoding and decoding

### encode

***Summary***

Encode converts the output from upstream Encodable object using a specified TopLevelEncoder. For example, use JSONEncoder or PropertyListEncoder..

***Constraints on connected publisher***

- Available when Output conforms to Encodable.

***docs***

https://developer.apple.com/documentation/combine/publishers/encode

***Usage***

- unit tests illustrating using encode and decode: `UsingCombineTests/EncodeDecodeTests.swift`

***Details***

The encode operator takes a single parameters:

- `encoder` an instance of an object conforming to TopLevelEncoder, frequently an instance of JSONEncoder() or PropertyListEncoder().

```
fileprivate struct PostmanEchoTimeStampCheckResponse: Codable {
    let valid: Bool
}

let dataProvider = PassthroughSubject<PostmanEchoTimeStampCheckResponse, Never>()
    .encode(encoder: JSONEncoder())
    .sink { data in
        print(".sink() data received \(data)")
        let stringRepresentation = String(data: data, encoding: .utf8)
        print(stringRepresentation)
    })
```

Like the decode operator, the encode process can also fail and throw an error, so it returns a failure type of Error. With the compiler forcing type matching, the usual error condition is if you flow an optional value into the pipeline.

### decode

***Summary***

A very common operation is to want to use decode (or encode data in a pipline, so Combine provides an operator specifically suited to that task.

***Constraints on connected publisher***

- Available when Output conforms to Decodable.

***docs***

https://developer.apple.com/documentation/combine/publishers/decode

***Usage***

- Pattern 2.1: Making a network request with dataTaskPublisher
- Pattern 2.2: Stricter request processing with dataTaskPublisher
- Pattern 3.2: Using catch to handle errors in a one-shot pipeline
- Pattern 3.3: Retrying in the event of a temporary failure
- unit tests illustrating using encode and decode: `UsingCombineTests/EncodeDecodeTests.swift`

***Details***

The decode operator takes two parameters:

- `type` which is typically a reference to a struct you've defined
- `decoder` an instance of an object conforming to TopLevelDecoder, frequently an instance of JSONDecoder() or PropertyListDecoder().

Since decoding can fail, the operator will also return a failure type of Error. The data type returned by the operator is defined by the type you provided to decode.

```
let testUrlString = "https://postman-echo.com/time/valid?timestamp=2016-10-10"
// checks the validity of a timestamp - this one should return {"valid":true}
// matching the data structure returned from https://postman-echo.com/time/valid
fileprivate struct PostmanEchoTimeStampCheckResponse: Decodable, Hashable {
    let valid: Bool
}

let remoteDataPublisher = URLSession.shared.dataTaskPublisher(for: URL(string:
testUrlString)!)
    // the dataTaskPublisher output combination is (data: Data, response: URLResponse)
    .map { $0.data }
    .decode(type: PostmanEchoTimeStampCheckResponse.self, decoder: JSONDecoder())
```

## Working with multiple subscribers

### multicast

- multicast

## Debugging

### breakpoint

- breakpoint
  - Raises a debugger signal when a provided closure needs to stop the process in the debugger.

### breakpointOnError

- breakpointOnError
  - Raises a debugger signal upon receiving a failure.

### handleEvents

- handleEvents

### print

- print
  - Prints log messages for all publishing events.
  - requires Failure to be `<Never>`

## Scheduler and Thread handling operators

### receive

***Summary***

Receive defines the scheduler on which to receive elememts from the publisher.

***Constraints on connected publisher***

- *none*

***docs***

`receive`

***Usage***

- Pattern 1.2: Creating a subscriber with assign shows an example of using assign to set an a boolean property on a UI element.
- unit tests illustrating using an assign subscriber in a pipeline from a dataTaskPublisher with subscribe and receive: `UsingCombineTests/SubscribeReceiveAssignTests.swift`

***Details***

Receive takes a single required parameter (`on:`) which accepts a scheduler, and an optional parameter (`optional:`) which can accept SchedulerOptions. Scheduler is a protocol in Combine, with the conforming types that are commonly used of RunLoop, DispatchQueue and OperationQueue. Receive is frequently used with assign to make sure any following pipeline invocations happen on a specific thread, such as `RunLoop.main` when updating user interface objects. Receive effects itself and any opertors chained after it, but not previous operators. If you want to influence previously chained publishers (or operators) for where to run, use the subscribe operator.

```
examplePublisher.receive(on: RunLoop.main)
```

Receive takes a single

### subscribe

***Summary***

Subscribe defines the scheduler on which to run operators in a pipeline.

***Constraints on connected publisher***

- *none*

***docs***

`subscribe`

***Usage***

- Pattern 1.2: Creating a subscriber with assign shows an example of using assign to set an a boolean property on a UI element.

- unit tests illustrating using an assign subscriber in a pipeline from a dataTaskPublisher with subscribe and receive: `UsingCombineTests/SubscribeReceiveAssignTests.swift`

***Details***

Subscribe assigns a scheduler to any preceding pipeline invocations, and is often used to invoke a publisher on a background thread or queue. When used in this fashion, it is often used in coordination with receive to transfer data to another thread (such as the main runloop) for following operators or the subscriber.

Subscribe takes a single required parameter (`on:`) which accepts a scheduler, and an optional parameter (`optional:`) which can accept SchedulerOptions. Scheduler is a protocol in Combine, with the conforming types that are commonly used of RunLoop, DispatchQueue and OperationQueue.

Subscribe effects itself and any opertors chained before it, but not following operators. If you want to influence chained operators after subscribe for where to run, use the receive operator. The most comon example of this is receiving on `RunLoop.main`, critical when updating user interface objects.

```
networkDataPublisher
    .subscribe(on: backgroundQueue) ①
    .receive(on: RunLoop.main) ②
    .assign(to: \.text, on: yourLabel) ③
```

① the `subscribe` call requests the publisher (and any pipeline invocations before this in a chain) be invoked on the backgroundQueue.

② the `receive` call transfers the data to the main runloop, suitable for updating user interface elements

③ the `assign` call uses the assign subscriber to update the property `text` on a KVO compliant object, in this case `yourLabel`.

When creating a DispatchQueue to use with Combine publishers on background threads, it is recommended that you use a regular serial queue rather than a concurrent queue to allow Combine to adhere to its contracts. That is - don't create the queue with `attributes: .concurrent`.

## Type erasure operators

### eraseToAnyPublisher

- when you chain operators together in swift, the object's type signature accumulates all the various types, and it gets ugly pretty quickly.
- eraseToAnyPublisher takes the signature and "erases" the type back to the common type of AnyPublisher
- this provides a cleaner type for external declarations (framework was created prior to Swift 5's opaque types)
- `.eraseToAnyPublisher()`
- often at the end of chains of operators, and cleans up the type signature of the property getting asigned to the chain of operators

### eraseToAnySubscriber

### eraseToAnySubject

# Subjects

## currentValueSubject

- `CurrentValueSubject`

## PassthroughSubject

- `PassthroughSubject`

# Subscribers

For general information about subscribers and how they fit with publishers and operators, see Subscribers.

## assign

### *Summary*

Assign creates a subscriber used to update a property on a KVO compliant object.

### *Constraints on connected publisher*

- Failure type must be `<Never>`

### *docs*

`assign`

### *Usage*

- Pattern 1.2: Creating a subscriber with assign shows an example of using assign to set an a boolean property on a UI element.
- unit tests illustrating using an assign subscriber in a pipeline from a dataTaskPublisher with subscribe and receive: `UsingCombineTests/SubscribeReceiveAssignTests.swift`

### *Details*

Assign only handles data, and expects all errors or failures to be handled in the pipeline before it is invoked. The return value from setting up assign can be cancelled, and is frequently used when disabling the pipeline, such as when a viewController is disabled or deallocated. Assign is frequently used in conjunction with the receive operator to receive values on a specific scheduler, typically `RunLoop.main` when updating UI objects.

```
examplePublisher
    .receive(on: RunLoop.main) ②
    .assign(to: \.text, on: yourLabel) ③
```

## sink

### *Summary*

Sink creates an all-purpose subscriber. At a minimum, you provide a closure to receive values, and optionally a closure that receives completions.

### *Constraints on connected publisher*

- *none*

### *docs*

`sink`

### *Usage*

- Pattern 1.1: Creating a subscriber with sink shows an example of creating a sink that

receives both completion messages as well as data from the publisher.

- unit tests illustrating a sink subscriber and how it works: `UsingCombineTests/SinkSubscriberTests.swift`

***Details***

The simplest form of `.sink()` takes a single closure - by default this closure receives data (if provided by the attached publisher).

```
let examplePublisher = Just(5)

let _ = examplePublisher.sink { value in
    print(".sink() received \(String(describing: value))")
}
```

The closure you provide is invoked for every update that the publisher passes down, up until the completion. Be aware that the single closure form may be called repeatedly. How often it is called depends on the pipeline to which it is subscribing.

If you don't also include a closure to get the completion, you will not receive any information about failures. If an error or failure occurs and is handed down from the publisher the single closure form will not be called.

If you are creating a subscriber and want to receive failures, or see the completion messages at the end of pipeline, create a sink with two closures. The more complete sink has the two closures named `receiveCompletion` and `receiveValue`:

```
let examplePublisher = Just(5)

let _ = examplePublisher.sink(receiveCompletion: { err in
    print(".sink() received the completion", String(describing: err))
}, receiveValue: { value in
    print(".sink() received \(String(describing: value))")
})
```

The type that is passed into receiveCompletion is the enum `Subscribers.Completion`. The completion `.failure` incudes an Error wrapped within it, providing access to the underlying cause of the failure. To get to the error within the `.failure` completion, `switch` on the returned completion to determine if it is `.finished` or `.failure`, and then pull out the error.

When you chain a `.sink` subscriber onto a publisher (or pipeline), the result is cancellable. At any time before the publisher sends a completion, the subscriber can send a cancellation and invalidate the pipeline. After a cancel is sent, no further values will be received by either closure in the sink.

```
let simplePublisher = PassthroughSubject<String, Error>()
let cancellablePipeline = simplePublisher.sink { data in
  // do what you need with the data...
}

cancellablePublisher.cancel() // this invalidates the pipeline, no further data will
be received by the sink
```