Using Combine

Joseph Heck

Version 0.1, 2019-06-23

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 large cloud-based distributed systems. He has established development processes, CI and CD pipelines, and developed validation and operational automation for software solutions. Joe also builds teams 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 from GitHub.

Download the project

The project associated with this book requires Xcode v11 (which has been released as beta2 as of this writing) and MacOS 10.14 or later.

Welcome to Xcode

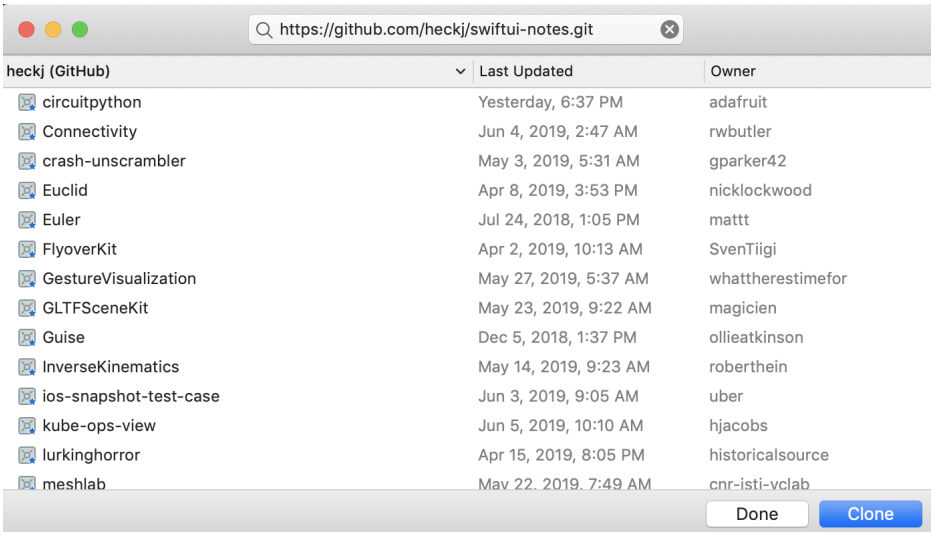
Version 11.0 beta 2 (11M337n)

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 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 strongly-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. As this happens over time, you can view the updates as a stream of objects.

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. Handling the flow of all these event streams, the timing of all the various pieces, errors when they happen, and coordinating how a system responds to all those events is at the heart of this kind of programming.

This solution is particularly effective when programming user interfaces, or for creating pipelines that process and transform 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 was also set up to be composed, and Combine is explicitly 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. NotificationCenter, URLSession, and Timer in the Foundation also now include 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 system that is "immediately" reactive to a variety of inputs. User interfaces fit very naturally into this pattern.

The classic examples revolve around the user experience with form validation, where the inputs are text fields and a button for submitting.

- 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 a 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 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 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 the **publisher** and the **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 the Publisher protocol, subscribing to one or more Publishers, and sending results to one (or more) 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 _ = Publishers.somePublisher
    .map { value -> String in
        // do something with the incoming value here
        // and return a string
        return "a string"
    }
    .sink { receivedValue in
        // sink is the subscriber and terminates the pipeline
        print("The end result was \(receivedValue)")
    }
```

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, as some of operators will either require conformance of a type, or will change the Failure output type.

Because the types are enforced, there are a number of Combine functions that are created to help with these transformations. Some operators are prefixed with `try` that indicate that they will return an <Error> type. In some cases, you may need to define the type being returned in the closure you're providing to the operator.

An example of this is map and tryMap. `map` 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.

To illustrate changing types within a pipeline, here is a short snippet thsat starts with a publisher that generates `<Int>`, `<Never>` and end with a subscription taking `<String>`, `<Never>`.

```
let _ = Publishers.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 creates an `<Int>`, `<Never>` type combination

② the `.map()` function takes in an `<Int>`, `<Never>` combination and transforms it into a `<String>`,

③ 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) displays this as the error message: `Unable to infer complex closure return type; add explicit type to disambiguate`.

Combine supports error handling by creating two streams - one for the functional case and one for the error case, and combining them together. We will see that in more detail in the section on patterns.

Lifecycle of Publishers and Subscribers

The interals of Combine are all driven by 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 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 built on subscribers and publishers communicating in a well defined sequence, or lifecycle.

- When the subscriber is attached to a publisher, it starts with a call to `.subscribe(Subscriber)`.
- 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()`
- 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, or terminating with an explicit completion enumeration.

Combine provides a number of convenience publishers:

`Publishers.Empty`	`Publishers.Fail`	`Publishers.Just`
`Publishers.Once`	`Publishers.Optional`	`Publishers.Sequence`
`Publishers.Deferred`	Publishers.Future	@Published

Other Apple APIs provide publishers as well

Combine allows you to create a publisher with the `@Published` property wrapper to provide a publisher for a specific property.

SwiftUI provides `@ObjectBinding` which can be used to create a publisher.

A number of other Apple APIs provide publishers as well:

- `NotificationCenter .publisher`

- `Timer .publish` and `Timer.TimerPublisher`
- `URLSession dataTaskPublisher`
- `RealityKit .Scene .publisher()`

Combine also includes mechanisms to allow you to create your own publishers with `Publishers.Future`. A future is initialized with a closure that eventually resolves to a Promise. This can be used to wrap any existing API (Apple or your own) that provides a completion closure to turn it into a 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	map
tryMap	flatMap	setFailureType

Filtering elements		
compactMap	tryCompactMap	filter
tryFilter	removeDuplicates	tryRemoveDuplicates
replaceEmpty	replaceError	

Reducing elements		
collect	collectByCount	collectByTime
ignoreOutput	reduce	tryReduce

Mathematic opertions on elements		
comparison	tryComparison	count

Applying matching criteria to elements		
allSatisfy	tryAllSatisfy	contains
containsWhere	tryContainsWhere	

Applying sequence operations to elements

first	firstWhere	tryFirstWhere
last	lastWhere	tryLastWhere
dropUntilOutput	dropWhile	tryDropWhile
concatenate	drop	prefixUntilOutput
prefixWhile	tryPrefixWhile	output

Combining elements from multiple publishers		
combineLatest	tryCombineLatest	merge
zip		

Handling errors		
assertNoFailure	catch	retry

Adapting publisher types		
switchToLatest		

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 `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.

There are two built-in subjects with Combine:

The first is `CurrentValueSubject`.

- CurrentValue remembers the current value so that when you attach a subscriber you can see 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 the sending of any requested data (or a failure) is under test control using the `.send()` function.

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

Subjects can also be useful for fanning out values to multiple subscribers.

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 Publishers.Future<String, Error> { promise in
            promise(.success(""))
            }.catch { _ in
                Publishers.Just("No user found")
            }.map { result in
                return "\(result) foo"
        }
}
```

The resulting type would reflect that composition:

```
Publishers.FlatMap<Publishers.Map<Publishers.Catch<Publishers.Future<String, Error>,
Publishers.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 Publishers.Future<String, Error> { promise in
            promise(.success(""))
            }.catch { _ in
                Publishers.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: Creating a subscriber with sink

If you are creating the subscriber, you can use `.sink()` to receive and process this information. The simplest form of `.sink()` takes a single closure - but by default this is the closure that receives data (if provided by the pipeline). If you don't also include a closure to get the completion, you will not receive any information about failures.

Remember that subscribers drive the execution of any Combine pipelines you create. If you end a pipeline with `.sink()` you are creating the subscriber and connecting it to the pipeline. By chaining it, it will implicitly start the lifecycle with the `subscribe` and request for unlimited data.

A simple sink is created with a single closure:

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

The closure gets invoked for every new update that the publisher creates, up until the completion. If an error or failure occurs, then the closure will simply never be called.

When you write the closure for a `sink` subscriber, be aware that your receiveValue closure may be called repeatedly. How often it is called depends on the pipeline to which it is subscribing.

If you are creating a subscriber and want to receive and handle 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 _ = remoteDataPublisher.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 within it, which provides access to the underlying cause of the failure. If

you want to get to the error in the subscriber, you can `switch` on the returned completion to determine if it is `.finished` or `.failure`, and then pull out the error:

```
.sink(receiveCompletion: { completion in
    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("GOT 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)")
})
```

Pattern 2: Making a network request with dataTaskPublisher

One of the common use cases is requesting JSON data from a URL and decoding it. This can be readily accomplished with Combine using dataTaskPublisher on URLSession. The data that is returns down the pipeline is a tuple: `(data: Data, response: URLResponse)`

The simplest case of using this might be:

```
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())
    .eraseToAnyPublisher()
```

With this example, the subscriber will get an instance of the struct IPInfo or will receive a completion notification with an error. The dataTaskPublisher makes a single request, and what you build in the pipeline will determine what the subscriber receives.

A failed URL request will result in a `.failure` completion with an encapsulated error. Likewise, if the returned data couldn't be decoded based on the structure, or wasn't JSON, then the result will be a `.failure` completion with an encapsulated decoding error from the decoder.

To have more control over what is considered a failure in the URL response, use a `tryMap` operator on the tuple response from dataTaskPublisher. An example of that might look like:

```
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'

enum testFailureCondition: Error {
    case invalidServerResponse
}

let remoteDataPublisher = URLSession.shared.dataTaskPublisher(for: myURL!)
    // the dataTaskPublisher output combination is (data: Data, response: URLResponse)
    .tryMap { data, response -> Data in
        // this give you the full capability to react to both the data and the
HTTPURLResponse
        // with the expectation that you will throw an error if a failure completion
is warranted.
        guard let httpResponse = response as? HTTPURLResponse,
            // this will casue an .failure completion on anything other than a 200
response
            httpResponse.statusCode == 200 else {
                throw testFailureCondition.invalidServerResponse
        }
        return data
    }
    .decode(type: IPInfo.self, decoder: JSONDecoder())
    .eraseToAnyPublisher()
```

The above example uses tryMap to allow for an error to be thrown within the closure. This lets you put in logic that inspects the specific status code, or you could inspect the data prior to sending it into `JSONDecoder`.

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.

Pattern 3: Handling errors within the pipline

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 3.1: 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 3.2: Using flatMap with catch to handle errors.

converting to a Never failure type using assertNoFailure

A brutal way to handle this is to crash the application when an error happens. Combine provides the operator `assertNoFailure()` for this purpose. This is useful 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>`.

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.

It is far more likely that you want to handle the error and not have the application crash.

Pattern 3.1: Using catch to handle errors in a one-shot pipeline

The `.catch()` operator is useful to recover from an error, but it can have unexpected side effects if you aren't familiar with how it works. `.catch()` handles errors by replacing the upstream publisher with another publisher that you provide as a return in a closure. 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())
    .catch { err in
        return Publishers.Just(IPInfo(ip: "8.8.8.8"))
    }
    .eraseToAnyPublisher()
```

Now the `remoteDataPublisher` can be used with

```
.receive(on: RunLoop.main)
.assign(to: \.isEnabled, on: yourButton)
```

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. This means 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 detail and an example.

retrying in the event of a temporary failure

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

```
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()
```

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 3.2: Using flatMap with catch to handle errors

The `flatMap()` operator is exactly what we need in this use case.

You can think of the `flatMap()` operator as a way to inject values into your pipeline from a temporary one-shot publisher. For every element `flatMap()` receives, it invokes it's closure to create a publisher. That publisher is then sending values to any downstream subscriber.

This is a perfect mechanism for when you want to maintain updates up an upstream publisher, as it effectively creates one-shot publishers (or even small pipelines) that send a single value and then complete. The completion from the created one-shot publishers terminates in the flatMap and isn't passed to downstream subscribers. To use this with error handling, we can create a one-shot publisher, or pipeline, that starts with the value provided upstream. Then we create the pipeline to do any potentially failing work, and construct it with a `.catch` as we saw earlier to provide a fallback value.

A diagram version of this pipelines 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)
    }
}
```

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

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.

This sample is originally from the WWDC session, but the API and example has evolved with the beta's of Combine since that presentation. `tryCatch` in particular appears to have been removed. I suspect `mapError` is an appropriate replacement, but that detail has yet to be confirmed.

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

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

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: Coordinating a sequence of asynchronous operations

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

Using Future to turn an an asynchronous call into publisher

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

Pattern 8: binding

- binding to SwiftUI
  - validating forms
  - UX responsiveness - live updates to view properties
  - handling error within property update
  - retry for remote service

simple case - data validation with external service

```
@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 N: Testing pipelines

test strategies with combine

- testing streams/pipelines
- testing publishers
- testing subscribers

using PassthroughSubject and creative sinks

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

Publishers.Empty

Summary

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

docs

`Publishers.Empty`

Usage

- Pattern 3.1: 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 3.2: 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.Just

Summary

`just` provides a single result and then terminates.

docs

`Publishers.Just`

Usage

n/a

Details

- Just → `<SomeType>`, `<Never>`
  - often used in error handling

Publishers.Once

Summary

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

docs

`Publishers.Once`

Usage

n/a

Details

n/a

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

Publishers.Future

Summary

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

docs

Publishers.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)`

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`
- part of URLSession
    - dataTaskPublisher
    - two versions, on taking a type `URL`, the other `URLSession
    - outputs `URLSession.DataTaskPublisher`

```
var 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

- map
  - you provide a closure that gets the values and chooses what to publish
  - there's a variant `tryMap` that that transforms all elements from the upstream publisher with a provided error-throwing closure.

tryMap

- tryMap

flatpMap

- flatMap
  - collapses nil values out of a stream
  - used with error recovery or async operations that might fail (ex: Future)
  - requires Failure to be `<Never>`

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 Pattern 3: Handling errors within the pipline for more detail on how you can design error handling.

assertNoFailure

- assertNoFailure
  - Raises a fatal error when its upstream publisher fails, and otherwise republishes all received input.

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 publisher

- *none*

Documentation reference

- `Publishers.Catch`

Usage

- Pattern 3.1: 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 3.2: 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 Publishers.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 (`Publishers.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.

retry

- retry
    - multiple variants - once or by a provided count
    - forces Failure type of Never on output

mapError

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

Adapting publisher types

- 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

- timeout
  - Terminates publishing if the upstream publisher exceeds the specified time interval without producing an element.
  - requires Failure to be `<Never>`

Encoding and decoding

encode

- encode
  - Encodes the output from upstream using a specified TopLevelEncoder. For example, use JSONEncoder or PropertyListEncoder
  - Available when Output conforms to Encodable.

decode

- decode
  - common operating where you hand in a type of decoder, and transform data (ex: JSON) into an object
  - can fail, so it returns an error type
  - Available when Output conforms to Decodable.
  - → `<SomeType>`, `<Error>`

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

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

subscribe

- subscribe(on:)

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

assign

- `assign`

key-path assignment **ex:** `Subscribers.Assign(object: exampleObject, keyPath: \.someProperty)` ex: `.assign(to: \.isEnabled, on: signupButton)` **Assigns the value of a KVO-compliant property from a publisher.** requires Failure to be `<Never>`

sink

- `sink`
  - you provide a closure where you process the results
  - ex:

```
let cancellablePublisher = somePublisher.sink { data in
  // do what you need with the data...
}

cancellablePublisher.cancel() // to kill the stream before it's complete
```