# Combine Intro

Daniel H Steinberg

## A Workshop

**Winter 2020-1**

**iOS SG**

http://dimsumthinking.com

# Combine Intro

**Daniel H Steinberg**

## A Workshop

**Winter 2020-1**

**iOS SG**

**http://dimsumthinking.com**

# Xcode 12 : Swift 5.3

# Introducing Combine

## A Workshop

## Recommended Settings

The ePub is best viewed in scrolling mode using the original fonts. The ePub and Mobi versions of this book are best read in single column view.

## Contact

Instructor: Daniel H Steinberg

Dim Sum Thinking at http://dimsumthinking.com email: inquiries@dimsumthinking.com

View the complete Course List.

## Copyright

## Legal

Every precaution was taken in the preparation of this material. The publisher and author assume no responsibility for errors and omissions, or for damages resulting from the use of the information contained herein and in the accompanying code downloads.

The sample code is intended to be used to illustrate points made in the text. It is not intended to be used in production code.

SECTION 1

# Warm up

We are going to begin with a UIKit app and enable some of it before starting our move to Combine.

Before we get started, I want to point out that we will have a completely working Rock Paper Scissors app in a few minutes.

The point of this example is not to build this app. The point is to learn combine in the context of building this app.

## The Project

You'll find the RockPaperScissors project in the 00 folder of the code download.

Open it and run it.

The simulator should pop up and display a screen with some images containing question marks, a few labels, a segmented control and a button.

## View Controller

To begin with the view controller contains four outlets - one to each Image view and one to the segmented control. It also contains an action for the button.

*01 /RockPaperScissors/RockPaperScissors/ViewController.swift*

```
class ViewController: UIViewController {
  @IBOutlet private weak var resultView: UIImageView!
  @IBOutlet private weak var playerView: UIImageView!
  @IBOutlet private weak var responseView: UIImageView!
  @IBOutlet private weak var playerSelector: UISegmentedControl!

  @IBAction func enterChoice(_ button: UIButton) {
  }
}
```

As you move to SwiftUI and Combine, the way we work in UIKit is going to feel intrusive and less easy to reason about.

As an analogy, I feel the same about UIKit vs SwiftUI and/or Combine as I did about ObjectiveC vs Swift. There is nothing wrong with the techniques on the left side of the vs but you will come to like the techniques on the right side.

So let's begin by implementing this as we would in UIKit.

For now we will do nothing with the first action.

## Actions

In UIKit, when a button is tapped an action is called. This was a brilliant mechanism that allowed Apple to write UIButton without it having to know what the button actually does. The action will proscribe the behavior.

When the button is tapped, we want to retrieve the choice the player has made using the segmented control and display the image on the screen.

You'll find a function in *HandPosition.swift* named `setHandPosition(for:)` that allows us to convert an `Int` representing the selected button into a `HandPosition`. There is a computed property named `imageName` that returns the name of the SF Symbol representing that position.

Add this code to our action.

*01 /RockPaperScissors/RockPaperScissors/ViewController.swift*

```
  @IBAction func enterChoice(_ button: UIButton) {
    let player = setHandPosition(for: playerSelector.selectedSegmentIndex)

    playerView.image = UIImage(systemName: player.imageName)
  }
```

Run the app. When you tap the button you should see the image in the middle-left slot.

## A Response

To get the response, we use a second function in *HandPosition.swift* this is called `randomHandPosition()`.

*01 /RockPaperScissors/RockPaperScissors/ViewController.swift*

```
  @IBAction func enterChoice(_ button: UIButton) {
    let player = setHandPosition(for: playerSelector.selectedSegmentIndex)
    let response = randomHandPosition()

    playerView.image = UIImage(systemName: player.imageName)
    responseView.image = UIImage(systemName: response.imageName)
  }
}
```

Run the app again. Now you should be seeing the images for both the player choice and the response.

## The Result

This time we use the function `gameResult(for: against:)` from `GameResult.swift` and the computed properties `imageName` and `uiImageColor` to represent the result as a win, loss, or draw.

*01 /RockPaperScissors/RockPaperScissors/ViewController.swift*

```
  @IBAction func enterChoice(_ button: UIButton) {
    let player = setHandPosition(for: playerSelector.selectedSegmentIndex)
    let response = randomHandPosition()
    let result = gameResult(for: player, against: response)

    playerView.image = UIImage(systemName: player.imageName)
    responseView.image = UIImage(systemName: response.imageName)
    resultView.image = UIImage(systemName: result.imageName)
    resultView.tintColor = result.uiImageColor
  }
```

Run the app and you should see both hand positions and the result of the game.

In two sections we add some complexity more typical of the apps we write.

Next, let's do a quick example of Combine to show how it can be used in place.

SECTION 2

# Introducing Combine

Let's introduce three of the four main pillars of Combine: Publishers, Operators, and Subscribers.

We're also going to remove some of the functionality of this app to keep us focused.

## Selecting a value

The button action does too much. For now, let's add a property to hold the value of the `playerChoice`. When the button is tapped, we'll update this value.

*02/RockPaperScissors/RockPaperScissors/ViewController.swift*

```
class ViewController: UIViewController {
  @IBOutlet private weak var resultView: UIImageView!
  @IBOutlet private weak var playerView: UIImageView!
  @IBOutlet private weak var responseView: UIImageView!
  @IBOutlet private weak var playerSelector: UISegmentedControl!
  private var playerChoice = 0

  @IBAction func enterChoice(_ button: UIButton) {
    playerChoice = playerSelector.selectedSegmentIndex
  }
}
```

Run the app. Nothing seems to be happening.

I find it helpful in these situations to add a `didSet`.

*02/RockPaperScissors/RockPaperScissors/ViewController.swift*

```
class ViewController: UIViewController {
  @IBOutlet private weak var resultView: UIImageView!
  @IBOutlet private weak var playerView: UIImageView!
  @IBOutlet private weak var responseView: UIImageView!
  @IBOutlet private weak var playerSelector: UISegmentedControl!
  private var playerChoice = 0 {
    didSet {
      print(playerChoice)
    }
  }

  @IBAction func enterChoice(_ button: UIButton) {
    playerChoice = playerSelector.selectedSegmentIndex
  }
}
```

Now we can see that `playerChoice` is being updated.

## Publisher

We can just publish the new value every time `playerChoice` updates using `@Published`.

Publishers can publish a finite number of times or a continuing number of times. If a publisher completes it will send the message that it is finished. This one publishes as long as the view controller is alive.

Publishers can also fail for a variety of reasons. This one can never fail. How could it fail publishing an Int when it changes? Failing publishers will indicate why they fail with an Error. If they can't possibly fail, their error type is Never.

*02/RockPaperScissors/RockPaperScissors/ViewController.swift*

```
class ViewController: UIViewController {
  @IBOutlet private weak var resultView: UIImageView!
  @IBOutlet private weak var playerView: UIImageView!
  @IBOutlet private weak var responseView: UIImageView!
  @IBOutlet private weak var playerSelector: UISegmentedControl!
  @Published private var playerChoice = 0 {
    didSet {
      print(playerChoice)
    }
  }

  @IBAction func enterChoice(_ button: UIButton) {
    playerChoice = playerSelector.selectedSegmentIndex
  }
}
```

`@Published` is a property wrapper that (in this case) gives us a publisher of Ints that never fails.

`playerChoice` is an `Int` while `$playerChoice` is a `Publisher<Int, Never>`.

`@Published` doesn't require that we import combine.

Notes:

`@Published` doesn't require that we import combine.

`@Published` works like the `CurrentValueSubject`. It publishes the existing value to subscribers and every subsequent value. A `PassthroughSubject` doesn't give you the current value just every value after you subscribe.

`@Published` actually doesn't publish anything unless someone is subscribed.

## Subscriber

We attach a subscriber to a publisher. There are three basic built in subscribers. One is `sink` and the other two are `assign`. We begin with `sink`.

You'll notice there are two forms of `sink`. One receives the completion and value and the other just receives a value. For now, because our publisher never fails or finishes we won't receive a completion. so we just use the version that accepts a value and we implement it using a trailing closure.

Set up the pipeline from the publisher `$playerChoice` to our subscriber. We'll do this in `viewDidLoad()`.

*02/RockPaperScissors/RockPaperScissors/ViewController.swift*

```
class ViewController: UIViewController {
  @IBOutlet private weak var resultView: UIImageView!
  @IBOutlet private weak var playerView: UIImageView!
  @IBOutlet private weak var responseView: UIImageView!
  @IBOutlet private weak var playerSelector: UISegmentedControl!
  @Published private var playerChoice = 0 {
    didSet {
      print(playerChoice)
    }
  }

  override func viewDidLoad() {
    super.viewDidLoad()
    $playerChoice.sink{[weak self] int in  // there's a warning
      let player = setHandPosition(for: int)
      self?.playerView.image = UIImage(systemName: player.imageName)
    }
  }

  @IBAction func enterChoice(_ button: UIButton) {
    playerChoice = playerSelector.selectedSegmentIndex
  }
}
```

There's a warning. Also the app builds and runs but nothing happens when you tap the button.

That involves another piece of the pipeline.

## Subscription

The connection from a publisher to a subscriber must be kept alive by a subscription. `sink` returns a token that we need to retain in order to do so. The token is a `Cancellable`. If you've worked with Notifications this should feel familiar.

We declare a property named `cancellable` of type `AnyCancellable?` so it is initialized to `nil`. `cancellable` can't be something that would go out of scope - so we need it to be a property.

Note - this is the first time we will have to import `Combine`.

Then we set `cancellable` equal to what we get back from our subscription.

Since this will now work, let's eliminate the didSet.

*02/RockPaperScissors/RockPaperScissors/ViewController.swift*

```
import UIKit
import Combine

class ViewController: UIViewController {
  @IBOutlet private weak var resultView: UIImageView!
  @IBOutlet private weak var playerView: UIImageView!
  @IBOutlet private weak var responseView: UIImageView!
  @IBOutlet private weak var playerSelector: UISegmentedControl!
  @Published private var playerChoice = 0 {
    didSet {
      print(playerChoice)
    }
  }
  private var cancellable: AnyCancellable?

  override func viewDidLoad() {
    super.viewDidLoad()
    cancellable =
      $playerChoice.sink{[weak self] int in
        let player = setHandPosition(for: int)
        self?.playerView.image = UIImage(systemName: player.imageName)
      }
  }

  @IBAction func enterChoice(_ button: UIButton) {
    playerChoice = playerSelector.selectedSegmentIndex
  }
}
```

Run the app. The button now works.

The only remaining issue is that we start with a rock because of the way `@Published` works.

## Operator

A lot of the work in Combine is done by modifying our values between the publisher and the subscriber. We do this with

Operators.

Operators are not subscribers - we still need a downstream subscriber or no values are emitted by the publisher. Think of the operators as passively receiving (without requesting) values from upstream, making modifications, and passing the modified values downstream.

Operators can influence which items we receive, transform the items, split or connect pipelines.

This time we want to eliminate the first element in the stream so we use the `dropFirst()` operator.

*02/RockPaperScissors/RockPaperScissors/ViewController.swift*

```
import UIKit
import Combine

class ViewController: UIViewController {
  @IBOutlet private weak var resultView: UIImageView!
  @IBOutlet private weak var playerView: UIImageView!
  @IBOutlet private weak var responseView: UIImageView!
  @IBOutlet private weak var playerSelector: UISegmentedControl!
  @Published private var playerChoice = 0
  private var cancellable: AnyCancellable?

  override func viewDidLoad() {
    super.viewDidLoad()
    cancellable =
      $playerChoice
      .dropFirst()
      .sink{[weak self] int in
        let player = setHandPosition(for: int)
        self?.playerView.image = UIImage(systemName: player.imageName)
      }
  }

  @IBAction func enterChoice(_ button: UIButton) {
    playerChoice = playerSelector.selectedSegmentIndex
  }
}
```

This works perfectly.

## So ...

On the one hand, we've used a lot more code to do a lot less than our existing version.

On the other hand, you've been introduced to the four components of Combine and have seen them working together.

Next, we add some gratuitous complexity to our app to structure more like what you might actually be dealing with. In addition, we want to keep our ViewControllers small and light.

SECTION 3

# Spreading Out

We're going to build out our model and add something in between the view controller and the model.

I don't care what you call any of this. Some people call them controllers, others view models or presenters. I'm going to use controller to generically refer to these things. I'm going to place some of these classes in the model but I don't really feel strongly one way or another about this. It's just a way of organizing our code. I leave the arguments on the proper way to do this to you and your team.

## Hand

Add a new Swift file named `Hand` to the `Model` group.

`Hand` contains a single property named `handPosition` which we initialize to `rock`.

*03/RockPaperScissors/RockPaperScissors/Model/Hand.swift*

```
class Hand {
  var handPosition: HandPosition = .rock
}
```

We could easily create a player and response as two instance of `Hand` using our functions `setHandPosition(for:)` and `randomHandPosition()`, but it could be useful to distinguish between two types `Player` and `Response` that are modified differently.

## Player

`Player` is a subclass of `Hand`. Its `handPosition` is initialized to `rock` but we can modify it based on the segmented control.

Create a new Swift file named `Player` in `Model`.

*03/RockPaperScissors/RockPaperScissors/Model/Player.swift*

```
class Player: Hand {
  func setPosition(to int: Int) {
    handPosition = setHandPosition(for: int)
  }
}
```

Now we can create and modify and instance of `Player`.

## Response

Similarly, create a Swift file named `Response` in `Model`. It too will be a subclass of `Hand` but its `handPosition` is updated to a random value.

*03/RockPaperScissors/RockPaperScissors/Model/Player.swift*

```
class Response: Hand {
  func nextPosition() {
    handPosition = randomHandPosition()
  }
}
```

Next we create a class that manages player and response.

## Game

Create a Swift file named `Game`. To begin with it has private properties `player` and `response`.

*03/RockPaperScissors/RockPaperScissors/Game.swift*

```
class Game {
  private let player = Player()
  private let response = Response()
}
```

The view controller will know about `Game` but nothing else. `Game` will not know about view controller.

Add this private property for `game` to `ViewController`.

*03/RockPaperScissors/RockPaperScissors/ViewController.swift*

```
class ViewController: UIViewController {
  @IBOutlet private weak var resultView: UIImageView!
  @IBOutlet private weak var playerView: UIImageView!
  @IBOutlet private weak var responseView: UIImageView!
  @IBOutlet private weak var playerSelector: UISegmentedControl!
  @Published private var playerChoice = 0
  private var cancellable: AnyCancellable?
  private let game = Game() // ...
}
```

So how does `ViewController` contact the player to ask for the next value?

## Relaying the action

Add this function to an extension of `Game`.

*03/RockPaperScissors/RockPaperScissors/Game.swift*

```
class Game {
  private let player = Player()
  private let response = Response()
}

extension Game {
  func setPlayer(_ int: Int){
    player.setPosition(to: int)
  }
}
```

Now `ViewController` sends `game` the `setPlayer` message to pass along the selection.

*03/RockPaperScissors/RockPaperScissors/ViewController.swift*

```
class ViewController: UIViewController {
  @IBOutlet private weak var resultView: UIImageView!
  @IBOutlet private weak var playerView: UIImageView!
  @IBOutlet private weak var responseView: UIImageView!
  @IBOutlet private weak var playerSelector: UISegmentedControl!
  @Published private var playerChoice = 0
  private var cancellable: AnyCancellable?
  private let game = Game()

  override func viewDidLoad() {
    super.viewDidLoad()
    cancellable =
      $playerChoice
      .dropFirst()
      .sink{[weak self] int in
        let player = setHandPosition(for: int)
        self?.playerView.image = UIImage(systemName: player.imageName)
      }
  }

  @IBAction func enterChoice(_ button: UIButton) {
    game.setPlayer(playerSelector.selectedSegmentIndex)
  }
}
```

I love that we pass functions around the same way we pass around Strings or other types.

## Sending now works

Add a didSet to `Hand` to verify that our connection works.

*03/RockPaperScissors/RockPaperScissors/Model/Hand.swift*

```
class Hand {
  var handPosition: HandPosition = .rock {
    didSet {
      print("set hand to:", handPosition)
    }
  }
}
```

Run the app and you can see that player's hand position is being set.

How do we get this information back to ViewController?

## Notification Center

Now that we know we're setting the `handPosition` when we tap the button, we could replace the `print` with a post to the `NotificationCenter`.

The posting does not use Combine. We do, however, need to import Foundation

*03/RockPaperScissors/RockPaperScissors/Model/Hand.swift*

```
import Foundation

class Hand {
  var handPosition: HandPosition = .rock {
    didSet {
      NotificationCenter
        .default
        .post(name: NSNotification.Name("player"),
              object: self)
    }
  }
}
```

I will keep pointing out the parts that don't use Combine as I think it helps us notice where we do use Combine.

The advantage of Combine being an Apple API is that they've integrated it into their existing APIs. For example we can ask `NotificationCenter` for a publisher and add that to a pipeline like this.

*03/RockPaperScissors/RockPaperScissors/ViewController.swift*

```
class ViewController: UIViewController {
  @IBOutlet private weak var resultView: UIImageView!
  @IBOutlet private weak var playerView: UIImageView!
  @IBOutlet private weak var responseView: UIImageView!
  @IBOutlet private weak var playerSelector: UISegmentedControl!
  @Published private var playerChoice = 0
  private var cancellable: AnyCancellable?
  private let game = Game()

  override func viewDidLoad() {
    super.viewDidLoad()
    cancellable =
      NotificationCenter
      .default
      .publisher(for: Notification.Name("player"))
      .sink{[weak self] notification in
        guard let player = notification.object as? Player else {return}
        self?.playerView.image = UIImage(systemName: player.handPosition.imag
      }
  }

  @IBAction func enterChoice(_ button: UIButton) {
    game.setPlayer(playerSelector.selectedSegmentIndex)
  }
}
```

This works as before using notifications and Combine.

Let's find a more direct route back.

SECTION 4

# Pub Sub

Let's return to a more typical route from publisher to subscriber.

## Publishing Hand

We can publish changes to `handPosition` in `Hand` using `@Published`.

*04/RockPaperScissors/RockPaperScissors/Model/Hand.swift*

```
import Foundation

class Hand {
  @Published var handPosition: HandPosition = .rock
}
```

We can leave the import of `Foundation` in or switch to `Combine`.

*04/RockPaperScissors/RockPaperScissors/Model/Hand.swift*

```
import Combine

class Hand {
  @Published var handPosition: HandPosition = .rock
}
```

Let's subscribe from `Game`.

## Preparation

We will use a method named `subscribeToPlayer()` to subscribe in `Game`.

*04/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() {
  }
}
```

`Game` needs to have a cancellable and a place to put the image name. We'll add a didSet there to confirm we're receiving it. We also need to call `subscribeToPlayer()` perhaps from `init()`.

*04/RockPaperScissors/RockPaperScissors/Game.swift*

```
import Combine

class Game {
  private let player = Player()
  private let response = Response()
  private var cancellable: AnyCancellable?
  private(set) var playerImageName: String = "questionmark" {
    didSet {
      print("received:", playerImageName)
    }
  }

  init() {
    subscribeToPlayer()
  }
}
```

Now we're ready to build our pipeline.

## Pub to Sub

Take a moment and implement `subscribeToPlayer()`.You should be able to run and check that it works. Remember how to get rid of the first value.

*04/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() {
    cancellable
      = player.$handPosition
      .dropFirst()
      .sink{[weak self]  handPosition in
        self?.playerImageName = handPosition.imageName
      }
  }
}
```

Let's make one more adjustment.

## Move the Cancellable

I think it's cleaner to move the cancellable to where we're calling `subscribeToPlayer()` so that the chain begins with a publisher and ends with a subscriber.

Make this adjustment to `subscribeToPlayer()`.

*04/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() -> AnyCancellable {
    cancellable
      = player.$handPosition
      .dropFirst()
      .sink{[weak self]  handPosition in
        self?.playerImageName = handPosition.imageName
      }
  }
}
```

Now `subscribeToPlayer()` returns the `Cancellable` from the `sink()`. We hold on to it when we call `subscribeToPlayer()`.

We need to get our value to our view controller. That's next.

# Republishing

We need to some how relay the updates to `Game` so that `ViewController` reflects them.

## @Published

The first step is to mark `playerImageName` as `@Published` in `Game`. We can remove the didSet.

*05/RockPaperScissors/RockPaperScissors/Game.swift*

```
class Game {
  private let player = Player()
  private let response = Response()
  private var cancellable: AnyCancellable?
  @Published private(set) var playerImageName: String = "questionmark" {
    didSet {
      print("received:", playerImageName)
    }
  }

  init() {
    subscribeToPlayer()
  }
}
```

Now `ViewController` can subscribe to this publisher.

## Subscribing from ViewController

Once again we use `viewDidLoad`. Subscribe to `playerImageName` and use the name to set the image.

*05/RockPaperScissors/RockPaperScissors/ViewController.swift*

```
class ViewController: UIViewController {
  @IBOutlet private weak var resultView: UIImageView!
  @IBOutlet private weak var playerView: UIImageView!
  @IBOutlet private weak var responseView: UIImageView!
  @IBOutlet private weak var playerSelector: UISegmentedControl!
  private var cancellable: AnyCancellable?
  private let game = Game()

  override func viewDidLoad() {
    super.viewDidLoad()
    cancellable =
      game
      .$playerImageName
      .sink{[weak self] name in
        self?.playerView.image = UIImage(systemName: name)
      }
  }

  @IBAction func enterChoice(_ button: UIButton) {
    game.setPlayer(playerSelector.selectedSegmentIndex)
  }
}
```

## The main thread

Most UI gets done on the main thread, so we should probably take care to use an operator that pushes us back to the main thread before updating the UI.

*05/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() -> AnyCancellable {
    player.$handPosition
      .dropFirst()
      .receive(on: RunLoop.main)
      .sink{[weak self]  handPosition in
        self?.playerImage
          = UIImage(systemName: handPosition.imageName)
      }
  }
}
```

For GUI work we tend to use `RunLoop.main`. You'll also see people use `DispatchQueue.main`. As to the difference between them...

From Philippe Hausler (https://forums.swift.org/t/runloop-main-or-dispatchqueue-main-when-using-combine-scheduler/26635): RunLoop.main as a Scheduler ends up calling RunLoop.main.perform whereas DispatchQueue.main calls DispatchQueue.main.async to do work, for practical purposes they are nearly isomorphic. The only real differential is that the RunLoop call ends up being executed in a different spot in the RunLoop callouts whereas the DispatchQueue variant will perhaps execute immediately if optimizations in libdispatch kick in. In reality you should never really see a difference tween the two.

We've seen several operators including dropFirst(), filter(), compactMap(), and receive(on:). Next we look at map() and are introduced to another assign().

SECTION 6

# Map

In this section we introduce map and two versions of assign. We use map() to transform our item.

## Map

Think of what `map` does for Arrays. It essentially does the same thing for combine publishers.

Use it to convert a `HandPosition` to its `imageName` between `dropFirst()` and `receive()`.

*06/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() -> AnyCancellable {
    player.$handPosition
      .dropFirst()
      .map {handPosition in
        handPosition.imageName
      }
      .receive(on: RunLoop.main)
      .sink {[weak self] name in
        self?.playerImageName = name
      }
  }
}
```

Run the app. It works perfectly.

If you prefer we can use $0.

*06/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() -> AnyCancellable {
    player.$handPosition
      .dropFirst()
      .map {$0.imageName}
      .receive(on: RunLoop.main)
      .sink {[weak self] name in
        self?.playerImageName = name
      }
  }
}
```

But my favorite is the key path version.

*06/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() -> AnyCancellable {
    player.$handPosition
      .dropFirst()
      .map(\.imageName)
      .receive(on: RunLoop.main)
      .sink {[weak self] name in
        self?.playerImageName = name
      }
  }
}
```

The code works as before.

## Assign(to: on:)

We are in a very special case in which `sink()` is connected to a publisher that never fails and gives it the value it is assigning to a property. We can use a simpler subscriber `assign(to: on:)` like this.

*06/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() -> AnyCancellable {
    player.$handPosition
      .dropFirst()
      .map(\.imageName)
      .receive(on: RunLoop.main)
      .assign(to: \.playerImageName,
              on: self)
  }
}
```

There is no way to weakly hold on to `self` in this `assign` so we may have a memory leak. In our case there's an even nicer version of `assign`.

## Assign(to:)

In the case that our property we are assigning to is a publisher, we can use an even nicer version of `assign()` that manages the subscription for us.

*06/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() -> AnyCancellable {
    player.$handPosition
      .dropFirst()
      .map(\.imageName)
      .receive(on: RunLoop.main)
      .assign(to: &$playerImageName)
  }
}
```

Adjust the `init()` accordingly, and remove `cancellable`.

class Game { private let player = Player() private let response = Response() ~~private var cancellable: AnyCancellable?~~ @Published private(set) var playerImageName = " questionmark"  init() { subscribeToPlayer() } }

Next, let's look at how we might subscribe to multiple publishers from a single object.

SECTION 7

# Observable Objects

In this section we add properties to `Game` for the remaining publishers and simplify the subscription using an Observable Object.

## More publishers

Add publishers for our other attributes we want to pass along. Unfortunately, we need UIKit for UIColor.

*07/RockPaperScissors/RockPaperScissors/**Game.swift***

```
import Combine
import UIKit

class Game {
  private let player = Player()
  private let response = Response()
  ~~private var cancellable: AnyCancellable?~~
  @Published private(set) var playerImageName = "questionmark"
  @Published private(set) var responseImageName = "questionmark"
  @Published private(set) var resultImageName = "questionmark"
  @Published private(set) var resultImageColor = UIColor.secondaryLabel

  init() {
    subscribeToPlayer()
  }
}
```

Consider what we might need to do to subscribe to all of these in `ViewController`. I've highlighted our current subscription for `playerImageName`.

*07/RockPaperScissors/RockPaperScissors/ViewController.swift*

```
class ViewController: UIViewController {
  @IBOutlet private weak var resultView: UIImageView!
  @IBOutlet private weak var playerView: UIImageView!
  @IBOutlet private weak var responseView: UIImageView!
  @IBOutlet private weak var playerSelector: UISegmentedControl!
  private var cancellable: AnyCancellable?
  private let game = Game()

  override func viewDidLoad() {
    super.viewDidLoad()
    cancellable =
      game
      .$playerImageName
      .receive(on: RunLoop.main)
      .sink {[weak self] name in
        self?.playerView.image = UIImage(systemName: name)
      }
  }

  @IBAction func enterChoice(_ button: UIButton) {
    game.setPlayer(playerSelector.selectedSegmentIndex)
  }
}
```

We are NOT going to repeat that for each publisher.

## ObservableObject

Declare that `Game` conforms to `ObservableObject`.

*07/RockPaperScissors/RockPaperScissors/Game.swift*

```
class Game: ObservableObject {
  private let player = Player()
  private let response = Response()
  @Published private(set) var playerImageName = "questionmark"
  @Published private(set) var responseImageName = "questionmark"
  @Published private(set) var resultImageName = "questionmark"
  @Published private(set) var resultImageColor = UIColor.secondaryLabel

  init() {
    subscribeToPlayer()
  }
}
```

When any of the published items changes, `Game` announces to any objects observing in that it has changed.

Actually, as you'll see, that's not quite true. It announces it will change.

## Object will change

An ObservableObject has an ObjectWillChange publisher. It's just an announcement that the object will change. It doesn't send a value. It's up to us to listen for the changes.

Here's our first try.

*07/RockPaperScissors/RockPaperScissors/ViewController.swift*

```
  override func viewDidLoad() {
    super.viewDidLoad()
    cancellable =
      game
      .objectWillChange
      .sink{[weak self] _ in
        guard let self = self else {return}
        self.playerView.image = UIImage(systemName: self.game.playerImageName
      }
  }
```

Because we use `self` more than once we do the strong weak dance.

Take the app for a spin. You'll notice that the response is behind - it's the difference between willChange and didChange.

This API was built for SwiftUI which needs to know that something will change so that it can refresh the UI with the changes. UIKit doesn't work that way.

## RunLoop main

Again, ObservableObject is meant to be used with SwiftUI - I am demoing it with UIKit and that will require a hack.

Even though we're on the main thread, this unfortunate UIKit hack works. Add the `receive(on:)` to our chain in between `objectWillChange` and the `sink`.

*07/RockPaperScissors/RockPaperScissors/ViewController.swift*

```
  override func viewDidLoad() {
    super.viewDidLoad()
    cancellable =
      game
      .objectWillChange
      .receive(on: RunLoop.main)
      .sink{[weak self] _ in
        guard let self = self else {return}
        self.playerView.image = UIImage(systemName: self.game.playerImageName
      }
  }
```

## The other properties

Any time any of the published properties updates we will be told that game will change. We don't know why, so we update all of the properties. I've added a helper method.

*07/RockPaperScissors/RockPaperScissors/ViewController.swift*

```
class ViewController: UIViewController {
  @IBOutlet private weak var resultView: UIImageView!
  @IBOutlet private weak var playerView: UIImageView!
  @IBOutlet private weak var responseView: UIImageView!
  @IBOutlet private weak var playerSelector: UISegmentedControl!
  private var cancellable: AnyCancellable?
  private var game = Game()

  override func viewDidLoad() {
    super.viewDidLoad()
    cancellable =
      game
      .objectWillChange
      .receive(on: RunLoop.main)
      .sink{[weak self] _ in
        guard let self = self else {return}
        self.playerView.image = self.image(self.game.playerImageName)
        self.responseView.image = self.image(self.game.responseImageName)
        self.resultView.image = self.image(self.game.resultImageName)
        self.resultView.tintColor = self.game.resultImageColor
      }
  }

  private func image(_ name: String) -> UIImage? {
    UIImage(systemName: name)
  }

  @IBAction func enterChoice(_ button: UIButton) {
    game.setPlayer(playerSelector.selectedSegmentIndex)
  }
}
```

Let's add another placeholder publisher just to test this.

*07/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() {
    player.$handPosition
      .dropFirst()
      .map(\.imageName)
      .receive(on: RunLoop.main)
      .assign(to: &$playerImageName)

    Just("sun.dust")
      .assign(to: &$responseImageName)
  }
}
```

`Just` is a publisher that publishes its contents, completes, and never fails.

When you tap the button you'll see both images update.

## Delay

Some operators allow you to adjust timing. For example, we can delay the update five seconds like this.

*07/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() {
    player.$handPosition
      .dropFirst()
      .map(\.imageName)
      .receive(on: RunLoop.main)
      .assign(to: &$playerImageName)

    Just("sun.dust")
      .delay(for: 5, scheduler: RunLoop.main)
      .assign(to: &$responseImageName)
  }
}
```

Now you should see the player image update when you tap the button and the response update five seconds later.

## Clean up

Remove the `Just` sequence before continuing.

*07/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() {
    player.$handPosition
      .dropFirst()
      .map(\.imageName)
      .receive(on: RunLoop.main)
      .assign(to: &$playerImageName)

    Just("sun.dust")>
      .delay(for: 5, scheduler: RunLoop.main)
      .assign(to: &$responseImageName)
  }
}
```

We are going to make no more changes to this UI. All of our changes will be on the back end - so I want to show you this same app working with SwiftUI.

SECTION 8

# SwiftUI

I've implemented the UI in SwiftUI and kept the model virtually the same.

Honestly we only have to make two changes to `Game`.

## UIKit to SwiftUI

The only change we have to make to our work so far is that the `UIColor` changes to a SwiftUI `Color`. This means that we must import SwiftUI instead of UIKit in `Game`.

*08/RockPaperScissors/RockPaperScissors/Game.swift*

```
import Combine
import UIKit
import SwiftUI

class Game: ObservableObject {
  private let player = Player()
  private let response = Response()
  @Published private(set) var playerImageName = "questionmark"
  @Published private(set) var responseImageName = "questionmark"
  @Published private(set) var resultImageName = "questionmark"
  @Published private(set) var resultImageColor
    = Color.secondary UIColor.secondaryLabel

  init() {
    subscribeToPlayer()
  }
}
```

The rest of the code is UI code.

## Listening for Game updates

Here's where we register for `Game` updates.

*08/RockPaperScissors/RockPaperScissors/View/GameView.swift*

```
struct GameView {
  @StateObject var game = Game()
}
```

Here's where we pass `game` on to the two parts of the board:

*08/RockPaperScissors/RockPaperScissors/View/GameView.swift*

```
extension GameView: View {
    var body: some View {
      VStack {
        GamePanel(game: game)
        PlayersChoice(game: game)
      }
    }
}
```

The `GamePanel` needs to get updates so it decorates `game` as an observed object.

*08/RockPaperScissors/RockPaperScissors/View/GameView.swift*

```
struct GamePanel {
  @ObservedObject var game: Game
}
```

Whenever `game` updates, the components using it update as well.

*08/RockPaperScissors/RockPaperScissors/View/GameView.swift*

```
extension GamePanel: View {
  var body: some View {
    VStack(spacing: 40) {
      ResultView(imageName: game.resultImageName,
                 imageColor: game.resultImageColor)

      HStack(alignment: .bottom, spacing: 60) {
        PositionView(name: "Player",
                     imageName: game.playerImageName)

        PositionView(name: "Response",
                     imageName: game.responseImageName)
      }
    }
  }
}
```

That's it. The code runs as before.

We'll continue with this SwiftUI version of our app.

# Publishing The Response

Since `playerImageName` is a Publisher, we can subscribe to it and perform an action whenever it changes. In particular, whenever the `playerImageName` is set, we will ask `response` to create a new random `HandPosition`.

## Cancellables

We're going to need to store more than one cancellable token before we're through. Create an empty set of cancellables in `Game`.

*09/RockPaperScissors/RockPaperScissors/Game.swift*

```
class Game: ObservableObject {
  private let player = Player()
  private let response = Response()
  private var cancellables = Set<AnyCancellable>()
  @Published private(set) var playerImageName = "questionmark"
  @Published private(set) var responseImageName = "questionmark"
  @Published private(set) var resultImageName = "questionmark"
  @Published private(set) var resultImageColor = Color.secondary

  init() {
    subscribeToPlayer()
  }
}
```

We'll add our first token in a moment.

## Elicit a response

When the `playerImageName` is set, we will request a response.

Implement this in a method named `requestResponse()` in `Game`.

Begin with our pipeline from the publisher to the subscriber.

*09/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() {
    player.$handPosition
      .dropFirst()
      .map(\.imageName)
      .receive(on: RunLoop.main)
      .assign(to: &$playerImageName)
  }

  func requestResponse() {
    // not done yet
    $playerImageName
      .dropFirst()
      .sink{[weak self] _ in
        self?.response.nextPosition()
      }
  }
}
```

We need to store this in our set of cancellables. Return the `Cancellable` from `requestResponse()`.

*09/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() {
    player.$handPosition
      .dropFirst()
      .map(\.imageName)
      .receive(on: RunLoop.main)
      .assign(to: &$playerImageName)
  }

  func requestResponse() -> AnyCancellable {
    $playerImageName
      .dropFirst()
      .sink{[weak self] _ in
        self?.response.nextPosition()
      }
  }
}
```

Don't forget to call this method from `init()` and add to `cancellables` there.

*09/RockPaperScissors/RockPaperScissors/Game.swift*

```
  init() {
    subscribeToPlayer()
    requestResponse().store(in: &cancellables)
  }
```

Next we have to subscribe to the response. You do it.

## Subscribe to response

You should create something almost exactly like `subscibeToPlayer()`.

*09/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() {
    player.$handPosition
      .dropFirst()
      .map(\.imageName)
      .receive(on: RunLoop.main)
      .assign(to: &$playerImageName)
  }

  func requestResponse() -> AnyCancellable {
    $playerImageName
      .dropFirst()
      .sink{[weak self] _ in
        self?.response.nextPosition()
      }
  }

  func subscribeToResponse() {
    response.$handPosition
      .dropFirst()
      .map(\.imageName)
      .receive(on: RunLoop.main)
      .assign(to: &$responseImageName)
  }
}
```

Also add the call to `init()`.

*09/RockPaperScissors/RockPaperScissors/Game.swift*

```
  init() {
    subscribeToPlayer()
    requestResponse().store(in: &cancellables)
    subscribeToResponse()
  }
```

There is way too much duplicated code.

## Identifying duplicate code

Take a look at the repeated code.

*09/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() {
    player.$handPosition
      .dropFirst()
      .map(\.imageName)
      .receive(on: RunLoop.main)
      .assign(to: &$playerImageName)
  }

  func requestResponse() -> AnyCancellable {
    $playerImageName
      .dropFirst()
      .sink{[weak self] _ in
        self?.response.nextPosition()
      }
  }

  func subscribeToResponse() {
    response.$handPosition
      .dropFirst()
      .map(\.imageName)
      .receive(on: RunLoop.main)
      .assign(to: &$responseImageName)
  }
}
```

I often leave the `receive(on:)`s here because I prefer to keep them close to the `assign()` but you could certainly move them.

It's a matter of taste and I don't feel strongly either way. Let's move them along with the other code.

I also want to implement this in a property instead of a function just to show you this technique. The property needs to be `lazy` because it can't be evaluated before `Hand` exists.

## De-duping

Add `imageName` to `Hand`. It contains the repeated code. We also need to import Foundation for `RunLoop`.

*09/RockPaperScissors/RockPaperScissors/Model/Hand.swift*

```
import Combine
import Foundation

class Hand {
  @Published var handPosition: HandPosition = .rock

  lazy private(set) var imageName
    = $handPosition
    .dropFirst()
    .map(\.imageName)
    .receive(on: RunLoop.main)
}
```

Option-Click on `imageName` to view its type. It is

```
Publishers.MapKeyPath<Publishers.Drop<Published<HandPosition>.Publisher>, Str
```

That's awful. Also, it exposes the innards of what we've done. Before bundling this up to be consumed, we erase its type.

## Any Publisher

We erase to any publisher so we only expose the output and error types. We specify this type

*09/RockPaperScissors/RockPaperScissors/Model/Hand.swift*

```
class Hand {
  @Published var handPosition: HandPosition = .rock

  lazy private(set) var imageName: AnyPublisher<String, Never>
    = $handPosition
    .dropFirst()
    .map(\.imageName)
    .receive(on: RunLoop.main)
    .eraseToAnyPublisher()
}
```

This makes it clear that we're returning a Publisher of a `String` that never fails.

## Simplifying the code

Back to `Game` we can use this `imageName` property. Note we don't use `$imageName` as `imageName` is already a publisher.

*09/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() {
    player.imageName
      .assign(to: &$playerImageName)
  }

  func requestResponse()  -> AnyCancellable {
    $playerImageName
      .dropFirst()
      .sink{[weak self] _ in
        self?.response.nextPosition()
      }
  }

  func subscribeToResponse() {
    response.imageName
      .assign(to: &$responseImageName)
  }
}
```

We now have three publisher to subscriber subscriptions in action and we've set up these connections with functions and lazy

properties.

When we tap the button both player and response images update.

Next, let's combine two publishers and then split them.

# Joining And Splitting

Let's combine the player choice and the response to determine a winner.

## Scorer

Create a new file named `Scorer` in Model that is a struct that contains two properties: `player` and `response`.

*10/RockPaperScissors/RockPaperScissors/Model/**Scorer.swift***

```
struct Scorer {
  let player: Player
  let response: Response
}
```

We use the `gameResult()` function in our pipeline.

## Combining two publishers

Let's create a pipeline that takes the `HandPosition` publishers from `player` and `response` and combines them into a tuple that is then mapped using `gameResultForPlayer`.

In other words, this chain combines two publishers of `HandPosition`s into a single publisher of `GameResult`s.

We can either use `combineLatest()` or `zip()` to combine our publishers.

`combineLatest()` emits a tuple of both of the published values when either publisher changes. This is often handy but would be the wrong choice here.

Imagine we have a value for player and a value for response. With `combineLatest()`, as soon as the player's choice changes it would be compared to the previous value for response. Then response is updated and `combineLatest()` fires again with the value we want.

In this case we want `zip()` which waits until both publishers have updated.

## Combining the results

We'll use a lazy initialized property to create our publisher. Don't forget to import Combine.

*10/RockPaperScissors/RockPaperScissors/Model/Scorer.swift*

```
import Combine

struct Scorer {
  let player: Player
  let response: Response

  lazy private(set) var result
    = player.$handPosition
    .zip(response.$handPosition)
}
```

The first to get through `zip` will be rock vs rock so we need to drop it.

*10/RockPaperScissors/RockPaperScissors/Model/Scorer.swift*

```
struct Scorer {
  let player: Player
  let response: Response

  lazy private(set) var result
    = player.$handPosition
    .zip(response.$handPosition)
    .dropFirst()
}
```

We now have a publisher of a tuple of `HandPosition`s so we can use `map()` to convert this to a publisher of `GameResult`s.

*10/RockPaperScissors/RockPaperScissors/Model/Scorer.swift*

```
struct Scorer {
  let player: Player
  let response: Response

  lazy private(set) var result
    = player.$handPosition
    .zip(response.$handPosition)
    .dropFirst()
    .map(gameResult)
}
```

Finally, let's clean up pushing to the main thread and by erasing to any publisher.

*10/RockPaperScissors/RockPaperScissors/Model/**Scorer.swift***

```
import Combine
import Foundation

struct Scorer {
  let player: Player
  let response: Response

  lazy private(set) var result: AnyPublisher<GameResult, Never>
    = player.$handPosition
    .zip(response.$handPosition)
    .dropFirst()
    .map(gameResult)
    .receive(on: RunLoop.main)
    .eraseToAnyPublisher()
}
```

We could have done all of this in `Game` but I think it's nicer to have this localized modular approach.

## Create Scorer in Game

Lazily create an instance of `Scorer` in `Game`

*10/RockPaperScissors/RockPaperScissors/Game.swift*

```
class Game: ObservableObject {
  private let player = Player()
  private let response = Response()
  lazy private var scorer = Scorer(player: player,
                                   response: response)
  private var cancellables = Set<AnyCancellable>()
  @Published private(set) var playerImageName = "questionmark"
  @Published private(set) var responseImageName = "questionmark"
  @Published private(set) var resultImageName = "questionmark"
  @Published private(set) var resultImageColor = Color.secondary

  init() {
    subscribeToPlayer()
    requestResponse().store(in: &cancellables)
    subscribeToResponse()
  }
}
```

It needs to be `lazy` as it depends on `player` and `response`.

## Displaying the Result

Add this stub for `subscribeToResult()` to `Game`.

*10/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() {
    player.imageName
      .assign(to: &$playerImageName)
  }

  func requestResponse() -> AnyCancellable {
    $playerImageName
      .dropFirst()
      .sink{[weak self] _ in
        self?.response.nextPosition()
      }
  }

  func subscribeToResponse() {
    response.imageName
      .assign(to: &$responseImageName)
  }

  func subscribeToResult() -> AnyCancellable {

  }
}
```

Add a call to `subscribeToResult()` to the `init`.

*10/RockPaperScissors/RockPaperScissors/Game.swift*

```
class Game: ObservableObject {
  private let player = Player()
  private let response = Response()
  lazy private var scorer = Scorer(player: player,
                                   response: response)
  private var cancellables = Set<AnyCancellable>()
  @Published private(set) var playerImageName = "questionmark"
  @Published private(set) var responseImageName = "questionmark"
  @Published private(set) var resultImageName = "questionmark"
  @Published private(set) var resultImageColor = Color.secondary

  init() {
    subscribeToPlayer()
    requestResponse().store(in: &cancellables)
    subscribeToResponse()
    subscribeToResult().store(in: &cancellables)
  }
}
```

Here are the steps.

- Start with the pubisher we just created in `Scorer`.
- Use `sink` to set the `resultImageName` and the `resultImageColor`.
- Store the `Cancellable` in `cancellables`.

*10/RockPaperScissors/RockPaperScissors/Game.swift*

```
extension Game {
  func subscribeToPlayer() {
    player.imageName
      .assign(to: &$playerImageName)
  }

  func requestResponse() -> AnyCancellable {
    $playerImageName
      .dropFirst()
      .sink{[weak self] _ in
        self?.response.nextPosition()
      }
  }

  func subscribeToResponse() {
    response.imageName
      .assign(to: &$responseImageName)
  }

  func subscribeToResult() -> AnyCancellable {
    scorer.result
      .sink{[weak self] result in
        guard let self = self else {return}
        self.resultImageName = result.imageName
        self.resultImageColor = result.imageColor
      }
  }
}
```

This now runs perfectly.

Let's look back at our code before moving on.

SECTION 11

# URLSession

Let's fetch our result using a URL Session. We'll also deal with possible errors and finishing.

## Preparation

Control - Click on Models and add the file from *Extras* named *URLConstants.swift*.

We're going to use `nextPosition()` to ask a server for a number between 0 and 2 to use to create a `HandPosition`. We also need to import Combine and Foundation.

*11/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
import Combine
import Foundation

class Response: Hand {
  private var cancellable: AnyCancellable?

  func nextPosition() {
    handPosition = randomHandPosition()
  }
}
```

Let's start building out `nextPosition()`.

## The data task

Use the helper method to fetch the URL for getting a random int.

*11/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
  func nextPosition() {
    guard let randomIntURL = intURL() else {return}
  }
```

Apple has wrapped `URLSession` with Combine. Create a data task publisher like this.

*11/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
  func nextPosition() {
    guard let randomIntURL = intURL() else {return}
    cancellable
      = URLSession.shared
      .dataTaskPublisher(for: randomIntURL)
  }
```

With the closure based API we can get back optional data, response, and an error. With the publisher we get back output which is a tuple containing Data and URLResponse. The error is reported in the error side of the house.

We're interested in pulling out the data if there is any.

*11/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
  func nextPosition() {
    guard let randomIntURL = intURL() else {return}
    cancellable
      = URLSession.shared
      .dataTaskPublisher(for: randomIntURL)
      .map(\.data)
  }
```

So barring an error, we have data at this point.

## Transforming the data

We need to turn the data into a string using the utf8 encoding. This can fail in which case there is no valid string. The result will be nil. We use compactMap so that nothing is passed on if we end up with nil here.

*11 /RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
  func nextPosition() {
    guard let randomIntURL = intURL() else {return}
    cancellable
      = URLSession.shared
      .dataTaskPublisher(for: randomIntURL)
      .map(\.data)
      .compactMap {data in
        String(data: data,
               encoding: .utf8)
      }
  }
```

If there is a String, we want to convert it into an Int. This could also fail and return nil so we use compactMap again.

*11/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
  func nextPosition() {
    guard let randomIntURL = intURL() else {return}
    cancellable
      = URLSession.shared
      .dataTaskPublisher(for: randomIntURL)
      .map(\.data)
      .compactMap {data in
        String(data: data,
               encoding: .utf8)
      }
      .compactMap {string in
        Int(string)
      }
  }
```

At the end of this chain we either have output or we have an error or we had nil along the way so we don't get to the end.

## The Kitchen Sink

This time let's use the full version of `sink`.

*11 /RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
  func nextPosition() {
    guard let randomIntURL = intURL() else {return}
    cancellable
      = URLSession.shared
      .dataTaskPublisher(for: randomIntURL)
      .map(\.data)
      .compactMap {data in
        String(data: data,
               encoding: .utf8)
      }
      .compactMap {string in
        Int(string)
      }
      .sink(receiveCompletion: {completion in
        switch completion {
        case .finished:
          print("finished")
        case .failure(let error):
          print("error in datatask:", error)
        }
      }, receiveValue: {int in
        self.handPosition = setHandPosition(for: int)
      })
  }
```

Run the app. You should see " finished"  in the console and the values should update on screen.

SECTION 12

# JSON

We're going to hit a different endpoint that delivers us JSON instead of an Int.

## Codable

JSON is so much easier now that we have the `Codable` protocol. Declare that `HandPosition` is codable.

*12/RockPaperScissors/RockPaperScissors/Model/HandPosition.swift*

```
enum HandPosition: String, CaseIterable, Codable {
  case rock
  case paper
  case scissors
}
```

We're going to bundle up our `HandPosition` in a type we call `RandomPosition`. Add this to the top of `HandPosition.swift`.

*12/RockPaperScissors/RockPaperScissors/Model/HandPosition.swift*

```
struct RandomPosition: Codable {
  let position: HandPosition
}

enum HandPosition: String, CaseIterable, Codable {
  case rock
  case paper
  case scissors
}
```

Now to modify our URLSession.

## URLSession

We'll start at the top of `nextPosition()` and start making adjustments. The URL is our first difference.

*12/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
  func nextPosition() {
    guard let randomPositionURL = handPositionURL() else {return}
    cancellable
      = URLSession.shared
      .dataTaskPublisher(for: randomPositionURL)
      .map(\.data)
```

If the task doesn't error out, we have data at this point.

## JSON

Next use the decoder operator to take the data and a designated decoder to transform it into a given type. Of course this can fail.

*12/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
  func nextPosition() {
    guard let randomPositionURL = handPositionURL() else {return}
    cancellable
      = URLSession.shared
      .dataTaskPublisher(for: randomPositionURL)
      .map(\.data)
      .decode(type: RandomPosition.self,
              decoder: JSONDecoder())
```

So we have a publisher of RandomPositions that can fail.

## Using the returned value

Our sink looks mostly the same.

If we get a `RandomPosition` we'll ask it for its `position`.

*12/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
  func nextPosition() {
    guard let randomPositionURL = handPositionURL() else {return}
    cancellable
      = URLSession.shared
      .dataTaskPublisher(for: randomPositionURL)
      .map(\.data)
      .decode(type: RandomPosition.self,
              decoder: JSONDecoder())
      .sink(receiveCompletion: {completion in
        switch completion {
        case .finished:
          print("finished")
        case .failure(let error):
          print("error in JSON decoding:", error)
        }
      }, receiveValue: {randomPosition in
        self.handPosition = randomPosition.position
      })
  }
```

Before we leave this section, let's see an error.

## Failure

Change the URL to the `intUR` but keep everything else the same.

*12/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
  func nextPosition() {
    guard let randomPositionURL = intURL() else {return}
```

This should and does fail. We see this in the console.

```
failed error in JSON decoding: (...) "Expected to
decode
```

```
Dictionary<String, Any> but found a number instead.",
underlyingError: nil)) error in JSON decoding:
```

Let's handle the error differently.

## Catch

We can catch errors the same way we do with do try catch. There is a catch operator that we can use like this.

*12/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
  func nextPosition() {
    guard let randomPositionURL = intURL() else {return}
    cancellable
      = URLSession.shared
      .dataTaskPublisher(for: randomPositionURL)
      .map(\.data)
      .decode(type: RandomPosition.self,
              decoder: JSONDecoder())
      .catch {error in
        Just(RandomPosition(position: randomHandPosition()))
      }
      .sink(receiveCompletion: {completion in
        switch completion {
        case .finished:
          print("finished")
        case .failure(let error):
          print("error in JSON decoding:", error)
        }
      }, receiveValue: {randomPosition in
        self.handPosition = randomPosition.position
      })
  }
```

Now when the JSON decoder fails an error is thrown and we trigger the catch. Notice catch must return a publisher. We use

Just and create a random HandPosition wrapped in a RandomPosition.

We are now making the network call, failing, and generating a local value.

In the next section we look more carefully at errors.

SECTION 13

# Errors

Let's look at a couple of ways of dealing with errors. The URLSession returned and finished. What happens if we have a non-ending pipeline?

This is not a practical example for this project but it does demonstrate the techniques you'll find useful.

## Preparation

Add a private published property to `Response` named `int`. Update `nextPosition()` to pick a random Int between 1 and 100. We won't need `cancellable` any more but we will set up our subscription in a method named `tensRemover`.

*1.3/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
import Combine
import Foundation


class Response: Hand {
  private var cancellable: AnyCancellable?
  @Published private var int = 0 {
    didSet {
      print("Int set to:", int)
    }
  }

  override init() {
    super.init()
    tensRemover()
  }

  func nextPosition() {
    int = .random(in: 1...100)
  }

  private func tensRemover() {
  }
}
```

Run the app.

We see the player choice but no response and no result. I actually like this a lot. When the response doesn't happen, nothing else happens.

## A function that throws

In the Extras folder you'll find a file named *ResponseExtras.swift*. Add it to Model. It contains an error type and a method that throws an error.

*1.3/RockPaperScissors/RockPaperScissors/Model/ResponseExtra.swift*

```
struct DivisibleByTen: Error {}

extension Response {
  func restrictedPosition(from int: Int) throws -> HandPosition {
    print(int)
    guard !int.isMultiple(of: 10) else {throw DivisibleByTen()}
    return setHandPosition(for: int % 3)
  }
}
```

If our input is not divisible by 10 we'll return a valid HandPosition.

## Mapping a throwing function

Currently we are mapping using a function that either returns a valid HandPosition or fatal errors out. What happens instead if we use a throwing function?

When using `map()` for arrays, it rethrows so if we map a throwing function then our map will rethrow an error and if we map a non-throwing function then our map won't.

Combine is architected differently. If we have a function that throws, we have to use `tryMap` not `map`.

In particular, this won't compile.

*1.3/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
  private func tensRemover() {
    $int
      .dropFirst()
      .map(restrictedPosition) // can't use throwing function in map
      .assign(to: &$handPosition)
  }
```

We need `tryMap`. It won't compile either but that's because we aren't handling the error.

*1 3/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
  private func tensRemover() {
    $int
      .dropFirst()
      .tryMap(restrictedPosition) //need to handle the error
      .assign(to: &$handPosition)
  }
```

In the previous section we handled the error in two ways: in the `sink` and in `catch`.

Let's add a catch to see a possible issue that we will address in a moment.

## Catch

Add a catch that provides a HandPosition.

*1 3/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
  private func tensRemover() {
    $int
      .dropFirst()
      .tryMap(restrictedPosition)
      .catch {error in
        Just(randomHandPosition())
      }
      .assign(to: &$handPosition)
  }
```

It might be also handy to add a print here as well.

*13/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
  private func tensRemover() {
    $int
      .dropFirst()
      .tryMap(restrictedPosition)
      .catch {error in
        Just(randomHandPosition())
      }
      .print()
      .assign(to: &$handPosition)
  }
```

Run the app. You'll see the int and the result each time the button is tapped. It shouldn't take you long to hit a multiple of 10. You'll then issue one more hand position and then complete.

The `Just` has become the new source of the publishing chain so it is the last sent value.

## FlatMap

We're going to wrap part of the pipeline in a closure.

*13/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
      { .tryMap(self.restrictedPosition)
          .catch {error in
            Just(randomHandPosition())
          }
      }
```

We need to take the `Int` from `$int` and pass it on to this chain. We do it with a Just.

*1.3/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
????.{int in
      Just(int)
        .tryMap(self.restrictedPosition)
        .catch {error in
          Just(randomHandPosition())
        }
    }
```

That leaves us with a function from `Int -> Publisher<HandPosition>` wrapped in a publisher. This is the signature of a flatMap!

*1.3/RockPaperScissors/RockPaperScissors/Model/Response.swift*

*1.3/RockPaperScissors/RockPaperScissors/Model/Response.swift*

```
private func tensRemover() {
  $int
     .dropFirst()
    .flatMap {int in
      Just(int)
        .tryMap(self.restrictedPosition)
        .catch {error in
          Just(randomHandPosition())
        }
    }
    .print()
    .assign(to: &$handPosition)
}
```

Now this continues indefinitely. It doesn't terminate on a catch.