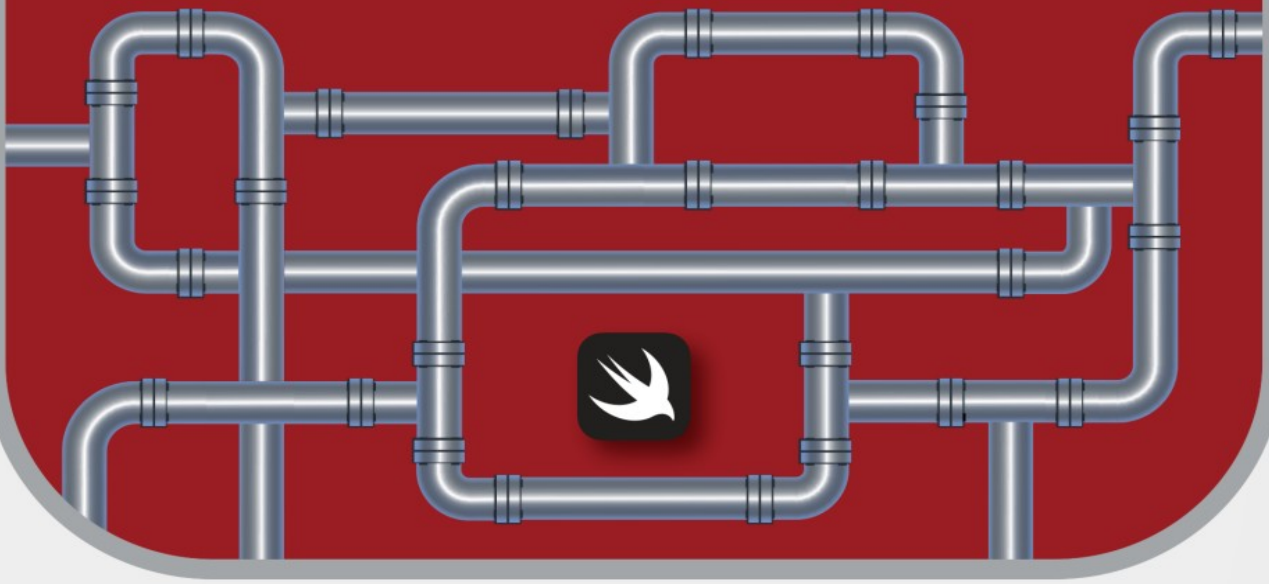
A Combine Kickstart

DANIEL H STEINBERG



Introducing the Declarative Framework for Processing Values over Time

Editors Cut

A Combine Kickstart

Introducing The Declarative Framework

For Processing Values Over Time

by Daniel H Steinberg

Editors Cut

Copyright

ISBN-13: 978-1-944994-02-0

Book Version

This is version 0.3 for Swift 5.3, Xcode 12.4, macOS Big Sur, and iOS 14 released February 2021. All code has been tested on Apple Silicon.

Code Download

Visit https://github.com/editorscut/ec011CombineKickstart for all of the code for this book.

Run it in Xcode 12 or higher. All code is written in Swift.

Recommended Settings

The ePub is best viewed in scrolling mode on an iPad. On smaller devices I also choose landscape. For some reason that I don't understand, scrolling mode is supported by Apple's Books app on the iPad but not on the Mac. If you view this book in Apple's Books app, choose "Let lines break naturally" in Preferences > General.

Submit Errata

Submit your errata here for the book or for the source code by selecting New Issue. Please provide the book version listed above, chapter, section, and page number in your issue so that I can find it and, if possible, resolve it quickly.

Official Links

Please check http://developer.apple.com for additional resources including videos, sample code, documentation, and forums. You'll also find information on what is required to take advantage of these resources.

Apple has posted videos, slides, and sample code from the Worldwide Developers Conference.

Legal

Table Of Contents

Combine Latest

In the previous section we merged two streams of `DieView`s into a single stream of `DieView`s by using `merge()` to interleave them as we received elements from either one.

In this section we will use `combineLatest()` to combine two streams of `DieView`s into a single stream that contains tuples of `DieView`s.

`combineLatest()` waits until it has a pair to combine - one from each stream.

As soon as it does, it published the pair. When either stream sends a new element, `combineLatest()` publishes a new pair that consists of the new element paired with the element that didn't change.

Once we build it, you can see this in action.

Preparing to Display Pairs of Dice

We're going to display this combined stream so we need a published property `combinedViews` that is an array of pairs of `DieViews`.

Unlike `merge()`, `combineLatest()` can combine streams of different types. For instance you can combine a publisher of `Int`s that never fails with a publisher of `String`s that never fails to get a publisher of

`(Int, String)` that never fails. The error type must be the same but the output types don't have to be.

06/06/Dice/Dice/Board.swift

```
class Board: ObservableObject {
  private var die = Die()
  private var greenDie = Die()
  @Published private(set) var dieView = DieView()
  @Published private(set) var greenDieView = DieView.green()
  @Published private(set) var dieViews = [DieView]()
  @Published private(set) var greenDieViews = [DieView]()
  @Published private(set) var mergedDieViews = [DieView]()
  @Published private(set) var combinedViews
                                  = [(DieView, DieView)]()
  @Published private(set) var isRunning = false
  @Published private(set) var tallies = Array(repeating: 0,
                                              count: 6)
  private var cancellables = Set<AnyCancellable>()
}
```

You'll find `CombinedHistoryView` which allows us to display this list of pairs visually. Add it to `ContentView`.

06/06/Dice/Dice/Views/ContentView.swift

```
extension ContentView: View {
  var body: some View {
    VStack (spacing: 20) {
      HStack (spacing: 20) {
        board.dieView
        board.greenDieView
      }
      HStack (spacing: 50) {
        Button("Start",
               action: board.start)
          .disabled(board.isRunning)
        Button("Stop",
               action: board.stop)
          .disabled(!board.isRunning)
      }
      Text("History")
      DieHistoryView(dieViews: board.dieViews)
      DieHistoryView(dieViews: board.greenDieViews)
      Text("Merged")
      DieHistoryView(dieViews: board.mergedDieViews)
      Text("Combined")
      CombinedHistoryView(combinedViews: board.combinedViews)
    }
  }
}
```

We've connected our view to its source. Next, let's build our pipeline.

The Combine Latest Pipeline

The code for using `combineLatest()` will look almost exactly like the code we wrote for the `merge()` pipeline. Note the difference in `scan()`'s initial value. It's an empty array of a pair of `DieView`s instead of an empty array of `DieView`s.

06/06/Dice/Dice/*Board.swift*

```swift
extension Board {
  private func connect() {
    die = Die()
    greenDie = Die()
    dieViewPipeline()
    greenDieViewPipeline()
    historyViewsPipelines()
    mergedViewsPipeline()
    combinedViewsPipeline()
  }

  // ...

  private func mergedViewsPipeline() {
    Publishers.Merge($dieView.dropFirst(),
                     $greenDieView.dropFirst())
      .scan([DieView]()){(dieViewArray, nextView) in
        dieViewArray + [nextView]
      }
      .assign(to: &$mergedDieViews)
  }

  private func combinedViewsPipeline() {
    $dieView.dropFirst()
      .combineLatest($greenDieView.dropFirst())
      .scan([(DieView, DieView)]()){(dieViewArray, nextView) in
        dieViewArray + [nextView]
      }
      .assign(to: &$combinedViews)
  }
}
```

Before we run the app, you may want to use the `Publishers.CombineLatest` operator instead of the `combineLatest()`

method.

06/06/Dice/Dice/Board.swift

```
private func combinedViewsPipeline() {
  Publishers.CombineLatest($dieView.dropFirst(),
                           $greenDieView.dropFirst())
    .scan([(DieView, DieView)]()){(dieViewArray, nextView) in
      dieViewArray + [nextView]
    }
    .assign(to: &$combinedViews)
}
```

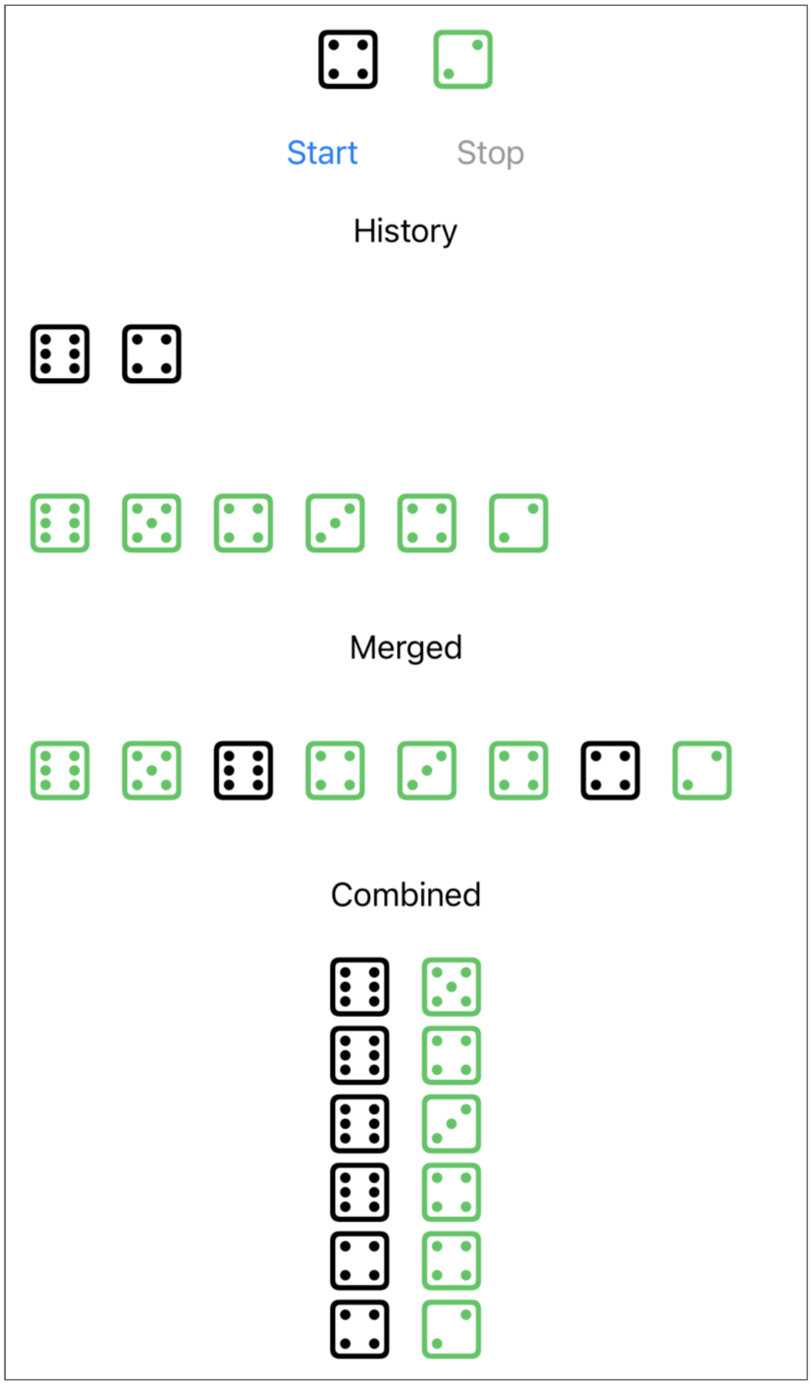
Here's where I think I'm coming down on using the operator's initializer versus using the convenience method.

If the operator is a step in the pipeline that takes a single upstream publisher then the convenience method feels like the right way to go. Think of examples we've seen using `dropFirst()`, `map()`, and `share()`.

On the other hand, if we're starting pipeline by gathering up two or more publishers then I prefer to use the actual operator. Think of examples such as `Publishers.Merge`, `Publishers.CombineLatest`, and (spoiler alert) `Publishers.Zip` that you'll meet in the next section.

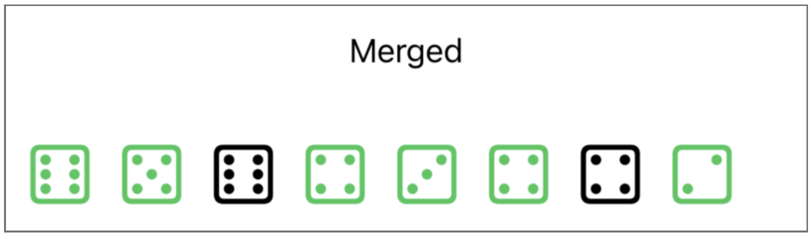
Understanding Combine Latest

Run the app. Tap start. Here's an image of a sample run.



The first pair in the Combined display is a black six and a green five.

The easiest way to understand the Combined display is to start with Merged and see the rolls in order.



First we roll a green six. No black die yet so nothing is published by `Publishers.CombineLatest`.

Then we roll a green five. Still no pair to publish

Next we roll a black six. Finally, there's a pair to publish. The latest values from each upstream publisher is a black six and a green five.

Next we receive a green four. `Publishers.CombineLatest` pairs the existing black six with this new value green four and that's the second pair we see in the Combined display.

Our next two values are a green three and a green four. This means our next two pairs are the same black six with the green three followed by the black six with the green four.

At last we roll a black four. This black four is paired with the existing green four and published by `Publishers.CombineLatest`.

Our final value is a green two and so our final pair is the black four with the green two.

Once there is a pair to publish, when either value updates, `Publishers.CombineLatest` publishes the pair of values.

Classic example

It's funny to say that there's a classic example for a technology as new as Combine, but you'll often see form validation as an example of where you want to use `combineLatest()`.

Imagine you're filling in an online form and you make a mistake in a single field. Maybe you mistype your password a second time and the two values aren't equal or maybe you forget to check a checkbox that says you authorize the site to do something or other in a user agreement you didn't bother to read.

Whatever it is - you've made one mistake.

What you'd like to do is correct this mistake and have this single new value processed with all of the existing values. You definitely don't want to have to type everything in again.

Being able to use this new value with the existing values is what `combineLatest()` is for.

We'll look at the other situation in the next section when we look at the `zip()` operator.

Transforming the Tuple

Generally we don't just display the pairs as I've done here, but we process them in some way.

For example, comment out `combinedViewsPipeline()` and replace it with this pipeline that combines the latest values from `die.sharedRoll` and `greenDie.sharedRoll` then uses `map()` to calculate their sum. Notice the `+` takes two `Int`s and returns the sum as a single `Int`. We then print the result in `sink()`.

06/06/Dice/Dice/Board.swift

```
private func combinedViewsPipeline() {
  Publishers.CombineLatest(die.sharedRoll,
                           greenDie.sharedRoll)
    .map (+)
    .sink {int in print("Sum:", int)}
    .store(in: &cancellables)
}
```

If you're uncomfortable with using `map()` this way, we can instead write it with a trailing closure like this:

06/06/Dice/Dice/Board.swift

```
private func combinedViewsPipeline() {
  Publishers.CombineLatest(die.sharedRoll,
                           greenDie.sharedRoll)
    .map {$0 + $1}
    .sink {int in print("Sum:", int)}
    .store(in: &cancellables)
}
```

or like this:

06/06/Dice/Dice/Board.swift

```
private func combinedViewsPipeline() {
  Publishers.CombineLatest(die.sharedRoll,
                           greenDie.sharedRoll)
    .map {(die1, die2) in die1 + die2}
    .sink {int in print("Sum:", int)}
    .store(in: &cancellables)
}
```

Here's a sample run. The green five is ignored. The black four and green three are added to give us a sum of `7`, and so on.

```
Green Die View: receive value: (5)
Green Die View: receive value: (3)
Die View: receive value: (4)
Sum: 7
Green Die View: receive value: (5)
Sum: 9
Green Die View: receive value: (3)
Sum: 7
Green Die View: receive value: (1)
Sum: 5
Die View: receive value: (5)
Sum: 6
```

This is a case in which we might use a version of a convenience method for `combineLatest()` that combines two publishers and applies a transform.

06/06/Dice/Dice/Board.swift

```
private func combinedViewsPipeline() {
  die.sharedRoll
    .combineLatest(greenDie.sharedRoll, +)
    .sink {int in print("Sum:", int)}
    .store(in: &cancellables)
}
```

Again if you prefer to use a trailing closure, we can write this like this:

06/06/Dice/Dice/Board.swift

```
private func combinedViewsPipeline() {
  die.sharedRoll
    .combineLatest(greenDie.sharedRoll){$0 + $1}
    .sink {int in print("Sum:", int)}
    .store(in: &cancellables)
}
```

or like this:

06/06/Dice/Dice/Board.swift

```
private func combinedViewsPipeline() {
  die.sharedRoll
    .combineLatest(greenDie.sharedRoll){(die1, die2) in
      die1 + die2
    }
    .sink {int in print("Sum:", int)}
    .store(in: &cancellables)
}
```

Before moving on, remove this version of `combinedViewsPipeline()` and uncomment the previous version.

06/06/Dice/Dice/Board.swift

```
private func combinedViewsPipeline() {
  Publishers.CombineLatest($dieView.dropFirst(),
                           $greenDieView.dropFirst())
    .scan([(DieView, DieView)]()){(dieViewArray, nextView) in
      dieViewArray + [nextView]
    }
    .assign(to: &$combinedViews)
}
```

If you need them, there are versions of `combineLatest()` for combining up to four publishers.

Next, let's replace `combineLatest()` with `zip()`.