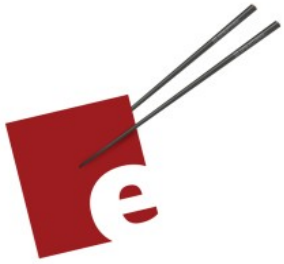


First Edition

# A SwiftUI Kickstart

DANIEL H STEINBERG

Introducing the
SwiftUI User Interface Framework

Editors Cut

# A SwiftUI Kickstart

## Introducing The SwiftUI User Interface Framework

by Daniel H Steinberg

Editors Cut

# Copyright



ISBN-13: 9 978-1-944994-00-6

# Book Version

This is version 0.6 for Swift 5.3, Xcode 12, and iOS 14 released October 10, 2020.

# Recommended Settings

The ePub is best viewed in scrolling mode using the original fonts. On smaller devices I also choose landscape. If you view this book in Apple's Books app, choose "Let lines break naturally".

# Legal

Every precaution was taken in the preparation of this book. 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.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks or service marks. Where those designations appear in this book, and Dim Sum Thinking, Inc. was aware of the trademark claim, the designations have been printed with initial capital letters or in all capitals.

This book uses terms that are registered trademarks of Apple Inc. for which the terms of use don't permit rendering them in all caps or initial caps. You can view a complete list of the trademarks and registered trademarks of Apple Inc. at http://www.apple.com/legal/trademark/appletmlist.html.

The Editor's Cut name and logo are registered trademarks of Dim Sum Thinking, Inc.

# Grids

In this section we use grids to layout our symbols. The grid views don't replace `List`, they replace `VStack` and `HStack`. We begin this section by backing up a bit and then we replace a `VStack` with a `LazyVGrid`. After that we look at several ways to layout our grid.

## Back up

Let's begin by backing up a bit. Here's the current state of our `List`.

*04/05/Symbols/Symbols/ContentView.swift*

```
struct ContentView: View {
  var body: some View {
    List(symbols){symbol in
      SystemLabel(name: symbol.name)
    }
  }
}
```

Let's back up to where we used `ScrollView`, `VStack`, and `ForEach`.

*04/05/Symbols/Symbols/ContentView.swift*

```
struct ContentView: View {
  var body: some View {
    ScrollView {
      LazyVStack(spacing: 20){
        ForEach(symbols){symbol in
          SystemLabel(name: symbol.name)
        }
      }
    }
  }
}
```

Let's convert the `LazyVStack` to a `LazyVGrid`.

## Introducing a Grid

Grids are either vertical or horizontal. If we use a vertical grid then we have to specify how the horizontal part, the columns, will be presented. Each row will be filled using that rule and then the grid will manage the vertical axis.

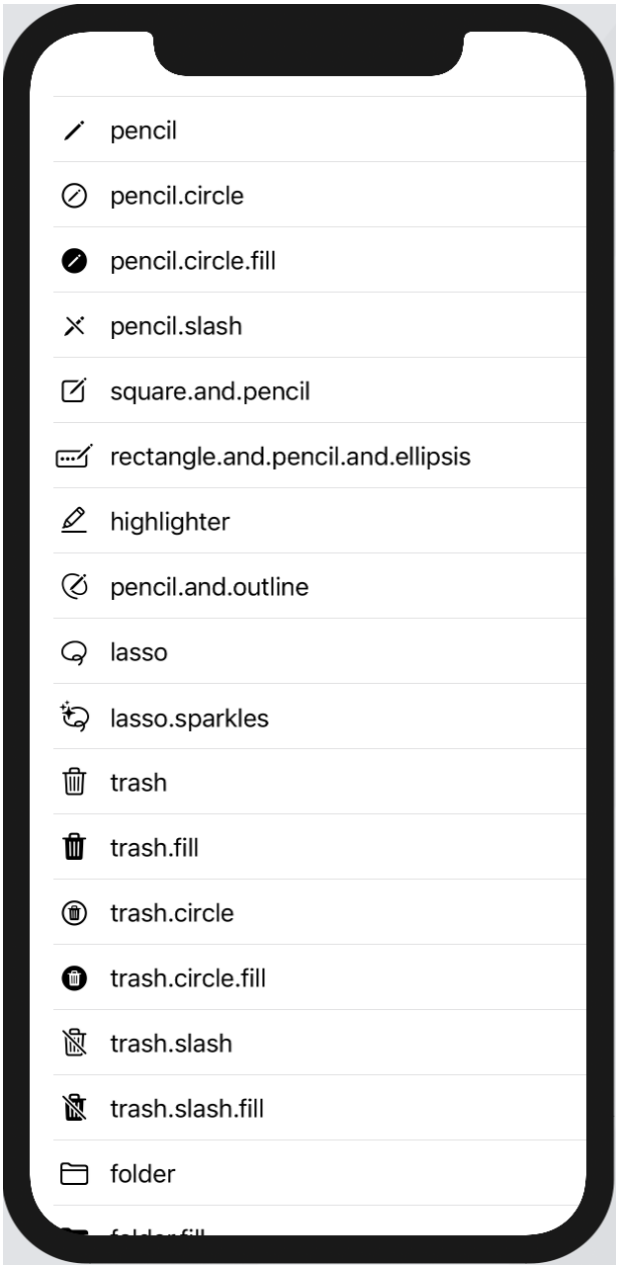
We'll specify the columns using an array of `GridItem`s. We then use that array in our `LazyVGrid`.

*04/05/Symbols/Symbols/ContentView.swift*

```
struct ContentView: View {
  let columns = [GridItem(.flexible())]

  var body: some View {
    ScrollView {
      LazyVGrid(columns: columns,
                spacing: 20){
        ForEach(symbols){symbol in
          SystemLabel(name: symbol.name)
        }
      }
    }
  }
}
```

This looks the same as the `VStack` version.




Let's display only the symbols and experiment with the grid layout.

## Flexible

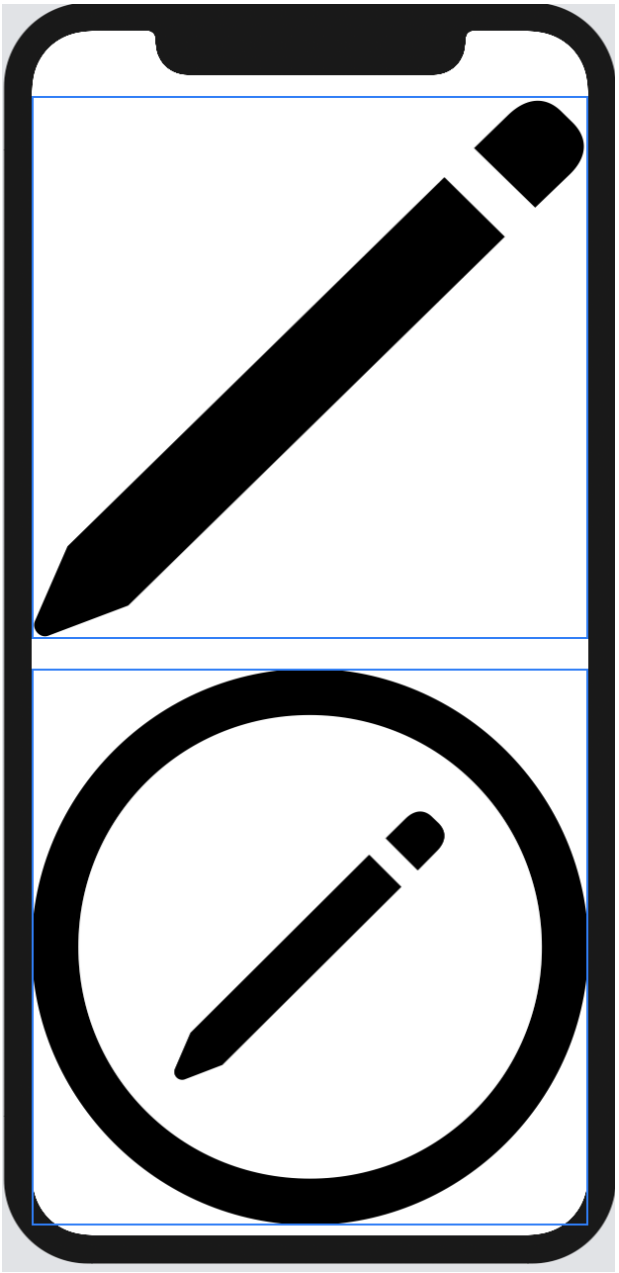
Replace `SystemLabel` with `Image` and allow the image to grow to fill the space allowed.

*04/05/Symbols/Symbols/ContentView.swift*

```
struct ContentView: View {
  let columns = [GridItem(.flexible())]

  var body: some View {
    ScrollView {
      LazyVGrid(columns: columns,
                spacing: 20){
        ForEach(symbols){symbol in
          Image(systemName: symbol.name)
            .resizable()
            .scaledToFit()
        }
      }
    }
  }
}
```

I've selected the line containing `Image` so you can see the blue rectangle that the image is being sized to fit.



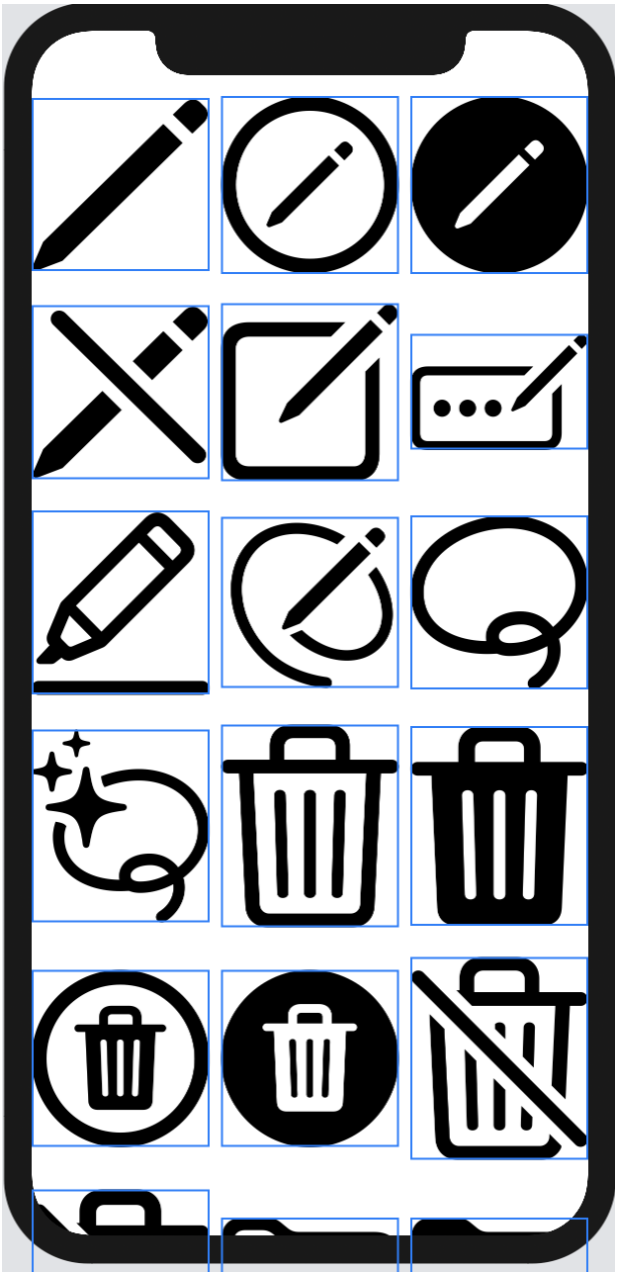
The `columns` are an array containing a single `GridItem` that is `flexible` so it grows to fill the space. As you typed in `.flexible()` you should have seen a completion that allows you to specify a minimum and maximum value.

Now, check out what happens if we add two more columns of `flexible()` items.

*04/05/Symbols/Symbols/ContentView.swift*

```
struct ContentView: View {
  let columns = [GridItem(.flexible()),
                 GridItem(.flexible()),
                 GridItem(.flexible())]
  //...
}
```

Again, I've selected `Image` so you can see the bounding rectangles. Note that after the spacing is accounted for, the remaining horizontal space is divided into three equal pieces to accomodate the `flexible` items.



You're beginning to feel the power or grids. But there's more!

## Fixed

We can also specify the size of the items in a column using the `fixed GridItem`.

*04/05/Symbols/Symbols/ContentView.swift*

```
struct ContentView: View {
  let columns = [GridItem(.fixed(200)),
                 GridItem(.flexible()),
                 GridItem(.flexible()),
                 GridItem(.flexible())]
  //...
}
```

This sets the first column to be a fixed width of 200 and splits the remainder among the three remaining columns. It looks like this.

We use fixed and flexible either by themselves or in combination with each other to specify a layout for a specified number of columns. There is another type of `GridItem`.

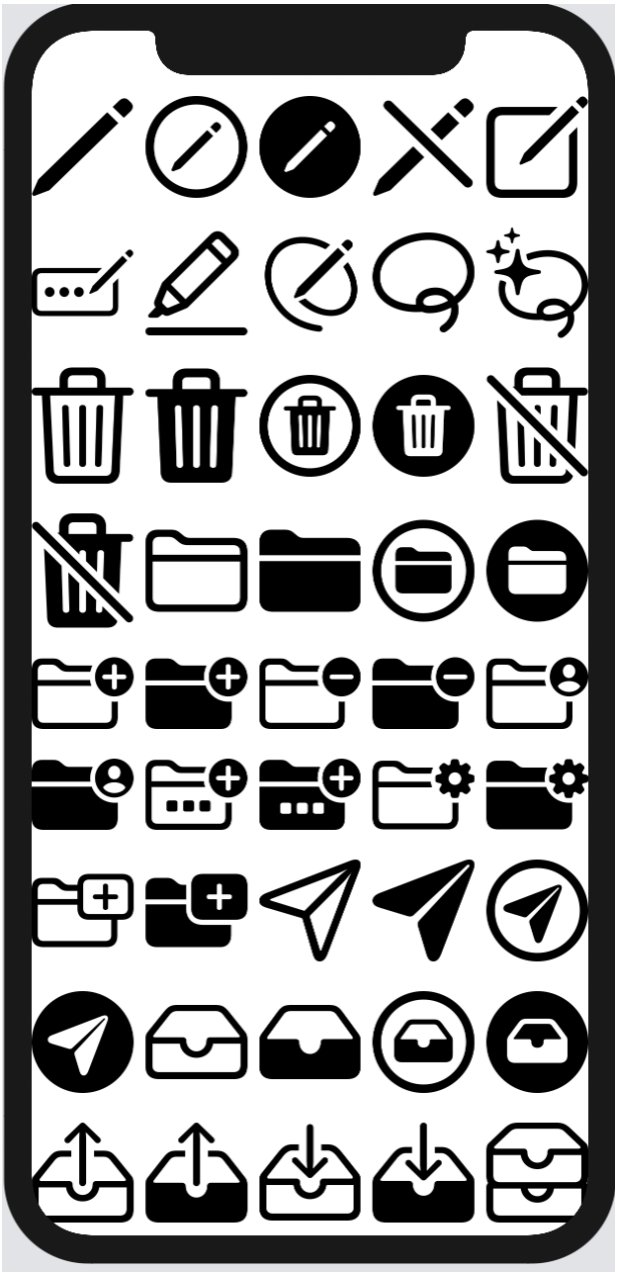
## Adaptive

The third type of `GridItem` is `adaptive`. In this final example we use the `adaptive` choice with the restriction that the symbols must be at least 60 wide. The result is to fit as many symbols in a row as we can of equal size so long as they are at least 60 wide.

*04/05/Symbols/Symbols/ContentView.swift*

```
struct ContentView: View {
  let columns = [GridItem(.adaptive(minimum: 60))]
  //...
}
```

In this screenshot of the preview you can see even rows that each have five symbols across.



This is a great setting in which to experiment with different settings for `columns` and view the results. Play a little bit before we return to `List`s in the next section.