IS 543 Fall 2023

Mobile Platform Development

Project 1 Questions

Fresh Look at SwiftUI

Lists, Forms, Navigation, Etc.

Dr. Stephen W. Liddle
782 TNRB, office hours Tu 9:30-10:30am, Th 2-3pm
is543@byu.edu
801-422-8792 (rings on my cell)
Zoom: https://bit.ly/liddlezoom

Today

- Project 1 questions?
- A fresh look at SwiftUI
- Navigation, split views, and lists
  `NavigationStack`
  `NavigationLink`
  `List`
  `Form`
  `Section`

Project 1

- What Project 1 questions do you have?
- How to turn in Project 1

  Place a README file in the root folder of your project describing your experience on this project.

  In your README, include thoughts on what you did well or not so well. This is your chance not only to explain your work, but to lobby for the grade you feel you earned.

  Fill out the rubric and place it in the same root folder of your project.

  Before zipping up your project, rename the folder "Project 1 Lastname Firstname". Do this is so that when I unzip the file, it goes into a folder named "Project 1 Lastname Firstname". This will help me keep everyone's project separate and properly identified. You'll receive 1 point for following this naming convention.

  Turn in this project by uploading the zip file of your project to Learning Suite.

MVC vs. Declarative UI

Review

- Model View Controller (MVC) is the traditional iOS app architecture
  - It is orderly, well understood, and works reasonably well
  - Build an app using a storyboard, combining potentially many coordinating MVC units
  - But it's easy to create massive controllers, and that's not great architecture
- Swift UI follows the "Declarative UI" architectural pattern
  - You manage the state of an app and then declare a UI specification that renders updates to the UI whenever the state changes
  - This is the React approach
  - Also, Google's Flutter project follows this approach
- Declarative vs. Imperative UI:
  - With imperative, you call methods on widgets when events occur and the UI needs to change
  - With declarative, a widget calls setState() to make state changes, then the framework decides which parts of the UI need to be modified

Why Declarative UI?

Review

- Managing state changes can be tricky

  Lots of moving parts:

  - User performs touch event
  - Network request starts or finishes
  - Timer fires
  - Operation fails or succeeds

  And from all that you want to change the UI to reflect the state of the app

  Transitions from state to state can be very complex

  Order dependencies and concurrent processing can cause interesting bugs

- Solution

  Manage the state (only) and declare how the UI can be built from the state

  Then let the framework generate any UI updates needed for any state changes that occur

  I.e. you notify the framework of a state change and it re-renders the UI

MVVM Architecture

Review

@Observable
ObservableObject
@Published
objectWillChange.send()
.environmentObject()

might "interpret"

publishes "something changed"

calls "intent" function

modifies the Model

notices changes

automatically observes publications, pulls data, and rebuilds

ViewModel

Binds View to Model
Interpreter
Processes Intent

Model

UI Independent
Data + Logic
"The Truth"

@ObservedObject
@Binding
.onReceive
@EnvironmentObject

View

Reflects the Model
Stateless
Declared
Reactive

SwiftUI

- From the ground up, new architecture to (mostly) replace UIKit
  There are ways to integrate UIKit and SwiftUI
- SwiftUI is (currently) code-centric
  Rather than drag and drop UI with Interface Builder, it's (currently) most common to type code
- Fundamental unit of UI code is `View`
  Views are lightweight
  It is common to nest views (deeply)
  `View` is just a protocol:

```
public protocol View {
    associatedtype Body : View

    @ViewBuilder var body: Self.Body { get }
}
```

Yes, the definition is recursive (!)

That's okay — just define a body that returns some View

Simple SwiftUI View

- To create your own `View`, write a `struct` that conforms to the `View` protocol (i.e. it defines a `body` computed property)

```
struct MyBeautifulView : View {
    var body: some View {
        Text("Ain't it great?")
    }
}
```

Have you seen Swift's keyword `some` before?

Formally, it's an "opaque result type", but you can think of it as syntax meaning "return something that conforms to View"; and you can read more about it if you'd like.

TLDR; "`some`" makes generic types more implementation independent

Simple SwiftUI View

- To create your own `View`, write a `struct` that conforms to the `View` protocol (i.e. it defines a `body` computed property)

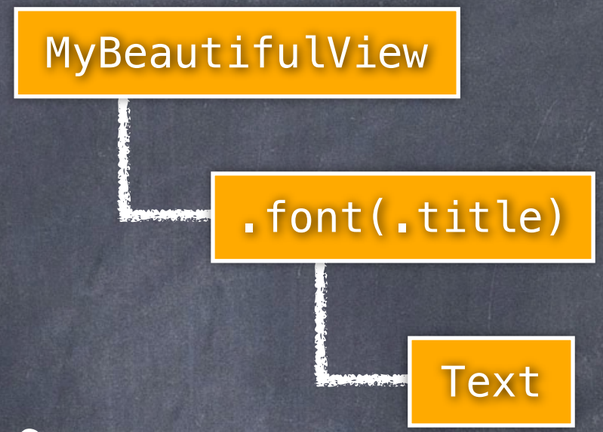
```
struct MyBeautifulView : View {
   var body: some View {
        Text("Ain't it great?")
    }
}
```

- UIKit has fat views with LOTS of properties built in
- SwiftUI views are lightweight and inherit to add more properties, e.g. by using modifiers to wrap views

Simple SwiftUI View

- A modifier wraps a view in another view

```
struct MyBeautifulView : View {

    var body: some View {
        Text("Ain't it great?")
            .font(.title)
    }
}
```



- With lightweight views, nesting them is almost free

  Analogous to a function call in a language like Java with a good optimizing compiler

  In UIKit/MVC development, you would try really hard not to nest views deeply

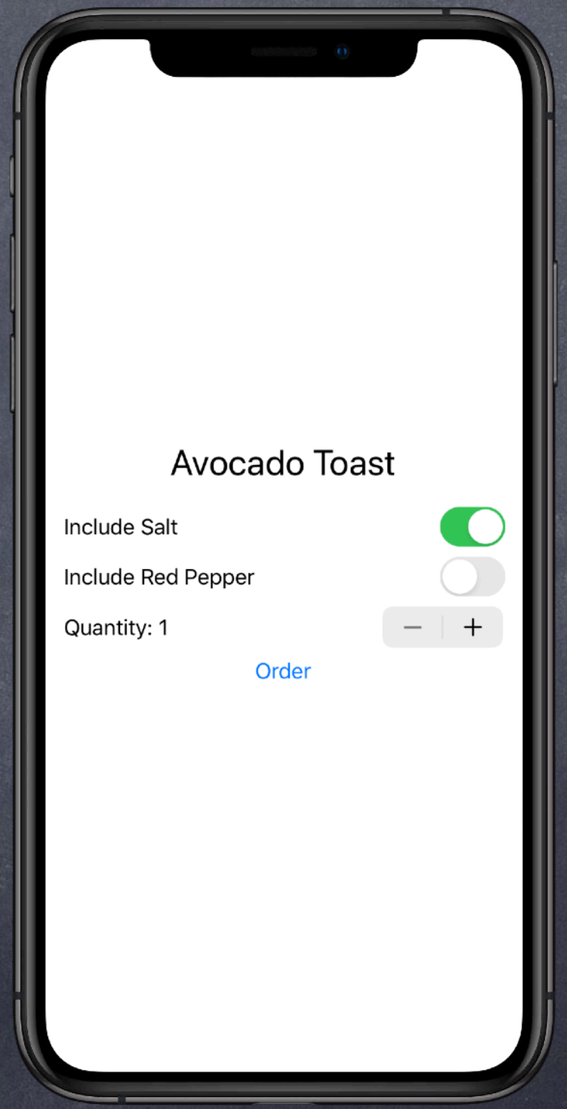
  SwiftUI is the opposite — it's good practice

  Like React, SwiftUI's rendering engine sees what changes need to be made and manages the process efficiently; having a rich view hierarchy helps this process

SwiftUI

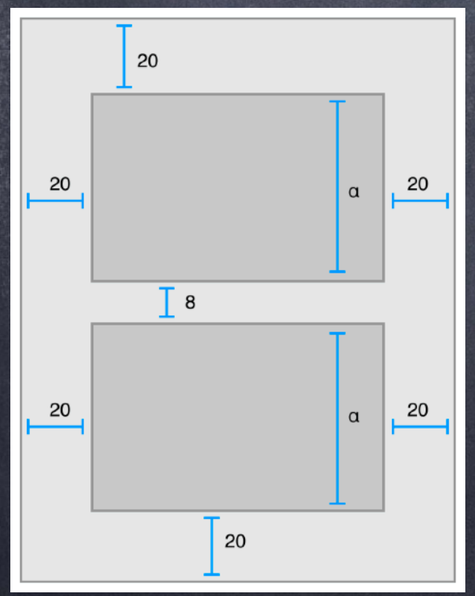
- Avocado toast example

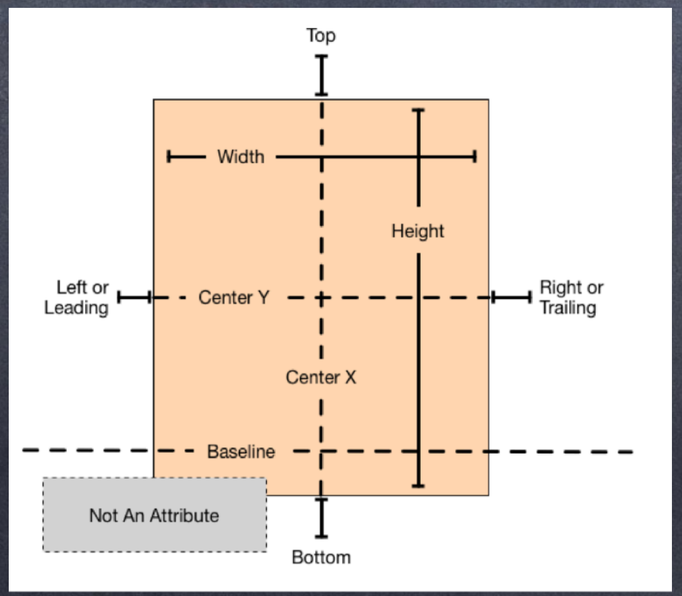
```
VStack {
    Text("Avocado Toast").font(.title)

    Toggle(isOn: $order.includeSalt) {
        Text("Include Salt")
    }
    Toggle(isOn: $order.includeRedPepperFlakes) {
        Text("Include Red Pepper Flakes")
    }
    Stepper(value: $order.quantity, in: 1...10) {
        Text("Quantity: \(order.quantity)")
    }

    Button(action: submitOrder) {
        Text("Order")
    }
}
.padding()
```

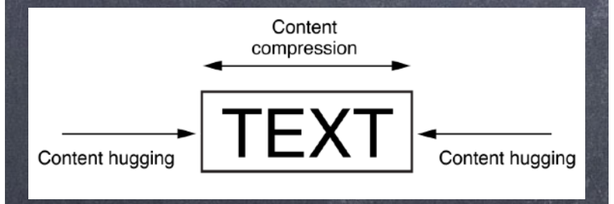


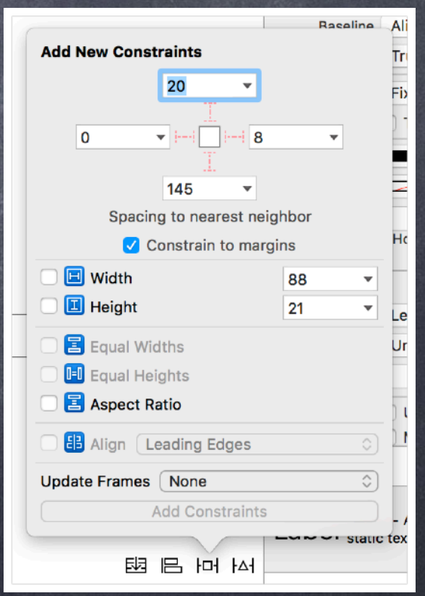
Layout in SwiftUI

- We try to use HStacks and VStacks with padding, alignment, Spacers, and so forth
- This tends to be easier to specify and manage than UIKit's "Autolayout" constraint management mechanism









Built-in View Types

- Object library shows these view types:

iOS View Types

Actually, some of these are containers too

- Controls

  `Button, ColorPicker, DatePicker, EditButton, Gauge, Label, LabeledContent, Link, MultiDatePicker, NavigationLink, PasteButton, Picker, ProgressView, RenameButton, SecureField, ShareLink, SignInWithAppleButton, Slider, Stepper, Text, TextEditor, TextField, Toggle`

- Container controls

  `ControlGroup, DisclosureGroup, Form, Group, GroupBox, List, NavigationSplitView, NavigtationStack, OutlineGroup, ScrollView, Section, TabView, Table`

- Paints and shapes

  `AngularGradient, EllipticalGradient, LinearGradient, RadialGradient`
  `Capsule, Circle, ContainerRelativeShape, Ellipse, Rectangle, RoundedRectangle`

- Layout views

  `GeometryReader, {H|V|Z}Stack, Lazy{H|V}Grid, ScrollViewReader, Spacer, Lazy{H|V}Stack, ViewThatFits`

- Other views

  `AsyncImage, Canvas, Color, ContentUnavailableView, Divider, EmptyView, Image, Menu, Path, TimelineView`

View Containers

- We commonly use view containers to nest views
- `VStack` is a good example

```
VStack {
    Image(...)
    Text(...)
}
```

Hmm... What is this?

That's a trailing closure! Here's how `VStack` is defined:

```
public struct VStack<Content: View>: View {
    public init(
        alignment: HorizontalAlignment = .center,
        spacing: Length? = nil,
        @ViewBuilder content: () -> Content
    )
}
```

In SwiftUI it's common to specify a closure for content-like parameters (making view nesting easier)

View Containers

- `Button` is also, in fact, a container

```
Button(action: {}) {
    Image(systemName: "star.fill")
    Text("Order")
}
```

This places an image side-by-side with a piece of text in the button

View Containers

- `Button` is also, in fact, a container

```
Button(action: {}) {
    Image(systemName: "star.fill")
    Text("Order")
}
```

This is common enough to merit its own `View` type, `Label`:

```
Button(action: {}) {
    Label("Order", systemImage: "star.fill")
}
```

Binding State to Views

- You can bind state variables to other code like views

  Mark the state variable with the attribute `@State` and use `$` to signal binding

- `Stepper` is a good example

```
struct OrderForm : View {
    @State private var order: Order

    var body: some View {
        Stepper(value: $order.quantity, in: 1...10) {
            Text("Quantity: \(order.quantity)")
        }
    }
}
```

Making a List Is Easy

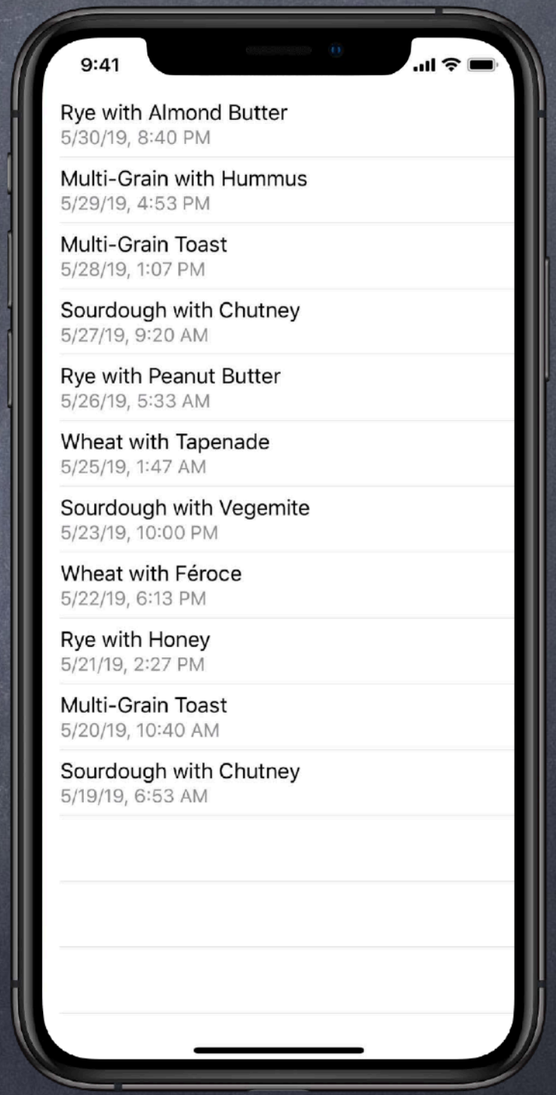
- Making a List is easy:

  Pass an array to List and return a configured View for each array element

  Here we have the typical title/subtitle cell:

```
List(previousOrders) { order in
    VStack(alignment: .leading) {
        Text(order.summary)
        Text(order.purchaseDate)
            .font(.subheadline)
            .foregroundColor(.secondary)
    }
}
```

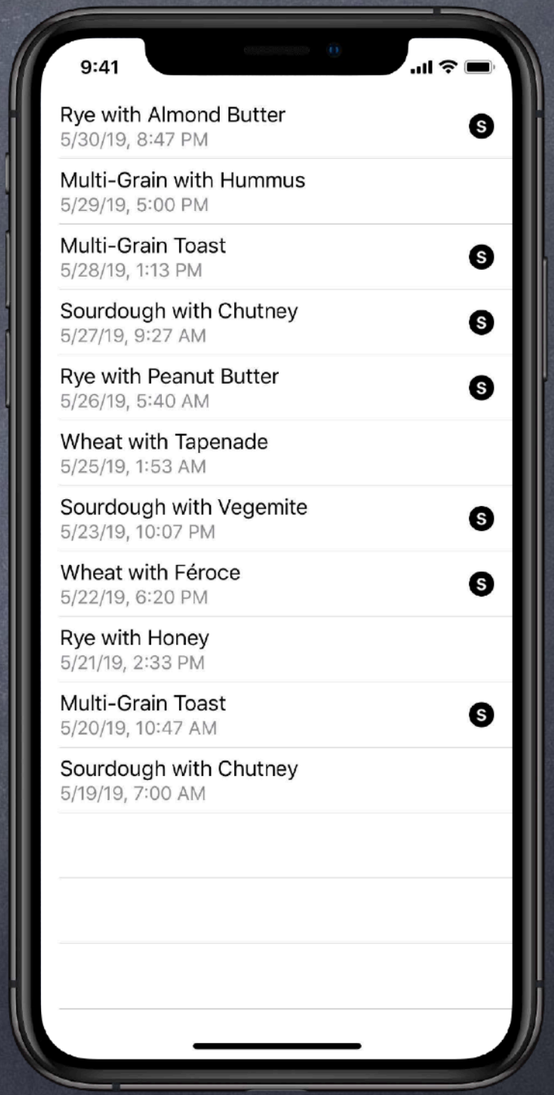
  You can use HStacks and VStacks and Spacers to present more information as you wish



Making a List Is Easy

- Add icons with an HStack and a Spacer:

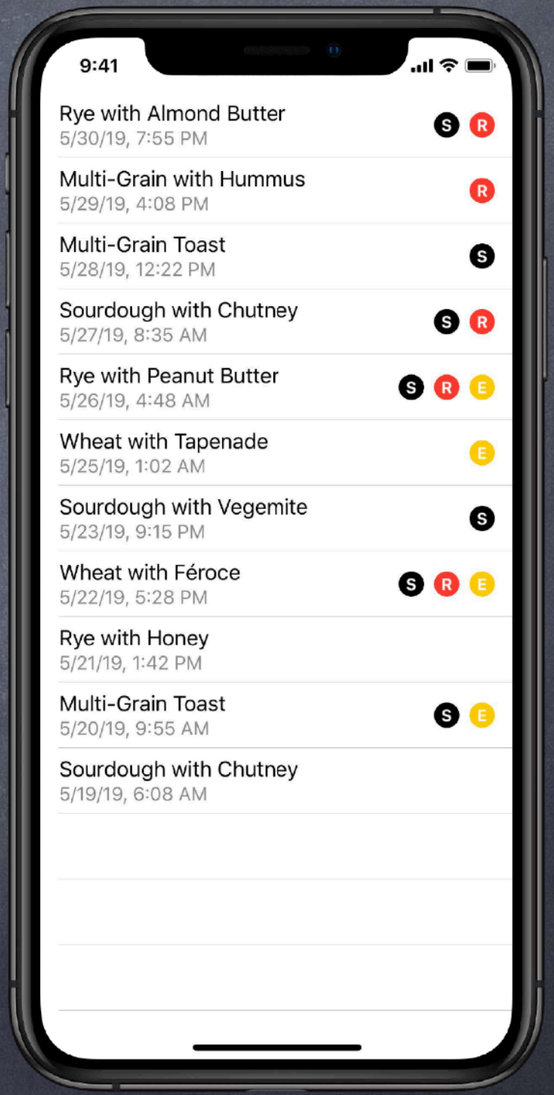
```
List(previousOrders) { order in
    HStack {
        VStack(alignment: .leading) {
            Text(order.summary)
            Text(order.purchaseDate)
                .font(.subheadline)
                .foregroundColor(.secondary)
        }
        Spacer()
        if order.includeSalt {
            SaltIcon()
        }
    }
}
```



Making a List Is Easy

- Use ForEach when you have a list of options:

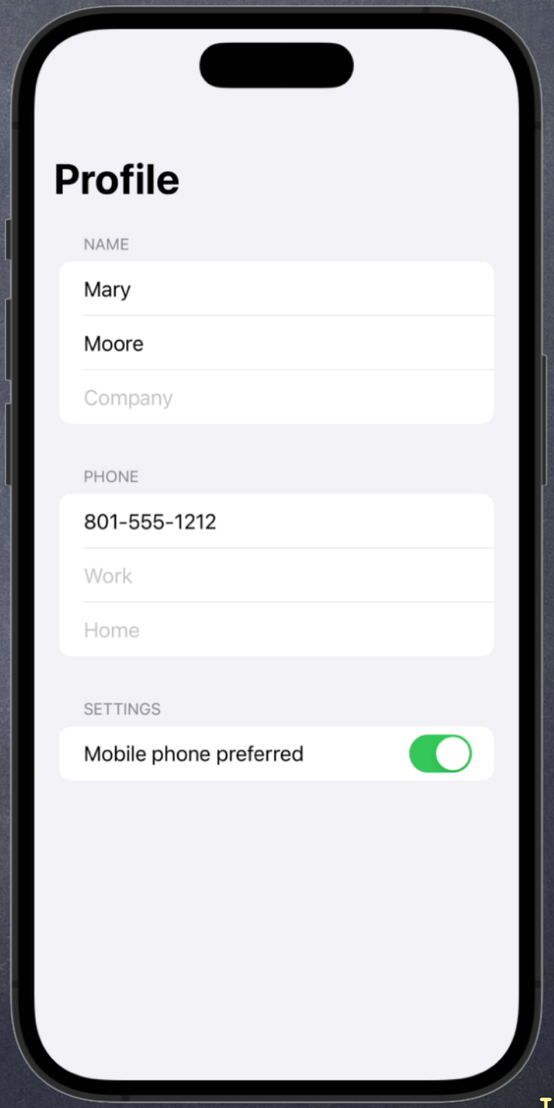
```
struct OrderCell : View {
    var order: CompletedOrder
    var body: some View {
        HStack {
            VStack(alignment: .leading) {
                Text(order.summary)
                Text(order.purchaseDate)
                    .font(.subheadline)
                    .foregroundColor(.secondary)
            }
            Spacer()
            ForEach(order.toppings) { topping in
                ToppingIcon(topping)
            }
        }
    }
}
```



Making a Form Is Also Easy

- Use Form for settings or inspectors

```
var body: some View {
    NavigationStack {
        Form {
            Section(header: Text("Name")) {
                TextField("First name", text: $firstName)
                TextField("Last name", text: $lastName)
                TextField("Company", text: $company)
            }
            Section(header: Text("Phone")) {
                TextField("Mobile", text: $mobilePhone)
                TextField("Work", text: $workPhone)
                TextField("Home", text: $homePhone)
            }
            Section(header: Text("Settings")) {
                Toggle("Mobile phone preferred", isOn: $mobilePreferred)
            }
        }
        .navigationBarTitle("Profile")
    }
}
```



Layout for Multiple Devices

- `NavigationView` is a multi-purpose container
  - It provides a navigation bar (though this can be hidden if you wish)
  - It automatically supports master/detail split view on larger devices
  - Its first child is the master view
  - Its second child is the detail view
- On an iPad, master and detail will be side-by-side
  - On an iPhone, you need the master to link to the detail e.g. with `NavigationLink`
  - So the detail view pushes onto the navigation stack (on top of the master)
  - Various devices behave somewhat differently with this
- Messy! In iOS 16+, now use `NavigationStack` and `NavigationSplitView`
  - You get better control over view presentation, container configuration, and programmatic navigation
  - See https://apple.co/3y86vHW for details

Converting Old NavigationViews

- For one-column navigation, instead of this:

```
NavigationView {
    /* content */
}
.navigationViewStyle(.stack)
```

- Use this:

```
NavigationStack {
    /* content */
}
```

Converting Old NavigationViews

- For multiple-column or variable-column navigation (e.g. where it's 1-column for iPhone but 2-column for iPad), instead of this:

```
NavigationView {
    /* column 1 */
    /* column 2 */
}
```

- Use this:

```
NavigationSplitView {
    /* column 1 */
} detail: {
    /* column 2 */
}
```

and

```
NavigationView {
    /* column 1 */
    /* column 2 */
    /* column 3 */
}
```

```
NavigationSplitView {
    /* column 1 */
} content: {
    /* column 2 */
} detail: {
    /* column 3 */
}
```

`Color` vs. `UIColor`

- `Color`
  - What this symbol means in SwiftUI varies by context
  - Acts as a color specifier, e.g., `.foregroundColor(Color.green)`
  - Can also act like a `ShapeStyle`, e.g., `.fill(Color.blue)`
  - Can also act like a `View`, e.g., `Color.white` can appear wherever a `View` can appear
  - Due to this multifaceted role, its API is limited mostly to creation/comparison
- `UIColor`
  - Is used to manipulate colors
  - Also has many more built-in colors than `Color`, including "system-related" colors
  - Can be interrogated and can convert between color spaces
  - For example, you can get the RGBA values from a `UIColor`
  - Once you have desired `UIColor`, employ `Color(uiColor:)` to use it in one of the roles above

Image vs. UIImage

- Image
  - Primarily serves as a View
  - Is not a type for vars that hold an image (i.e. a .jpeg or .png or some such); that's UIImage
  - Access images in your Assets.xcassets (in Xcode) by name using Image(_ name: String)
  - Also, many, many system images available via Image(systemName:)
  - Search all these system images in the SF Symbols app (available at developer.apple.com/design)
  - While you're there, study the Human Interface Guidelines (a must for App Store submission)
  - You can control the size of system images with .imageScale() view modifier
  - System images are also very useful as masks (for gradients, for example)
- UIImage
  - Is the type for actually creating/manipulating images and storing in vars
  - Very powerful representation of an image
  - Multiple file formats, transformation primitives, animated images, etc.
  - Once you have the UIImage you want, use Image(uiImage:) to display it

Coming Up...

- Continue learning SwiftUI techniques and patterns
  - We need to learn how to read/write data in the filesystem and DBMS
  - Also how to interact with the network
  - There are a number of additional SwiftUI views, techniques, patterns yet to explore
  - And we need to learn something about UIKit too

Project 1

- What Project 1 questions do you have?
- How to turn in Project 1

  Place a README file in the root folder of your project describing your experience on this project.

  In your README, include thoughts on what you did well or not so well. This is your chance not only to explain your work, but to lobby for the grade you feel you earned.

  Fill out the rubric and place it in the same root folder of your project.

  Before zipping up your project, rename the folder "Project 1 Lastname Firstname". Do this is so that when I unzip the file, it goes into a folder named "Project 1 Lastname Firstname". This will help me keep everyone's project separate and properly identified. You'll receive 1 point for following this naming convention.

  Turn in this project by uploading the zip file of your project to Learning Suite.

Today's Devotional

- Sister Kristin M. Yee
  Second Counselor, Relief Society General Presidency
  The Church of Jesus Christ of Latter-day Saints

  Earned an MPA here at BYU Marriott
  Spoke in general conference a year ago:
  "Beauty for Ashes: The Healing Path of Forgiveness"
  Worked as an artist and producer at Disney
  Is quite a good painter