

NOTES

Course

CS193p – Spring 2020
Developing Apps for iOS

Professor

Paul Hegarty

University

Stanford

Version: 1.2
Date: June 6, 2020

1	Lec01. Course Logistics and intro to SwiftUI	8
1.1	Create a new SwiftUI project	8
1.2	ContentView.swift	8
1.2.1	Canvas	9
1.2.2	import SwiftUI	9
1.2.3	Struct.....	9
1.2.4	struct ContentView: View	10
1.2.5	Text	10
1.2.6	var body: some View {.....	10
1.2.7	return Text("Hello There, World!")	12
1.3	Build a UI that looks a little bit more like a card.	13
1.3.1	Emoji	13
1.3.2	Rounded rectangle.....	15
1.3.3	Combine the rounded rectangle with the text showing the ghost emoji	15
1.3.4	Views, Shapes and .stroke().....	16
1.3.5	Orange foreground color.....	18
1.3.6	Padding	19
1.3.7	Fill.....	20
1.3.8	stroke(linewidth: 3)	21
1.3.9	Dark mode in simulator	22
1.3.10	font(Font.largeTitle)	23
1.4	Multiple cards.....	24
1.4.1	#TBC	25
1.4.2	25
1.4.3	Fronts and backs of cards	26
1.4.4	Conditional fronts and backs of cards	27
2	Lec02. MVVM and the Swift Type System	28
2.1	MVVM (Model-View-ViewModel).....	29
2.1.1	MVVM - Model	30
2.1.2	MVVM - View.....	31
2.1.3	MVVM - ViewModel.....	32
2.1.4	MVVM - How does MVVM work?	33
2.1.5	MVVM – Swift syntax	34
2.1.6	MVVM – What if the View wants to change the Model?	35
2.1.7	MVVM – This is it.....	37
2.2	Types.....	38
2.2.1	Structs and classes – Stored vars, computed vars	38
2.2.2	Structs and classes – Constant lets and functions.....	39
2.2.3	Structs and classes – Initializers	39

2.2.4	What's the difference between structs and classes?	40
2.2.5	Generics	41
2.2.6	Functions as types	42
2.3	Back to the demo.....	43
2.3.1	Model – Create MemoryGame.swift	43
2.3.2	Model – MemoryGame.Card struct.....	44
2.3.3	Model – choose(card:) function.....	44
2.3.4	Model – isFaceUp and isMatched variables	45
2.3.5	Model – MemoryGame<CardContent>.....	45
2.3.6	ViewModel – Create EmojiMemoryGame.swift.....	46
2.3.7	ViewModel - Import Foundation.....	46
2.3.8	ViewModel – Create a property to access the Model.....	46
2.3.9	ViewModel - Why ViewModel should be a class?	47
2.3.10	ViewModel - Intents	48
2.3.11	ViewModel – Intents -	49
2.3.12	ViewModel – Access to the Model.....	50
2.3.13	ViewModel – Class EmojiMemoryGame has no initializers.....	51
2.3.14	How many cards are in the game? – numberOfPairsOfCards: 2 – (Closures)	51
2.3.15	Closures – init(numberOfPairsOfCards: Int)	52
2.3.16	Closures – cardContentFactory: (Int) -> CardContent	53
2.3.17	Closures – createCardContent(pairIndex: Int) -> String.....	54
2.3.18	Closures - { (pairIndex: Int) -> String in return "😊" }.....	54
2.3.19	Closures - { _ in "😊" }	55
2.3.20	Closures – Return a different emoji for each pair of cards.....	57
2.3.21	View – var card: MemoryGame<String>.Card	59
2.3.22	SceneDelegate – let contentView = ContentView(viewModel: game).....	61
2.3.23	63
2.3.24	ContentView – iterate over the cards.....	65
2.3.25	Summary	66
2.3.26	Run – isFaceUp = false	67
2.3.27	Express an intent.....	68
2.3.28	Run	70
2.4	Assignment I: Memorize.....	71
2.4.1	Required Task 1	71
2.4.2	Required Task 2	71
2.4.3	Required Task 3	71
2.4.4	Required Task 4	72
2.4.5	Required Task 5	72
2.4.6	Required Task 6	72
2.4.7	Hint 1	72
2.4.8	Hint 2	73
2.4.9	Hint 3	73

2.4.10	Hint 4	73
2.4.11	Hint 5	73
2.4.12	Hint 6	73
2.4.13	Hint 7	73
2.4.14	Hint 8	74
2.4.15	Hint 9	74
3	Lec03. Reactive UI + Protocols + Layout.....	75
3.1	Reactive Swift demo	76
3.1.1	Refactoring ContentView -	76
3.1.2	Model	77
3.1.3	Model – Reserved word mutating in a struct	78
3.1.4	Use of identifiable	79
3.1.5	Run	80
3.2	Publish/Subscribe	80
3.2.1	Publish – ViewModel – ObservableObject-.....	80
3.2.2	Publish – ViewModel – @Published	82
3.2.3	Subscribe – View – @ObservedObject.....	83
3.2.4	Run	84
3.2.5	Recap	84
3.3	Protocols	86
3.3.1	86
3.3.2	Layout	86
3.3.3	86
3.4	Back to the Demo	86
3.4.1	Adjust the font manually	87
3.4.2	Adjust the font with GeometryReader.....	87
3.4.3	Documentation - GeometryReader.....	88
3.4.4	Documentation – GeometryProxy	89
3.4.5	90
3.4.6	Run	90
3.4.7	91
3.5	Coding style	92
3.5.1	93
3.5.2	94
4	Lec04. Grid + enum + Optionals	95
4.1	Create a Grid	96
4.1.1	How ZStack works	97
4.1.2	View – Grid(viewModel.cards) { card in	97
4.1.3	Grid – Create Grid.swift.....	98
4.1.4	Grid - Arguments with no labels	99
4.1.5	Grid - @scaping closure	100
4.1.6	Grid - ViewModel - How does Swift accomplish keeping local variables in closures or future calls?	101

4.1.7	Grid - where Item: Identifiable.....	103
4.1.8	Grid - where itemView: View	104
4.1.9	Grid - GeometryReader.....	105
4.1.10	Grid - GridLayout.swift.....	106
4.1.11	Grid - Create GridLayout.....	106
4.1.12	Grid - Position the view.....	107
4.1.13	Grid – Recap	108
4.1.14	Grid - Run.....	109
4.1.15	Grid - View - Padding.....	110
4.1.16	Search an item – Item is an identifiable Card struct	111
4.1.17	Search an item - Extension Array where Element: Identifiable {	113
4.1.18	Search an item - Extension Array where Element: Identifiable { - Array+Identifiable.swift	115
4.2	enum	116
4.2.1	enum – Associated Data	116
4.2.2	enum – Setting the value of an enum.....	117
4.2.3	enum –.....	117
4.2.4	enum –.....	117
4.2.5	enum –.....	118
4.2.6	enum –.....	118
4.2.7	enum –.....	118
4.2.8	enum –.....	119
4.2.9	enum –.....	126
4.3	Back to the demo.....	127
4.3.1	Optional – In the function	127
4.3.2	Optional - Calling the function - Grid	128
4.3.3	Optional - Calling the function - MemoryGame	129
4.4	Play de game	130
4.4.1	131
4.4.2	Equatable	132
4.4.3	Equatable - Documentation	134
4.4.4	Two cards matching.....	135
4.4.5	Turn all the cards face-down	136
4.4.6	EmojiMemoryGameView	137
4.4.7	138
4.4.8	140
4.4.9	140
4.4.10	141
4.4.11	141
4.4.12	141
4.4.13	141
4.4.14	141
4.4.15	142

4.4.16.....	142
4.4.17.....	142
5 Lec05. ViewBuilder + Shape + ViewModifier.....	141
5.1.1.....	141
5.1.2.....	141
6 Lec06. Animation.....	143
6.1.1.....	143
6.1.2.....	143

72. The 5 Step Approach to solve any programming problem

1 Lec01. Course Logistics and intro to SwiftUI

Stanford's CS193P-2020 "Developing Applications for iOS using SwiftUI".
Web site <https://cs193p.sites.stanford.edu/>

The Xcode development environment is used to demonstrate the basics of SwiftUI's declarative interface for composing user-interfaces.

1.1 Create a new SwiftUI project

Single View Application | Memorize

1.2 ContentView.swift

19:07 **ContentView.swift**

- This is a Swift file.

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

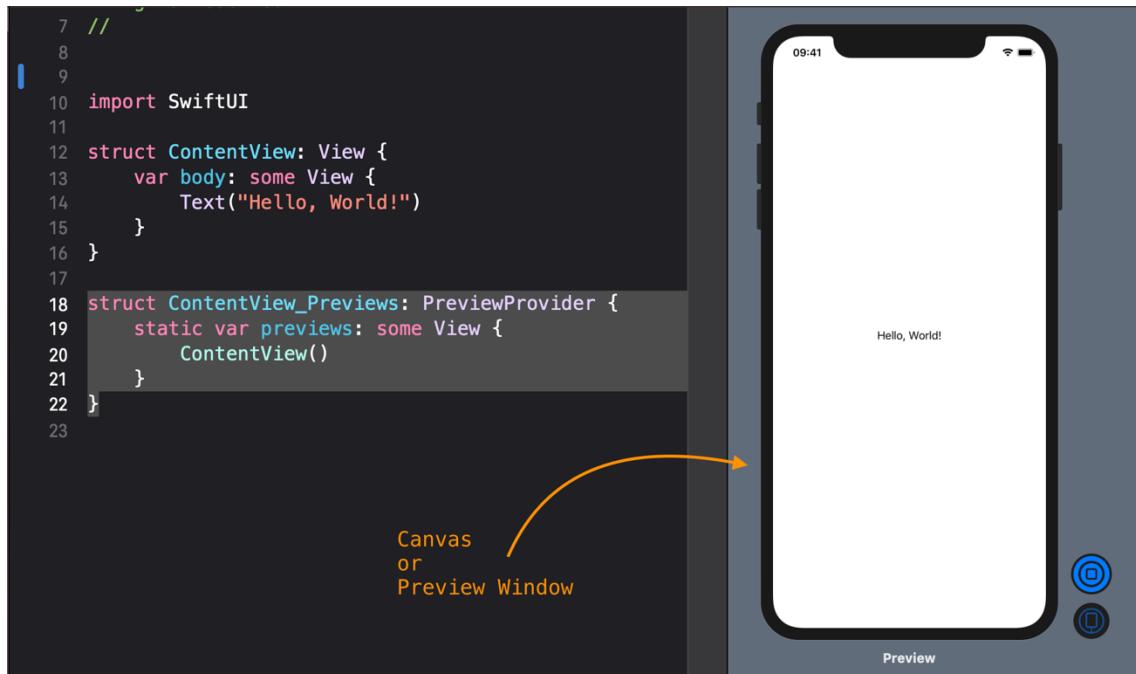
```
struct ContentView_Previews: PreviewProvider {
```

- It provides some glue between the code that you're writing and the **Canvas** or **Preview Window**.

1.2.1 Canvas

We can see what's happening in our UI, kind of in real time by hitting the **Resume** button.

- What it does is compile our code and essentially run it right here.



1.2.2 import SwiftUI

21:10 **import SwiftUI**

- This is a **package** in Swift to deal with UI.
- Sometimes we're going to be writing code that is not UI stuff
 - in that case we won't import **SwiftUI**.
 - We'll be importing a different package called **Foundation**.
 - **Foundation** is kind of **arrays** and **dictionaries** and **strings**.
 - The **SwiftUI** package depends on **Foundation**,
 - so, if you imported **SwiftUI** you get that one too automatically.

1.2.3 Struct

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}
```

struct ContentView

- Declaration of a **struct**.
 - A **struct** in Swift is a container for **variables**, it also has behavior through **functions**.
- This **struct** is called **ContentView**.

1.2.4 struct ContentView: View

```
struct ContentView: View {
```

- **View** is a very interesting little part of this **struct**'s declaration.
 - It essentially means that this **struct** is going to behave like a **view**.
 - Or some might say it's going to function like a **view**.
 - Or some would even say it "**is a**" **view**.
 - Although if I use the statement that it "is a" view, some people will think that's object-oriented programming like the superclass.
 - But, it's not. **This is functional programming**.
 - And that's why we may be more likely to say something like **ContentView functions like a view or it behaves like a view**.
- A view is just a rectangular area on screen.

1.2.5 Text

```
Text("Hello, World!")
```

- **Text** also behaves like a **view**.

1.2.6 var body: some View {

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}
```

var

- Swift **variables** have the keyword **var**.
 - Inside a **struct** like this **var**, we call them **properties**.

body

- It's the name of the property

:some View {

- It's the **type** of the **body** property.

:some

- This is kind of an interesting type.
- What this essentially means is that the type of this property is **any type**, any struct, as long as it behaves like a view.
 - So, if you want to behave like a view, you have to have a property called **body** that also is another thing that behaves like a view.

You could kind of think of views as legos

- Of course there are some brick legos that are like the basic thing like text,
 - but then you can put legos together to make something else, and that thing that you put them together,
 - you could call that a new lego.
 - It's like a combine lego or a more powerful lego.
- For example, if you were building a lego house, you probably would make little lego furniture, a lego couch or lego kitchen table.
 - If you consider that couch and the kitchen table as legos, then you're putting those together to make the house, you could even then have the house be a lego that you put together to make a lego neighborhood,
 - or the lego world, lego universe.
 - That's the way we want to think of making these views.
 - The difference between a lego and a view is that ***there're special kinds of views that are used to combine legos.***
 - We've got these basic brick lego like ***text***
 - then we've got ***other kinds of views that are combiners, view combiners.***
- Our body returns one single some view of some sort.
 - It might be returning a combiner.
 - If it returns a combiner view, then it could have tons and tons of views inside that are combined.

{ ... }

- The value of the property is not stored in memory,
 - instead this var is ***computed***.
 - So every time someone, namely the system, asks for the value of this property,
 - the code inside the curly braces gets executed.
 - Whatever the value that's returned, that's what the value of the ***body*** is.

1.2.7 return Text("Hello There, World!")

```
return Text("Hello There, World!")
```

- This code seems kind of minimalistic, but it will probably make more sense if I put the `return` keyword.
 - It wasn't there because Swift loves to leave things out and make it so you have to type as little code as possible.

```
Text("Hello There, World!")
```

- If you have a **one-line function** that returns a value, **then you can leave this `return` out**.
 - It will just **infer** that.

```
return Text("Hello There, World!")
```

- If you're saying, it's returning text, What's going on here?
 - Well, of course text behaves like a view, so `Text` is "some view".
 - In other words, somewhere off in Apple's code there is some line that looks like this:

```
struct Text: View {  
}
```

- and that's why we can return a text as the value of body because it's some view.

In fact, we could even just type `Text` instead of `some View`,

```
struct ContentView: View {  
    var body: Text {  
        Text("Hello, World!")  
    }  
}
```

- that would be valid because to behaves like a view your `var body` just has to return something that is a view, and of course, text is a view.
 - So, **why don't we type `Text`?**
 - Why do we do this `some View`?
 - As our body of our view gets more complicated aww start using these view-combiners, it's going to be constantly changing which kind of view we're returning.
 - Right now we're returning a text, but eventually we're going to be returning view-combining views and we want let the compiler figure it out for us.
 - So this `some View` is basically saying to the compiler,
 - "Go look in my code right here,
 - figure out what it's returning,
 - make sure that it behaves like a view,
 - and then use that as the type of the body."

1.3 Build a UI that looks a little bit more like a card.

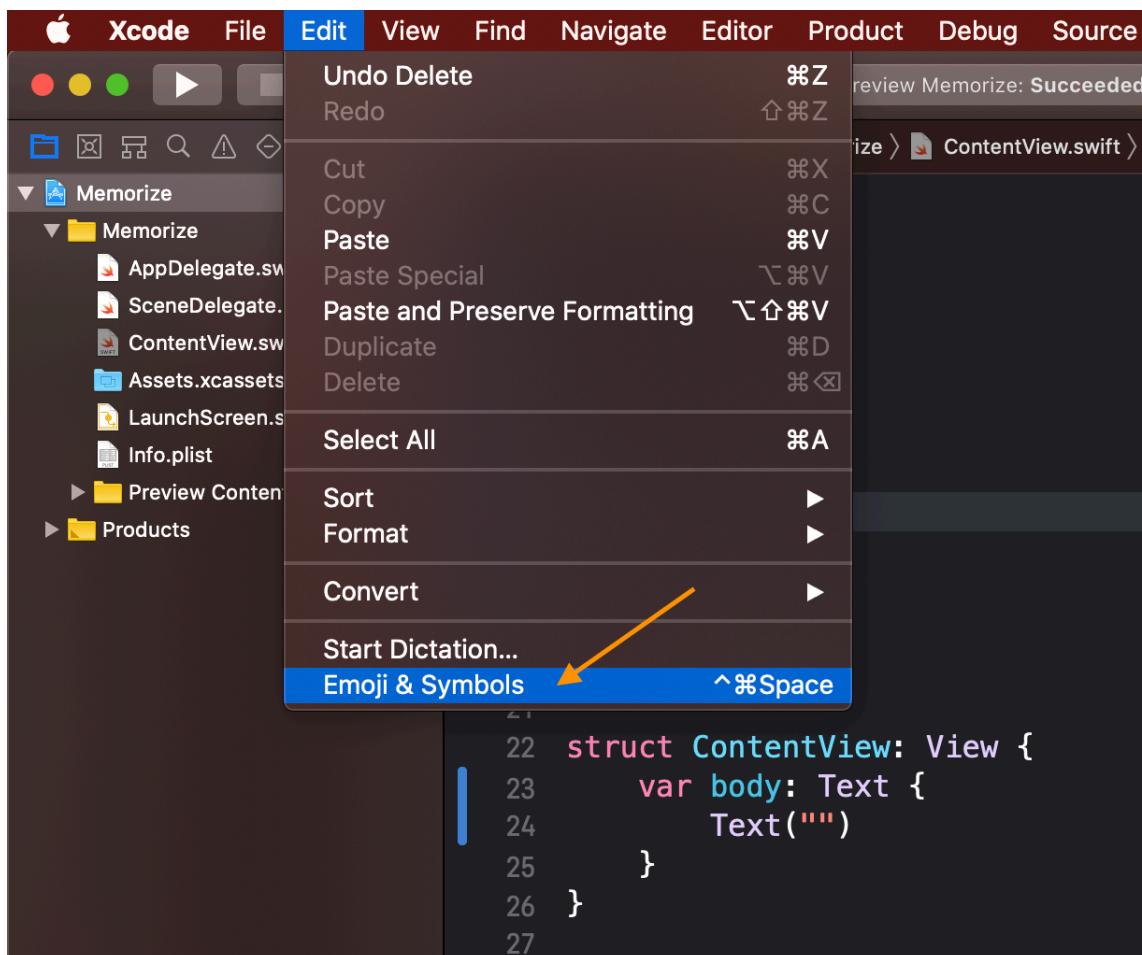
32:40 Let's remind ourselves what that's going to look like:



1.3.1 Emoji

That's going to be really easy, because an emoji is just a piece of text.

- Edit | *Emoji & Symbols*





```
struct ContentView: View {  
    var body: Text {  
        Text("👻")  
    }  
}
```

1.3.2 Rounded rectangle

```
9  
10  
11 import SwiftUI  
12  
13 struct ContentView: View {  
14     var body: some View {  
15         // return Text("👻")  
16         return RoundedRectangle(cornerRadius: 10.0)  
17     }  
18 }  
19  
20  
21  
22  
23 struct ContentView_Previews: PreviewProvider {  
24     static var previews: some View {  
25         ContentView()  
26     }  
27 }  
28 |
```



1.3.3 Combine the rounded rectangle with the text showing the ghost emoji

37:31 I'm going to combine the rounded rectangle with the text showing the ghost emoji returning `some View`.

- The `view` I'm going to return is a `ZStack`.
 - `ZStack` is just a struct that behaves like a `view`, just like `RoundedRectangle`, so is `Text` and `ContentView`.

```
struct ContentView: View {  
    var body: some View {  
        return ZStack {  
            RoundedRectangle(cornerRadius: 10.0)  
            Text("👻")  
        }  
    }  
}
```

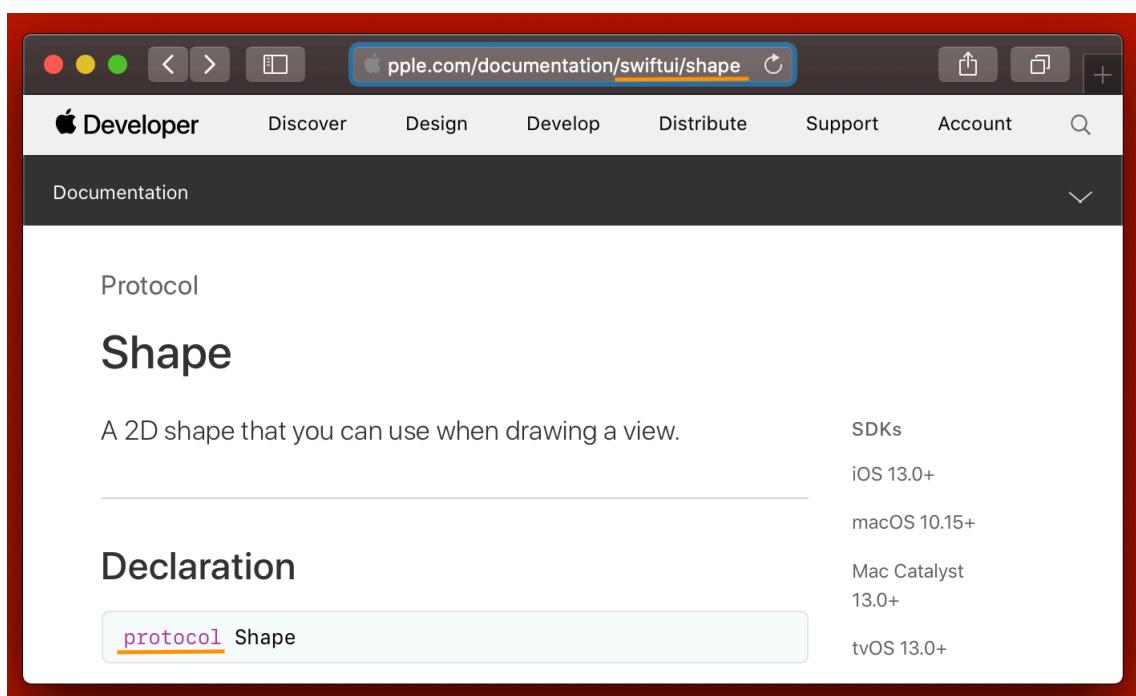
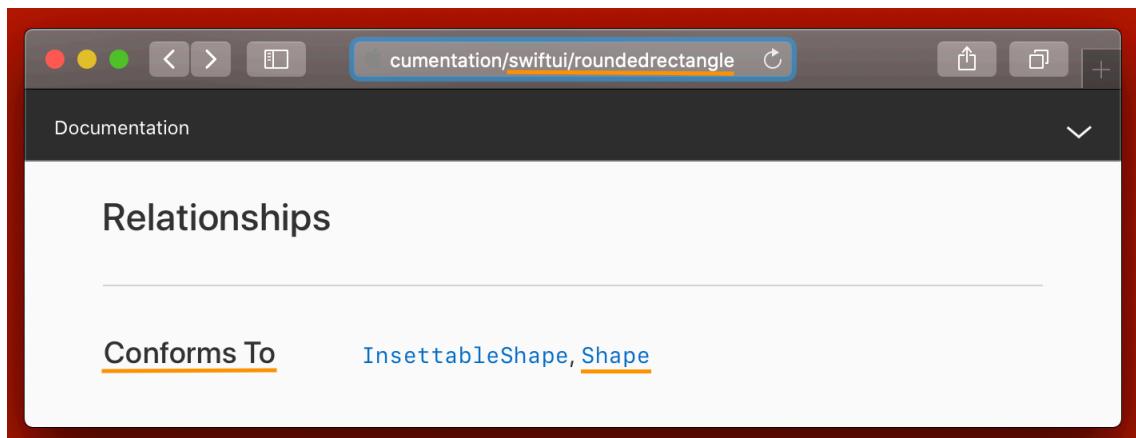
1.3.4 Views, Shapes and .stroke()

39:33 We call a function on `RoundRectangle` called `stroke()`.

```
struct ContentView: View {  
    var body: some View {  
        return ZStack {  
            RoundedRectangle(cornerRadius: 10.0).stroke()  
            Text("👻")  
        }  
    }  
}
```

`RoundedRectangle` behaves like a `View` and also as a `Shape`.

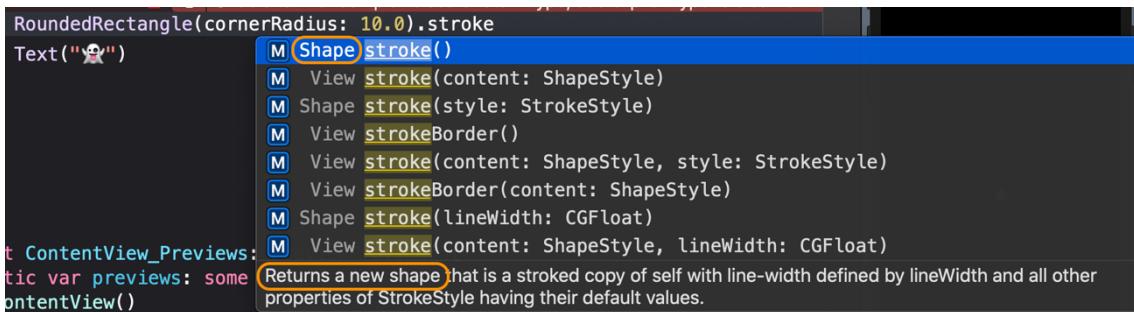
- Other `shapes` are `Circle` or `Capsule`.



The screenshot shows a web browser window with the URL apple.com/documentation/swiftui/shape. The page title is "Relationships". Below it, under "Inherits From", it lists "Animatable, View".

Shapes can all be stroked by calling this function on them called `stroke()`.

- `stroke()` is an interesting function that returns a `Shape`.



DECLARATION

```
func stroke(style: StrokeStyle) -> some Shape
```

This pattern of calling a function on a view or a shape to return another view is very common.

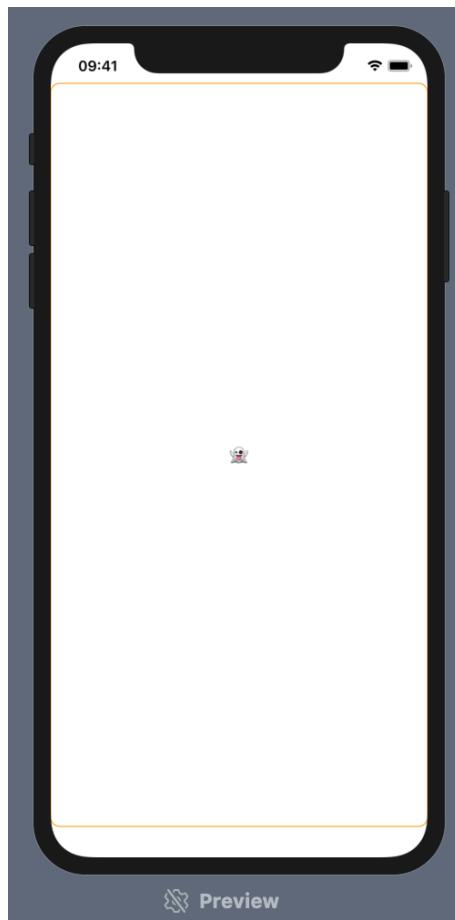
```
RoundedRectangle(cornerRadius: 10.0).stroke()
```

- `stroke()` actually can take arguments but it works perfectly fine without its arguments.
- Its default behavior consist of stroking a one-point line around the edges of the `shape`.

1.3.5 Orange foreground color

41:17

```
struct ContentView: View {  
    var body: some View {  
        return ZStack {  
            RoundedRectangle(cornerRadius: 10.0).stroke()  
            Text("👻")  
        }  
        .foregroundColor(Color.orange)  
    }  
}
```



1.3.6 Padding

44:24 We can do padding with another view function called `padding()`.

```
struct ContentView: View {  
    var body: some View {  
        return ZStack {  
            RoundedRectangle(cornerRadius: 10.0).stroke()  
            Text("👻")  
        }  
        .padding()  
        .foregroundColor(Color.orange)  
    }  
}
```

Notice the subtle difference between `padding()` and `foregroundColor(_)`

- `padding()` pads the `ZStack`
 - The entire `Zstack` gets some padding around it and everything in the `ZStack` is drawn in that little padded area.
- Whereas `foregroundColor(_)` doesn't really make sense for a `ZStack`.
 - A `ZStack` doesn't have a color. All it does is it contains other views,
 - so it gets passed down into the environment for all the views on the inside of it

1.3.7 Fill

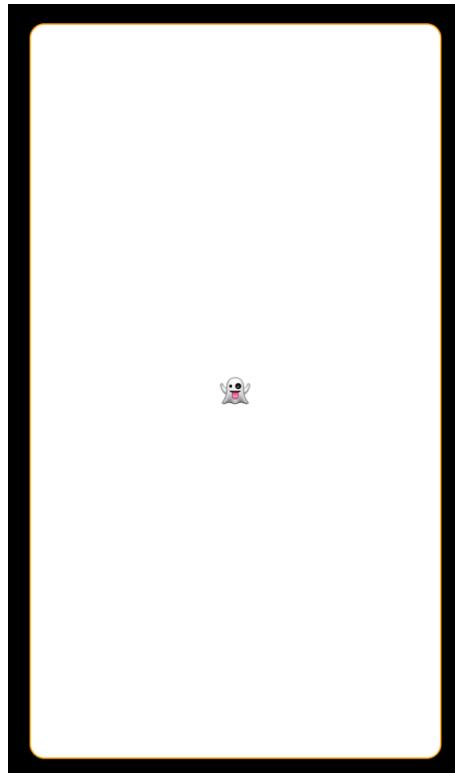
Unfortunately, there's no function you can send to a RoundedRectangle that says stroke it with one color and then fill it with another color.

- But that's no problem because we're in a **ZStack** right now.
 - I'm just going to create another *rounded rectangle*, and this one I will fill instead of stroke.

```
struct ContentView: View {  
    var body: some View {  
        return ZStack {  
            RoundedRectangle(cornerRadius: 10.0).fill(Color.white)  
            RoundedRectangle(cornerRadius: 10.0).stroke()  
            Text("👻")  
        }  
        .padding()  
        .foregroundColor(Color.orange)  
    }  
}
```

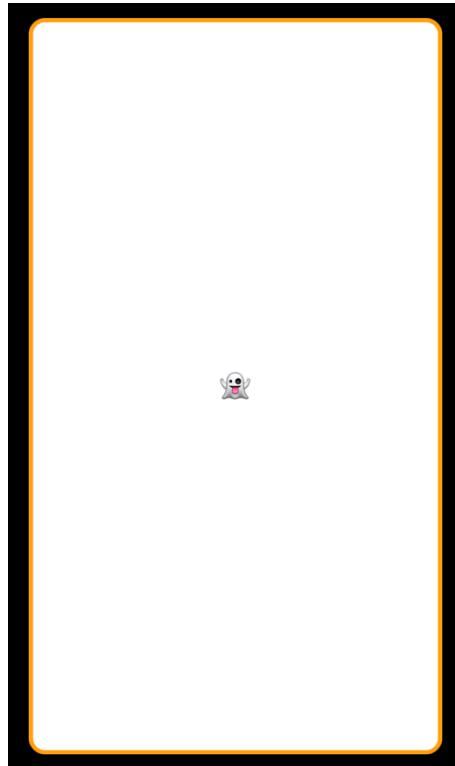
`RoundedRectangle(cornerRadius: 10.0).fill(Color.white)`

- The color white overrides the `foregroundColor(Color.orange)`

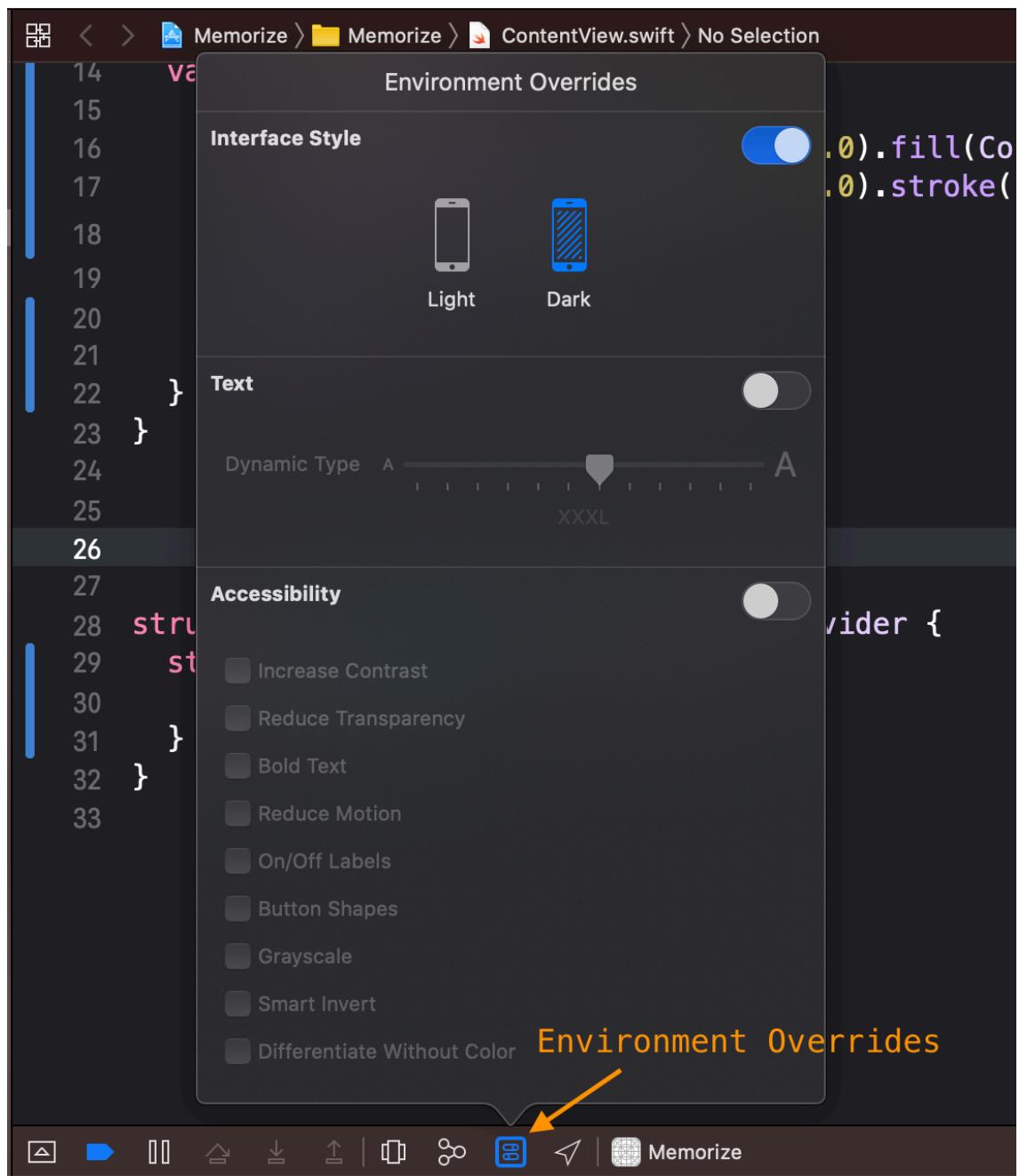


1.3.8 stroke(lineWidth: 3)

```
struct ContentView: View {
    var body: some View {
        return ZStack {
            RoundedRectangle(cornerRadius: 10.0).fill(Color.white)
            RoundedRectangle(cornerRadius: 10.0).stroke(lineWidth: 3)
            Text("👻")
        }
        .padding()
        .foregroundColor(Color.orange)
    }
}
```



1.3.9 Dark mode in simulator



1.3.10 font(Font.largeTitle)

47:44

```
struct ContentView: View {
    var body: some View {
        return ZStack {
            RoundedRectangle(cornerRadius: 10.0).fill(Color.white)
            RoundedRectangle(cornerRadius: 10.0).stroke(lineWidth: 3)
            Text("👻").font(Font.largeTitle)
        }
        .padding()
        .foregroundColor(Color.orange)
    }
}
```

The font can actually be put on the `ZStack`

- it will set the font environment for all texts.
 - So if I had multiple Texts inside the ZStack, they would all take the font environment for all of them.

```
struct ContentView: View {
    var body: some View {
        return ZStack {
            RoundedRectangle(cornerRadius: 10.0).fill(Color.white)
            RoundedRectangle(cornerRadius: 10.0).stroke(lineWidth: 3)
            Text("👻") //.font(Font.largeTitle)
        }
        .padding()
        .foregroundColor(Color.orange)
        .font(Font.largeTitle)
    }
}
```

1.4 Multiple cards

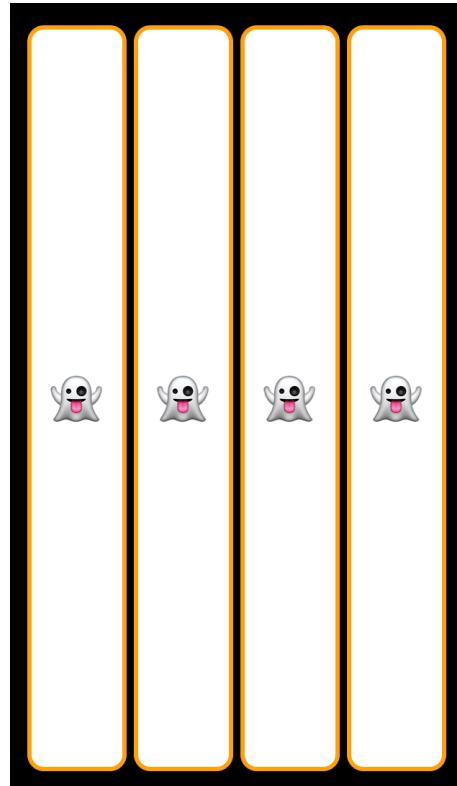
48:54 Now I want multiple cards because I only have one card here.

- I'm going to do that by returning a different kind of view,
 - another combining view called **ForEach**.

DECLARATION

```
Struct ForEach<Data,  
           ID,  
           Content> where Data : RandomAccessCollection,  
           ID : Hashable
```

```
struct ContentView: View {  
    var body: some View {  
        HStack {  
            ForEach(0..<4) { index in  
                ZStack {  
                    RoundedRectangle(cornerRadius: 10.0).fill(Color.white)  
                    RoundedRectangle(cornerRadius: 10.0).stroke(lineWidth: 3)  
                    Text("👻")  
                }  
            } // ForEach  
        } // HStack  
        .padding()  
        .foregroundColor(Color.orange)  
        .font(Font.largeTitle)  
    } // body  
} // ContentView
```



1.4.1 #TBC

54:47 There's another thing we can take away.

1.4.2

56:29 Wouldn't it be cool if I just create a new struct called `CardView` which also behaves like a view?

```
struct ContentView: View {
    var body: some View {
        HStack {
            ForEach(0..<4) { index in
                CardView()
            }
        } // HStack
        .padding()
        .foregroundColor(Color.orange)
        .font(Font.largeTitle)
    } // body
} // ContentView

struct CardView: View {
    var body: some View {
        ZStack {
            RoundedRectangle(cornerRadius: 10.0).fill(Color.white)
            RoundedRectangle(cornerRadius: 10.0).stroke(lineWidth: 3)
            Text("Hello")
        } // ZStack
    } // body
} // CardView
```

This is a way we can use the views to factor out our code to do **encapsulation**,

- to make our code look even simpler and nicer.

1.4.3 Fronts and backs of cards

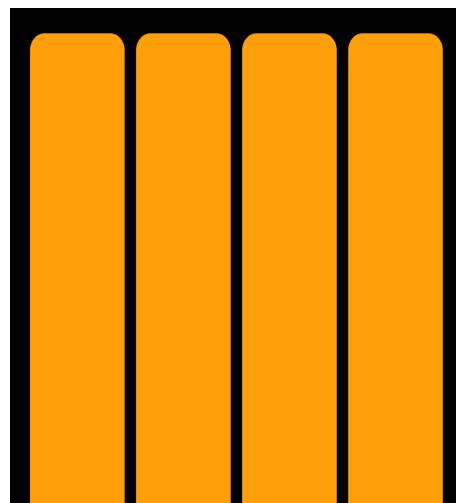
Right now, we only have the front of the card shown.

```
struct ContentView: View {
    var body: some View {
        HStack {
            ForEach(0..<4) { index in
                CardView()
            }
        } // HStack
        .padding()
        .foregroundColor(Color.orange)
        .font(Font.largeTitle)
    } // body
} // ContentView

struct CardView: View {
    var body: some View {
        ZStack {
//            RoundedRectangle(cornerRadius: 10.0).fill(Color.white)
//            RoundedRectangle(cornerRadius: 10.0).stroke(lineWidth: 3)
//            Text("👻")
            RoundedRectangle(cornerRadius: 10.0).fill() // This line is highlighted in red
        } // ZStack
    } // body
} // CardView
```

Of course, we don't want to fill with color white.

- We just want to **fill with whatever our environment color is.**
 - which is going to get from `.foregroundColor(Color.orange)`
 - notice that this *foreground color* not only applies to everything in the `HStack`
 - `ForEach` passes it on down,
 - so that it applies to the `CardView`,
 - which passes it on down,
 - which applies to the `ZStack`,
 - which passes it on down,
 - so that it applies to all these things.



1.4.4 Conditional fronts and backs of cards

But of course, we want to have fronts of cards and backs of cards somehow be conditional.

- The card is either face up or face down.

```
struct ContentView: View {
    var body: some View {
        HStack {
            ForEach(0..<4) { index in
                CardView(isFaceUp: false)
            }
        } // HStack
        .padding()
        .foregroundColor(Color.orange)
        .font(Font.largeTitle)
    } // body
} // ContentView

struct CardView: View {
    var isFaceUp: Bool

    var body: some View {
        ZStack {
            if isFaceUp {
                RoundedRectangle(cornerRadius: 10.0).fill(Color.white)
                RoundedRectangle(cornerRadius: 10.0).stroke(lineWidth: 3)
                Text("👻")
            } else {
                RoundedRectangle(cornerRadius: 10.0).fill()
            }
        } // ZStack
    } // body
} // CardView
```

In Swift all variables are strongly typed, and all variables have to have an initial value.

2 Lec02. MVVM and the Swift Type System

The series of video lectures given to Stanford University students in Spring of 2020 continues with a conceptual overview of the architectural paradigm underlying the development of applications for iOS: **MVVM**.

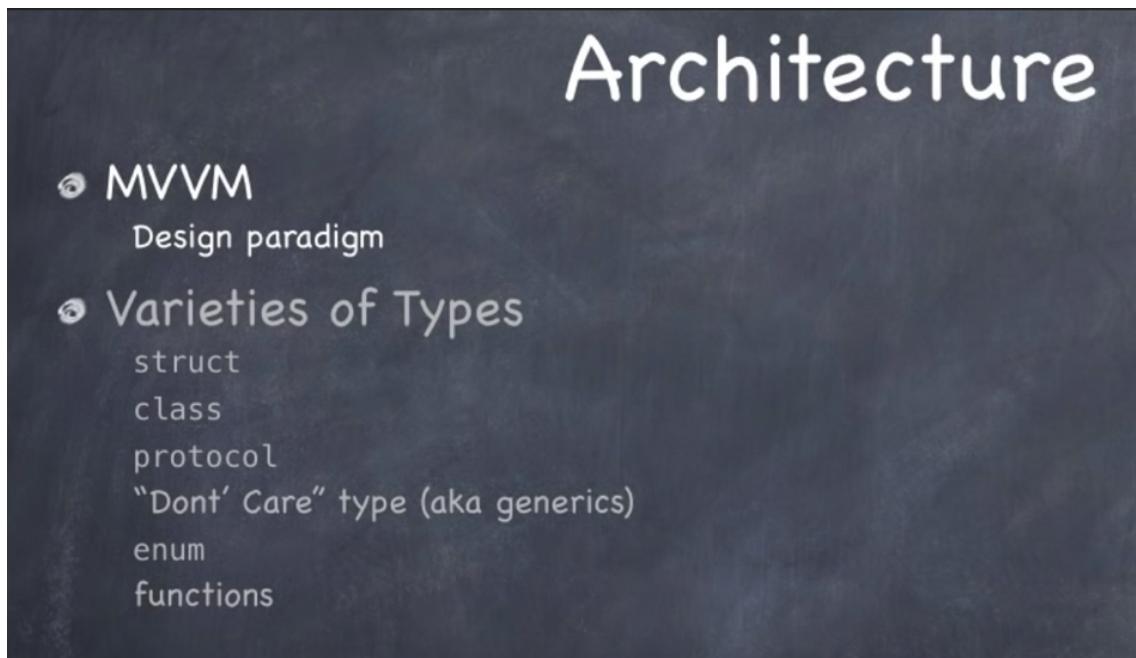
- In addition a key concept in the Swift Programming Language, its type system, is explained. The Memorize demonstration continues, incorporating **MVVM**.

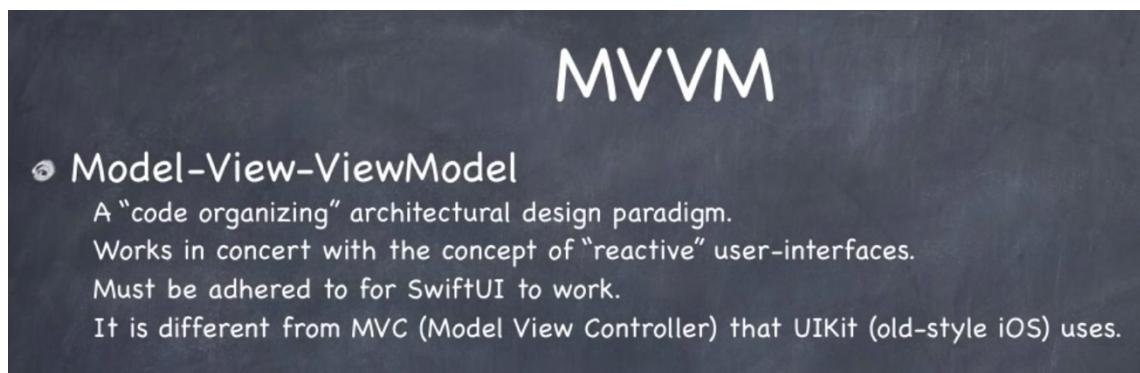
It is impossible to develop applications for iOS using SwiftUI without using the **MVVM** architecture for organizing your code.

- This lecture explains what that is and then demonstrates how it works in our demonstration application.
- SwiftUI development happens entirely in the programming language Swift.
 - Swift is unique in its support of most modern language features, including both **object-oriented programming** and **functional programming**.
 - Since **functional programming** is new to most Stanford students, this lecture starts the process of explaining how it works by covering the basics of Swift's type system, including **structs** and **classes, generics** and **functions as types**.

The demonstration then moves to the next level using the **MVVM architecture** (including creating a **Model**, a **ViewModel**, expressing **user's intent from the View**) and utilizing Swift features like **generics** and **functions as types**.

After this lecture, students take over the development of Memorize for their first assignment.





A “code organizing” architectural design paradigm

- Basically, a place to determine where all your code lives in your app.

MVVM shares a lot with **MVC** in that we’re trying to separate the **Model** which is our backed-end of our app, with the **View**.



Let’s talk about the Model and the View first and then we’ll talk about how MVVM hooks them up together.



UI Independent

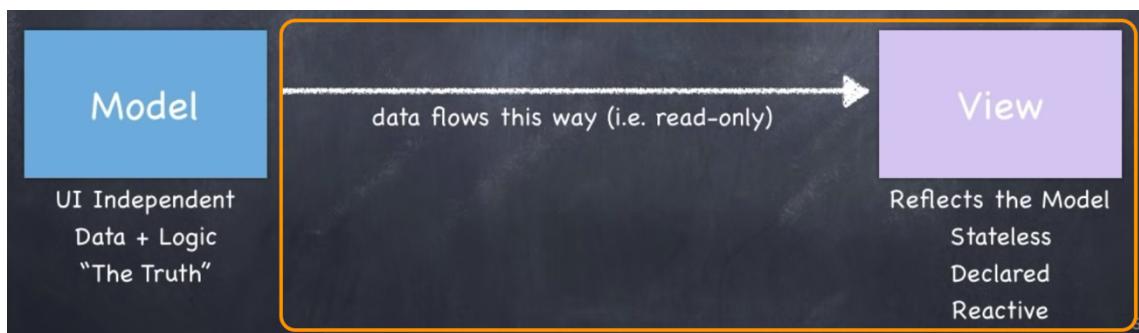
- The Model doesn't import **SwiftUI**, for example.

Data + Logic

- Encapsulate the data and the logic about what your application does.
 - In the case of our card matching game,
 - the data
 - are the cards
 - and the logic
 - what happens when I choose a card,
 - how do I match,
 - how many points do I get when I match,
 - what happens if I have a mismatch?
- All of that logic and the card data lives in the Model.

"The truth"

- We're never going to store that data somewhere else
 - and don't have two different versions of it.



Reflects the Model

---- > data flows this way (i.e. read-only)

- The data is always flowing from the Model to the View.
- We're always going to try and make our View look just like our Model.
 - However our view draws what's in the Model, it's always going to reflect the state of the game.

Stateless

- Because all the state about the game is in the Model.
 - The View itself doesn't need to have any state.

Declared

- **Declarative** means we're just going to declare that the View looks this way and we're only going to actually change anything on screen when the Model changes.
- In our code, notice we don't call functions to put things in places, we just create *rounded rectangles*, *texts* and *stacks*, and place them where we want in the UI.
 - The only functions we call are **modifiers**,
 - they change the look of things, and do it right in place.
- That's different than the old way of doing iOS apps and also a lot of other systems that have been around for years, which we would call **imperative**.

why is the imperative model so bad for UI?

The main reason has to do with **time**.

- Functions are called over time.
 - Put the button here and then later, we're going to arrange this over here and then later, something's going to happen here.
 - So if you want to understand how your UI is built, you need this other dimension of **time** to know when this function can be called and what function call depends on some other call happening to change your UI at any time so you have to be always on guard and ready for that.
 - That is a nightmare to manage and almost impossible to prove that your UI really works because you can't call every function in every possible order.
- Whereas **declarative**, you can always look at what you've declared for your UI and see this is what this thing does.
 - At **anytime**, it's really **time independent**.
 - It also localizes all the code.
 - All the code to draw your UI all right in front of you.

That code we wrote yesterday, that was it, that was the entire code for showing the UI of our cards and there wasn't some code some other where else that was going to call a function on this and mess it up.

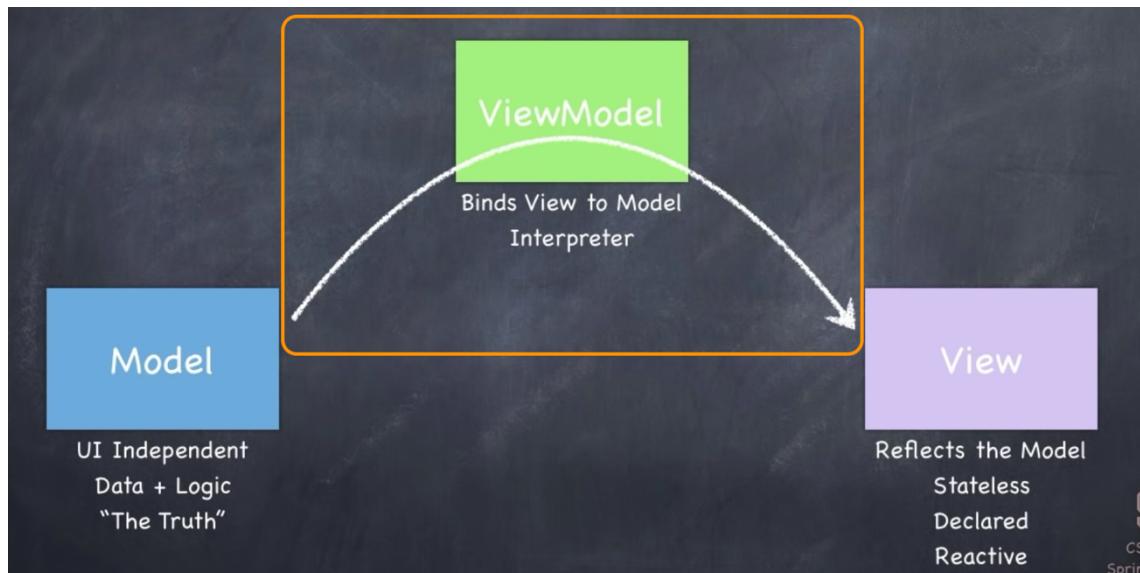
- Because our views are structs that are actually read-only by default,
 - no one is allowed to call a function that would change it.
 - It's not even possible to do. So that way, you can be sure that this view is always going to look like exactly what you see in the code.

Reactive

- The View reacts to changes in the Model.
 - Anytime the Model changes, it's going to automatically update the View, because I told you the View is stateless.
- At any time, you should be able to say make it look like the Model.

2.1.3 MVVM - ViewModel

07:30

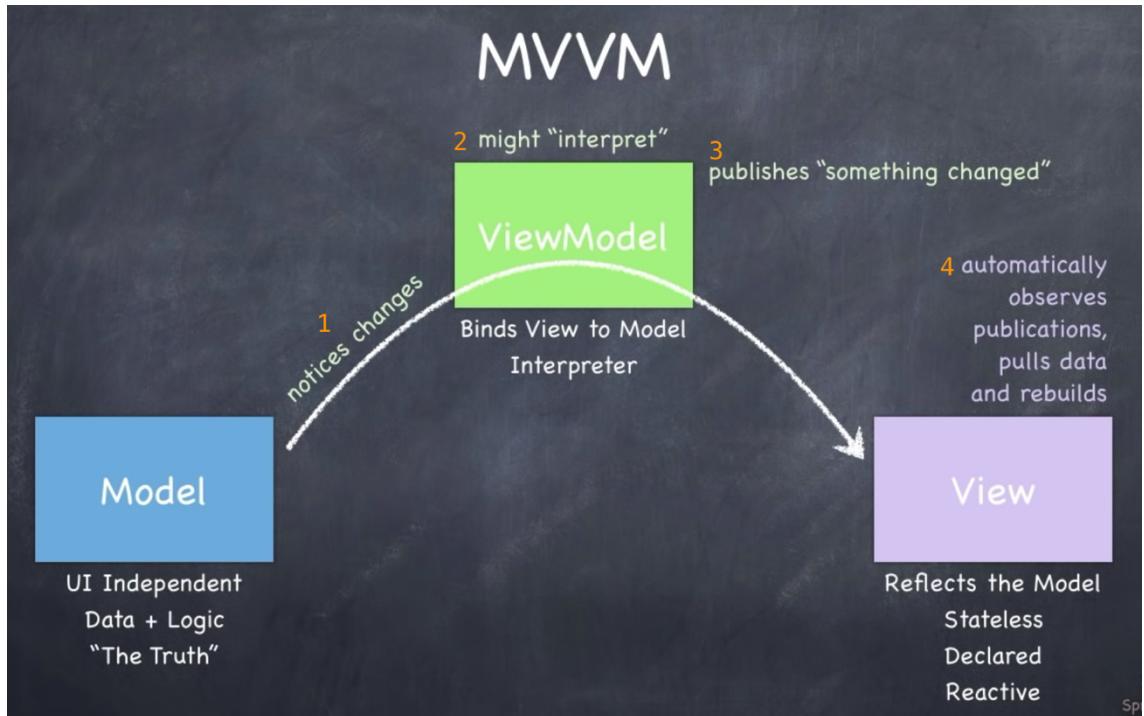


Binds View to Model

- The ViewModel's job is to **bind the View to the Model**.
 - When the Model changes, the View get reflected.

Interpreter

- Along the way, as it does that binding between the Model and the View, it might be **interpreting the Model for the View**
 - because we want the View to be very simple since we're writing it in a declarative way, we don't want it to have a lot of code in there that's like converting from one data type to another and things like that.
 - So, **we're going to ask the ViewModel to do that**.
- Our Model in our game that we're writing here, you could imagine that it is a SQL database or it's something over the network where you're making HTTP requests.
 - It can be quite complicated over there and your ViewModel can simplify that, boil it down into maybe some simpler data structures that it can pass to the View that will let the View be simple code that draws it.
- We're going to interpose the ViewModel between the Model and the View.



1. --- > notices changes

- ViewModel is always trying to notice changes in the Model
 - If your Model is a struct, it's actually quite easy.
 - Since structs are copied around when they are passed to functions, Swift knows when a struct has changed.
 - It can track when a struct changes .
 - But if the Model were a SQL database, when it changes, the ViewModel gets notified.

2. might "interpret"

- When that data changes, it might convert it to some other format.

3. publishes "something changed"

- Afterwards what it does is publishing something changed to anybody who is interested.
 - It doesn't actually have any pointers to any views.
 - **The ViewModel does not talk directly to its views.**

4. automatically observes publications, pulls data and rebuilds.

- In fact, the View **subscribes** to that **publication**, and when it sees that something has changed, it goes back to the ViewModel and asks,
 - what's the current state of the world?
 - I'm going to draw myself to match that state of the world.
 - The reason it doesn't go directly to the Model, is because the ViewModel might be doing this interpreting or it might be protecting the Model from some nefarious View that could do something bad to the Model.

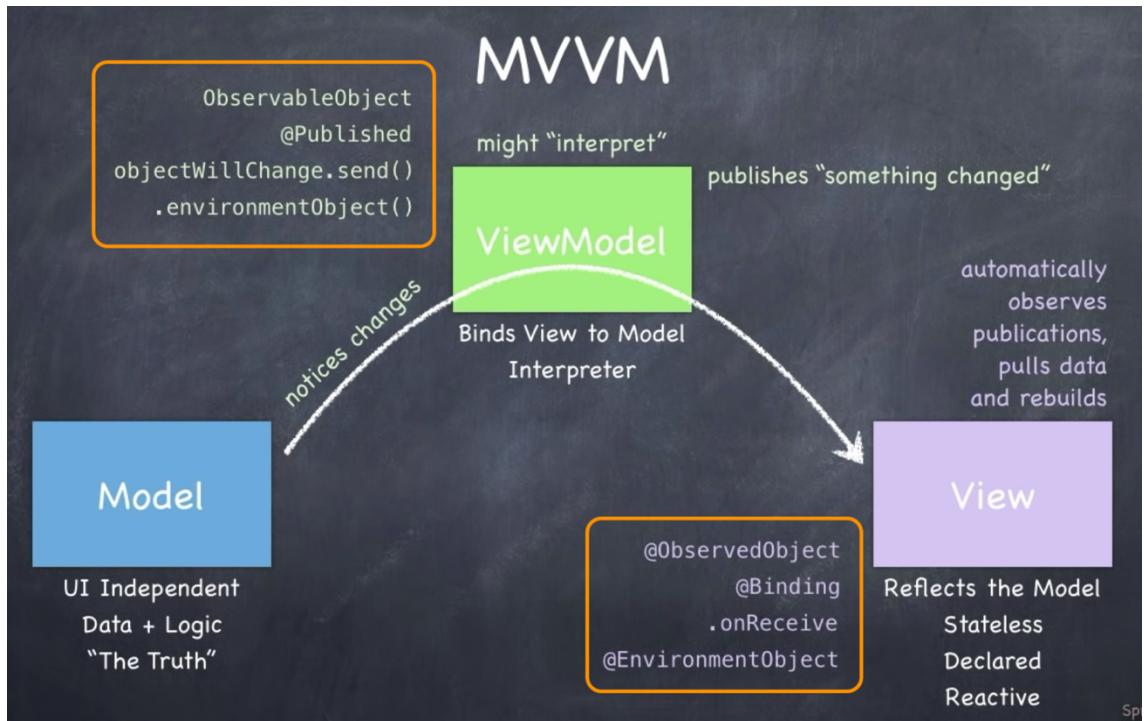
SUMMARY

ViewModel notices changes in the Model and anytime something has changed, then the Views just observe those changes and then it pulls the data from the ViewModel and redraws itself.

2.1.5 MVVM – Swift syntax

11:12 As the quarter goes on, we're going to see the Swift syntax for making this all work.

- I've put some of the syntax up here like ObservableObjects and onRecieve, objectWillchange and so on.
- Actually, we're going to see it right at the end of the demo today.

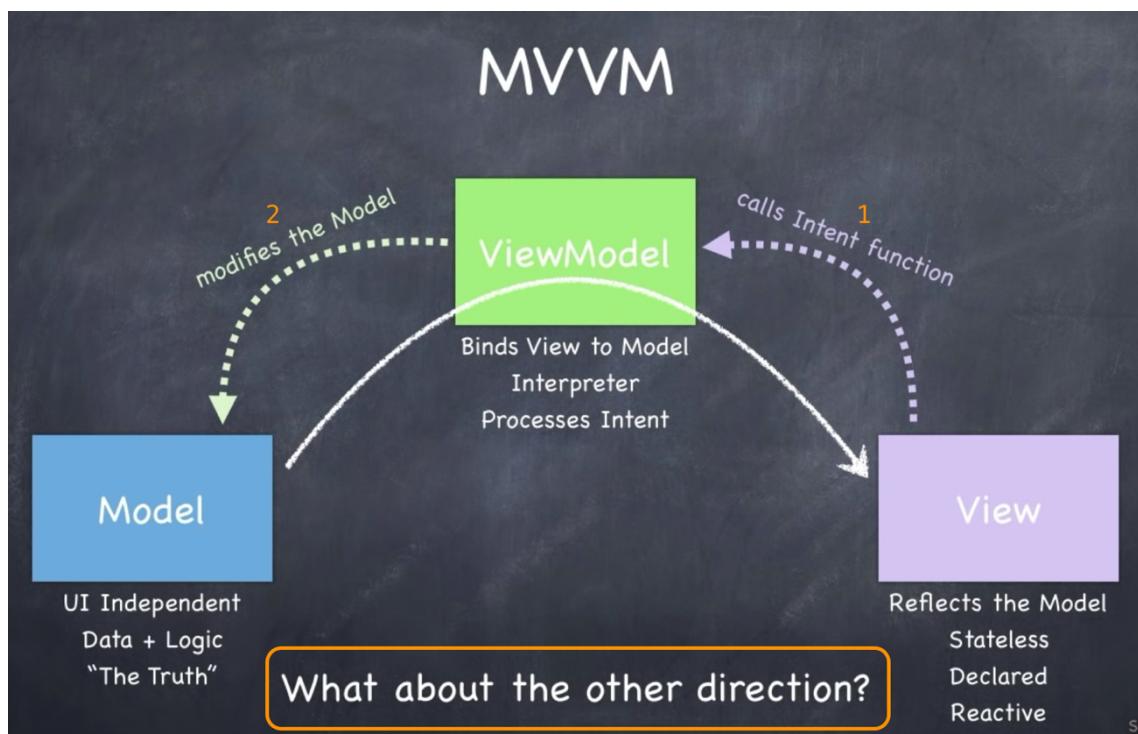


2.1.6 MVVM – What if the View wants to change the Model?

11:39 We add another responsibility for the ViewModel which is to **process the user's intent**.

There's another somewhat related architecture called **Model-View_Intent**,

- which makes even more clear that when the user wants to do something, they go through this intent.
- Apple's iOS SwiftUI design does not implement an intent system,
 - so I'm just going to talk about intent as a concept here.
- An intent is some user intent.
 - A classic example here in our memory game
 - the user is going to have the **intent of choosing a card**.
 - It's up to the ViewModel to process these intents and it does this by making functions available to the View to call to make the intent clear.

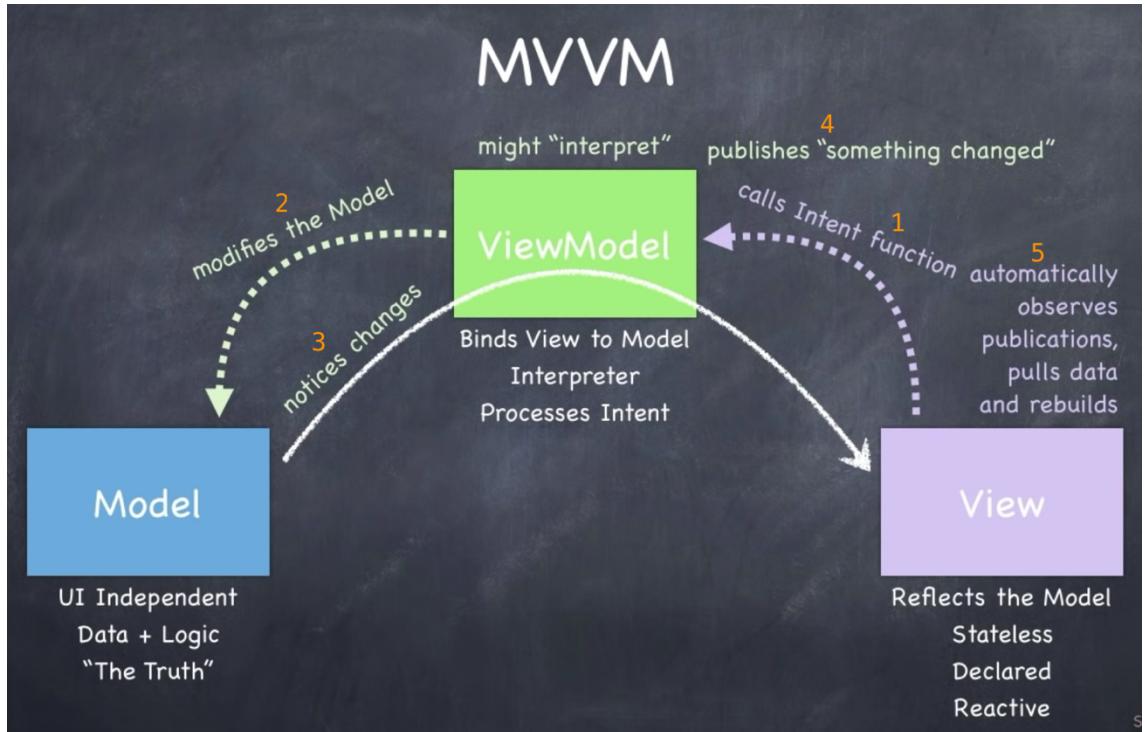


1. --- > calls Intent function

- Whenever a gesture happens, the View is going to call an intent function in the ViewModel.
 - It's just a documentation thing in our ViewModel's code

2. --- > modifies the Model

- When the ViewModel receives a function like this called on them, they're going to **modify the Model** and again,
 - the ViewModel knows all about the Model and how it's represented,
 - if it's SQL, it's going to be issuing SQL commands to change the Model,
 - if it's a struct, then maybe it's just setting vars or calling functions in the Model to modify.
 - You can do whatever makes sense to express that user's intent in changing the Model.



3. --- > notices changes

- Now the Model is changing. What happens next? Well, the exact intent we talked about before.
 - The ViewModel notices the change that it just made,

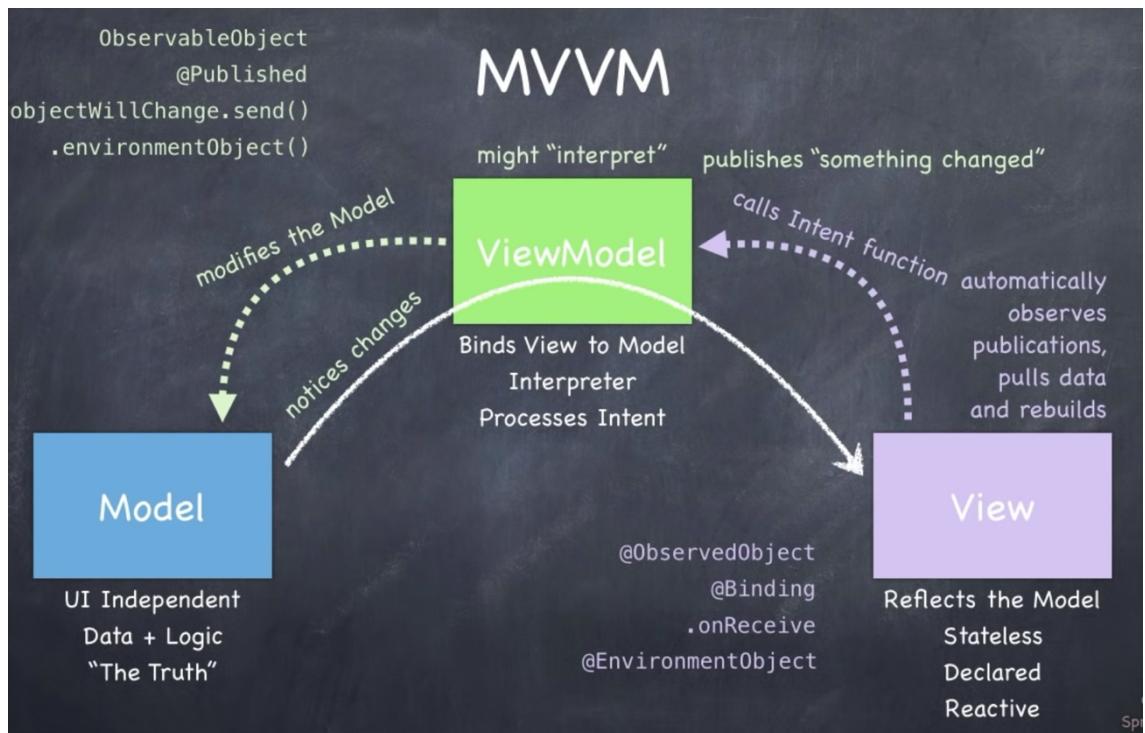
4. publishes “something changed”

- it publishes something changed

5. automatically observes publications, pulls data and rebuilds

- and then the View sees that something changed and it automatically redraws itself.

This whole picture is the MVVM architecture with its Swift keywords.



The key to all this is just understanding each of these three thing's roles because they're going to be very clearly defined in the code.

Architecture

• MVVM

Design paradigm

• Varieties of Types

`struct`

`class`

`protocol`

“Dont’ Care” type (aka generics)

`enum`

functions

2.2.1 Structs and classes – Stored vars, computed vars

struct and class

• Both `struct` and `class` have ...

... pretty much exactly the same syntax.

stored `vars` (the kind you are used to, i.e., stored in memory)

computed `vars` (i.e. those whose value is the result of evaluating some code)

```
var body: some View {  
    return Text("Hello World")  
}
```

- They also both can have **computed variables** like we saw from the demo last time.
- `body` is a **computed variable**.
 - Its value is computed each time

• Both struct and class have ...

... pretty much exactly the same syntax.

stored vars (the kind you are used to, i.e., stored in memory)

computed vars (i.e. those whose value is the result of evaluating some code)

constant lets (i.e. vars whose values never change)

functions

```
func multiply(operand: Int, by: Int) -> Int {
    return operand * by
}

multiply(operand: 5, by: 6)

func multiply(_ operand: Int, by otherOperand: Int) -> Int {
    return operand * otherOperand
}

multiply(5, by: 6)
```

The arguments have labels, actually each parameter can have two labels.

- The function `multiply(...)` has two labels.
 - The first parameter has the label **under bar** (_) and the label `operand`
 - the second one has the label `by` and the label `otherOperand`.
 - See how there're two, a blue one and a purple one for each of the arguments, and why are there two?
 - The blue ones are used by callers of the function
 - the purple ones are used inside the function.
 - The **under bar** (_) label, means no label.

18:40

• Both struct and class have ...

... pretty much exactly the same syntax.

stored vars (the kind you are used to, i.e., stored in memory)

computed vars (i.e. those whose value is the result of evaluating some code)

constant lets (i.e. vars whose values never change)

functions

initializers (i.e. special functions that are called when creating a struct or class)

```
struct MemoryGame {
    init(numberOfPairsOfCards: Int) {
        // create a game with that many pairs of cards
    }
}
```

Put an `init(...)` function inside the struct which takes that as an argument.

- I can have any number of these initializers as I want, each of which taking a different argument.

struct	class
Value type	Reference type
Copied when passed or assigned	Passed around via pointers
Copy on write	Automatically reference counted
Functional programming	Object-oriented programming
No inheritance	Inheritance (single)
"Free" <code>init</code> initializes ALL vars	"Free" <code>init</code> initializes NO vars
Mutability must be explicitly stated	Always mutable
Your "go to" data structure	Used in specific circumstances
Everything you've seen so far is a struct (except <code>View</code> which is a protocol)	The ViewModel in MVVM is always a class (also, UIKit (old style iOS) is class-based)

Passed around via pointers

- Reference types live in the heap.
- When you create an instance of a class, the storage for them is in the heap.
 - That's just like stored in memory and when I pass it around, I'm passing around pointers to it.
 - So a lot of people might have a pointer to the same instance somewhere.

Copied when passed or assigned

- Structs are not passed around by pointer, they are copied.

Automatically reference counted

- A class, on the other hand, you're passing pointers to it, so instead is **counting the references**.
 - Seeing how many pointers there are to this thing, and it happens automatically.
 - when finally, no one is left pointing to the instance class in the heap, then the memory gets freed up out of the heap.
 - So that's called **automatic reference counting**.

Most things that you see are structs.

- Arrays, dictionaries, ints, bools, doubles

Functional programming

- Structs are kind of basically built to support a kind of programming called **functional programming**.
 - *Functional programming* focuses on the functionality of things

Object-oriented programming

- OOP focuses on encapsulating the data and the functionality into some container, an object.

Generics

• Sometimes we just don't care

We may want to manipulate data structures that we are “type agnostic” about.
 In other words, we don’t know what type something is and we don’t care.
 But Swift is a strongly-typed language, so we can’t have variables and such that are “untyped.”
 So how do we specify the type of something when we don’t care what type it is?
 We use a “don’t care” type (we call this feature “generics”) ...

• Example of a user of a “don’t care” type: Array

Awesome example of generics: `Array`.
 An `Array` contains a bunch of things and it doesn’t care at all what type they are!
 But inside `Array`’s code, it has to have variables for the things it contains. They need types.
 And it needs types for the arguments to `Array` functions that do things like adding items to it.
 Enter ... GENERICS.

• How Array uses a “don’t care” type

`Array`’s declaration looks something like this ...

```
struct Array<Element> {
    ...
    func append(_ element: Element) { ... }
}
```

The type of the argument to `append` is `Element`. A “don’t care” type.
`Array`’s implementation of `append` knows nothing about that argument and it does not care.
`Element` is not any known struct or class or protocol, it’s just a placeholder for a type.

The code for using an `Array` looks something like this ...

```
var a = Array<Int>()
a.append(5)
a.append(22)
```

When someone uses `Array`, that’s when `Element` gets determined (by `Array<Int>`).

Note that `Array` has to let the world know the names of all of its “don’t care” types in its API.
 It does this with the `< >` notation on its struct declaration `Array<Element>` above.
 That’s how users of `Array` know that they have to say what type `Element` actually is.
`var a = Array<Int>()`
 It is perfectly legal to have multiple “don’t care” types in the above (e.g. `<Element, Foo>`)

• Type Parameter

I will often refer to these types like `Element` in `Array` as a “don’t care” type.
 But its actual name is Type Parameter.
 Other languages most of you may know (e.g. Java) have a similar feature.
 However, Swift combines this with protocols to take it all to the next level ...

Functions as Types

• Functions are people* too! (* er, types)

You can declare a variable (or parameter to a func or whatever) to be of type "function".

The syntax for this includes the types of the arguments and return value.

You can do this anywhere any other type is allowed.

Examples ...

```
(Int, Int) -> Bool // takes two Ints and returns a Bool
(Double) -> Void // takes a Double and returns nothing
() -> Array<String> // takes no arguments and returns an Array of Strings
() -> Void // takes no arguments and returns nothing (this is a common one)
```

All of the above are just types. No different than Bool or View or Array<Int>. All are types.

```
var foo: (Double) -> Void // foo's type: "function that takes a Double, returns nothing"
func doSomething(what: () -> Bool) // what's type: "function, takes nothing, returns Bool"
```

Example ...

```
var operation: (Double) -> Double
```

This is a var called `operation`.

It is of type "function that takes a Double and returns a Double".

Here's a simple function that takes a Double and returns a Double ...

```
func square(operand: Double) -> Double {
    return operand * operand
}
```

```
operation = square // just assigning a value to the operation var, nothing more
```

```
let result1 = operation(4) // result1 would equal 16
```

Note that we don't use argument labels (e.g. `operand:`) when executing function types.

```
operation = sqrt // sqrt is a built-in function which happens to take and return a Double
let result2 = operation(4) // result2 would be 2
```

• Closures

It's so common to pass functions around that we are very often "Inlining" them.

We call such an inlined function a "closure" and there's special language support for it.

We'll cover this in the demo and again later in the quarter.

Remember that we are mostly doing "functional programming" in SwiftUI.

As the very name implies, "functions as types" is a very important concept in Swift. Very.

2.3 Back to the demo

36:40

Back to the Demo

• MVVM and Types in Action

Now that we know about MVVM, let's implement it in our Memorize application

In doing so, we'll see a lot of what we just talked about ...

We're going to use the special `init` function (in both our Model and our ViewModel)

We're going to use generics in our implementation of our Model

We're going to use a function as a type in our Model

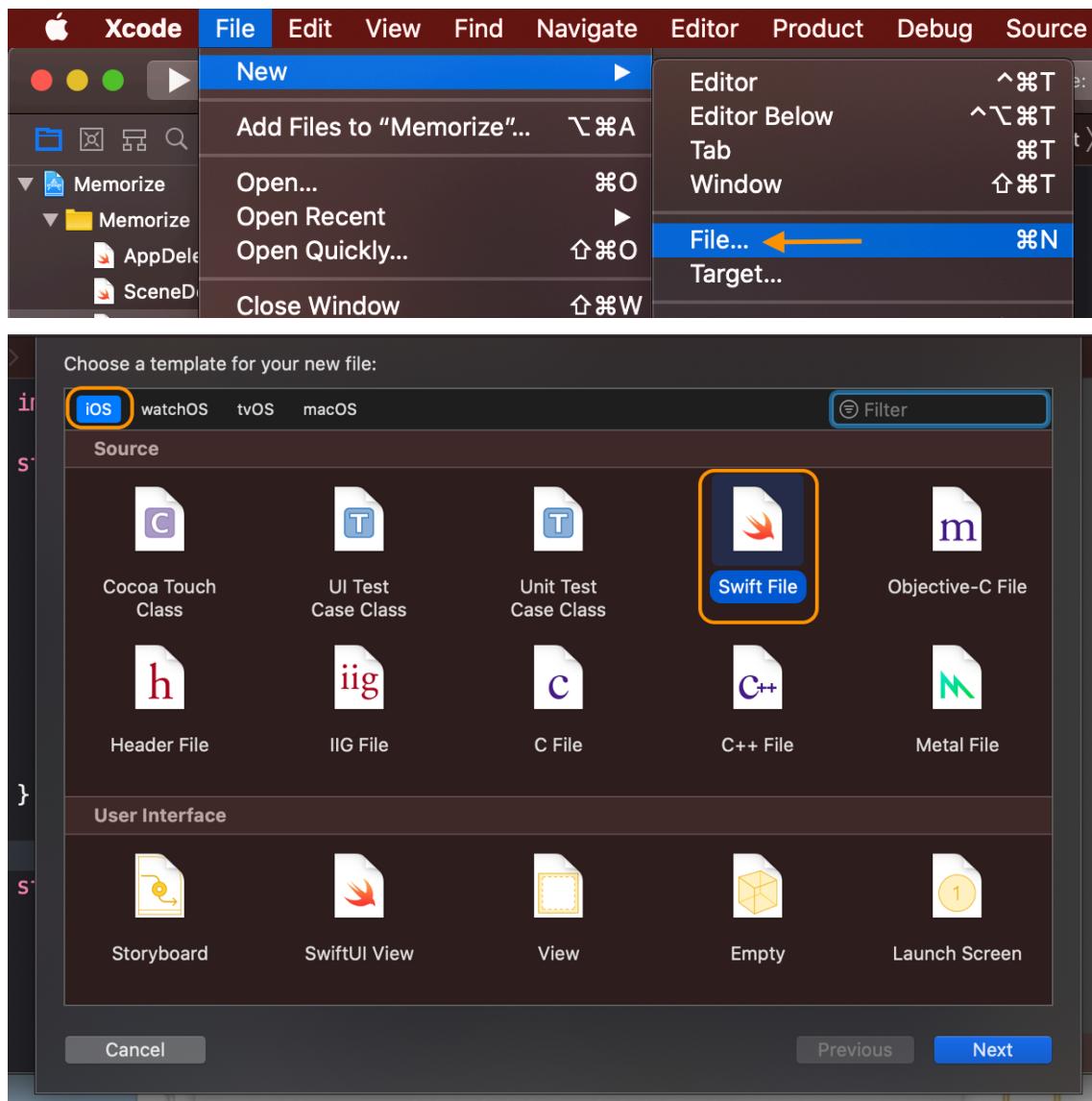
We're going to see a `class` for the first time (our ViewModel will be a class)

We're going to implement an "Intent" in our MVVM

And finally, we will make our UI "reactive" through our MVVM design

Whew! Let's get started ...

2.3.1 Model – Create MemoryGame.swift



Save as: **MemoryGame**

2.3.2 Model – MemoryGame.Card struct

When I create a struct that's going to represent my Model I'm always asking myself, what does this Model do?

- I put the vars and functions in place that could really describe what this thing does.
 - So, when I think of a memory game, the most important thing I'm thinking about is it should have come cards.

```
import Foundation

struct MemoryGame {
    var cards: Array<Card>

    struct Card {
    }
}
```

struct Card {

- Notice that I put the `Card` struct inside the `MemoryGame` struct.
 - The full name is actually `MemoryGame.Card`.
 - Nesting structs inside structs, it's mostly a naming, a name spacing thing
 - so that we know that this is not a playing card or some other kind of random card, it is a memory game's card.
 - Has some other slight benefits that you'll see along the way.

2.3.3 Model – choose(card:) function

41:22 Our memory game also needs a way to choose a Card.

- So, you're going to see here your first definition of a Swift function.

```
struct MemoryGame {
    var cards: Array<Card>

    func choose(card: Card) {
        print("card chosen: \(card)")
    }

    struct Card {
    }
}
```

2.3.4 Model – isFaceUp and isMatched variables

44:15 We're going to have to really obviously decide what a card looks like.

- What's important about a card and one thing we know a card has is whether it's face-up or not.
- Also, I'm going to need to know whether a card is matched.

```
struct Card {  
    var isFaceUp: Bool  
    var isMatched: Bool  
}
```

2.3.5 Model – MemoryGame<CardContent>

44:37

- I guess I also need to know its content. What type is this var going to be?
 - I could imagine building a card game with
 - images
 - var content: Image
 - emoji,
 - words,
 - maybe with numbers
 - var content: Int
 - It's almost like we don't really care.
 - We're doing a UI independent game playing where we can put anything we want on the cards.
 - So, I'm going to call this CardContent.
 - That's a type I just made up, my don't care type, and of course, if I do a don't care type, I'm required up here to say MemoryGame<CardContent> to declare to the world that I'm a generic type and I have this don't care that you if you want to use MemoryGame you're going to have to tell me what this is.
 - In our game, once we start using this Model, we're going to say MemoryGame<String> because an emoji is just a character in the string so we're going to say <String> and that's going to define what kind of memory game this is.
 - This is a really awesome simple example of this don't care business because really, this memory game does not care what's on these cards.

```
struct MemoryGame<CardContent> {  
    var cards: Array<Card>  
  
    func choose(card: Card) {  
        print("card chosen: \(card)")  
    }  
  
    struct Card {  
        var isFaceUp: Bool  
        var isMatched: Bool  
        var content: CardContent  
    }  
}
```

2.3.6 ViewModel – Create EmojiMemoryGame.swift

46:27 File | New | **File...**

iOS | **Swift File**

- It's not a SwiftUI View, it is a UI thing but is not an actual View, it's the ViewModel.

Save As: **EmojiMemoryGame**

- It's a specific kind of memory game that happen to use emoji as the thing it draws.

2.3.7 ViewModel - Import Foundation

47:14

```
import Foundation
```

It's importing Foundation, but here I could actually import **SwiftUI** if I want.

- I'm not actually going to do UI in here, I'm going to be doing all my UI in **ContentView.swift**.
 - The **ViewModel** is essentially a UI thing because it knows how this is going to be drawn on screen.
 - That's in fact some of its purpose in life, is to take the UI independent **Model** memory game and translate it to have it displayed in some way.
 - In this case, as an emoji memory game,

2.3.8 ViewModel – Create a property to access the Model

```
import Foundation

class EmojiMemoryGame {
    var model: MemoryGame<String>
}
```

```
var model: MemoryGame
```

- What our ViewModel needs here is some sort of **var** that it can access the Model through.
 - I'm actually calling this **var model**.
 - which you probably wouldn't call any **var model** because that's a concept, but I'm calling it here just for instructor purposes.
 - You'd really probably call this **var** something like **game**, something more descriptive of what it is.

```
var model: MemoryGame<String>
```

- As in our emoji game the contents of the cards are strings, we substitute the type **<CardContent>** for **<String>**.

2.3.9 ViewModel - Why ViewModel should be a class?

```
class EmojiMemoryGame {
```

- The biggest advantage of a class is that it's easy to share because instance of classes lives in the heap and it has pointers to it.
 - It means that all of our views could have pointers to it.
 - When we start building complicated UIs we're going to have lots of views and many of them are going to want to look through these class instances.
- but the classes biggest strength is also its greatest weakness.
 - The problem with lots of different people pointing to the same ViewModel is that if any one of them kind of messes it up, it ruins the party for everybody and especially in these circumstances.
- Here's my analogy: **Imagine that there's a house.**
 - Inside this house live our views.
 - An **instance of EmojiMemoryGame ViewModel is the front door** because, ViewModels are doorways or portals to access the Model.
 - So the **Model is the outside world**.
 - All views share and look the world through the same doorway.
 - They all have pointers to the same doorway.
 - That's a good thing because they're all seeing the outside world in exactly the same way through this same doorway, so our UI is always going to be nice and **self-consistent**.
 - They're all seeing the same thing.
 - There's a big problem with our front door because it's wide open.
 - The variable **var model: MemoryGame<String>**, which any of our views could look at and they could go, for example, find a card in there and they could set it to be **isMatched**, and that could really mess up our game.
 - There are some things we can do to mitigate this problem of all sharing this same instance, but still have the advantage of them all sharing.
 - One is we can close the door making the **model** var **private**.
 - That means that the model var can only be accessed by **EmojiMemoryGame**.
 - Now, none of the views can look out the door.
 - How can we find a middle ground there?
 - One way we can do that is by using a little different private called **private(set)**
 - It's like **the door is closed but it's a glass door**.
 - This means **only EmojiMemoryGame can modify the Model but everyone else can still see the Model**.

```
class EmojiMemoryGame {  
    private(set) var model: MemoryGame<String>  
}
```

- But now nobody can choose any cards either because views can't get through the glass door to choose a card.
 - So that's where **intents** come in
 - One of the ViewModel's jobs is to **interpret user intent**.

2.3.10 ViewModel - Intents

54.52 I'm going to provide functions that allow the views to access the outside world.

- In our analogy, you can imagine there's a high-tech door with like a video doorbell intercom system or something and these views are going to press the intercom button and talk to the outside world and say, please choose this card.
 - Then the ViewModel which is the door, it can obviously talk to the Model directly and tell it to do things.
- These user intents are kind of things that the views would say into the intercom.
 - Things that they want to happen in the game.

```
// MARK: - Intent(s)
```

```
func choose(card: Card){  
}
```

- This is an intent that the user might have to choose a card.

```
func choose(card: MemoryGame<String>.Card){  
    model.choose(card: card)  
}
```

```
func choose(card: MemoryGame<String>.Card){
```

- We have to make sure to give it its full name with all the parts of its type.

```
model.choose(card: card)
```

- Ask the model to choose that card.
 - Keep in mind, our Model might be a SQL database and we should issue a bunch of SQL commands in here to make this kind of the intent by the user come to fruition.

2.3.11 ViewModel – Intents -

56:51

```
class EmojiMemoryGame {
    private(set) var model: MemoryGame<String>

    var cards: Array<MemoryGame<String>.Card> {
        return model.cards
    }

    // MARK: - Intent(s)

    func choose(card: MemoryGame<String>.Card){
        model.choose(card: card)
    }
}
```

```
private(set) var model: MemoryGame<String>
```

- We might even want to be more restrictive.
- For example, we might really want that door to be closed and instead of looking through the glass door, you're going to use the video doorbell's video.
 - You know how video doorbell works, people come to the door and you can see them there on little video screen.

```
var cards: Array<MemoryGame<String>.Card> {
    return model.cards
}
```

- So, the analogy here of a little video screen is that we can provide vars and functions that let people look at the Model in constricted ways.
 - We obviously want people to be able to see the cards in the Model, so maybe I'll create my own var cards.

2.3.12 ViewModel – Access to the Model

57:48

```
class EmojiMemoryGame {
    private var model: MemoryGame<String>

    // MARK: - Access to the Model
    //

    var cards: Array<MemoryGame<String>.Card> {
        return model.cards
    }

    // MARK: - Intent(s)

    func choose(card: MemoryGame<String>.Card) {
        model.choose(card: card)
    }
}
```

`model.cards`

- The ViewModel might be doing some interpretation here,
 - either to try and message the Model's data into some form that is more consumable by the View,
 - or it might actually be having to do some work, like maybe this data, this Model is coming from over the network and it has to be doing some network requests.
 - But if we have a choice between adding some complexity to the ViewModel here to message the data so that view is simpler we're always going to make the trade-off in that direction.
 - We want our View to be as simple as possible, so it's really part of the ViewModel's job to present the Model to the Views in a way that's easily consumable by the Views.
- Of course, this is a one-liner that returns something, so we don't need the `return` keyword.

`var cards: Array<MemoryGame<String>.Card> {`

- I prefer this little more closed non-glass door, kind of approach to things, but occasionally, it makes sense to do that `private(set)` and let people have a glass door and see the Model, but you're always going to want to have the functions that play the role of intents.

`// MARK: - Intent(s)`

- The `intents` functions are almost like *documentation*.
 - It's letting all the Views know, people who are writing View code, *here are the things you can do to change the Model*.

2.3.13 ViewModel – Class EmojiMemoryGame has no initializers

59:48

The screenshot shows the Xcode interface with the file `EmojiMemoryGame.swift` open. A warning is displayed: "Class 'EmojiMemoryGame' has no initializers". A tooltip provides the reason: "1. Stored property 'model' without initial value prevents synthesized initializers".

```
8
9 import Foundation
10
11 class EmojiMemoryGame {           ⓘ Class 'EmojiMemoryGame' has no initializers
12     private var model: MemoryGame<String>
13
14     ⓘ 1. Stored property 'model' without initial value
15     // prevents synthesized initializers
16     // ****
17     // MARK: - Access to the Model
18     //
19
20     var cards: Array<MemoryGame<String>.Card> {
21         model.cards
22     }
```

```
struct MemoryGame<CardContent> {
    var cards: Array<Card> // <- Not good place to initialize the cards
```

- It's really up to the `MemoryGame` to decide which cards are face up or face down.
 - That's why `MemoryGame` has to initialize the cards.
 - But it's a bit of a problem because it doesn't really know for example, how many cards are in the game.

2.3.14 How many cards are in the game? – `numberOfPairsOfCards: 2` – (Closures)

1:01:51 Where is the number of cards going to be communicated from our ViewModel that's trying to create its Model over to the `MemoryGame`?

```
class EmojiMemoryGame {
    private var model = MemoryGame<String>(c
    ⓘ M MemoryGame<String> (cards: Array<MemoryGame<String>.Card>)
```

```
class EmojiMemoryGame {
    private var model = MemoryGame<String>(cards:
        Array<MemoryGame<String>.Card>)
```

```
private var model = MemoryGame<String>(cards:
    Array<MemoryGame<String>.Card>)
```

- The place to know how many cards are in the game is here.
 - Instead of having created a memory game by giving it the cards, be nice if we just create the memory game by saying...

```
class EmojiMemoryGame {
    private var model = MemoryGame<String>(numberOfPairsOfCards: 2)
```

- ... create a memory game with this number of pairs of cards,
 - 2 pairs, 5 pairs or whatever number of cards.
 - Then this memory game would say, okay I will go and create this many pairs of cards and I'll set them all up properly and do all that.
- The bottom line here is that we want to create this memory game with some number of pairs of cards,
 - The way we are going to manage it is with an init.

2.3.15 Closures – init(numberOfPairsOfCards: Int)

1:02:56

```
struct MemoryGame<CardContent> {
    var cards: Array<Card> // <- Not good place to initialize the cards

    func choose(card: Card) {
        print("card chosen: \(card)")
    }

    init(numberOfPairsOfCards: Int) {
        cards = Array<Card>()
    }

    struct Card {
        var isFaceUp: Bool
        var isMatched: Bool
        var content: CardContent
    }
}
```

`init(numberOfPairsOfCards: Int)` {

- You don't have to say `func init`, you can just say `init` because *inits are, by definition, functions.*
- `init needs to initialize all of our vars`
 - because we are not allowed to have a memory game without all of its vars initialized.

`cards = Array<Card>()`

- We create an empty array of cards.

```
struct MemoryGame<CardContent> {

    ...
    init(numberOfPairsOfCards: Int) {
        cards = Array<Card>()

        for pairIndex in 0..
```

`for pairIndex in 0..`

- We need to create these many pairs of cards, and add those to this array.

`Card(isFaceUp: false, isMatched: false, content: CardContent))`

- For struct we don't need any `init` function because we initialize the values through its constructor.

2.3.16 Closures – cardContentFactory: (Int) -> CardContent

I don't really know how to create the content because this content is of type `CardContent`, which for me, is a **don't care**.

- I don't even know what that is. It could be an Image, Int, String, I don't know.
- How could I possibly know how to create one of these things?
- Who does know how to create the content on this card?
 - It is `EmojiMemoryGame` because he is responsible of creating memory games.
 - We give `EmojiMemoryGame` an opportunity to do this little creation of the content in `MemoryGame`.
 - We're going to do that with a function.

```
struct MemoryGame<CardContent> {  
    var cards: Array<Card> // <- Not good place to initialize the cards  
    ...  
    init(numberOfPairsOfCards: Int,  
         cardContentFactory: (Int) -> CardContent) {  
        cards = Array<Card>()  
  
        for pairIndex in 0..            let content = cardContentFactory(pairIndex)  
  
            cards.append(Card(isFaceUp: false,  
                               isMatched: false,  
                               content: Content))  
            cards.append(Card(isFaceUp: false,  
                               isMatched: false,  
                               content: Content))  
        }  
    }  
}
```

`cardContentFactory: (Int) -> CardContent`

- I'm just going to add another argument to my `init`, I'm going to call it `cardContentFactory`.
 - The type of this argument is a **function** that takes an `Int` and returns the `CardContent` type.
 - Functions are first class types in Swift.
 - You can pass functions around.
 - In other languages, passing functions around can be a torture because you have to pass pointers on them and all kinds of crazy things.
 - Here, you just explain the types of the arguments and the return and you can pass a function around.

`let content = cardContentFactory(pairIndex)`

- Every time you have any kind of variable that doesn't actually vary, it's a **constant**.

2.3.17 Closures – createCardContent(pairIndex: Int) -> String

```
12 class EmojiMemoryGame {  
13     private var model = MemoryGame<String>(numberOfPairsOfCards: 2,  
14                                         cardContentFactory: (Int) -> String)  
15 }
```

cardContentFactory: (Int) -> String

- The second argument is a *function* that takes an **Int** and returns a **CardContent** which we know has to be a *string*.

```
func createCardContent(pairIndex: Int) -> String {  
    return "😊"  
}
```

```
class EmojiMemoryGame {  
    private var model = MemoryGame<String>(numberOfPairsOfCards: 2,  
                                             cardContentFactory: createCardContent)
```

```
func createCardContent(pairIndex: Int) -> String {
```

- No errors, no warnings. This is all perfectly legal way to do this.
 - However, we would never do it this way because we don't want to have to be creating these extra little functions to do that. Instead, we would in-line this right here.

2.3.18 Closures - { (pairIndex: Int) -> String in return "😊" }

1:13:57 Watch carefully now because I'm going to go through the process of how we take the `createCardContentfunction(pairIndex:)` function and in-line it over `createCardContent`.

- This in lining a function in Swift is called a **closure** because it actually captures the information from the surroundings that it needs to work

```
class EmojiMemoryGame {  
    private var model = MemoryGame<String>(numberOfPairsOfCards: 2,  
                                             cardContentFactory: (pairIndex: Int) -> String {  
                                                 return "😊"  
                                            })
```

```
(pairIndex: Int) -> String { return "😊" }
```

- We cut the `createCardContentfunction(pairIndex:)` function, except for its name and then we paste it over `createCardContent`

```
class EmojiMemoryGame {  
    private var model = MemoryGame<String>(numberOfPairsOfCards: 2,  
                                             cardContentFactory: { (pairIndex: Int) -> String in  
                                                 return "😊"  
                                            })
```

- This almost works as is, but there's one thing I always have to do when I do this is to take the curly brace `{`, cut it, replace it whit the word `in`, then paste the curly brace at the start.
- Essentially, **the curly braces have to surround the entire inline function**,
 - that's why we move the curly brace out in front and use the word `in` to separate the content of the closure from the arguments of the closure.

2.3.19 Closures - { _ in "😊" }

Type inference is really nice in a language where everything has to be strongly typed.

- Following we are going to reduce our code thanks to type inference.

```
class EmojiMemoryGame {
    private var model = MemoryGame<String>(numberOfPairsOfCards: 2,
                                              cardContentFactory: { (pairIndex: Int) -> String in
                                                return "😊"
                                              })
}

struct MemoryGame<CardContent> {
    var cards: Array<Card> // <- Not a good place to initialize the
                           cards

    func choose(card: Card) {
        print("card chosen: \(card)")
    }

    init(numberOfPairsOfCards: Int,
         cardContentFactory: (Int) -> CardContent) {
        self.numberOfPairsOfCards = numberOfPairsOfCards
        self.cardContentFactory = cardContentFactory
    }
}

class EmojiMemoryGame {
    private var model = MemoryGame<String>(numberOfPairsOfCards: 2,
                                              cardContentFactory: { (pairIndex: Int) -> String in
                                                return "😊"
                                              })
}

(pairIndex: Int) -> String { return "😊" })
```

- Swift knows the type of `cardContentFactory` which is a function that takes an `int` and returns a `CardContent`.
 - So, we don't need to say this is an `Int`, and we don't need to say this return a `string`.

```
class EmojiMemoryGame {
    private var model = MemoryGame<String>(numberOfPairsOfCards: 2,
                                              cardContentFactory: { pairIndex in
                                                return "😊"
                                              })
}
```

```
cardContentFactory: {pairIndex} {
    • We don't even really need the parentheses surrounding pairIndex.
```

```
class EmojiMemoryGame {
    private var model = MemoryGame<String>(numberOfPairsOfCards: 2,
                                              cardContentFactory: { pairIndex in return "😊"})
```



```
return "😊"
```

- We know this is a one-line function that returns and emoji string,
 - so, we don't need the word `return` here.

```
class EmojMemoryGame {  
    private var model =  
        MemoryGame<String>(numberOfPairsOfCards: 2) { pairIndex in "😊" }
```

```
MemoryGame<String>(numberOfPairsOfCards: 2) { pairIndex in "😊" }
```

- Even more, we know that if you have a curly brace thing, that is the last argument (*trailing closure*) , we can do the same thing we did with the last argument for `ForEach` or `HStack`, which is to *get rid of the label of the argument* and put the curly brace floating outside

```
class EmojMemoryGame {  
    private var model =  
        MemoryGame<String>(numberOfPairsOfCards: 2) { _ in "😊" }
```

- Even more than that, notice that since we're just always returning a smiley face, we don't really even need this `pairIndex` here, but you can't delete it, you have to mark it with an underbar just to say yeah, I know this is supposed to take an argument but I don't need it.

2.3.20 Closures – Return a different emoji for each pair of cards.

1:18:44 We don't want to have every pair of cards have the smiley face

```
class EmojiMemoryGame {  
    private var model: MemoryGame<String> = createMemoryGame()  
  
    func createMemoryGame() -> MemoryGame<String> {  
        let emojis: Array<String> = ["👻", "🎃"]  
  
        return MemoryGame<String>(numberOfPairsOfCards: 2) { pairIndex in  
            return emojis[pairIndex]  
        }  
    }  
  
    let emojis: Array<String> = ["👻", "🎃"]
```

- This is a constant array.

```
return emojis[pairIndex]
```

- My little card factory is just going to return emojis

ERROR:

Cannot use instance member 'CreateMemoryGame' within property initializer; property initializers run before 'self' is available.

```
class EmojiMemoryGame {  
    private var model: MemoryGame<String> = createMemoryGame()  
        ! Cannot use instance member 'createMemoryGame'  
        within property initializer; property initializers run before  
        'self' is available  
  
    func createMemoryGame() -> MemoryGame<String> {  
        let emojis: Array<String> = ["👻", "🎃"]  
  
        return MemoryGame<String>(numberOfPairsOfCards: 2) {  
            pairIndex in  
            return emojis[pairIndex]  
        }  
    }
```

I've told you that we cannot in Swift have any variable that's not initialized to something.

- What's even more restrictive than that is that **we cannot use any functions on our class or struct until all of these are initialized.**

```
class EmojiMemoryGame {
    private var model: MemoryGame<String> =
        EmojiMemoryGame.createMemoryGame()

    static func createMemoryGame() -> MemoryGame<String> {
        let emojis: Array<String> = ["🎃", "🧙‍♀️"]

        return MemoryGame<String>(numberOfPairsOfCards: 2) { pairIndex in
            return emojis[pairIndex]
        }
    }
}
```

```
static func createMemoryGame() -> MemoryGame<String> {
```

- I want to use the `createMemoryGame()` function on my instance to create a memory game but I can't until the `model` property is initialized.
- This is now a static utility function to create our memory game.

2.3.21 View – var card: MemoryGame<String>.Card

1:26:41 Let's go back to our View and use our ViewModel.

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        HStack {
            ForEach(0..<4) { index in
                CardView(isFaceUp: false)
            }
        } // HStack
        .padding()
        .foregroundColor(Color.orange)
        .font(Font.largeTitle)
    } // body
} // ContentView

struct CardView: View {
    var isFaceUp: Bool

    var body: some View {
        ZStack {
            if isFaceUp {
                RoundedRectangle(cornerRadius: 10.0).fill(Color.white)
                RoundedRectangle(cornerRadius: 10.0).stroke(lineWidth: 3)
                    .text("👻")
            } else {
                RoundedRectangle(cornerRadius: 10.0).fill()
            }
        } // ZStack
    } // body
} // CardView

var isFaceUp: Bool
```

- Currently, our CardView has the `isFaceUp` property but really, it should be getting that `isFaceUp` from the card that it's viewing.
 - So, I'm going to change this var from `isFaceUp` to be a card.

```
struct CardView: View {
    var card: MemoryGame<String>.Card
```

```
import SwiftUI

struct ContentView: View {

    var body: some View {
        HStack {
            ForEach(0..<4) { index in
                CardView(card: ...)
            }
        } // HStack
        .padding()
        .foregroundColor(Color.orange)
        .font(Font.largeTitle)
    } // body
} // ContentView

struct CardView: View {
    var card: MemoryGame<String>.Card

    var body: some View {
        ZStack {
            if card.isFaceUp {
                RoundedRectangle(cornerRadius: 10.0).fill(Color.white)
                RoundedRectangle(cornerRadius: 10.0).stroke(lineWidth: 3)
                Text(card.content)
            } else {
                RoundedRectangle(cornerRadius: 10.0).fill()
            }
        } // ZStack
    } // body
} // CardView
```

`CardView(card: ...)`

- It's not going to take in `isFaceUp` anymore, it wants to take some sort of Card.

2.3.22 SceneDelegate – let contentView = ContentView(viewModel: game)

We're going to have to find some way to provide it with a card.

- We're going to get those cards through our ViewModel.

```
import SwiftUI

struct ContentView: View {
    var viewModel: EmojiMemoryGame

    var body: some View {
        HStack {
            ForEach(0..<4) { index in
                CardView(card: ...)
            }
        } // HStack
        .padding()
        .foregroundColor(Color.orange)
        .font(Font.largeTitle)
    } // body
} // ContentView

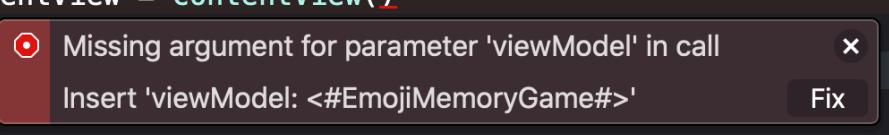
struct CardView: View {
    var card: MemoryGame<String>.Card

    var body: some View {
        ZStack {
            if card.isFaceUp {
                RoundedRectangle(cornerRadius: 10.0).fill(Color.white)
                RoundedRectangle(cornerRadius: 10.0).stroke(lineWidth: 3)
                Text(card.content)
            } else {
                RoundedRectangle(cornerRadius: 10.0).fill()
            }
        } // ZStack
    } // body
} // CardView

var viewModel: EmojiMemoryGame
```

- How are we going to create an instance of `EmojiMemoryGame`?
 - In wherever this content view is being created.
 - The place is in `SceneDelegate.swift`.

```
// Create the SwiftUI view that provides the window contents.
let contentView = ContentView()
```



A screenshot of Xcode showing a code completion tooltip. The tooltip is a red box containing the text "Missing argument for parameter 'viewModel' in call" with a red circle icon, "Insert 'viewModel: <#EmojiMemoryGame#>'", and a "Fix" button. The background shows the Swift code for creating the SwiftUI view.

```
import UIKit
import SwiftUI

class SceneDelegate: UIResponder, UIWindowSceneDelegate {

    var window: UIWindow?

    func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options connectionOptions: UIScene.ConnectionOptions) {

        // Create the SwiftUI view that provides the window contents.
        let game = EmojiMemoryGame()
        let contentView = ContentView(viewModel: game)

        // Use a UIHostingController as window root view controller.
        if let windowScene = scene as? UIWindowScene {
            let window = UIWindow(windowScene: windowScene)
            window.rootViewController = UIHostingController(rootView: contentView)
            self.window = window
            window.makeKeyAndVisible()
        }
    }

    struct ContentView_Previews: PreviewProvider {
        static var previews: some View {
            ContentView(viewModel: EmojiMemoryGame())
        }
    }
}
```

1:31:11 How do we use the `viewModel` property to get the cards that we're going to show?

```
struct ContentView: View {
    var viewModel: EmojiMemoryGame

    var body: some View {
        HStack {
            ForEach(0..<4) { index in
                CardView(card: ...)
            }
        } // HStack
        .padding()
        .foregroundColor(Color.orange)
        .font(Font.largeTitle)
    } // body
} // ContentView
```

`ForEach(0..<4) { index in`

- Right now, we're using a range that shows 4 cards.

I told you this could be any iterable thing.

- How about if we just make this to be our ViewModel's cards?

```
var body: some View {
    HStack {
        ForEach(viewModel.cards) { index in
            CardView(card: ...)
        }
    } // HStack
    .padding()
    .foregroundColor(Color.orange)
    .font(Font.largeTitle)
} // body
} // ContentView
```

I kinda mislead you a little when I said this could be any iterable thing.

- It actually is any *iterable* thing where the things that's iterating over are what is called **Identifiable**.
 - If they're not a range of int, they have to be **identifiable**.
 - Why do these have to be **identifiable**?
 - For example, let's say you want to do animation, and let's say these cards are moving around.
 - Moving into a different order or something like that.
 - This **ForEach** needs to be able to identify which card is which so that the view it creates for each card can keep that view in sync with the cards.

Right now if we look at MemoryGame the Card struct, it's not identifiable, there's no way to identify them.

- In fact, right now, they're all the same.
 - Two cards that match would be the same because they have the same content

```
struct Card {  
    var isFaceUp: Bool  
    var isMatched: Bool  
    var content: CardContent  
}
```

Swift has a formalism, a formal mechanism for identifying something, making something identifiable, and it does it with something I like to call **constrains and gains**.

- That's when you require a struct to do a certain thing, you constrain it to do a certain thing, and if it does, then it gains certain capabilities.

```
struct MemoryGame<CardContent> {  
    var cards: Array<Card> // <- Not good place to initialize the cards  
  
    func choose(card: Card) {  
        print("card chosen: \(card)")  
    }  
  
    init(numberOfPairsOfCards: Int,  
         cardContentFactory: (Int) -> CardContent) {  
        cards = Array<Card>()  
  
        for pairIndex in 0..            let content = cardContentFactory(pairIndex)  
  
            cards.append(Card(content: content,  
                               id: pairIndex*2))  
            cards.append(Card(content: content,  
                               id: pairIndex*2+1))  
        }  
    }  
  
    struct Card: Identifiable {  
        var isFaceUp: Bool = false  
        var isMatched: Bool = false  
        var content: CardContent  
        var id: Int  
    }  
}
```

```
struct Card: Identifiable {
```

- **Identifiable** like **View** is what's called a **protocol** and that's the heart of this **constrains and gains** business.
- Unfortunately, you don't gain much with it, except where **you gain the ability to be identified**.
 - The constraint of **Identifiable** is that you have to have a var called **id**.
 - Luckily, it can be any type you want.

2.3.24 ContentView – iterate over the cards.

1.35:45 Of course, this is no longer the **index** in the range.

```
struct ContentView: View {  
    var viewModel: EmojiMemoryGame  
  
    var body: some View {  
        HStack {  
            ForEach(viewModel.cards) { index in  
                CardView(...)  
            }  
        }  
    }  
}
```

Cannot convert value of type '(UnboundedRange_)->()' to expected argument type 'MemoryGame<String>.Card'

This argument is the card that's in the array.

- You already know that this is an inline function.
 - **card** is the argument of the closure and it's just iterating through these cards in this array.

```
struct ContentView: View {  
    var viewModel: EmojiMemoryGame  
  
    var body: some View {  
        HStack {  
            ForEach(viewModel.cards) { card in  
                CardView(card: card)  
            }  
        } // HStack  
        .padding()  
        .foregroundColor(Color.orange)  
        .font(Font.largeTitle)  
    } // body  
} // ContentView  
  
struct CardView: View {  
    var card: MemoryGame<String>.Card  
  
    var body: some View {  
        ZStack {  
            if card.isFaceUp {  
                RoundedRectangle(cornerRadius: 10.0).fill(Color.white)  
                RoundedRectangle(cornerRadius: 10.0).stroke(lineWidth: 3)  
                Text(card.content)  
            } else {  
                RoundedRectangle(cornerRadius: 10.0).fill()  
            }  
        } // ZStack  
    } // body  
} // CardView
```

CardView(card: card)

- For each card we're going to pass the corresponding card The card that is in the array

2.3.25 Summary

1:36.19 And that's it

- This is how we attach our *Model* to our *View* through our *ViewModel*.

Our *ViewModel* provide essentially a window or a portal on to our *Model* through this *cards* array.

- And through the `choose(card:)` function.

```
class EmojiMemoryGame {
    private var model: MemoryGame<String> =
        EmojiMemoryGame.createMemoryGame()

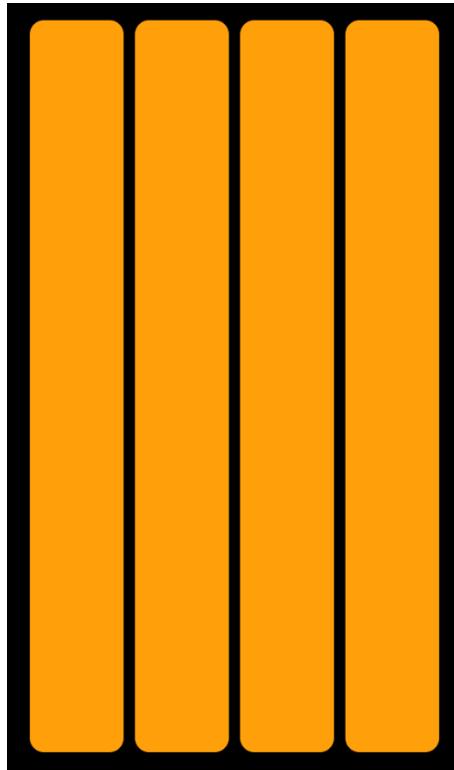
    static func createMemoryGame() -> MemoryGame<String> {
        let emojis: Array<String> = ["🂱", "🂲", "🂳", ]

        return MemoryGame<String>(numberOfPairsOfCards: emojis.count)
    }

    var cards: Array<MemoryGame<String>.Card> {
        model.cards
    }

    func choose(card: MemoryGame<String>.Card) {
        model.choose(card: card)
    }
}
```

2.3.26 Run – isFaceUp = false



We've got four face-down cards because when we created our memory game we said we wanted two pairs of cards.

```
class EmojiMemoryGame {
    private var model: MemoryGame<String> =
        EmojiMemoryGame.createMemoryGame()

    static func createMemoryGame() -> MemoryGame<String> {
        let emojis: Array<String> = ["👨‍🍳", "🧙‍♀️"]

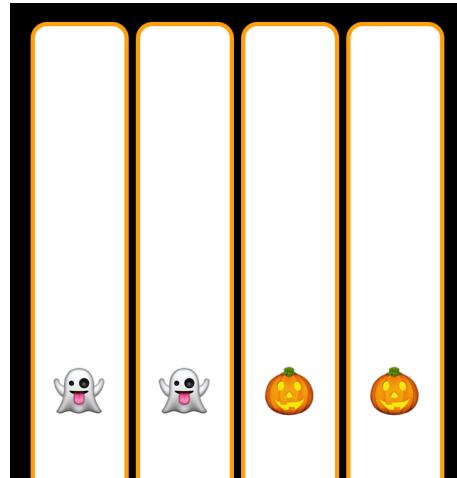
        return MemoryGame<String>(numberOfPairsOfCards: 2) { pairIndex
            in
                return emojis[pairIndex]
        }
    }
}
```

And in `MemoryGame` we have all our cards start facedown.

```
struct Card: Identifiable {
    var isFaceUp: Bool = false
    var isMatched: Bool = false
    var content: CardContent
    var id: Int
}
```

Let's change in our Model to be true

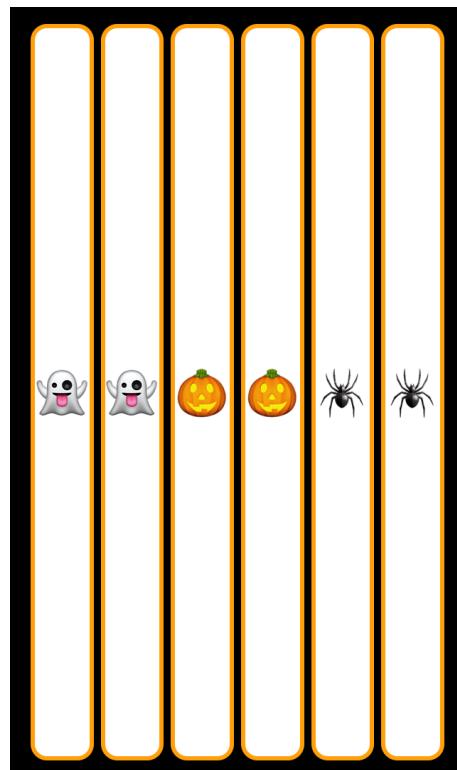
```
struct Card: Identifiable {
    var isFaceUp: Bool = true
    var isMatched: Bool = false
    var content: CardContent
    var id: Int
}
```



Now, we add another emoji.

```
class EmojiMemoryGame {
    private var model: MemoryGame<String> =
        EmojiMemoryGame.createMemoryGame()

    static func createMemoryGame() -> MemoryGame<String> {
        let emojis: Array<String> = ["👻", "🎃", "🕷"]
        return MemoryGame<String>(numberOfPairsOfCards: emojis.count)
    }
}
```



2.3.27 Express an intent

1:38:44 I want to be able to tap on a card.

`onTapGesture` is a function that takes another function as an argument.

- The argument `perform` takes no arguments and it returns nothing.

```
struct ContentView: View {  
    var viewModel: EmojiMemoryGame  
  
    var body: some View {  
        HStack {  
            ForEach(viewModel.cards) { card in  
                CardView(card: card).onTapGesture(perform: { })  
            }  
        } // HStack  
        .padding()  
        .foregroundColor(Color.orange)  
        .font(Font.largeTitle)  
    } // body  
} // ContentView
```

```
ForEach(viewModel.cards) { card in  
    CardView(card: card).onTapGesture(perform: { })  
}
```

`CardView(card: card).onTapGesture(perform: { })`

- `{ }` is a function that takes no arguments and it returns nothing.

```
ForEach(viewModel.cards) { card in  
    CardView(card: card).onTapGesture(perform: {  
        self.viewModel.choose(card: card)  
    })
```

`self.viewModel.choose(card: card)`

- This is an intent to choose a card

```
CardView(card: card).onTapGesture {  
    self.viewModel.choose(card: card)  
}
```

- Because `perform` is the only argument and thus, the last argument, we don't need that on here.
- We make it a little more readable by putting this little embedded function on its own line.

2.3.28 Run

The screenshot shows the Xcode interface with the ContentView.swift file open. The code defines an HStack of cards, each with an orange foreground color and large title font. The output window at the bottom shows four cards being chosen, each with a different emoji face.

```
import SwiftUI
struct ContentView: View {
    var viewModel: EmojiMemoryGame
    var body: some View {
        HStack {
            ForEach(viewModel.cards) { card in
                CardView(card: card).onTapGesture {
                    self.viewModel.choose(card: card)
                }
            }
        }
        .padding()
        .foregroundColor(Color.orange)
        .font(Font.largeTitle)
    }
}
```

card chosen: Card(isFaceUp: true, isMatched: false, content: "😊", id: 0)
card chosen: Card(isFaceUp: true, isMatched: false, content: "😊", id: 1)
card chosen: Card(isFaceUp: true, isMatched: false, content: "😅", id: 2)
card chosen: Card(isFaceUp: true, isMatched: false, content: "😅", id: 3)

```
struct MemoryGame<CardContent> {
    var cards: Array<Card> // <- Not good place to initialize the cards
    func choose(card: Card) {
        print("card chosen: \(card)")
    }
    init(numberOfPairsOfCards: Int,
         cardContentFactory: (Int) -> CardContent) {
        cards = Array<Card>()

        for pairIndex in 0..
```

2.4 Assignment I: Memorize

1:42:56

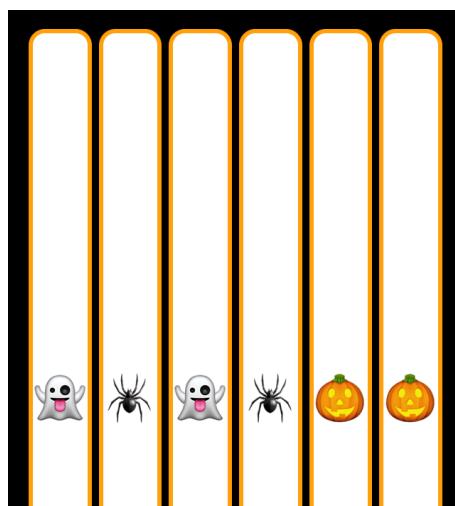
2.4.1 Required Task 1

Get the Memorize game working as demonstrated in lectures 1 and 2. Type in all the code. Do not copy/paste from anywhere.

2.4.2 Required Task 2

Currently the cards appear in a predictable order (the matches are always side-by-side, making the game very easy).

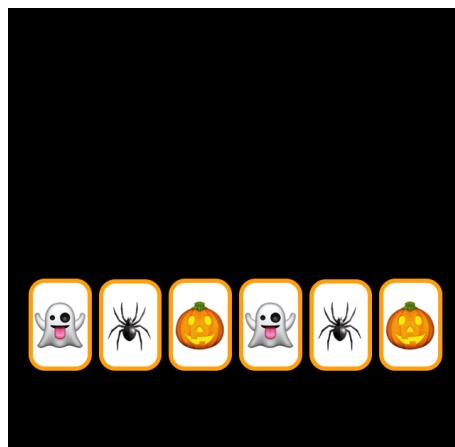
- Shuffle the cards.



2.4.3 Required Task 3

Our cards are currently arranged in a single row (we'll fix that next week).

- That's making our cards really tall and skinny (especially in portrait) which doesn't look very good.
- Force each card to have a width to height ratio of 2:3 (this will result in empty space above and/or below your cards, which is fine).

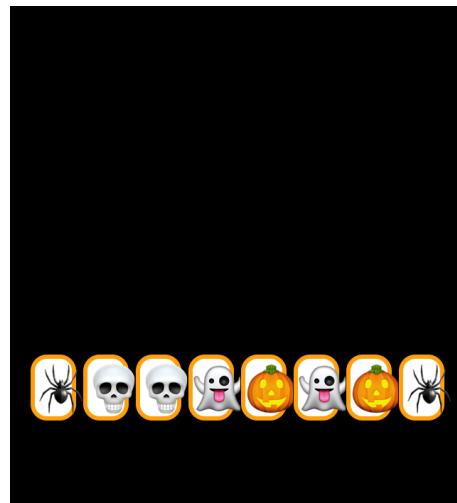


DECLARATION

```
Func aspectRatio(_ aspectRatio: CGFloat? = nil,  
                 contentMode: ContentMode) -> some View
```

2.4.4 Required Task 4

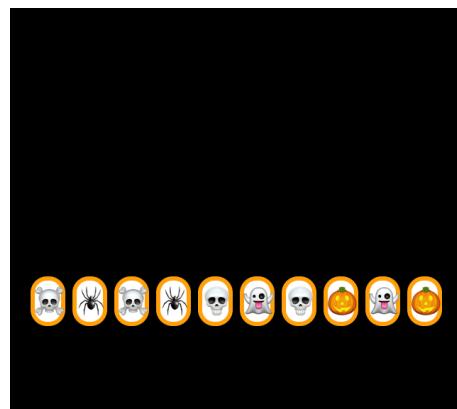
Have your game start up with a random number of pairs of cards between 2 pairs and 5 pairs.



2.4.5 Required Task 5

When your game randomly shows 5 pairs, the font we are using for the emoji will be too large (in portrait) and will start to get clipped.

- Have the font adjust in the 5 pair case (only) to use a smaller font than `.largeTitle`.
- Continue to use `.largeTitle` when there are 4 or fewer pairs in the game.



2.4.6 Required Task 6

Your UI should work in portrait or landscape on any iOS device.

- In landscape your cards will be larger (but still 2:3 aspect ratio).
- This probably will not require any work on your part (that's part of the power of SwiftUI), but be sure to experiment with running on different simulators in Xcode to be sure.

2.4.7 Hint 1

Economy is valuable in coding. The easiest way to ensure a bug-free line of code is not to write that line of code at all.

- Required Tasks 2, 3 and 4 (and possibly even 5) can each be done with an addition or change to **a single line of code**
 - (this is a Hint, not a Required Task, so you are in no way required to do that).

2.4.8 Hint 2

We haven't implemented the "reactive" part of SwiftUI yet (i.e. when the View automatically updates every time the Model changes),

- so you'll be re-launching your application over and over in the simulator (and observing it in the Preview pane) as you test your code.

2.4.9 Hint 3

Shuffling the cards might be easier than you think.

- Be sure to familiarize yourself with the documentation for Array.

DECLARATION
mutating func shuffle()

2.4.10 Hint 4

Be sure to do your shuffling in the proper place in your MVVM.

- Is shuffling a UI thing (i.e. View or ViewModel) or a Model thing?

2.4.11 Hint 5

Another area of the documentation you will want to become very familiar with is the [documentation for the View protocol](#).

- There's a lot there (View is very powerful), and you are of course not required to understand every function in [View](#) (yet), but at least scanning/searching through it will both give you an idea of what's there and [likely be helpful in solving your aspect ratio problem](#).

2.4.12 Hint 6

The function [`Int.random\(in:ClosedRange<Int>\)`](#) can generate a random integer in any range,

- for example, [`let random = Int.random\(in: 15...75\)`](#) would generate a random integer between 15 and 75 (inclusive).

Int.random(in:)

DECLARATION
Static func random(in range: ClosedRange<Int>) -> Int

2.4.13 Hint 7

You'll definitely need to look at the documentation for [Font](#).

- You are welcome to use one of the other built-in fonts listed there (i.e. you don't have to call any of Font's functions like system or custom).
- Just pick whichever built-in gives you the size you want since the font does not affect what an emoji looks like (other than size).

2.4.14 Hint 8

Required Task 5 can also be implemented in a single line of code.

- But you'll likely need to use Swift's "ternary" operator (e.g. `let x = truthTest ? foo : bar`) to do so.
- It is very common to use ternary operators to affect the arguments to modifiers of Views.

2.4.15 Hint 9

One thing we are not addressing in this assignment is that our emoji font looks **too small** in landscape. We'll be fixing this in a much more powerful way next week.

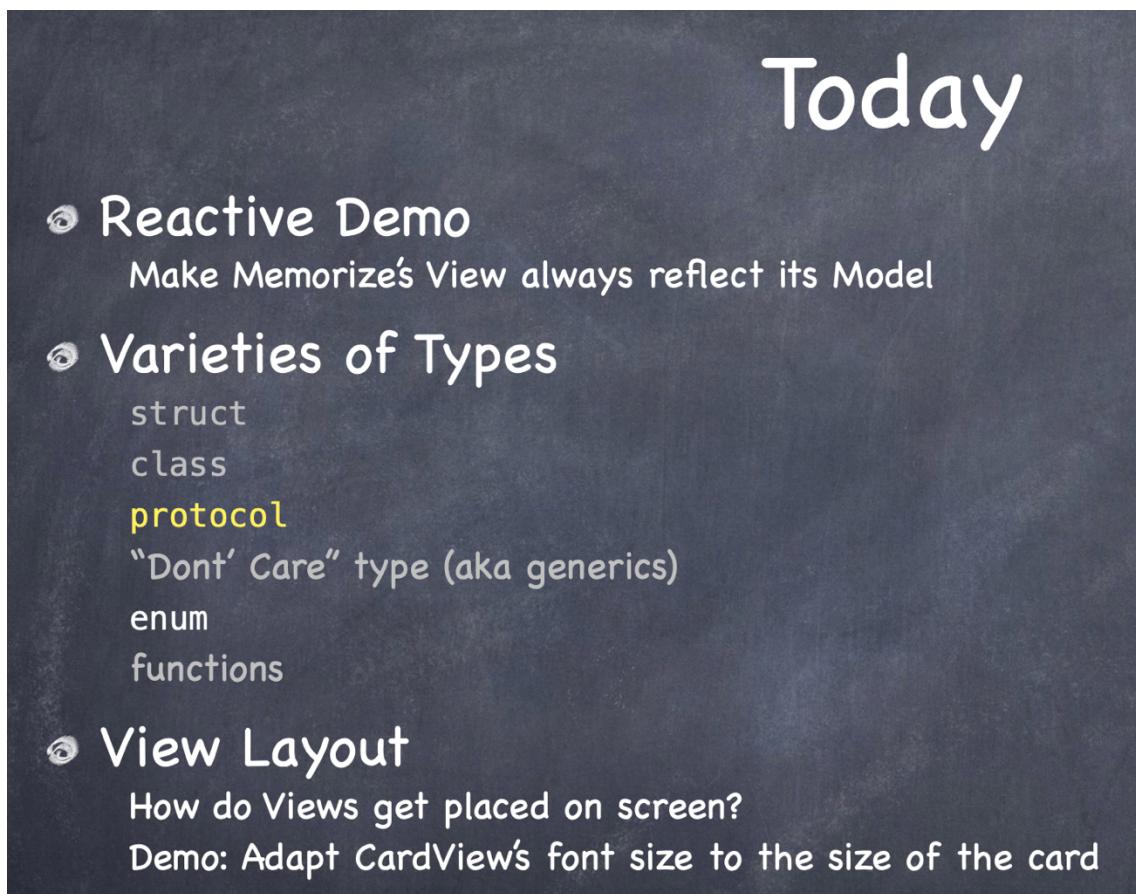
3 Lec03. Reactive UI + Protocols + Layout

The third of the lectures given to Stanford students in Spring quarter of 2020 demonstrates the mechanisms supporting SwiftUI's **reactive UI paradigm** by enhancing Memorize to flip the cards over when they are tapped.

Protocols, an important Swift language feature, are covered in detail as is some of the API for laying out Views, including the "Stacks", **View modifiers** and GeometryReader.

A fundamental principle of MVVM in SwiftUI is the **reactive, declarative approach** to building UIs.

- The View of our MVVM is always automatically reflecting the state in our Model, creating a single source of "truth" for the heart of the application's logic and storage.
- Added to that, we formalize the concept of capturing the user's "intent" to do something and using that to change the Model appropriately.
- Going back to our demo, we apply this to our Memorize application by using **@ObservedObject** and **@Published** to make tap gestures cause cards to flip over.
- After that, we dive back into an exploration of a very important **functional-/protocol-oriented-programming** topic: **protocols**.
- Finally, we go over many of the ways we can lay out the graphical elements of our UI on screen, from **HStack**, **VStack** and **ZStack**, alignment and spacing, and **GeometryReader**, a mechanism for adjusting our appearance to the space allocated to us in the UI.

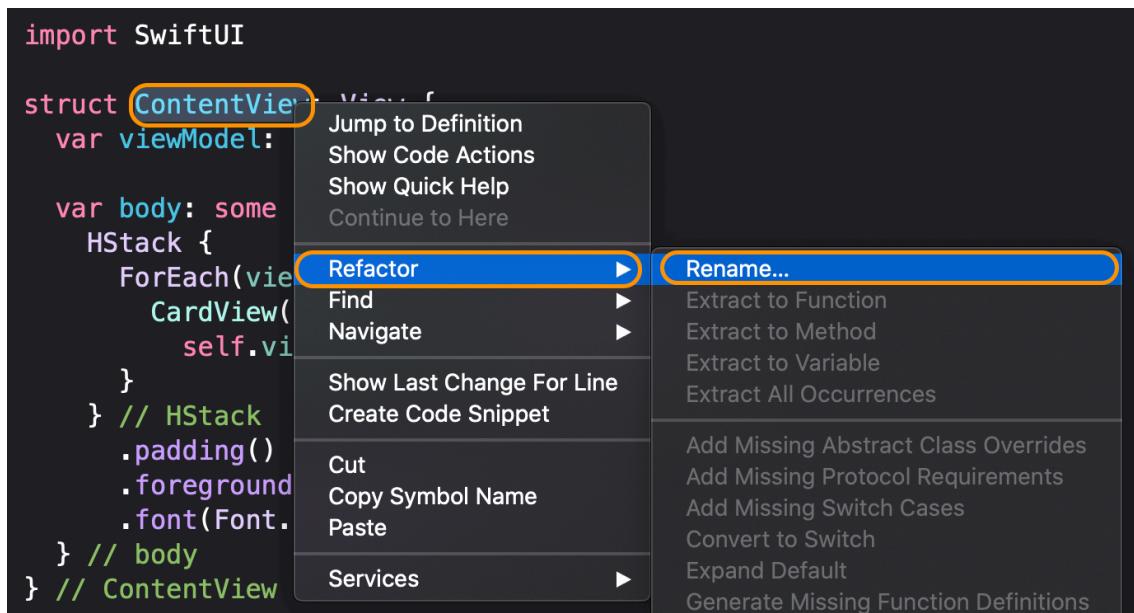


3.1 Reactive Swift demo

3.1.1 Refactoring ContentView -

07:55 If we're going to change the name of a var or of a type, we are going to use refactoring.

- Right click on `ContentView`



```
ContentView.swift
File Name: ContentView.swift
1 // ContentView.swift
2 // Memorize
3
4 struct ContentView: View {
5     var viewModel: EmojiMemoryGame
6 }
7
8 static var previews: some View {
9     ContentView(viewModel: EmojiMemoryGame())
10 }
```

```
All (6) Code (3) File Names (1) Comments (2) Strings (0) Other (0)  
ContentView.swift  
File Name: EmojiMemoryGameView.swift  
1 // Click on "+" to also  
2 // ContentView + left rename the name of the  
3 // Memorize struct in the comments.  
15  
16 struct EmojiMemoryGameView: View {  
17     var viewModel: EmojiMemoryGame  
29 } // body  
30 } // ContentView  
31  
51     static var previews: some View {  
52         EmojiMemoryGameView(viewModel: EmojiMemoryGame())  
53     }  
54 }
```

3.1.2 Model

10:15 Our topic today is going to be **reactive**.

- SwiftUI is reactive.
- When changes happen in the Model, they automatically are going to show up in the View.
- Currently our UI is not reactive.

We are going to make the cards flip over.

In Model code where currently we just say card chosen, I'm going to start implementing my game logic starting with flipping the card over.

3.1.3 Model – Reserved word mutating in a struct

- 17:30 We're changing `isFaceUp` in `cards` array to have it be toggled, and we still have an error
- “Cannot assign to property: ‘self’ is immutable.

It's not saying that the card is **immutable** because **the card is not, it's a variable**.

- If it were a `let`, it would be **immutable**.

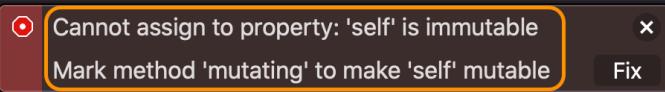
But it's `self` itself that is **immutable**.

- In other words, **our own function doesn't seem to be able to modify our self**.
- How do we do anything then if we can't do that?
- Value types Swift is not doing **bitwise copy**, it's not actually copying them from one place in memory to another when you pass them around.
 - That's the semantic, that's how you have to imagine it's happening, but it's really only **actually making those copies** in memory **when you start changing something** which we are doing here.
 - We are changing the `cards` array, and this is our actual property, so we're not making a copy, it's actually changing the array, but this method changes our self because it changes our cards.
 - We have to let Swift know that by saying that this is a **mutating function**.
 - So **all functions that modify self, have to be marked mutating in a struct**.
 - This is not true in a class. Because classes are in the heap, we can change things through a pointer.

```
struct MemoryGame<CardContent> {
    var cards: Array<Card> // <- Not a good place to initialize the cards
```

```
func choose(card: Card) {
    print("card chosen: \(card)")

    let chosenIndex: Int = self.index(of: card)
    self.cards[chosenIndex].isFaceUp = !self.cards[chosenIndex].isFaceUp
}
```



```
struct MemoryGame<CardContent> {
    var cards: Array<Card> // <- Not good place to initialize the cards
```

```
mutating func choose(card: Card) {
    print("card chosen: \(card)")

    let chosenIndex: Int = index(of: card)
    cards[chosenIndex].isFaceUp = !cards[chosenIndex].isFaceUp
}
```

```
func index(of: Card) -> Int {
```

- This function is just getting the index of a Card in the card's array.
 - It's not actually changing anything.
 - So, it does not need to be **mutating** here.

```

    init(numberOfPairsOfCards: Int,
         cardContentFactory: (Int) -> CardContent) {
    ...
}

```

```
init(numberOfPairsOfCards: Int, cardContentFactory: (Int) -> CardContent){
```

- Our initializer is implicitly changing our self.
 - We're creating our self.
 - We're setting all our variables here.
 - Therefore, this is mutating.
 - You don't say mutating in it because by default all *inits* are *mutating*.

3.1.4 Use of identifiable

19:26

```

struct MemoryGame<CardContent> {
    var cards: Array<Card> // <- Not good place to initialize the cards

    mutating func choose(_ card: Card) {
        print("card chosen: \(card)")

        let chosenIndex: Int = index(of: card)
        cards[chosenIndex].isFaceUp = !cards[chosenIndex].isFaceUp
    }

    func index(of card: Card) -> Int {
        for index in 0..<cards.count {
            if cards[index].id == card.id {
                return index
            }
        }
        return 0 // TODO: bogus!
    }
}

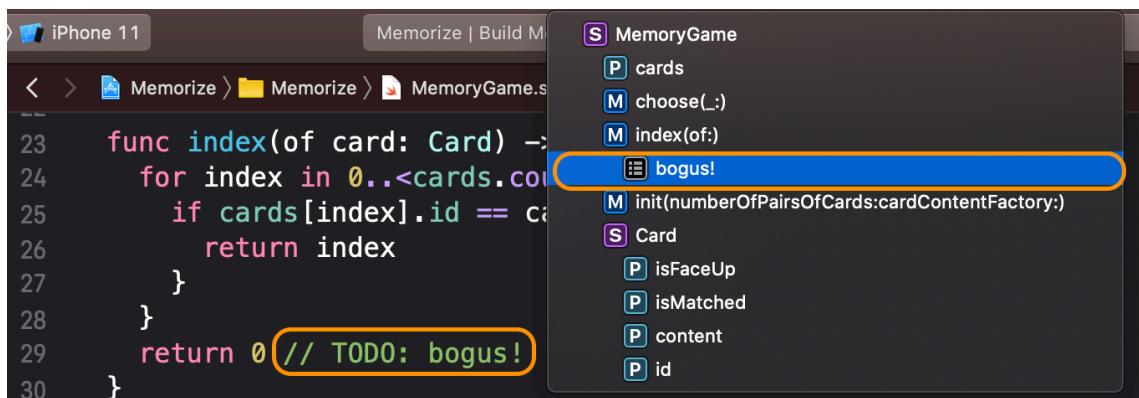
```

```
if cards[index].id == card.id {
```

- Since `card` is *identifiable*, we can look at its `id` and uniquely see which card it is.

```
return 0 // TODO: bogus!
```

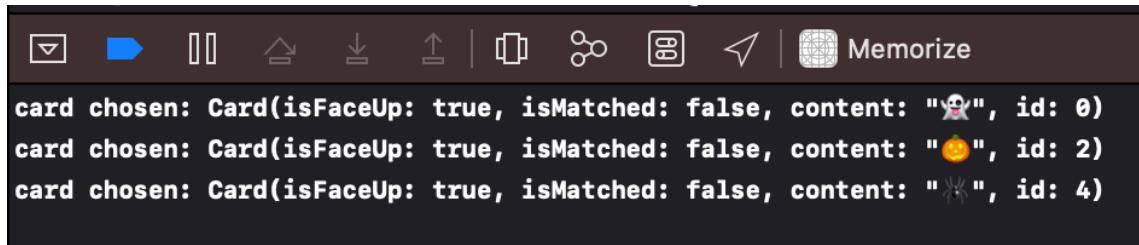
- What if this for loop goes all the way through and never finds that Card?
 - Swift actually has a fantastic thing to return when this happens.
 - For the time being, we will *return 0*.
 - And I'm going to put a `TODO:` mark that you can go back later, and find to remind yourself you need to fix this.



- If I click on `bogus!`, it takes me right to `TODO: bogus!`

3.1.5 Run

24:04



```
card chosen: Card(isFaceUp: true, isMatched: false, content: "💀", id: 0)
card chosen: Card(isFaceUp: true, isMatched: false, content: "🎃", id: 2)
card chosen: Card(isFaceUp: true, isMatched: false, content: "🕷", id: 4)
```

It's definitely accessing the Model because we're getting the cards chosen but **the cards are not flipped over because we haven't yet implemented the reactive behavior.**

- With reactive, when you change the Model, the UI should update.

3.2 Publish/Subscribe

3.2.1 Publish – ViewModel – ObservableObject–

The way we make the ViewModel participate in the reactive behavior is using a *constraints and gains* thing called `ObservableObject`.

```
class EmojiMemoryGame: ObservableObject {
    private var model: MemoryGame<String> =
        EmojiMemoryGame.createMemoryGame()
```

```
class EmojiMemoryGame: ObservableObject {
```

- The constraints in `ObservableObject` are almost none.
 - You don't have to implement any vars or funcs.
 - The tiny little constraint is that it only works for classes.
- The gain is that we get the var `objectWillChange`.

```
class EmojiMemoryGame: ObservableObject {
    private var model: MemoryGame<String> =
        EmojiMemoryGame.createMemoryGame()
```

```
var objectWillChange: ObservableObjectPublisher
```

```
static func createMemoryGame() -> MemoryGame<String> {
    let emojis: Array<String> = ["💀", "🎃", "🕷"]
    return MemoryGame<String>(numberOfPairsOfCards: emojis.count)
{ pairIndex in
    return emojis[pairIndex]
}
```

```
var objectWillChange: ObservableObjectPublisher
```

- In fact, you don't have to put it here like we did with `body`, you get it for free behind the scenes.
 - Therefore, it will not be here.

- This var is not really of type `ObservableObjectPublisher`.
 - It's a little more complicated than that.
 - We don't even know or care what it is really, but there're two things about this var that we need to know.
 - One is that it's a **publisher**, meaning that **it can publish to anyone who's interested**, and our Views are going to be interested when something changes.
 - The other is that this var can only be sent one function, which is the function `send()`.
 - If you call `send()` on `objectWillChange`, it's going to publish to the world something changed about this object or really something will change very soon so get ready and then react to it.
 - That's it, that's all we have to do to have our ViewModel participate in this.
 - Every time our Model changes, we want to do `objectWillChange.send()`.
 - For example, when function `choose(card:)` is called there is an intent.
 - Within this method clearly we're changing our Model. We know that `model.choose(card: card)` is calling a mutable function

```
func choose(card: MemoryGame<String>.Card) {
    objectWillChange.send()
    model.choose(card)
}
```

`objectWillChange.send()`

- This memory game will change
 - It's not saying how it changed, just it changed.
 - That's important to know because if this ViewModel changed, this portal on the Model has changed and Views that are looking to that portal need to redraw themselves.

However, in a significant app, you're going to have possibly a lots of intents and it's a little bit annoying to have to say `objectWillChange.send()` many times.

- It's even error prone. What if I forgot to write this line? Then you choose a card and it's like the car still don't flip over, why?
 - Because I forgot to put that in there.

3.2.2 Publish – ViewModel – @Published

```
class EmojiMemoryGame: ObservableObject {  
    @Published private var model: MemoryGame<String> =  
        EmojiMemoryGame.createMemoryGame()  
  
    static func createMemoryGame() -> MemoryGame<String> {  
        let emojis: Array<String> = ["👻", "🎃", "✳️"]  
  
        return MemoryGame<String>(numberOfPairsOfCards: emojis.count)  
    }  
  
    var cards: Array<MemoryGame<String>.Card> {  
        model.cards  
    }  
  
    func choose(card: MemoryGame<String>.Card){  
        model.choose(card)  
    }  
}
```

```
@Published private var model: MemoryGame<String> =  
    =EmojiMemoryGame.createMemoryGame()
```

- Usually the way we deal with this, is we make this var **Published**.
- This is not a Swift keyword; It's something called a **property wrapper**.
 - *Property wrappers* add a little functionality around a property.
 - In this case **what this wrapper does is every time this property changes it call `objectWillChange.send()`**.

3.2.3 Subscribe – View – @ObservedObject

31:08 Right now, we've got our ViewModel that is publishing every time the Model changes.

- We just need to fix our View so that when ViewModel publishes, it redraws.

```
struct EmojiMemoryGameView: View {
    @ObservedObject var viewModel: EmojiMemoryGame

    var body: some View {
        HStack {
            ForEach(viewModel.cards) { card in
                CardView(card: card).onTapGesture {
                    self.viewModel.choose(card: card)
                }
            }
        } // HStack
        .padding()
        .foregroundColor(Color.orange)
        .font(Font.largeTitle)
    } // body
} // EmojiMemoryGameView

struct CardView: View {
    var card: MemoryGame<String>.Card

    var body: some View {
        ZStack {
            if card.isFaceUp {
                RoundedRectangle(cornerRadius: 10.0).fill(Color.white)
                RoundedRectangle(cornerRadius: 10.0).stroke(lineWidth: 3)
                Text(card.content)
            } else {
                RoundedRectangle(cornerRadius: 10.0).fill()
            }
        } // ZStack
    } // body
} // CardView
```

`@ObservedObject var viewModel: EmojiMemoryGame`

- It says, the `viewModel` var has an `ObservableObject` in it, which it does, `EmojiMemoryGame` is an `ObservableObject`.
 - Every time it says `objectWillChange.send()`, redraw.
 - And of course redrawing `EmojiMemoryGameView`, is going to cause `CardView(card: card).onTapGesture` redraw as well.

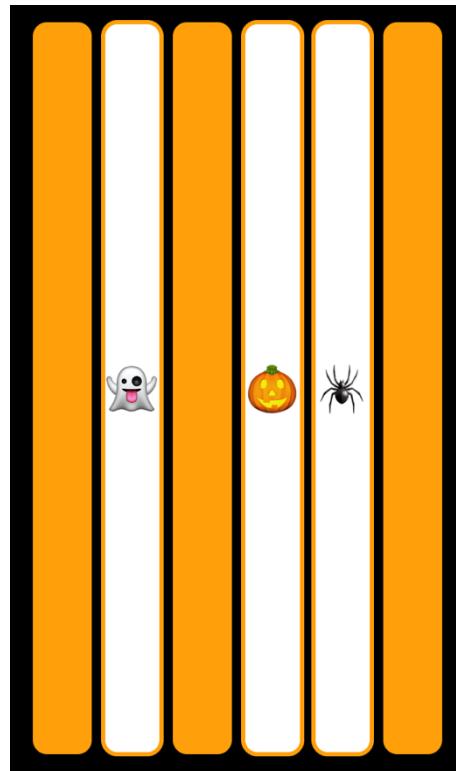
You might kind of feel that this could be really inefficient, what if the Model is changing all the time, are we going to be redrawing our Views every single time?

- Yes and no.
 - Yes, we're going to be reacting to `objectWillChange` and redrawing, but SwiftUI is smart about seeing whether something actually changed.
 - If we flip one card over, it's not going to redraw every single card, just the one that changed and it knows because the cards are **`identifiable`**.
 - You starting to see why this `ForEach` and the array of cards forced us to make this **`identifiable`**? It helps it to understand this one changed so I actually need to redraw that.

3.2.4 Run

33:22 Every time our Model changes, @Published wrapper notices that and as we're an ObservableObject our View can Mark that it's interested in that publishing and therefore it's going to redraw.

This is how we do *reactive programming*.



3.2.5 Recap

We're clicking on one card...

```
import SwiftUI

struct EmojiMemoryGameView: View {
    @ObservedObject var viewModel: EmojiMemoryGame

    var body: some View {
        HStack {
            ForEach(viewModel.cards) { card in
                CardView(card: card).onTapGesture {
                    self.viewModel.choose(card: card)
                }
            } // HStack
            .padding()
            .foregroundColor(Color.orange)
            .font(Font.largeTitle)
        } // body
    } // EmojiMemoryGameView
```

CardView(card: card).onTapGesture {

- ... onTapGesture is called...

self.viewModel.choose(card: card)

- ... then it's expressing this intent in the ViewModel...

```
class EmojiMemoryGame: ObservableObject {  
    ...  
    func choose(card: MemoryGame<String>.Card){  
        model.choose(card)  
    }  
}
```

- ... the ViewModel is then asking the Model ...

```
struct MemoryGame<CardContent> {  
    var cards: Array<Card> // <- Not good place to initialize the cards  
  
    mutating func choose(_ card: Card) {  
        print("card chosen: \(card)")  
  
        let chosenIndex: Int = index(of: card)  
        cards[chosenIndex].isFaceUp = !cards[chosenIndex].isFaceUp  
    }  
}
```

```
cards[chosenIndex].isFaceUp = !cards[chosenIndex].isFaceUp
```

- ... the Model reverse the card ...

```
class EmojiMemoryGame: ObservableObject {  
    @Published private var model: MemoryGame<String> =  
        EmojiMemoryGame.createMemoryGame()
```

```
@Published private var model
```

- ... and it's easy for the ViewModel to notice that this has changed, and it gets published `objectWillChange.send()`

```
struct EmojiMemoryGameView: View {  
    @ObservedObject var viewModel: EmojiMemoryGame
```

```
@ObservedObject var viewModel
```

- ... The View observes those `objectWillChange.send()` and is redrawing

3.3 Protocols

34:24

3.3.1

3.3.2 Layout

3.3.3

3.4 Back to the Demo

1:15:05

Layout

Make CardView proportion itself to the size its “offered” by its container.

How do we deal with “magic numbers” in our code in Swift?

Next lecture we’ll lay out our cards in a grid instead of using an HStack.

I have a short demo I’m going to do just to show you how GeometryReader works.

Make CardView proportion itself to the size its “offered” by its container.

- What we’re going to do is what I’ve been talking about, make our emoji font size better to the space we’re offered using GeometryReader.

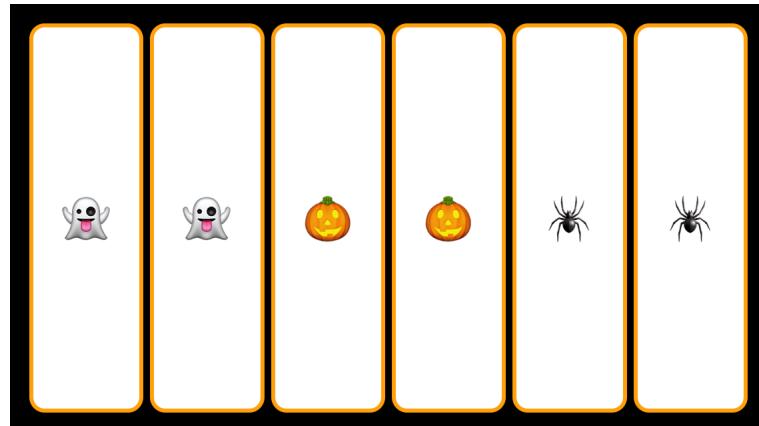
How do we deal with “magic numbers” in our code in Swift?

- While I’m there, I may do a quick little thing where I’m going to show you the best way or the kind of way we’ve all agreed on to collect **magic numbers** in our code.
 - We already have one in `cornerRadios 10`,
 - that `10` is a *magic number*, it really shouldn’t be embedded in our code.
- There’s kind of canonical way in Swift to take that out and put it in its own little space, so it’s well documented and typed.

3.4.1 Adjust the font manually

1:16:17 In your homework, you were asked to adjust the font choice to fit really small cards because large title was too big.

- That fix is just for small cards, but I made you do that almost to realize that that wasn't a good solution especially when we're in landscape where even large title is way too small.



3.4.2 Adjust the font with GeometryReader

What we really want for our card is to pick a font that uses all the space.

- We're going to do that with a special view called **GeometryReader**.

```
struct CardView: View {
    var card: MemoryGame<String>.Card

    var body: some View {
        GeometryReader(content: { geometry in
            ZStack {
                if self.card.isFaceUp {
                    RoundedRectangle(cornerRadius: 10.0).fill(Color.white)
                    RoundedRectangle(cornerRadius: 10.0).stroke(lineWidth: 3)
                    Text(self.card.content)
                } else {
                    RoundedRectangle(cornerRadius: 10.0).fill()
                }
            } // ZStack
        })
    } // body
} // CardView
```

`GeometryReader { geometry in`

- Has one argument which is the *content* that it's going to display inside of itself, which is just another view like our `card`, but it has an argument called `geometry`

28 struct CardView: View {
 29 var card: MemoryGame<String>.Card
 30 @Click
 31 var body: some View {
 32 GeometryReader(content: { geometry in

Summary
A container view that defines its content as a function of its own size and coordinate space.

Declaration

```
struct GeometryReader<Content> where Content : View
```

Discussion
This view returns a flexible preferred size to its parent layout.

[Open in Developer Documentation](#)

DECLARATION
`@frozen struct GeometryReader<Content> where Content : View`

<Content>

- It's got a **don't care** called **Content** that conforms the **view** protocol.

SwiftUI > Drawing and Animation > S GeometryReader > M init(content: (GeometryProxy) -> Content)

Initializer

init(content:)

No overview available.

Language	SDKs	Framework
Swift	iOS 13.0+ macOS 10.15+ tvOS 13.0+ watchOS 6.0+ Xcode 11.0+ Mac Catalyst 13.0+	SwiftUI

Declaration

```
init(content: @escaping (GeometryProxy) -> Content)
```

- Content: takes a function that returns a don't care Content

SwiftUI > Drawing and Animation > GeometryProxy

Structure

GeometryProxy

A proxy for access to the size and coordinate space (for anchor resolution) of the container view.

Language	SDKs	Framework	On This Page
Swift	iOS 13.0+ macOS 10.15+ tvOS 13.0+ watchOS 6.0+ Xcode 11.0+ Mac Catalyst 13.0+	SwiftUI	Declaration Topics

Declaration

```
struct GeometryProxy
```

Topics

Instance Properties

```
var safeAreaInsets: EdgeInsets
    The safe area inset of the container view.

var size: CGSize
    The size of the container view.
```

Declaration

```
var size: CGSize { get }
```

CGSize

- It's going to tell us the width and height that we've been offered to draw within.

get

- It's read only

3.4.5

1:19:49 What I'm going to do is I'm going to have my CardView take ownership to setting its own font.

```
struct CardView: View {
    var card: MemoryGame<String>.Card

    var body: some View {
        GeometryReader(content: { geometry in
            ZStack {
                if self.card.isFaceUp {
                    RoundedRectangle(cornerRadius: 10.0).fill(Color.white)
                    RoundedRectangle(cornerRadius: 10.0).stroke(lineWidth:
3)
                    Text(self.card.content)
                } else {
                    RoundedRectangle(cornerRadius: 10.0).fill()
                }
            } // ZStack
            .font(Font.largeTitle)
        })
    } // body
} // CardView
```



```
ZStack {
    if self.card.isFaceUp {
        RoundedRectangle(cornerRadius: 10.0).fill(Color.white)
        RoundedRectangle(cornerRadius: 10.0).stroke(lineWidth:
3)
        Text(self.card.content)
    } else {
        RoundedRectangle(cornerRadius: 10.0).fill()
    }
} // ZStack
    .font(Font.system(size: min(geometry.size.width,
geometry.size.height)))
```



```
.font(Font.system(size: min(geometry.size.width,
geometry.size.height)))
```

- We create a *font* whose *size* is based on that *geometry size*.
- I'm just going to pick the minimum of those two to make sure that I pick a font that fits.
- I'm using the *size* var that we saw in **GeometryProxy**.

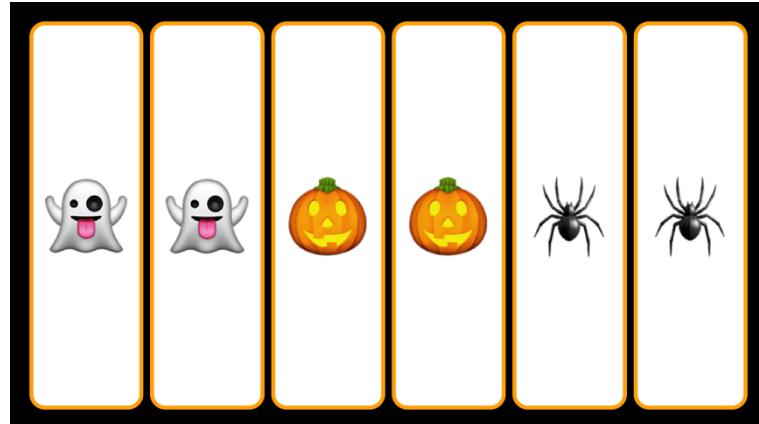
3.4.6 Run



1:21:26 This is actually a little too big.

- Probably I need some sort of constant multiplier

```
} // ZStack
    .font(Font.system(size: min(geometry.size.width,
        geometry.size.height) * 0.75))
```



3.5 Coding style

1:22:05 I'm going to talk about coding style because it's a little bit of a segue into how we're going to structure our View code to make it as readable as possible.

- This problem I'm worried about is these magic numbers.
 - We've created these blue magic numbers and started to sprinkle them out throughout our code.
 - That's not very good coding style.
 - SwiftUI is declarative. We're essentially declaring the UI directly here.
 - When we do that, these blue numbers end up being kind of knobs that we can turn to fine tune the way our View looks and get it just right.
 - Right now our knobs are just spread all over the place.
 - Really nice if we could have a *control panel*.

```
struct CardView: View {
    var card: MemoryGame<String>.Card

    var body: some View {
        GeometryReader { geometry in
            ZStack {
                if self.card.isFaceUp {
                    RoundedRectangle(cornerRadius: self.cornerRadius)
                        .fill(Color.white)
                    RoundedRectangle(cornerRadius: self.cornerRadius)
                        .stroke(lineWidth: self.edgeLineWidth)
                    Text(self.card.content)
                } else {
                    RoundedRectangle(cornerRadius: self.cornerRadius)
                        .fill()
                }
            } // ZStack
            .font(Font.system(size: min(geometry.size.width,
                                         geometry.size.height) *
self.fontSizeFactor))
        }
    } // body

    // MARK: - Drawing Constants

    let cornerRadius: CGFloat = 10.0
    let edgeLineWidth: CGFloat = 3
    let fontSizeFactor: CGFloat = 0.75
} // CardView

let cornerRadius: CGFloat = 10.0
```

- In this case to avoid errors, we won't let Swift to infer the numbers.

I'm going to show you a trick for avoiding this self dot thing in this common case of GeometryReader.

- Whenever you do a GeometryReader, the stuff inside is always going to complain about this self dot.

```
struct CardView: View {
    var card: MemoryGame<String>.Card

    var body: some View {
        GeometryReader { geometry in
            self.body(for: geometry.size)
        }
    } // body

    func body(for size: CGSize) -> some View {
        ZStack {
            if self.card.isFaceUp {
                RoundedRectangle(cornerRadius: cornerRadius)
                    .fill(Color.white)
                RoundedRectangle(cornerRadius: cornerRadius)
                    .stroke(lineWidth: edgeLineWidth)
                Text(self.card.content)
            } else {
                RoundedRectangle(cornerRadius: cornerRadius)
                    .fill()
            }
        } // ZStack
        .font(Font.system(size: min(size.width,
                                     size.height) * fontScaleFactor))
    }
}
```

- This code now is no longer embedded inside the this GeometryReader like this, so you don't need these self dot.

We could do one other thing.
I could have a function

You might think this is a very simple expression, I don't really need to turn it into its own func here, but again, you can't make this too simple.

```
struct CardView: View {
    var card: MemoryGame<String>.Card

    var body: some View {
        GeometryReader { geometry in
            self.body(for: geometry.size)
        }
    } // body

    func body(for size: CGSize) -> some View {
        ZStack {
            if self.card.isFaceUp {
                RoundedRectangle(cornerRadius: cornerRadius)
                    .fill(Color.white)
                RoundedRectangle(cornerRadius: cornerRadius)
                    .stroke(lineWidth: edgeLineWidth)
                Text(self.card.content)
            } else {
                RoundedRectangle(cornerRadius: cornerRadius)
                    .fill()
            }
        } // ZStack
        .font(Font.system(size: fontSize(for: size)))
    }

    // *****
    // MARK: - Drawing Constants
    //

    let cornerRadius: CGFloat = 10.0
    let edgeLineWidth: CGFloat = 3

    func fontSize(for size: CGSize) -> CGFloat {
        min(size.width, size.height) * 0.75
    }
} // CardView
```

4 Lec04. Grid + enum + Optionals

The survey of the Swift type system completes with a discussion of `enum`.

An important language construct, **Optionals**, is both explained in slides and then demonstrated in **Memorize** as we fully implement the logic of the game.

Today

- ⌚ Grid
 - Demo of generics with protocols, functions as types, container view, etc.
- ⌚ Varieties of Types
 - struct
 - class
 - protocol
 - “Dont’ Care” type (aka generics)
 - enum
 - functions
- ⌚ Optional
 - An extremely important type in Swift (it’s an enum).

Today

- Today, I’m going to start off with a big demo, and it’s going to be to make our card game be in rows and columns, instead of all across in one row.

4.1 Create a Grid

01:17 We're going to create a **grid**

- by replacing our **HStack**...

```
func body(for size: CGSize) -> some View {  
    ZStack {  
        if self.card.isFaceUp {  
            RoundedRectangle(cornerRadius: cornerRadius)  
                .fill(Color.white)  
            RoundedRectangle(cornerRadius: cornerRadius)  
                .stroke(lineWidth: edgeLineWidth)  
            Text(self.card.content)  
        } else {  
            RoundedRectangle(cornerRadius: cornerRadius)  
                .fill()  
        }  
    } // ZStack  
    .font(Font.system(size: fontSize(for: size)))  
}
```

- for a **Grid**.
 - As there's no such thing as a **grid** in SwiftUI, we are going to have to write that.
 - It actually provides us with a great opportunity to learn a lot about how things like **ZStack** work.

```
struct EmojiMemoryGameView: View {  
    @ObservedObject var viewModel: EmojiMemoryGame  
  
    var body: some View {  
        Grid {  
            ForEach(viewModel.cards) { card in  
                CardView(card: card).onTapGesture {  
                    self.viewModel.choose(card: card)  
                }  
            } // HStack  
            .padding()  
            .foregroundColor(Color.orange)  
        } // body  
    } // EmojiMemoryGameView
```

Grid {

- Our grid is just going to have a simple view that it's going to replicate using a **ForEach**, to put a certain view at every spot in the row and column.

4.1.1 How ZStack works

```
func body(for size: CGSize) -> some View {
    ZStack {
        if self.card.isFaceUp {
            RoundedRectangle(cornerRadius: cornerRadius)
                .fill(Color.white)
            RoundedRectangle(cornerRadius: cornerRadius)
                .stroke(lineWidth: edgeLineWidth)
            Text(self.card.content)
        } else {
            RoundedRectangle(cornerRadius: cornerRadius)
                .fill()
        }
    } // ZStack
    .font(Font.system(size: fontSize(for: size)))
}
```

ZStack {

- **ZStack** takes an argument which is a function
 - The curly brace ({ }) means it's a function.
- In this case, takes a function that takes no arguments.
- The **view** that gets built here is quite powerful.
 - It could be a list of other views, an if-the, a combination thereof.
 - This particular function that can build this complicated views is called **View Builder**.

4.1.2 View – Grid(viewModel.cards) { card in

```
struct EmojiMemoryGameView: View {
    @ObservedObject var viewModel: EmojiMemoryGame

    var body: some View {
        Grid(viewModel.cards) { card in
            CardView(card: card).onTapGesture {
                self.viewModel.choose(card: card)
            }
        } // Grid
        .padding()
        .foregroundColor(Color.orange)
    } // body
} // EmojiMemoryGameView
```

- Our grid is really going to combine **HStack** like Grid except for 2D **HStack** if you want to think of it, with **ForEach** like this.

viewModel.cards

- We're going to take an **array** of **identifiable** things, like this cards,...

{ card in

- ... and then we're going to pass a function that takes **card** as one of one **identifiable** thing as its argument, and return a card view (**CardView**) to use to draw at that location in the **grid**.

4.1.3 Grid – Create Grid.swift

03.15 We're going to create our grid in its own file because this is really a very powerful reusable object.

- We could use it in all our apps that needed a Grid.

New | File | iOS | SwiftUI View

- Save As: **Grid**

```
import SwiftUI

struct Grid: View {
    var body: some View {
        Text("Hello, World!")
    }
}

struct Grid_Previews: PreviewProvider {
    static var previews: some View {
        Grid()
    }
}
```

struct Grid_Previews: PreviewProvider {

- This struct hooks up to our canvas on the right.
 - But this grid is completely generic.
 - If we were going to make this preview work, we would need to come up with some test data for it.
 - We're not going to do that in this demo

Grid(viewModel.cards) { card in

- First thing we're going to do is its two arguments.
 - First argument: *identifiable viewModel.cards*
 - Second argument: *a function*

```
struct Grid<Item, ItemView>: View {
    var items: [Item]
    var viewForItem: (Item) -> ItemView

    var body: some View {
        Text("Hello, World!")
    }
}
```

var items: [Item]

- *Item* for us in **Grid** is a don't-care.
 - We really don't care what that thing is.

var viewForItem: (Item) -> ItemView

- This second argument is a function that takes an item as the argument, and it returns some *ItemView*.
 - which is another don't-care for us.

struct Grid<Item, ItemView>: View {

- We really don't care what kind of view you provide for each item

```

struct EmojiMemoryGameView: View {
    @ObservedObject var viewModel: EmojiMemoryGame

    var body: some View {
        Grid(items: viewModel.cards, viewForItem: { card in
            CardView(card: card).onTapGesture {
                self.viewModel.choose(card: card)
            }
        }) // Grid
            .padding()
            .foregroundColor(Color.orange)
    } // body
} // EmojiMemoryGameView

```

`Grid(items: viewModel.cards, viewForItem: { card in`

- We have to name the first argument as `items`
- and because the `viewForItem` is the last argument to this function (*trailing closure*), we can just get rid of this label...

```

struct EmojiMemoryGameView: View {
    @ObservedObject var viewModel: EmojiMemoryGame

    var body: some View {
        Grid(items: viewModel.cards) { card in
            CardView(card: card).onTapGesture {
                self.viewModel.choose(card: card)
            }
        } // Grid
            .padding()
            .foregroundColor(Color.orange)
    } // body
} // EmojiMemoryGameView

```

`Grid(items: viewModel.cards) { card in`

- ... and have our function kind of float on the outside.
 - So, this is the last argument to our grid.

4.1.4 Grid - Arguments with no labels

06:02 Why `ForEach` doesn't have labels in their arguments?

- It's using technology that you know well to do that.

```

struct Grid<Item, ItemView>: View {
    var items: [Item]
    var viewForItem: (Item) -> ItemView

    init(_ items: [Item], viewForItem: (Item) -> ItemView) {
        self.items = items
        self.viewForItem = viewForItem
    }

    var body: some View {
        Text("Hello, World!")
    }
}

```

`self.items = items`

- This is a different reason to have to put `self` here

4.1.5 ERROR - @scaping closure

07:52 However, we still have a problem back here in our grid. This error right here.

- It says, **Assigning non-scaping parameter “ViewItemForItem” to an @scaping closure**

```
struct Grid<Item, ItemView>: View {  
    var items: [Item]  
    var viewForItem: (Item) -> ItemView  
  
    init(_ items: [Item], viewForItem: (Item) -> ItemView) {  
        self.items = items  
        self.viewForItem = viewForItem  
    }  
    var body: some View {  
        Text("Hello, World!")  
    }  
}
```

A screenshot of Xcode showing a code editor with the above Swift code. A yellow box highlights the line `self.viewForItem = viewForItem`. A tooltip window appears over this line with the text "Assigning non-escaping parameter 'viewForItem' to an @escaping closure". Below the tooltip, another message says "Parameter 'viewForItem' is implicitly non-escaping" with a "Fix" button.

In SwiftUI, since we're doing functional programming, almost everything is a value type, and the problem that this escaping closure thing is trying to address here is really rare in SwiftUI.

```
init(_ items: [Item], viewForItem: (Item) -> ItemView) {
```

- This *function* that's passed in here, that creates a view for a given item, is not actually used in this initializer.
 - We declare it in `viewForItem` property, and call it later.
 - and we are going to call it in our body later when we need to actually create the views for all of our items.

```
struct Grid<Item, ItemView>: View {  
    var items: [Item]  
    var viewForItem: (Item) -> ItemView  
  
    init(_ items: [Item],  
         viewForItem: @escaping (Item) -> ItemView) {  
        self.items = items  
        self.viewForItem = viewForItem  
    }  
}
```

```
viewForItem: @escaping (Item) -> ItemView) {
```

- That's why we have to mark this kind of function as `@escaping`.
 - You can think of it as *this function is going to escape from the initializer without getting called*.

Swift has to be careful and it's actually very powerful and knows how to deal with these closures that might get called later.

- Why can that be an issue?

```
struct EmojiMemoryGameView: View {
    @ObservedObject var viewModel: EmojiMemoryGame

    var body: some View {
        Grid(items: viewModel.cards) { card in
            CardView(card: card).onTapGesture {
                self.viewModel.choose(card: card)
            }
        } // Grid
        .padding()
        .foregroundColor(Color.orange)
    } // body
} // EmojiMemoryGameView
```

{ card in

- Let's look back here where we actually pass this closure in.
 - Whatever code is in here has to be able to be called later.
 - If we use any **variables**, especially if we use a local variable or something from this closure,
 - they **have to be around in the future** when this closure gets executed.

Swift accomplish this by making **function types** be **reference types**.

- Just like classes, our ViewModel are reference types that live in the heap.
 - They're stored in memory.
 - People have pointers to them.
 - Same thing with these functions that can be called later.
 - The things inside these closures might also live in the heap.

self.viewModel.choose(card: card)

- If any of these things are classes, for example, the keyword **self** is going to be a pointer to something in the heap.
- The problem we're trying to avoid here in Swift is self having some var in it that actually points to the closure, because we know the closure points to self inside here.
 - They're both going to be in the heap.
 - The closure and everything inside are kept in the heap because it's going to be executed later.
 - If anything inside the closure points back, we've got a situation where two things in the heap are pointing to each other.
 - The way that Swift cleans up memory is when nobody points to something anymore, it cleans up the memory and frees it up for someone else to use.
 - If two things are pointing to each other in the heap, they're never going to be able to go away.
 - That's called a **memory cycle**.

This whole thing is trying to make it so that we can detect *memory cycles* by seeing this `@escaping` functions.

Amazingly, this is also why you get this error when you omit `self`.

The screenshot shows a portion of a Swift file named `EmojiMemoryGameView.swift`. The code defines a `struct` `EmojiMemoryGameView` that conforms to `View`. It has a `@ObservedObject` property `viewModel` of type `EmojiMemoryGame`. The `body` property contains a `Grid` of cards. Tapping on a card triggers an `onTapGesture` closure. Inside this closure, there is a call to `viewModel.choose(card: card)`. A red tooltip box appears over this line, stating: "Reference to property 'viewModel' in closure requires explicit 'self.' to make capture semantics explicit". Below the tooltip, there is a button labeled "Insert 'self.'".

```
struct EmojiMemoryGameView: View {
    @ObservedObject var viewModel: EmojiMemoryGame

    var body: some View {
        Grid(viewModel.cards) { card in
            CardView(card: card).onTapGesture {
                viewModel.choose(card: card)
            }
        } // Grid
        .padding()
        .foregroundColor(Color.orange)
    } // body
} // EmojiMemoryGameView
```

When we don't put a `self.` in our `onTapGesture` function, we're going to get this error that says this requires explicit self to make the *capture semantics explicit*.

- **Capture semantics** means the fact that it's going to capture everything inside the `onTapGesture` function, and keep it in the heap so that when `onTapGesture` executes it in the future when someone taps on this card, this stuff is still around.

```
struct EmojiMemoryGameView: View {
    @ObservedObject var viewModel: EmojiMemoryGame

    var body: some View {
        Grid(items: viewModel.cards) { card in
            CardView(card: card).onTapGesture {
                self.viewModel.choose(card: card)
            }
        } // Grid
        .padding()
        .foregroundColor(Color.orange)
    } // body
} // EmojiMemoryGameView
```

There should be no problem because this `self.` does not live in the heap anyway.

- This `self.` is the `EmojiMemoryGameView` struct.
 - Structs are value types that don't live in the heap.
- That is the fix that has been publicly approved.
 - The fix is basically if `self`, in this case, inside of one of these scaping functions is going to held around in the heap, if `self` doesn't live in the heap, in other words, it's a value type, a struct or an enum, then you don't need to have this error come up

4.1.7 ERROR – Convert to 'Range <Int>' - where Item: Identifiable

15:24

```
 9 import SwiftUI
10
11 struct Grid<Item, ItemView>: View {
12     var items: [Item]
13     var viewForItem: (Item) -> ItemView
14
15
16     init(_ items: [Item],
17          viewForItem: @escaping (Item) -> ItemView) {
18         self.items = items
19         self.viewForItem = viewForItem
20     }
21
22
23     var body: some View {
24         ForEach(items) { item in
25             self.viewForItem(item) // Cannot convert value of type '[Item]' to expected argument type 'Range<Int>'
26             // Return type of property 'body' requires that 'ItemView' conforms to 'View'
27
28
29
30
31         self.viewForItem(item) // Cannot convert value of type 'Int' to expected type 'Item'
32     }
33 }
34 }
```

ForEach(items) { item in

- The argument items in this `ForEach` has to be an array of *identifiable* things.
 - That's a problem for us because `Item` is a *don't care*.
 - We have no idea what this thing is.
 - [Here is where we get constraints and gains into the act.](#)

```
struct Grid<Item, ItemView>: View where Item: Identifiable {
    var items: [Item]
    var viewForItem: (Item) -> ItemView

    init(_ items: [Item],
          viewForItem: @escaping (Item) -> ItemView) {
        self.items = items
        self.viewForItem = viewForItem
    }
}
```

struct Grid<Item, ItemView>: View where Item: Identifiable {

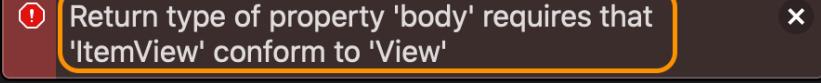
- We've created a grid that only works with this don't care when that don't care is *Identifiable*.
 - This can still be anything.
 - But [we care a little bit about Item](#).
 - [We care that it's identifiable](#).

```
import SwiftUI

struct Grid<Item, ItemView>: View where Item: Identifiable {
    var items: [Item]
    var viewForItem: (Item) -> ItemView

    init(_ items: [Item],
         viewForItem: @escaping (Item) -> ItemView) {
        self.items = items
        self.viewForItem = viewForItem
    }

    var body: some View {
        ForEach(items) { item in
            self.viewForItem(item)
        }
    }
}
```



‘ItemView’ conform to ‘View’

- `ItemView` is the return type of `viewForItem` function.
 - Of course, that makes sense because `ForEach` can only use views to have views for these items.
 - But right now it's a don't care as well.

```
import SwiftUI

struct Grid<Item, ItemView>: View where Item: Identifiable,
ItemView: View {
    var items: [Item]
    var viewForItem: (Item) -> ItemView

    init(_ items: [Item],
         viewForItem: @escaping (Item) -> ItemView) {
        self.items = items
        self.viewForItem = viewForItem
    }
```

```
struct Grid<Item, ItemView>: View where Item: Identifiable,
ItemView: View {
```

- You see how we're connecting **generics**; `Grid` is a **generic struct** with **protocols**. `Identifiable` and `View` are protocols, and we're using them to constrain the `Item` and `ItemView` don't-care to work.

4.1.9 Grid - GeometryReader

18:00 The next thing we have to do here is we're a container.

```
var body: some View {
    ForEach(items) { item in
        self.viewForItem(item)
    }
}
```

```
self.viewForItem(item)
```

- Grid contains all these views.
 - It puts them in rows and columns.
- We know the container's job is to take the space that's offered to them and divide it up amongst the things inside.
 - That means we need to figure out how much space has been allocated to us.
 - Geometry is going to allow us to find out how much space was given to the grid.

```
import SwiftUI

struct Grid<Item, ItemView>: View where Item: Identifiable,
ItemView: View {
    var items: [Item]
    var viewForItem: (Item) -> ItemView

    init(_ items: [Item],
          viewForItem: @escaping (Item) -> ItemView) {
        self.items = items
        self.viewForItem = viewForItem
    }

    var body: some View {
        GeometryReader { geometry in
            self.body(for: geometry.size)
        }
    }

    func body(for size: CGSize) -> some View {
        ForEach(items) { item in
            self.body(for: item, in: size)
        }
    }

    func body(for item: Item, in size: CGSize) -> some View {
        return viewForItem(item)
    }
}
```

All we have left to do, is to actually offer these views some of our space and then to position them.

4.1.10 Grid - GridLayout.swift

We need to do some math that I wrote in the file `GridLayout.swift`.

4.1.11 Grid - Create GridLayout

21:46

```
struct Grid<Item, ItemView>: View where Item: Identifiable,  
ItemView: View {  
    var items: [Item]  
    var viewForItem: (Item) -> ItemView  
  
    init(_ items: [Item],  
         viewForItem: @escaping (Item) -> ItemView) {  
        self.items = items  
        self.viewForItem = viewForItem  
    }  
  
    var body: some View {  
        GeometryReader { geometry in  
            self.body(for: GridLayout(itemCount: self.items.count,  
                                      in: geometry.size))  
        }  
    }  
  
    func body(for layout: GridLayout) -> some View {  
        ForEach(items) { item in  
            self.body(for: item, in: layout)  
        }  
    }  
  
    func body(for item: Item, in layout: GridLayout) -> some View {  
        return viewForItem(item)  
    }  
}
```

4.1.12 Grid - Position the view

23:16 Now we have to position the view,

```
import SwiftUI

struct Grid<Item, ItemView>: View where Item: Identifiable, ItemView: View {
    var items: [Item]
    var viewForItem: (Item) -> ItemView

    init(_ items: [Item],
         viewForItem: @escaping (Item) -> ItemView) {
        self.items = items
        self.viewForItem = viewForItem
    }

    var body: some View {
        GeometryReader { geometry in
            self.body(for: GridLayout(itemCount: self.items.count,
                                      in: geometry.size))
        }
    }

    func body(for layout: GridLayout) -> some View {
        ForEach(items) { item in
            self.body(for: item, in: layout)
        }
    }

    func body(for item: Item,
              in layout: GridLayout) -> some View {
        let index = self.index(of: item)
        return viewForItem(item)
            .frame(width: layout.itemSize.width,
                   height: layout.itemSize.height)
            .position(layout.location(ofItemAt: index))
    }

    func index(of item: Item) -> Int {
        for index in 0..

- o This function is kind of double bogus.
  - o We're still returning 0.
  - o And this function is exactly the same as the one we wrote in MemoryGame.
    - You'd never want to write the exact same code like this twice.

```

4.1.13 Grid – Recap

26:14

```
var body: some View {  
    GeometryReader { geometry in  
        self.body(for: GridLayout(itemCount: self.items.count,  
                                  in: geometry.size))  
    }  
}
```

```
self.body(for: GridLayout(itemCount: self.items.count, in:  
                           geometry.size))
```

- We get the space that's offered to us.

```
func body(for layout: GridLayout) -> some View {  
    ForEach(items) { item in  
        self.body(for: item, in: layout)  
    }  
}
```

```
func body(for layout: GridLayout) -> some View {
```

- We use this `GridLayout` to divide it up...

```
func body(for item: Item,  
         in layout: GridLayout) -> some View {  
    let index = self.index(of: item)  
    return viewForItem(item)  
        .frame(width: layout.itemSize.width,  
               height: layout.itemSize.height)  
        .position(layout.location(ofItemAt: index))  
}
```

```
.frame(width: layout.itemSize.width, height:  
layout.itemSize.height)
```

- ... and then we offer it to our little sub-views in here...

```
.position(layout.location(ofItemAt: index))
```

- ... and then we position them at those locations in the `GridLayout`.

4.1.14 Grid - Run

26:33



4.1.15 Grid - View - Padding

26:44 I would like a little bit of padding between the cards.

```
struct EmojiMemoryGameView: View {  
    @ObservedObject var viewModel: EmojiMemoryGame  
  
    var body: some View {  
        Grid(viewModel.cards) { card in  
            CardView(card: card).onTapGesture {  
                self.viewModel.choose(card: card)  
            }  
                .padding(5)  
        } // Grid  
        .padding()  
        .foregroundColor(Color.orange)  
    } // body  
} // EmojiMemoryGameView
```



4.1.16 Search an item – Item is an identifiable Card struct

27:13 Now that our grid is working, let's go and fix the bogus code that we have right here, which is this index of item.

What're we actually asking to happen?

```
struct MemoryGame<CardContent> {
    var cards: Array<Card>
    ...
    init(numberOfPairsOfCards: Int,
        cardContentFactory: (Int) -> CardContent) {
        cards = Array<Card>()

        for pairIndex in 0..
```

```
struct EmojiMemoryGameView: View {
    @ObservedObject var viewModel: EmojiMemoryGame

    var body: some View {
        Grid(viewModel.cards) { card in
            CardView(card: card).onTapGesture {
                self.viewModel.choose(card: card)
            }
            .padding(5)
        } // Grid
        .padding()
        .foregroundColor(Color.orange)
    } // body
} // EmojiMemoryGameView
```

```
struct Grid<Item, ItemView>: View where Item: Identifiable,
ItemView: View {
    var items: [Item]
    var viewForItem: (Item) -> ItemView

    init(_ items: [Item],
        viewForItem: @escaping (Item) -> ItemView) {
        self.items = items
        self.viewForItem = viewForItem
    }
}
```

```

var body: some View {
    GeometryReader { geometry in
        self.body(for: GridLayout(itemCount: self.items.count,
                                  in: geometry.size))
    }
}

func body(for layout: GridLayout) -> some View {
    ForEach(items) { item in
        self.body(for: item, in: layout)
    }
}

func body(for item: Item,
          in layout: GridLayout) -> some View {
    let index = self.index(of: item)
    return viewForItem(item)
        .frame(width: layout.itemSize.width,
               height: layout.itemSize.height)
        .position(layout.location(ofItemAt: index))
}

func index(of item: Item) -> Int {
    for index in 0..<items.count {
        if items[index].id == item.id {
            return index
        }
    }
    return 0 // TODO: bogus!
}

```

```

func body(for item: Item, in layout: GridLayout) -> some View {
    ○ We have an array of things that are Identifiable, and we have one of those things
        ○ and we want to look and see if we can find the Item which is an
            Identifiable Card struct.
                ■ struct Card: Identifiable {
    ○ That sounds to me more like an Array thing.

```

4.1.17 Search an item - Extension Array where Element: Identifiable {

27:50 Let's add a function to an Array to search the item.

```
extension Array where Element: Identifiable {
    func index(of item: Item) -> Int {
        for index in 0..
```

```
extension Array where Element: Identifiable {
```

- We are only going to search items to arrays where the elements in the array are **identifiable**
 - There it is again *constraints and gains*.
 - This extension is only going to add this function that we want to add to these kinds of arrays.

```
extension Array where Element: Identifiable {
    func index(of item: Element) -> Int {
        for index in 0..
```

```
if self[index].id == item.id {
```

- Before we were looking at items,
 - Now because this is an array function, items become **self**.

```
func index(of item: Element) -> Int {
```

- The type of this item, it's not **Item**.
 - It's **Element** because that's what we're looking for in an array.

```
extension Array where Element: Identifiable {
```

- We have an array whose **elements** are all *identifiable*.

```
func index(of item: Element) -> Int {
```

- What we're saying here is get the index of one of the **elements** of this **array** that's *identifiable*.

```

func body(for item: Item,
          in layout: GridLayout) -> some View {
    let index = items.firstIndex(matching: item)
    return viewForItem(item)
        .frame(width: layout.itemSize.width,
               height: layout.itemSize.height)
        .position(layout.location(ofItemAt: index))
    }
}

extension Array where Element: Identifiable {
    func firstIndex(matching: Element) -> Int {
        for index in 0..

```

`func firstIndex(matching: Element) -> Int {`

- This is correct and this does the right thing, but I actually don't like the naming that I've chosen here.
 - One thing about this is it doesn't actually return the index of this item.
 - It returns the first one that it finds.
 - Going through the array from zero on up, and when it finds one, it returns it.
 - So, this is really the **first index** of this item.

`func firstIndex(matching: Element) -> Int {`

- I also don't like the word `Item`.
 - In fact this is the **first index matching** that element.

`let index = items.firstIndex(matching: item)`

- Now we're asking the `array` to do the search.

Now we have to make some changes to the Model:

```

struct MemoryGame<CardContent> {
    var cards: Array<Card>

    mutating func choose(_ card: Card) {
        print("card chosen: \(card)")

        let chosenIndex: Int = cards.firstIndex(matching: card)
        cards[chosenIndex].isFaceUp = !cards[chosenIndex].isFaceUp
    }

    func index(of card: Card) -> Int {
        ...
    }
}

```

`func index(of card: Card) -> Int {`

- We no longer need this function.

30:45 Because this extension has nothing to do with the Grid struct, we are going to move this extension in its own file.

File | new | File... | iOS | Swift File

- Save as: **Array+Identifiable**
 - When we do extensions like this, we usually call the file Array + some sort of descriptor of what kind of thing this is.
 - So, this is kind of identifiable.

```
import Foundation

extension Array where Element: Identifiable {
    func firstIndex(matching: Element) -> Int {
        for index in 0..

```

```
extension Array where Element: Identifiable {
```

- Arrays that don't have identifiable elements don't see this function.

```
return 0
```

- I'm going to come back to the slides, and we're going to fix the **return 0**.
 - We need enums and optional types to do this.

enum

Another variety of data structure in addition to struct and class

It can only have discrete states ...

```
enum FastFoodMenuItem {
    case hamburger
    case fries
    case drink
    case cookie
}
```

An enum is a value type (like struct), so it is copied as it is passed around

It can only have discrete states...

- For example, the `FastFoodMenuItem` **enum** is either a **hamburger**, or it's **fries** or it's a **drink** or it's a **cookie**.
 - It can't be anything else.
 - It has to be one of these four things.**

4.2.1 enum – Associated Data

Associated Data

Each state can (but does not have to) have its own “associated data” ...

```
enum FastFoodMenuItem {
    case hamburger(numberOfPatties: Int)
    case fries(size: FryOrderSize)
    case drink(String, ounces: Int) // the unnamed String is the brand, e.g. "Coke"
    case cookie
}
```

Note that the drink case has 2 pieces of associated data (one of them “unnamed”)

In the example above, FryOrderSize would also probably be an enum, for example ...

```
enum FryOrderSize {
    case large
    case small
}
```

Each state can (but does not have to) have its own “associated data”

- What's cool about **enum** in Swift, unlike most other languages, is that each **of the discrete values can have some associated data with it** that's very specific to that particular discrete value.

Case `drink(String, ounces: Int)`

- Notice that **String** is **unnamed** right here.
 - This is a **tuple**.
 - So, all the rules of **tuples** which allow you to have labels or not labels, and have as many items as you want, **are valid as associated data values**.

4.2.2 enum – Setting the value of an enum

34.25

Setting the value of an enum

Just use the name of the type along with the case you want, separated by dot ...

```
let menuItem: FastFoodMenuItem = FastFoodMenuItem.hamburger(patties: 2)
var otherItem: FastFoodMenuItem = FastFoodMenuItem.cookie
```

```
let menuItem: FastFoodMenuItem =
FastFoodMenuItem.hamburger(numberOfPatties: 2)
```

- We set the value of an `enum` by just giving the name of the `enum` like `FastFoodMenuItem` dot the discrete value like `hamburger(numberOfPatties: 2)`.

```
let menuItem: FastFoodMenuItem =
FastFoodMenuItem.hamburger(numberOfPatties: 2)
```

- When you set the value of an `enum` you **must provide the associated data (if any)**

4.2.3 enum – Setting the value of an enum – Infer the type on one side of the assignment or the other

34.53

Setting the value of an enum

Swift can infer the type on one side of the assignment or the other (but not both) ...

```
let menuItem = FastFoodMenuItem.hamburger(patties: 2)
var otherItem: FastFoodMenuItem = .cookie
var yetAnotherItem = .cookie // Swift can't figure this out
```

```
var yetAnotherItem = .cookie
```

- Swift does not have enough information to infer that we're talking about fast food menu item.

4.2.4 enum – Checking an enum's state

Checking an enum's state

An enum's state is checked with a `switch` statement (i.e. `not if`) ...

```
var menuItem = FastFoodMenuItem.hamburger(patties: 2)
switch menuItem {
    case FastFoodMenuItem.hamburger: print("burger")
    case FastFoodMenuItem.fries: print("fries")
    case FastFoodMenuItem.drink: print("drink")
    case FastFoodMenuItem.cookie: print("cookie")
}
```

Note that we are ignoring the "associated data" above ... so far ...

4.2.5 enum – Checking an enum's state – Check hamburger(patties: 2)

```
var menuItem3 = FastFoodMenuItem.hamburger(patties: 2)
switch menuItem3 {
    case FastFoodMenuItem.hamburger(patties: 3):
        print("burger")
    case FastFoodMenuItem.fries(size: .large):
        print("fries")
    case FastFoodMenuItem.drink("Coke", ounces: 4):
        print("drink")
    case FastFoodMenuItem.cookie:
        print("cookie")
    default:
        print("no menu item")

case FastFoodMenuItem.hamburger(patties: 3):
```

- This code would print “burger” on the console.

4.2.6 enum – Checking an enum's state – Infer inside the switch

```
var menuItem4 = FastFoodMenuItem.fries(size: .large)
switch menuItem4 {
    case .hamburger(patties: 2):
        print("burger")
    case .fries(size: .large):
        print("fries")
    case .drink("Coke", ounces: 4):
        print("drink")
    case .cookie:
        print("cookie")
    default:
        print("no menu item")}
```

```
case .fries(size: .large):
```

- It is **not necessary to use the fully-expressed** `FastFoodMenuItem.fries` **inside the switch** (since Swift can infer the `FastFoodMenuItem` part of that).

4.2.7 enum – break

⌚ break

If you don't want to do anything in a given case, use `break` ...

```
var menuItem = FastFoodMenuItem.hamburger(patties: 2)
switch menuItem {
    case .hamburger: break
    case .fries: print("fries")
    case .drink: print("drink")
    case .cookie: print("cookie")
}
```

This code would print nothing on the console

So, `break` breaks out of the `switch`.

4.2.8 enum – default

37:00

• **default**

A switch must handle ALL POSSIBLE CASES (although you can **default** uninteresting cases) ...

```
var menuItem = FastFoodMenuItem.cookie
switch menuItem {
    case .hamburger: break
    case .fries: print("fries")
    default: print("other")
}
```

- **default** will happen if you didn't list a specific case for something.
- If the **menuItem** were a cookie, the above code would print "other".

4.2.9 enum – Associated data with 'let'

• What about the associated data?

Associated data is accessed through a switch statement using this **let** syntax ...

```
var menuItem = FastFoodMenuItem.drink("Coke", ounces: 32)
switch menuItem {
    case .hamburger(let pattyCount): print("a burger with \u2028(pattyCount) patties!")
    case .fries(let size): print("a \u2028(size) order of fries!")
    case .drink(let brand, let ounces): print("a \u2028(ounces)oz \u2028(brand)")
    case .cookie: print("a cookie!")
}
```

```
var menuItem9a = FastFoodMenuItem.hamburger(patties: 3)
switch menuItem9a {
case .hamburger(let pattyCount):
    print("a burger with \u2028(pattyCount) patties!")
case .fries(let size):
    print("a \u2028(size) order of fries!")
case .drink(let brand, let ounces):
    print("a \u2028(ounces)oz \u2028(brand)")
case .cookie:
    print("a cookie!")
}
```

case .hamburger(let pattyCount):

- Is going to **grab the associated data with hamburger** and assign it to the local variable **pattyCount** that's **only going to be valid for this print statement** or whatever that's happening after this hamburger.
- This code prints a burger with 3 patties!

```

var menuItem9b = FastFoodMenuItem.drink("Coke", ounces: 32)
switch menuItem9b{
case .hamburger(let pattyCount):
    print("a burger with \(pattyCount) patties!")
case .fries(let size):
    print("a \(size) order of fries!")
case .drink(let brand, let ounces):
    print("a \(ounces)oz \(brand)")
case .cookie:
    print("a cookie!")
}

```

```

enum FastFoodMenuItem {
    case hamburger(patties: Int)
    case fries(size: FryOrderSize)
    case drink(String, ounces: Int)
    case cookie
}

enum FryOrderSize {
    case large
    case small
}

```

`case .drink(let brand, let ounces):`

- The `drink` is going to grab the `brand` and the `ounces`.
 - Notice that `ounces` used to have a label when I declared it, but I didn't do that here and that's okay because this is a *tuple* .
 - With *tuples* you can use the labels if you want on both sides, declaring and using it.
 - And so, this is the exact same thing that's going on here with the associated value that you can grab.

Note that the local variable that retrieves the associated data can have a different name
 (e.g. pattyCount above versus patties in the enum declaration)

(e.g. brand above versus not even having a name in the enum declaration)

The associated value is actually just a single value that can, of course, be a *tuple*!

So you can do all the naming tricks of a *tuple* when accessing associated values via switch.

4.2.10 enum – Methods + Computed properties

• Methods yes, (stored) Properties no

An enum can have methods (and computed properties) but no stored properties ...

```
enum FastFoodMenuItem {  
    case hamburger(numberOfPatties: Int)  
    case fries(size: FryOrderSize)  
    case drink(String, ounces: Int)  
    case cookie  
  
    func isIncludedInSpecialOrder(number: Int) -> Bool { }  
    var calories: Int { // switch on self and calculate caloric value here }  
}
```

An enum's state is entirely which case it is in and that case's associated data, nothing more.

You can't have any stored properties.

- All the storage that goes with an *enum* is only the **associated values**,
 - which kind of **makes sense because an enum is a discrete thing**.
 - It really wouldn't make sense to have other data that applies to all of them. That wouldn't be discrete.

4.2.11 enum – Methods – switch self

39:02 If you have a *function* in your *enum*, and it's supposed to essentially tell you something about the *enum*, which is really common to have this kind of functions.

In an enum's own methods, you can test the enum's state (and get associated data) using *self* ...

```
enum FastFoodMenuItem {  
    ...  
    func isIncludedInSpecialOrder(number: Int) -> Bool {  
        switch self {  
            case .hamburger(let pattyCount): return pattyCount == number  
            case .fries, .cookie: return true // a drink and cookie in every special order  
            case .drink(_, let ounces): return ounces == 16 // & 16oz drink of any kind  
        }  
    }  
}
```

```

enum FastFoodMenuItemWithMethods {
    case hamburger(patties: Int)
    case fries(size: FryOrderSize)
    case drink(String, ounces: Int)
    case cookie
}

func isIncludedInSpecialOrder(number: Int) -> Bool {
    switch self {
        case .hamburger(let pattyCount):
            return pattyCount == number
        case .fries, .cookie:
            return true
        case .drink(_, let ounces):
            return ounces == 16
    }
}

var menuItem10a = FastFoodMenuItemWithMethods.hamburger(patties: 2)

if menuItem10a.isIncludedInSpecialOrder(number: 2) {
    print("yes, this menu item is included in special order")
}
// Prints out: yes, this menu item is included in special order

```

`func isIncludedInSpecialOrder(number: Int) -> Bool {`

- This function is going to say whether this `FastFoodMenuItemWithMethods` is included in a certain special order...

`switch self {`

- ...To do that, I'm going to switch on myself and see what I am, to see if I am included in that special order.

```

case .drink(_, let ounces):
    return ounces == 16
}
```

```

var menuItem10b = FastFoodMenuItemWithMethods.drink("Coke",
                                                    ounces: 16)

```

```

if menuItem10b.isIncludedInSpecialOrder(number: 16) {
    print("10b. yes, this menu item is included in special order")
}
// Prints out: yes, this menu item is included in special order

```

`case .drink(, let ounces):`

- Notice that it's getting the associated value in the switch for the `ounces`, but it's ignoring whether it's a Coke or Dr. Pepper.
 - We don't care. It doesn't matter in terms of whether it's included in a special order.

4.2.12 enum – Caselterable - allCases

40:20 *Enums* can do constraints and gains with protocols just like structs and classes can.

- There's a very interesting protocol called **CaseIterable**.
 - With this protocol you gain a static var called **allCases** that returns a collection.

• Getting all the cases of an enumeration

```
enum TeslaModel: CaseIterable {  
    case X  
    case S  
    case Three  
    case Y  
}  
  
Now this enum will have a static var allCases that you can iterate over.  
for model in TeslaModel.allCases {  
    reportSalesNumbers(for: model)  
}  
  
func reportSalesNumbers(for model: TeslaModel) {  
    switch model { ... }  
}
```

```
enum TeslaModel: CaseIterable {  
    case X  
    case S  
    case Three  
    case Y  
}  
  
for model in TeslaModel.allCases {  
    print("model: \(model)")  
}  
// Prints out:  
// model: X  
// model: S  
// model: Three  
// model: Y
```

4.3 Optional

40:57

Optionals are probably the most important *enum*.

➊ Optional

An Optional is just an enum. Period, nothing more.

It essentially looks like this ...

```
enum Optional<T> { // a generic type, like Array<Element> or MemoryGame<CardContent>
    case none
    case some(<T>) // the some case has associated value of type T
}
```

You can see that it can only have two values: is set (some) or not set (none).

In the is set case, it can have some associated data tagging along (of don't care type T).

Where do we use Optional?

Any time we have a value that can sometimes be "not set" or "unspecified" or "undetermined".

e.g., the return type of `firstIndex(matching:)` if the matching thing is not in the Array.

e.g., an index for the currently-face-up-card in our game when the game first starts.

This happens surprisingly often.

That's why Swift introduces a lot of "syntactic sugar" to make it easy to use Optionals ...

4.3.1 Optional – Declaring with the syntax T

➊ Optional

Declaring something of type `Optional<T>` can be done with the syntax `T?`

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
var hello: String?  
var hello: String? = "hello"  
var hello: String? = nil
```

```
var hello: Optional<String> = .none  
var hello: Optional<String> = .some("hello")  
var hello: Optional<String> = .none
```

```
var hello1a: String?  
var hello1b: String? = nil  
var hello1c: Optional<String> = .none
```

```
var hello1d: String? = "hello"  
var hello1f: Optional<String> = .some("hello")
```

```
var hello1a: String?
```

- Declaring something of type `Optional<T>` can be done with the syntax `T?`
- This is declaring that the var `hello1a` is of type `Optional` whose associated value is a `String`.
 - We would call this an **Optional String**.

```
var hello1b: String? = nil
```

- You can assign it the value `nil` (`Optional.none`).
 - So, the keyword `nil` in Swift means `Optional.none`.
 - That means the `none` set case of an Optional is `nil`.

```
var helloId: String? = "hello"
```

- Or you can assign it something of the type `T` (`Optional.some` with associated value that value).
- Swift is smart enough to know that that means set this `optional` to the `some` case and use `hello` as the `associated value`.

```
var hello1a: String?
```

- Note that `optionals` always start out with an implicit `= nil`.
 - This is nice because remember that in structs and classes, all vars have to have an initial value.
 - It's not really uninitialized.
 - It does get initialized to the `none` case.

4.3.2 Optional – Accessing the values

46:02

⌚ Optional

You can access the associated value either by force (with `!`) ...

Or "safely" using `if let` and then using the safely-gotten associated value in `{ }` (`else` allowed too).

```
enum Optional<T> {
    case none
    case some(<T>)
}

let hello: String? = ...
print(hello!)

if let safeHello = hello {
    print(safeHello)
} else {
    // do something else
}

switch hello {
    case .none: // raise an exception (crash)
    case .some(let data): print(data)
}
```



CS193p
Spring 2020

```
let hello2: String? = "Hi"
print(hello2!)
```

```
print(hello2!)
```

- If you put an exclamation point `(!)` after a var that is an optional,
 - it will assume that it's in the `some` case, the set case, and get you the associated value.
 - But if you're not, it crashes your program.
- We call it **force unwrapping**, because you're forcing that thing to be unwrapped and give me that String.
 - It can actually be quite useful in cases where you know this is never supposed to be the case.

```
if let safeHello2 = hello2 {
    print(safeHello2)
} else {
    // do something else
}
```

There is a safe way to do it, and that's by assigning it to a safe variable.

- You do that with `if let`.

4.4.1 Optional – In the function

```
extension Array where Element: Identifiable {  
    func firstIndex(matching: Element) -> Int? {  
        for index in 0..            if self[index].id == matching.id {  
                return index  
            }  
        }  
        return nil  
    }  
}
```

```
func firstIndex(matching: Element) -> Int? {
```

- It's an *optional* whose associated value is an `Int`.

```
return nil
```

- This *optional* `Int`, allows us to return `nil` when we can find it.
 - It's really good at communicating to anyone who calls this, “*I couldn't find this.*”

This is no longer a bogus function.

We can use it in the two places where we call this.

4.4.2 Optional - Calling the function - Grid

I could kind of protect my customers by protecting this code.

```
func body(for item: Item,
          in layout: GridLayout) -> some View {
    let index = items.firstIndex(matching: item)

    return Group {
        if index != nil {
            viewForItem(item)
                .frame(width: layout.itemSize.width,
                       height: layout.itemSize.height)
                .position(layout.location(ofItemAt: index!))
        }
    }
}
```

```
return Group {
```

- **Group** is like a **ZStack** in that its function argument is a **view builder**.
 - However, it doesn't do anything to what's inside of here.
 - It allows you to do the if and then and all that, and you can still list the things, but it does not lay them out.

What does Group do if index is nil?

- It's going to return a group that has some sort of empty content.

To be honest, I really probably wouldn't have gone through all the trouble of doing this.

- I don't think there's really any reason to check if the index is **nil**
 - because it should never be nil...

```
func body(for item: Item,
          in layout: GridLayout) -> some View {
    let index = items.firstIndex(matching: item)
    return viewForItem(item)
        .frame(width: layout.itemSize.width,
               height: layout.itemSize.height)
        .position(layout.location(ofItemAt: index!))
}
```

```
.position(layout.location(ofItemAt: index!))
```

- ...So, I likely would have just let this index force unwrap happen.
 - And if it crashed my app, it crashed my app.
 - It never should, and so I want to find in development, and this code is a lot cleaner without all that.

```
let index = items.firstIndex(matching: item)!
return viewForItem(item)
    .frame(width: layout.itemSize.width,
           height: layout.itemSize.height)
    .position(layout.location(ofItemAt: index!))
}
```

```
let index = items.firstIndex(matching: item)!
```

- The other thing I might do is move the exclamation point from **index!** To where I'm passing this var to up here where I'm actually getting the value of this var.
 - So, get that first index, and immediately force unwrap it.
 - That turns the index variable into an **Int**, and we can pass it directly.

4.4.3 Optional - Calling the function - MemoryGame

57:44 Here we're going to use a little different strategy to get around the error that the optional must be unwrapped

```
struct MemoryGame<CardContent> {
    var cards: Array<Card>

    mutating func choose(_ card: Card) {
        print("card chosen: \(card)")

        if let chosenIndex = cards.firstIndex(matching: card) {
            cards[chosenIndex].isFaceUp = !cards[chosenIndex].isFaceUp
        } else {
            print("Card not found!")
        }
    }

    if let chosenIndex = cards.firstIndex(matching: card) {
```

- Notice that this `if let` allowed us to keep calling `Int` as the type of `chosenIndex`.
 - We don't need that, by the way, because it's going to infer it.

4.5 Play de game

59:08 We have to think conceptually how are we going to play our game.

- I'm going to run through, in my mind, the scenarios.

1. All of my cards are face-down.

- I click on a card and no matching happens then.
 - The card just flips face-up.

2. I got that one card up.

- I click on a second card.
 - That's when I really need to match.

3. If there's two cards face-up now and I press a third card

- I need to essentially turn those other two cards face-down whether they're matched or not.
- The card I just touched on, that's going to be the one that's face-up.

In those three scenarios, the only time I actually played a matching game is if there's one and only one card face-up at the time I touch on a new card.

- Therefore I need to detect that case...

4.5.1 var indexOfTheOneAndOnlyFaceUPCard: Int?

```
struct MemoryGame<CardContent> {
    var cards: Array<Card>

    var indexOfTheOneAndOnlyFaceUPCard: Int?

    mutating func choose(_ card: Card) {
        print("card chosen: \(card)")

        if let chosenIndex = cards.firstIndex(matching: card) {
            cards[chosenIndex].isFaceUp = !cards[chosenIndex].isFaceUp
        } else {
            print("Card not found!")
        }
    }
}
```

var indexOfTheOneAndOnlyFaceUPCard: Int?

- ... I'm going to have a var which keeps track of the index of the one and only face-up card.

4.5.2 ERROR – Use of unresolved identifier 'chosenIndex' – if let with coma

I really only want to pay attention to cards that are **not already face-up**.

- If a card is already face-up and I tap on it, I'm just going to ignore it.

of

```
struct MemoryGame<CardContent> {
    var cards: Array<Card>

    var indexOfTheOneAndOnlyFaceUPCard: Int?

    mutating func choose(_ card: Card) {
        print("card chosen: \(card)")

        if let chosenIndex = cards.firstIndex(matching: card) &&
            !cards[chosenIndex].isFaceUp { ⚡ Use of unresolved identifier 'chosenIndex'
            cards[chosenIndex].isFaceUp = !cards[chosenIndex].isFaceUp
        }
        else {
            print("Card not found!")
        }
    }
}
```

```
if let chosenIndex = cards.firstIndex(matching: card) &&
    !cards[chosenIndex].isFaceUp {
```

- With **if let** we cannot do an “and” like this...

```
struct MemoryGame<CardContent> {
    ...
    if let chosenIndex = cards.firstIndex(matching: card),
        !cards[chosenIndex].isFaceUp {
        cards[chosenIndex].isFaceUp = !cards[chosenIndex].isFaceUp
    }
}
```

```
if let chosenIndex = cards.firstIndex(matching: card),
    !cards[chosenIndex].isFaceUp
```

- ... instead of **&&** we have to use comma , ...

```
if let chosenIndex = cards.firstIndex(matching: card),
    !cards[chosenIndex].isFaceUp,
    !cards[chosenIndex].isMatched {
    cards[chosenIndex].isFaceUp = !cards[chosenIndex].isFaceUp
}
```

```
, !cards[chosenIndex].isMatched
```

... and you can even have more of these, like I might also want to ignore cards that **have already matched** with another card.

4.5.3 ERROR - Equatable - Binary operator '==' cannot be applied

After the if let I know that I've chosen a card that was fade-down and not yet matched.

- So, I just want to see if there's one and only one face-up card right now.

```
mutating func choose(_ card: Card) {  
    print("card chosen: \(card)")  
  
    if let chosenIndex = cards.firstIndex(matching: card),  
        !cards[chosenIndex].isFaceUp,  
        !cards[chosenIndex].isMatched {  
  
        if let potentialMatchIndex = indexOfTheOneAndOnlyFaceUpCard {  
            if cards[chosenIndex].content == cards[chosenIndex].content  
        }  
        cards[chosenIndex].isFaceUp = !cards[chosenIndex].isFaceUp  
    }  
}
```

The code shows a Swift function `choose` that prints the chosen card. It then checks if the chosen card is not face-up and not matched. If so, it finds the index of the only face-up card (which is guaranteed to exist due to the condition). It then compares the content of the chosen card with the content of the face-up card. This comparison triggers two error messages:

- Binary operator '==' cannot be applied to two 'CardContent' operands
- Expected '{' after 'if' condition

- The error says that **Binary operator '==' cannot be applied to two 'CardContent' operands.**
 - Why can't we do equals-equals on CardContent?
 - This is a generic memory game.
 - These are no emoji in here.
 - They're **CardContent**, which is a don't care for us.
 - In fact, **how does equals-equals work in Swift?**

Equals-equals in Swift is not built into the language.

- It uses a feature in Swift called **operators** that lets you associate an operator like this with a function.
- Equals-equals function is a type function that takes two arguments which is the two things on either side of the equals and it returns a bool, whether they're the same or not.
- But not every type has this equals-equals in it.
 - Only some types that can actually check for equality have that.

```

struct MemoryGame<CardContent> where CardContent: Equatable {
    var cards: Array<Card>

    var indexOfTheOneAndOnlyFaceUpCard: Int?

    mutating func choose(_ card: Card) {
        print("card chosen: \(card)")

        if let chosenIndex = cards.firstIndex(matching: card),
           !cards[chosenIndex].isFaceUp,
           !cards[chosenIndex].isMatched {

            if let potentialMatchIndex = indexOfTheOneAndOnlyFaceUpCard {
                if cards[chosenIndex].content ==
                    cards[potentialMatchIndex].content {

                    }

                }

                cards[chosenIndex].isFaceUp = !cards[chosenIndex].isFaceUp
            }
            else {
                print("Card not found!")
            }
        }
    }
}

```

```
struct MemoryGame<CardContent> where CardContent: Equatable {
```

- Luckily, that equals-equals function is in a protocol called **Equatable**.
 - We can use our constraints and gains here to say where our **CardContent** implements **Equatable**.

4.5.4 Equatable - Documentation

Topics

Equatable Requirements

static func == (Self, Self) -> Bool

Returns a Boolean value indicating whether two values are equal.

Required.

Required

- Equatable has **only one function that is required**.
 - It's part of the protocol and there's no default implementation by any extension anywhere, so you must implement this.

```
static func == (self, self) -> Bool
```

- Is a **static** function that takes two arguments and compares them.

4.5.5 Two cards matching

```
struct MemoryGame<CardContent> where CardContent: Equatable {
    var cards: Array<Card>

    var indexOfTheOneAndOnlyFaceUpCard: Int?

    mutating func choose(_ card: Card) {
        print("card chosen: \(card)")

        if let chosenIndex = cards.firstIndex(matching: card),
           !cards[chosenIndex].isFaceUp,
           !cards[chosenIndex].isMatched {

            if let potentialMatchIndex = indexOfTheOneAndOnlyFaceUpCard {
                if cards[chosenIndex].content ==
                    cards[potentialMatchIndex].content {
                    cards[chosenIndex].isMatched = true
                    cards[potentialMatchIndex].isMatched = true
                }
            }

            cards[chosenIndex].isFaceUp = !cards[chosenIndex].isFaceUp
        }
        else {
            print("Card not found!")
        }
    }

    cards[chosenIndex].isMatched = true
    cards[potentialMatchIndex].isMatched = true
```

- We want to turn all the Cards face-down, except for the one we just have chosen.

4.5.6 Turn all the cards face-down

```
struct MemoryGame<CardContent> where CardContent: Equatable {
    var cards: Array<Card>

    var indexOfTheOneAndOnlyFaceUpCard: Int?

    mutating func choose(_ card: Card) {
        print("card chosen: \(card)")

        if let chosenIndex = cards.firstIndex(matching: card),
           !cards[chosenIndex].isFaceUp,
           !cards[chosenIndex].isMatched {

            if let potentialMatchIndex = indexOfTheOneAndOnlyFaceUpCard
            {
                if cards[chosenIndex].content ==
                    cards[potentialMatchIndex].content {
                    cards[chosenIndex].isMatched = true
                    cards[potentialMatchIndex].isMatched = true
                }
                indexOfTheOneAndOnlyFaceUpCard = nil
            }
            else {
                print("Card not found!")
                for index in cards.indices {
                    cards[index].isFaceUp = false
                }
                indexOfTheOneAndOnlyFaceUpCard = chosenIndex
            } // if let potentialMatchIndex
            cards[chosenIndex].isFaceUp = true
        } // if let potentialMatchIndex

    } // mutating func choose(...)
```

4.5.7 EmojiMemoryGameView

When playing the cards that have already match don't turn back up.

- I really want to take this away

```
struct CardView: View {
    var card: MemoryGame<String>.Card

    var body: some View {
        GeometryReader { geometry in
            self.body(for: geometry.size)
        }
    } // body

    func body(for size: CGSize) -> some View {
        ZStack {
            if self.card.isFaceUp {
                RoundedRectangle(cornerRadius: cornerRadius)
                    .fill(Color.white)
                RoundedRectangle(cornerRadius: cornerRadius)
                    .stroke(lineWidth: edgeLineWidth)
                Text(self.card.content)
            } else {
                if !card.isMatched {
                    RoundedRectangle(cornerRadius: cornerRadius)
                        .fill()
                }
            }
        } // ZStack
        .font(Font.system(size: fontSize(for: size)))
    }
}
```

```
if !card.isMatched
```

- Notice we don't even need an else here.
 - View builder will send a view in SwiftUI called **EmptyView**.

4.5.8 Refactoring – State in sync

I'm a little concerned that I have to keep this state in sync with my changes to the cards.

- This is kind of an error-prone way to program when you have state in two places.

```
struct MemoryGame<CardContent> where CardContent: Equatable {
    var cards: Array<Card>

    var indexOfTheOneAndOnlyFaceUpCard: Int? {
        get {
            var faceUpCardIndices = [Int]()
            for index in cards.indices {
                if cards[index].isFaceUp {
                    faceUpCardIndices.append(index)
                }
            }
            if faceUpCardIndices.count == 1 {
                return faceUpCardIndices.first
            } else {
                return nil
            }
        } // get
        set {
            for index in cards.indices {
                if index == newValue {
                    cards[index].isFaceUp = true
                } else {
                    cards[index].isFaceUp = false
                }
            }
        } // set
    }

    mutating func choose(_ card: Card) {
        print("card chosen: \(card)")

        if let chosenIndex = cards.firstIndex(matching: card),
           !cards[chosenIndex].isFaceUp,
           !cards[chosenIndex].isMatched {

            if let potentialMatchIndex = indexOfTheOneAndOnlyFaceUpCard {
                if cards[chosenIndex].content ==
                    cards[potentialMatchIndex].content {
                    cards[chosenIndex].isMatched = true
                    cards[potentialMatchIndex].isMatched = true
                }
                cards[chosenIndex].isFaceUp = true
//                indexOfTheOneAndOnlyFaceUpCard = nil
            }
            else {
//                for index in cards.indices {
//                    cards[index].isFaceUp = false
//                }
                indexOfTheOneAndOnlyFaceUpCard = chosenIndex
            } // if let potentialMatchIndex
        } // if let potentialMatchIndex
    } // mutating func choose(...)
```

- Let's use a computed var instead
- Set – If someone sets the `indexOfTheOneAndOnlyFaceUpCard`, we turn all the other Cards face-down.
- Inside this set, there's a special variable called `newValue`.
 - `newValue` is whatever the people set this to.
- Now that this is calculated, I don't have to work so hard to make sure it's kept in sync.
- For example I don't even need to say that `indexOfTheOneAndOnlyFaceUpCard` is `nil`
- For – Kind of similar because I set the one and only face-up card to be this index, I don't need to set all the rest of them face-down.
 - That's going to be automatically done by the setter.

4.5.9 Refactoring – get & set

How can we reduce this code a bit?

```
struct MemoryGame<CardContent> where CardContent: Equatable {
    var cards: Array<Card>

    var indexOfTheOneAndOnlyFaceUpCard: Int? {
        get {
            var faceUpCardIndices = cards.indices.filter { (Int) -> Bool in
                code
            }
        }
    }
}
```

```
struct MemoryGame<CardContent> where CardContent: Equatable {
    var cards: Array<Card>

    var indexOfTheOneAndOnlyFaceUpCard: Int? {
        get {
            cards.indices.filter {
                cards[$0].isFaceUp
            }.only
        } // get
        set {
            for index in cards.indices {
                cards[index].isFaceUp = index == newValue
            }
        } // set
    }
}
```

4.5.10 Refactoring – Create Array+Only extension

Create an extension

File | New File | iOS | Swift File

- Save as: **Array+Only**

```
import Foundation

extension Array {
    var only: Element? {
        count == 1 ? first : nil
    }
}
```

5 Lec05. ViewBuilder + Shape + ViewModifier

The fifth iOS lecture given at Stanford during Spring quarter of 2020 expanded coverage of topics related to drawing on screen including the `@ViewBuilder` construct for expressing a conditional list of Views, the `Shape` protocol for custom drawing and `ViewModifier`, a mechanism for making incremental modifications to Views.

This lecture starts with an aside about access control (marking APIs so that they are revealed to the right “audiences” within an application) which is then applied throughout Memorize.

After the `Shape` protocol is discussed, a “pie shape” is added to the background of Memorize cards in demo in preparation for animating a countdown timer for scoring.

A custom `ViewModifier` is also added to Memorize to “cardify” any View (i.e. make the View appear to be on a card that has a front and a back).

5.1.1

5.1.2

5.1.3

5.1.4

5.1.5

5.1.6

5.1.7

5.1.8

5.1.9

5.1.10

6 Lec06. Animation

Animation was the topic of lecture 6 of Stanford Spring 2020's iOS Application Development course.

After covering some ancillary topics like local, ephemeral state in Views (@State) and property observers, the lecture goes into a deep dive on animation, including implicit vs. explicit animations, transitions, Shape animations, animating ViewModifiers and more.

- These concepts are then installed in the Memorize demo by animating the flipping of cards, creating a new game and giving bonus points for quick matches.

In SwiftUI, any changes to ViewModifiers or Shapes can be animated. Views can animate changes either implicitly (using the `animate` declaration) or explicitly (by wrapping a `withAnimation` function call around code that might cause changes, most notably Intent functions in the ViewModel) and can control the duration and “curve” of the animation.

The “arrival and departure” of Views on screen can also be animated using the transition declaration (which declares which ViewModifiers to use to draw Views before/after they arrive/depart from the screen).

- In Memorize, the Cardify `ViewModifier` and the Pie Shape are both made Animatable and animation is applied throughout.

6.1.1

6.1.2