NOTES

Course

CS193p – Spring 2020
Developing Apps for iOS
Lectures 4-6

Professor

Paul Hegarty

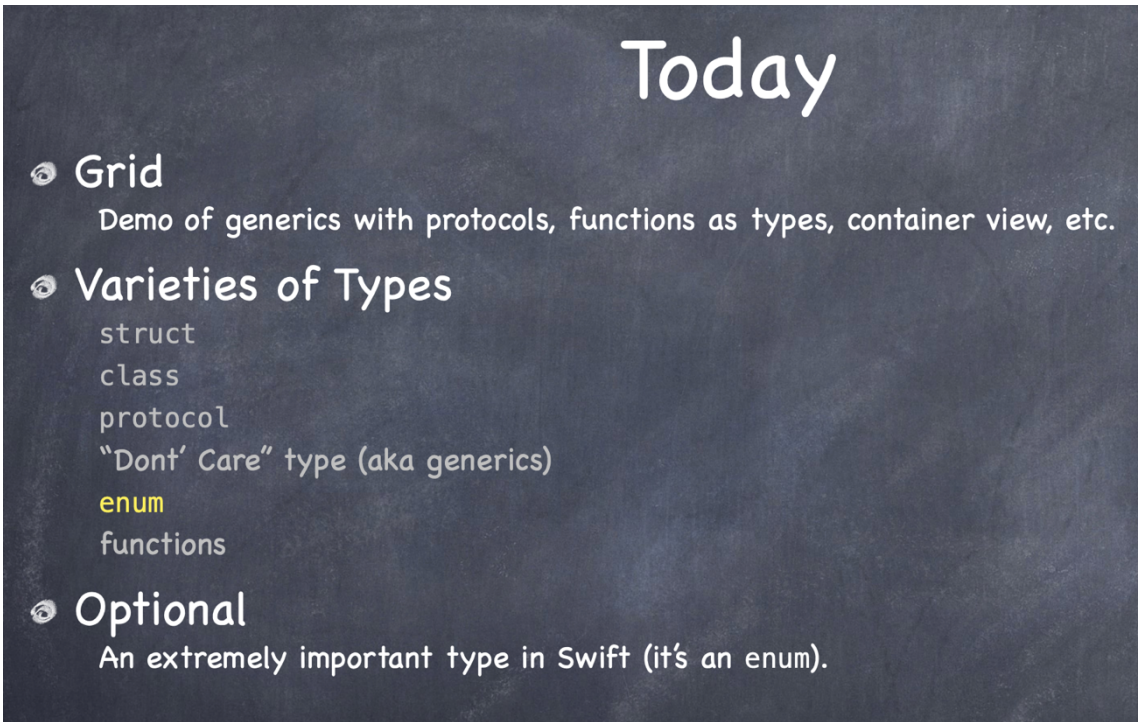
University

Stanford

Version: 1.0
Date: June 26, 2020

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

- 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 SwifUI, we arre 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 las 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

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

- It says, `Assigning non-scaping parameter "ViewForItem" 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!")
  }
}
```

Assigning non-escaping parameter 'viewForItem' to an @escaping closure

Parameter 'viewForItem' is implicitly non-escaping Fix

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***.

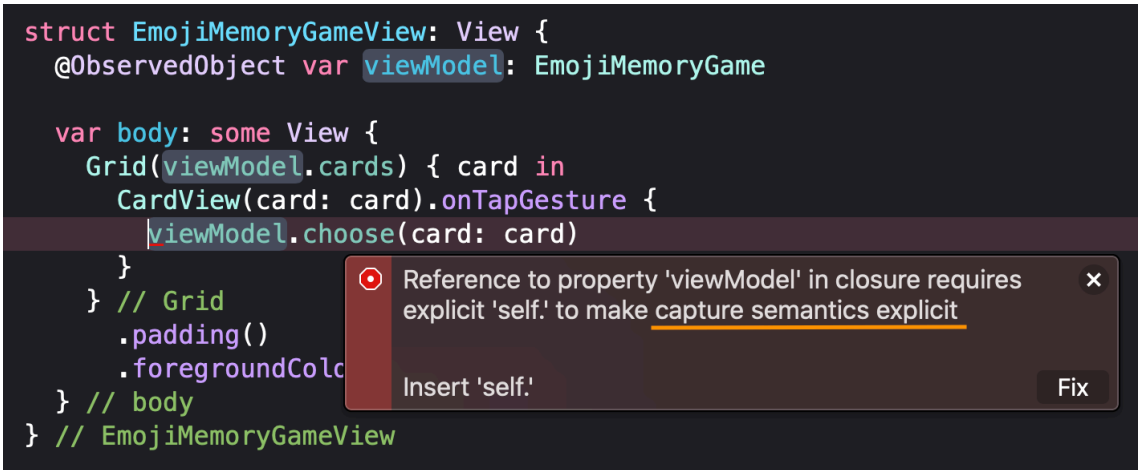
- Just like classes, our ViewModel are reference types that live in the heap.
  - They're stored in memory.
  - People have pointers to them.
  - Same thing with these functions that can be called later.
    - The things inside these closures might also live in the heap.

`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`.



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
26
27
28
29
30
31        self.viewForItem(item)
32      }
33    }
34  }
```

Cannot convert value of type '[Item]' to expected argument type 'Range<Int>'

Return type of property 'body' requires that 'ItemView' conform to 'View'

Cannot convert value of type 'Int' to exp...

`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 constrains 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***.

4.1.8 ERROR – 'ItemView' conform to 'View' - where ItemView: View

```
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
                  Return type of property 'body' requires that
                  'ItemView' conform to 'View'
      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-cares to work.

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 allocate 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..<items.count {
      if items[index].id == item.id {
        return index
      }
    }
    return 0 // TODO: bogus!
  }
}
```

`return 0 // TODO: bogus!`

- This function is kind of double bogus.
  - We're still returning 0.
  - 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.

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`.

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..<numberOfPairsOfCards {
      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 = true
    var isMatched: Bool = false
    var content: CardContent
    var id: Int
  }
```

```
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..<items.count {
      if items[index].id == item.id {
        return index
      }
    }
    return 0 // TODO: bogus!
  }
```

`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..<self.count {
      if self[index].id == item.id {
        return index
      }
    }
    return 0 // TODO: bogus!
  }
}
```

`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 and 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..<self.count {
      if self[index].id == matching.id {
        return index
      }
    }
    return 0 // TODO: bogus!
  }
}
```

`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 facts 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.

4.1.18 Search an item - Extension Array where Element: Identifiable { - Array+Identifiable.swift

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..<self.count {
      if self[index].id == matching.id {
        return index
      }
    }
    return 0 // TODO: bogus!
  }
```

`extension Array where Element: Identifiable {`

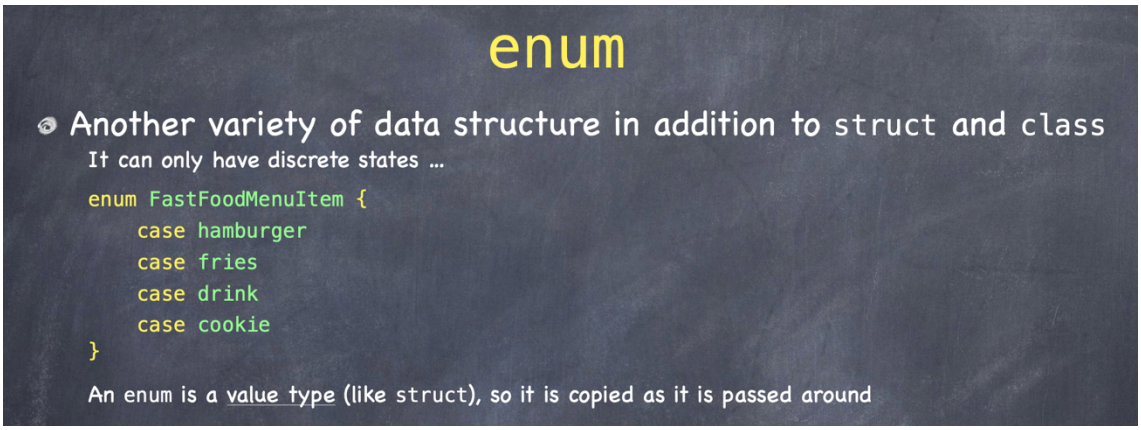
- Arrays that don't have identifiable elements don't see this function.

`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.

4.2 enum

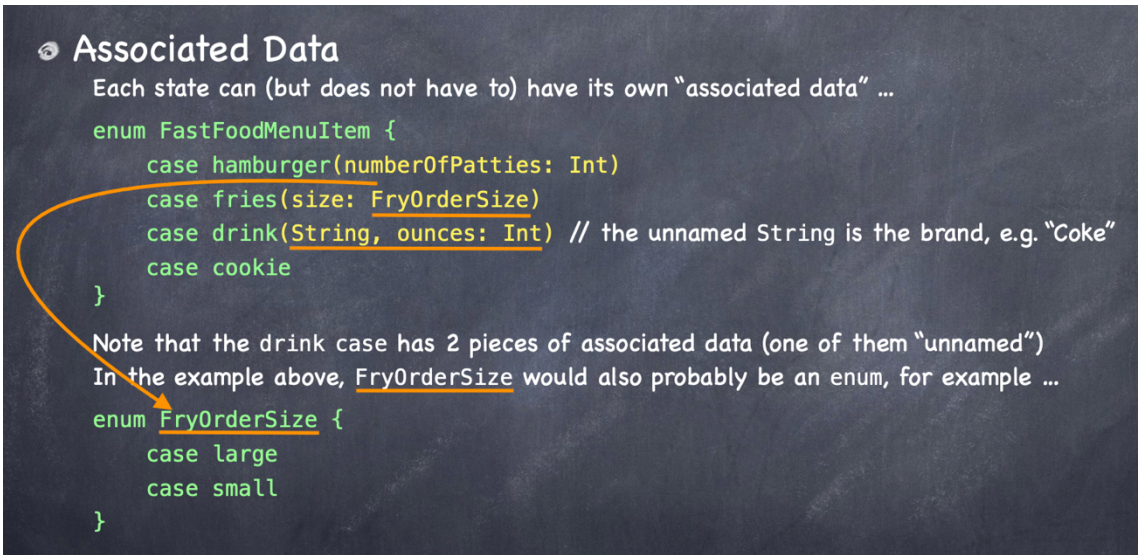
32:49



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



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

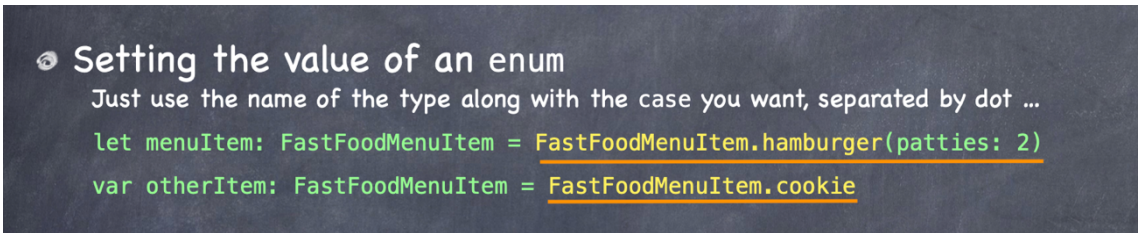
- What's cool about *enum* in Swift, unlike most other languages, is that each of the discrete values can have some ***associated data*** with it that's very specific to that particular discrete value.

`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




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

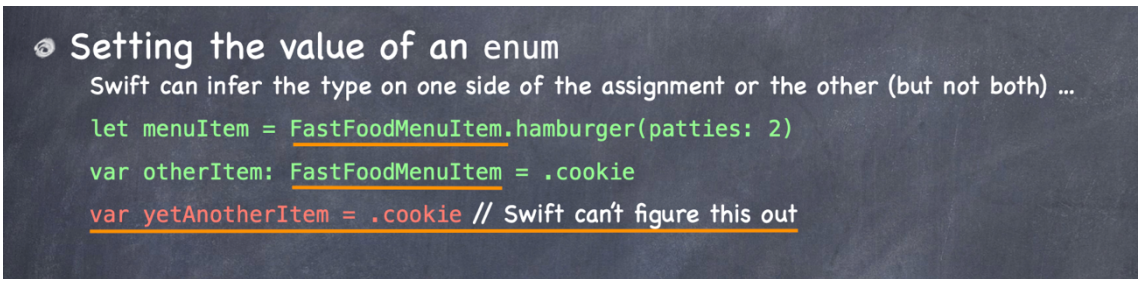
- We set the value of an *enum* by just giving the name of the *enum* like `FastFoodMenuItem` dot the discrete value like `hamburger(numberOfPatties: 2)`.

```
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 assignmetn or the other

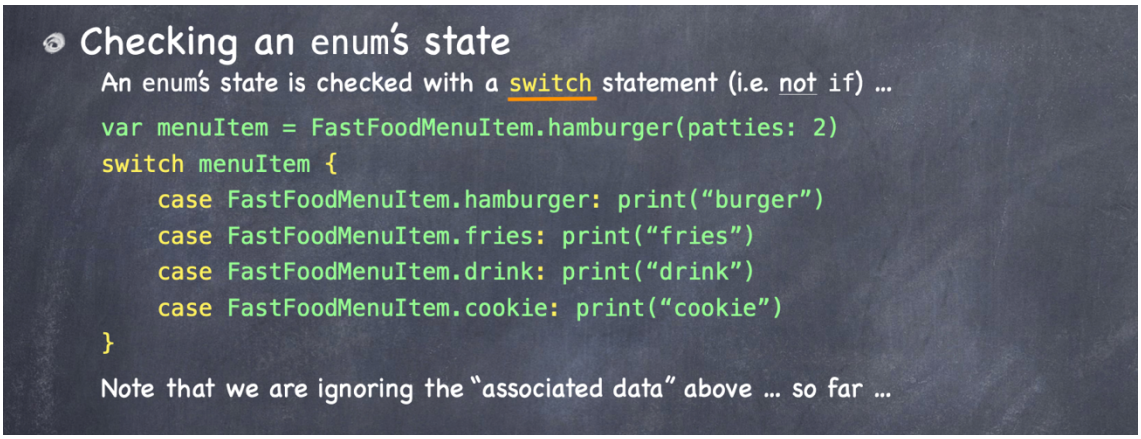
34:53




```
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



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.

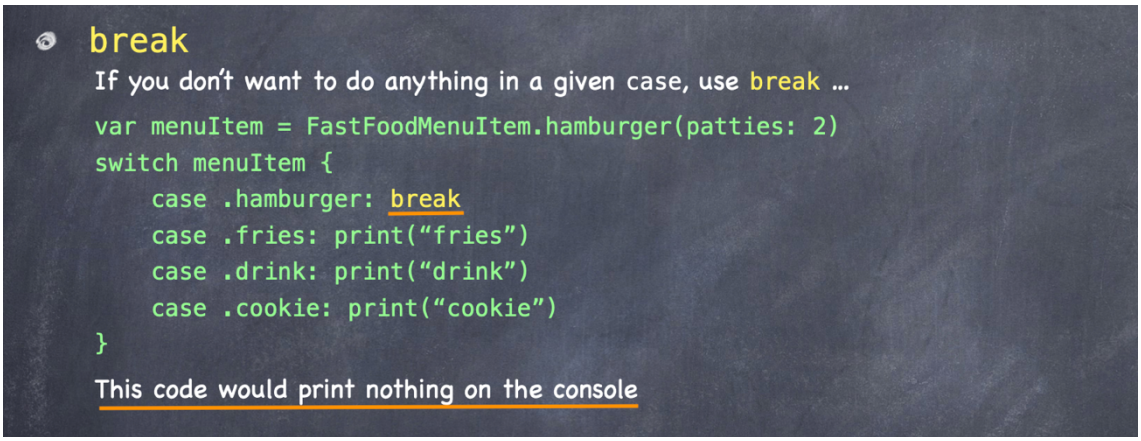
4.2.6 enum – Checking an enum's state - Infer inside the switch

```
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.

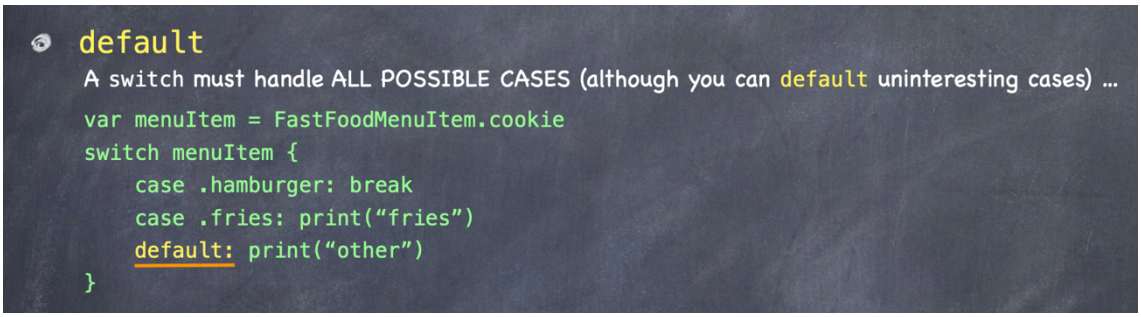
4.2.7 enum – break



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

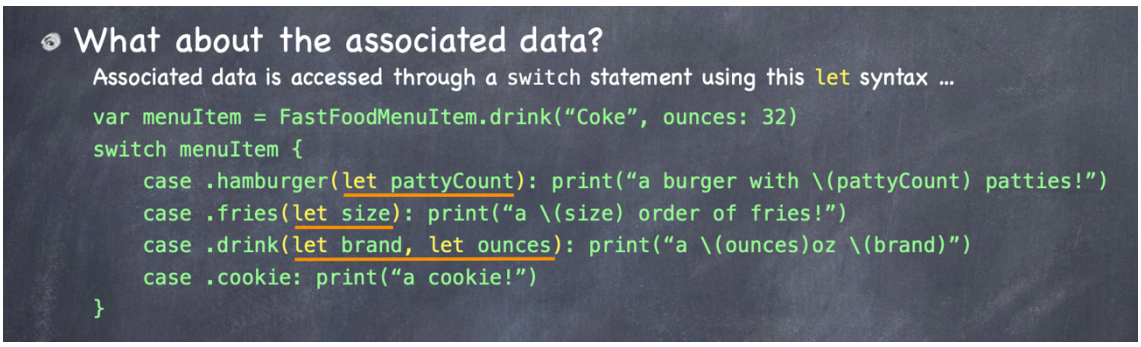
4.2.8 enum – default

37:00



- `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'




```
var menuItem9a = FastFoodMenuItem.hamburger(patties: 3)
switch menuItem9a {
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!")
}
```

`case .hamburger(let pattyCount):`

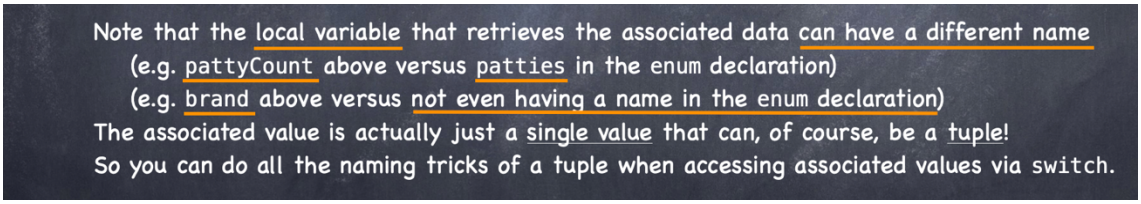
- Is going to grab the associated data with hamburger and assign it to the local variable `pattyCount` that's only going to be valid for this print statement or whatever that's happening after this hamburger.
- This code prints a burger with 3 patties!

```
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!")
}
```

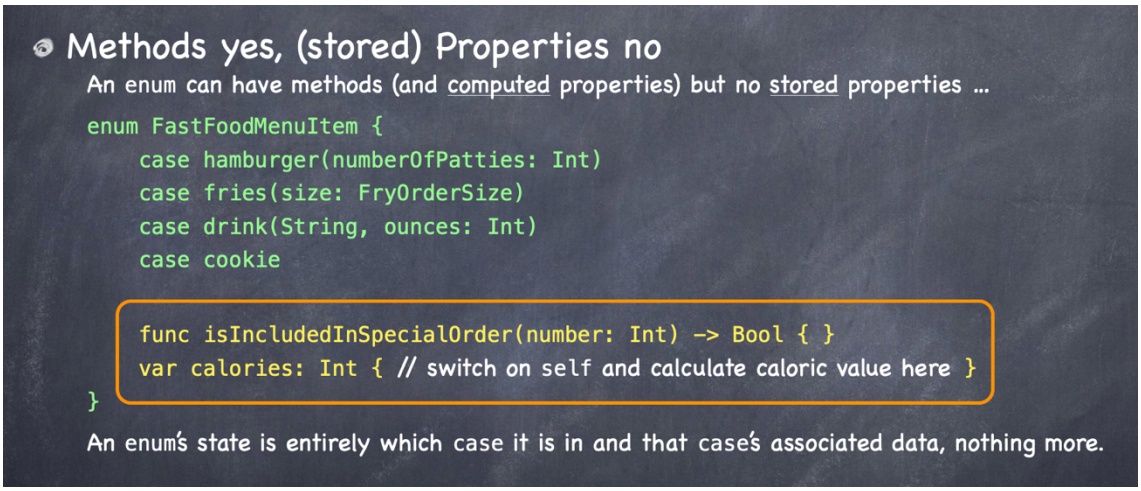
```
enum FastFoodMenuItem {
  case hamburger(patties: Int)
  case fries(size: FryOrderSize)
  case drink(String, ounces: Int)
  case cookie
}

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.



4.2.10 enum – Methods + Computed properties

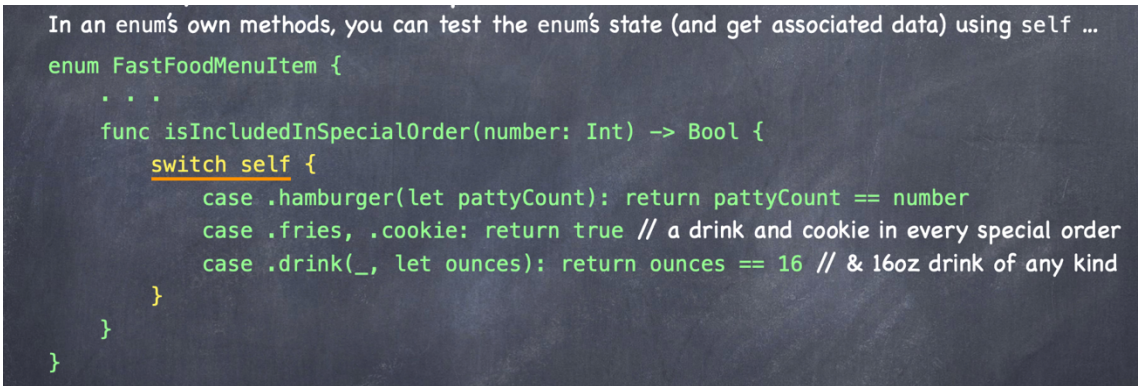


You can't have any stored properties.

- 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.



```
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
```

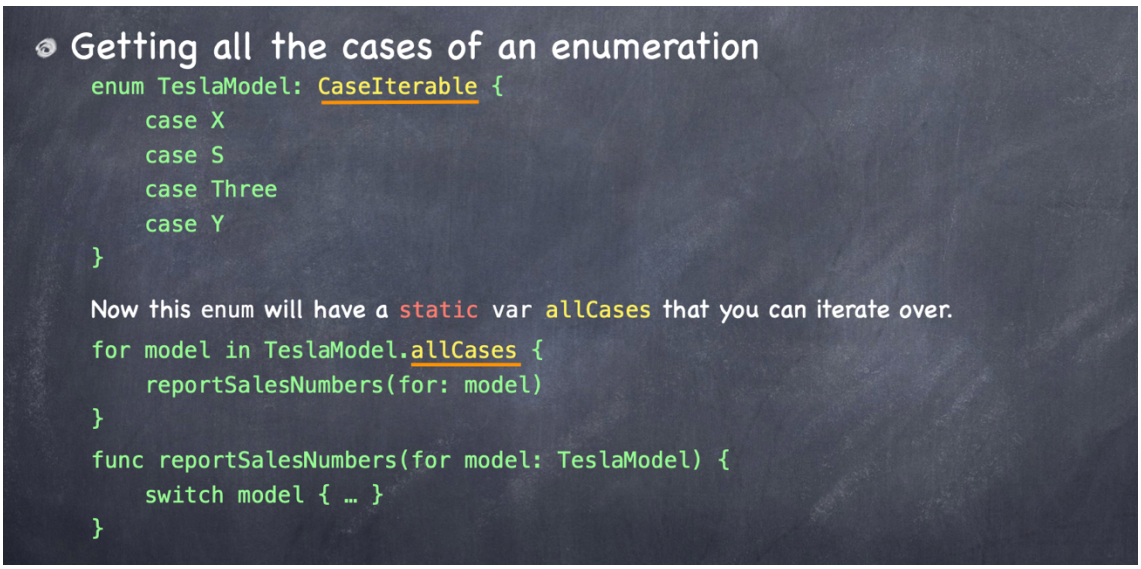
`case .drink(_, let ounces):`

- 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 – CaseIterable - allCases

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

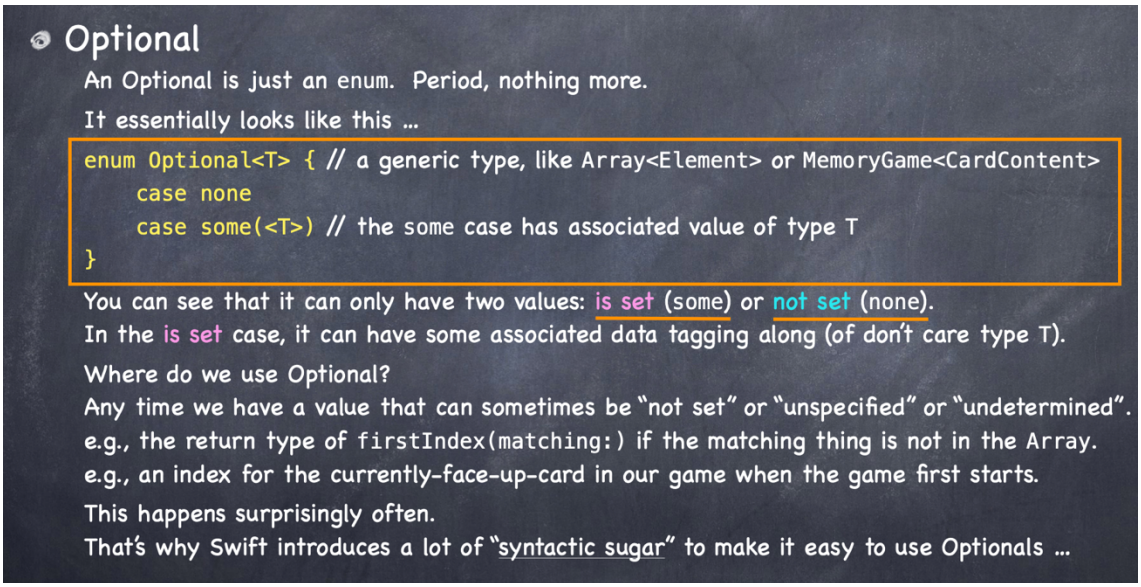
- There's a very interesting protocol called `CaseIterable`.
  - With this protocol you gain a static var called `allCases` that returns a collection.



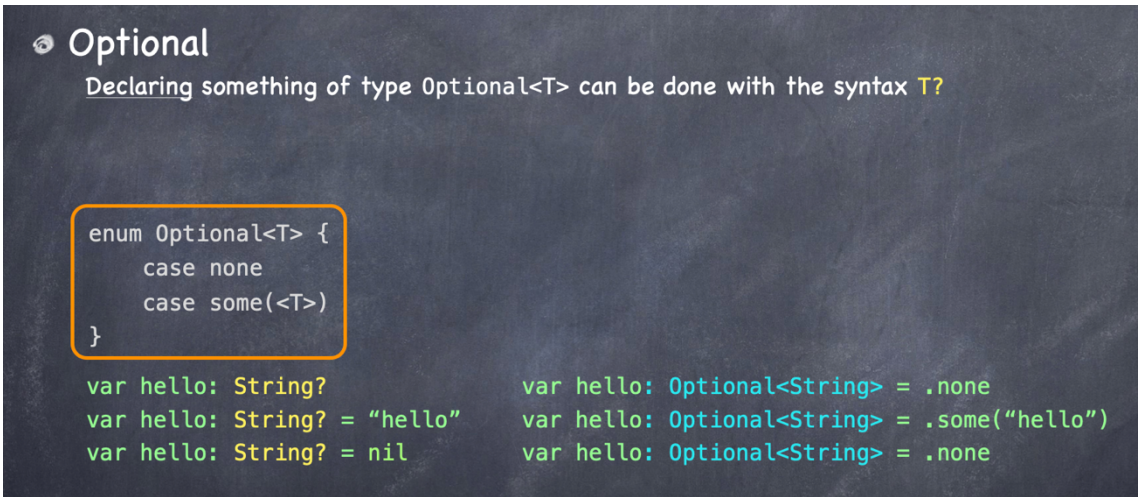

```swift
enum TeslaModel: CaseIterable {
  case X
  case S
  case Three
  case Y
}

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*.



4.3.1 Optional – Declaring with the syntax T




```
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 war `hello1a` is of type `Optional` whose associated value is a `String`.
    - We would call this an ***Optional String***.

`var hello1b: String? = nil`

- You can the 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 hello1d: String? = "hello"`

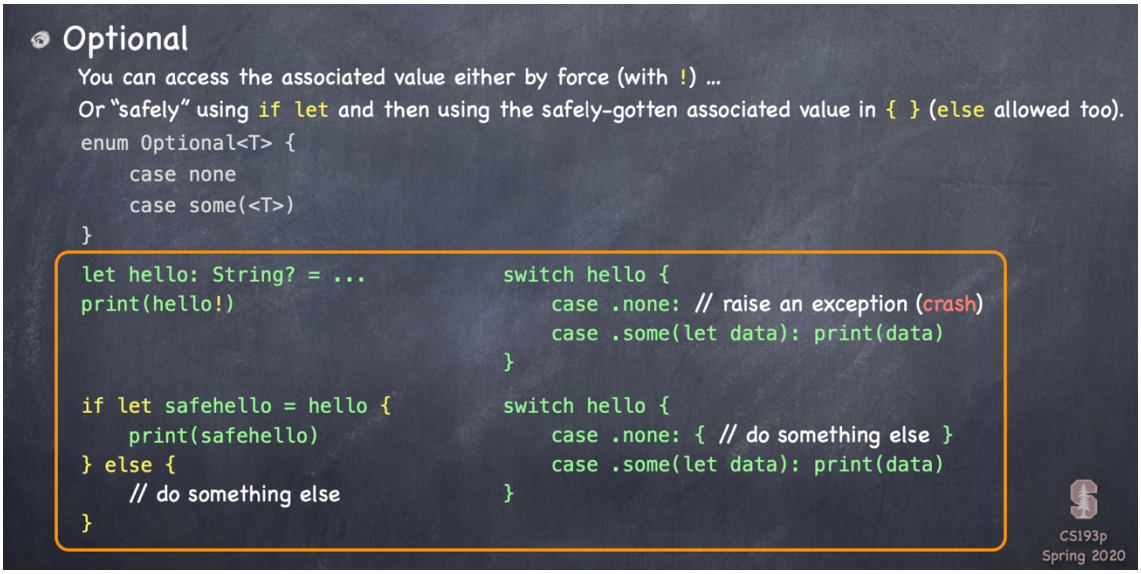
- Or you can assign it something of the type `T` (`Optional.some` with associated value that value).
- Swift is smart enough to know that that means set this ***optional*** to the `some` *case* and use hello as the *associated value*.

`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




```
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 I cases where you know this is never supposed to be the case.

```
if let safeHello2 = hello2 {
  print(safeHello2)
} else {
  // do something else
```

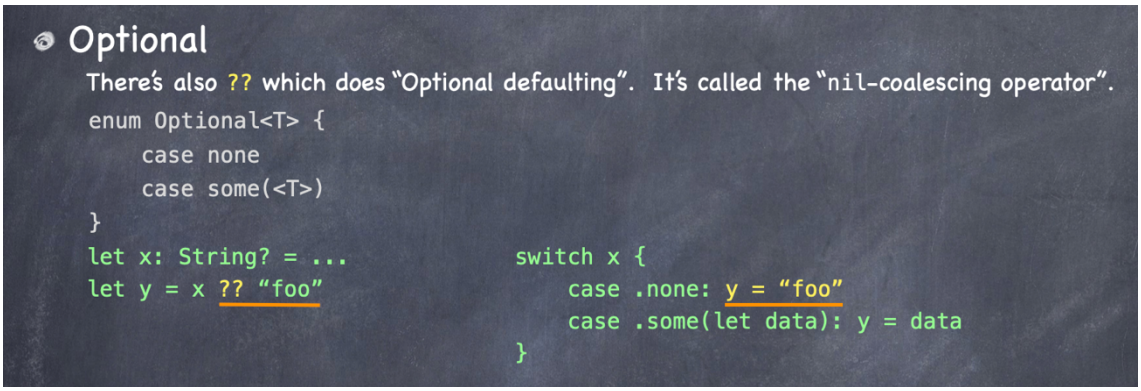
There is a safe way to do it, and that's by assigning it to a safe variable.

- You do that with if let.

4.3.3 Optional – Optional defaulting

48:35 Another syntactic sugar is the optional Defaulting

- This allows you to really simply provide a default when you are accessing an optional in case that optional is in the not set case, and so it's equal to nil.




```
let x: String? = nil
let y = x ?? "foo"
print("y: \(y)")
// Prints out: y: foo
```

`let x: String? = nil`

- The constant `x` is of type ***optional string*** and I set it so `nil`.

`let y = x ?? "foo"`

- If `x` is `nil`, use foo.

4.4 Back to the demo

50:04

4.4.1 Optional – In the function

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

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

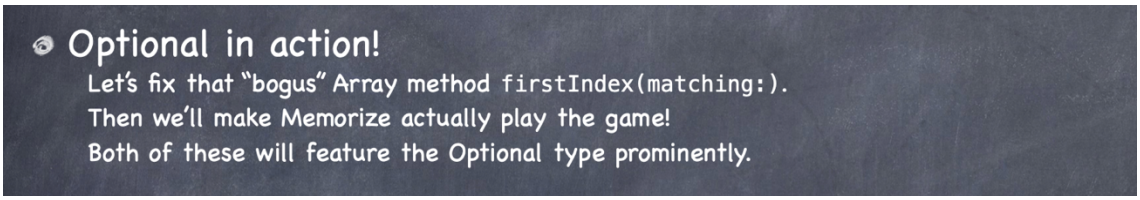
57:44 Here we're going to use a little different strategy to get around the error that the optional must be unwrapped

```
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.

4.4.4 Optional – Optionals in action!



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…

```
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 'choosenIndex' – 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
    }
```

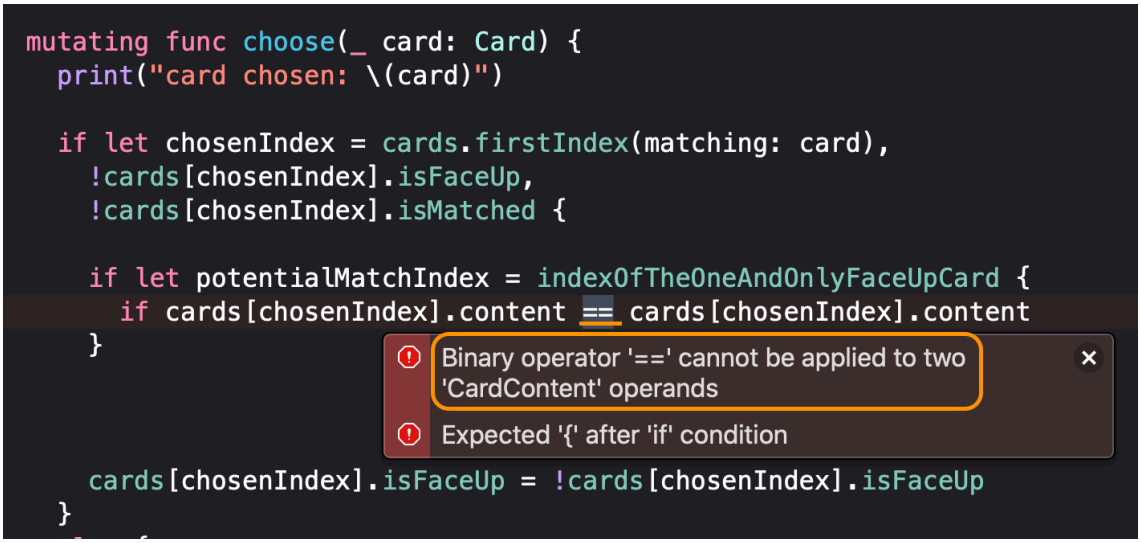
```
, !cards[chosenIndex].isMatched
```

… 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.



- 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 cate for us.
            - In fact, ***how does equals-equals work in Swift?***

Equals-equals in Swift is not built into the language.

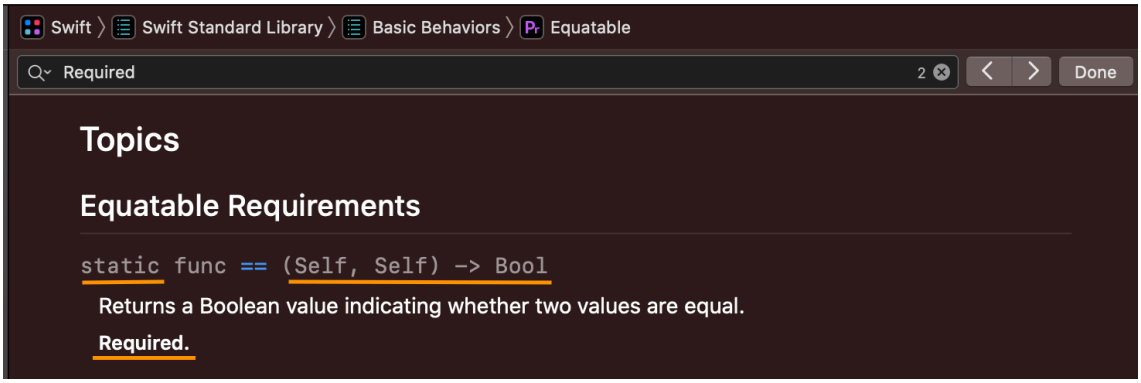
- It uses a feature in Swift called ***operators*** that lets you associate an operator like this with a function.
- Equals-equals function is a type function that takes two arguments which is the two things on either side of the equals and it returns a bool, whether they're the same or not.
- But not 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 constrains and gains here to say where our `CardContent` implements `Equatable`.

4.5.4 Equatable - Documentation



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.

```
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.

```
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(...)
```

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`.

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
  }
}
```

4.6 Assignment II: Memorize

4.6.1 Required Task 1

Get the Memorize game working as demonstrated in lectures 1 through 4. Type in all the code. Do not copy/paste from anywhere.

4.6.2 Required Task 2

Your game should still shuffle the cards.

4.6.3 Required Task 3

Architect the concept of a "***theme***" into your game.

- A *theme* consists of
  - a name for the theme,
  - a set of emoji to use,
  - a number of cards to show
    - (which, for at least one, but not all themes, should be random),
  - and an appropriate color to use to draw
    - (e.g. orange would be appropriate for a Halloween theme).

4.6.4 Required Task 4

Support at least 6 different themes in your game.

4.6.5 Required Task 5

A new theme should be able to be added to your game with a single line of code.

4.6.6 Required Task 6

Add a "New Game" button to your UI which begins a brand new game. This new game should have a randomly chosen theme. You can put this button anywhere you think looks best in your UI.

4.6.7 Required Task 7

Show the theme's name somewhere in your UI.

4.6.8 Required Task 8

Keep score in your game by giving 2 points for every match and penalizing 1 point for every previously seen card that is involved in a mismatch.

4.6.9 Required Task 9

Display the score in your UI in whatever way you think looks best.

4.6.10 Required Task 10

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

- The cards can have any aspect ratio you'd like.
- This probably will not require any work on your part (that's part of the power of SwiftUI), but be sure to continue to experiment with running on different simulators in Xcode to be sure.

4.6.11 Hint 1

Economy is still (and always) valuable in coding, but this week is not a collection of one-liners (though we do pay homage to the one-liner in Required Task 5).

4.6.12 Hint 2

Be careful to think carefully about

- what parts of your code are Model,
- what parts are ViewModel
- and what parts are the View.

This is a significant part of this assignment.

- Remember, nothing about how the game is *displayed* should be in the Model (it's UI independent), but all of the mechanics about how the game plays should be.

4.6.13 Hint 3

One way to think of the number of cards in a theme is that if it is a number specified in the theme, then it's that number of (pairs of) cards, but if it's not specified somehow, then it's random (but always less than the number of emoji in the theme, of course).

4.6.14 Hint 4

If you think of it that way, what sort of type would be good to represent the number of cards in a theme?

- Swift has a great type for something that is normally specified, but sometimes is not.

4.6.15 Hint 5

Your **New Game button** is allowed to be a Text with an onTapGesture (since you know how to do that) or you can use a ***Button*** (see the documentation for how to use that).

- The difference is that
  - a *Button* automatically adjusts itself to look like what a button is supposed to look like on the platform it is on (iOS, Apple TV, Apple Watch),
  - whereas a ***Text*** is always going to look like a piece of text on all platforms.

Because of this, *Button* is way better.

4.6.16 Hint 6

Example themes:

- animals (🐶 🐔 🦄),
- sports (🏀 🏈 ⚾),
- faces (😀 😢 😉).

4.6.17 Hint 7

A card has “already been seen” only if it has, at some point, been face up and then is turned back face down.

- So tracking “seen” cards is probably something you’ll want to do when you turn a card face down.

4.6.18 Hint 8

If you flipped over a 🐠 + 👻 , then flipped over a ✏️ + 🏀 , then flipped over two 👻 s, your score would be 2 because you’d have scored a match (and no penalty would be incurred for the flips involving 🐠 , ✏️ or 🏀 because they have not (yet) been involved in a mismatch, nor was the 👻 ever involved in a mismatch).

If you then flipped over a 🐠 + 🐼 , then flipped 🏀 + 🐠 , your score would drop 3 full points down to -1 overall because the 🐠 had already been seen (on the very first flip) and subsequently was involved in two separate mismatches (scoring -1 for each mismatch) and the 🏀 was also involved in a mismatch after already having been seen (-1).

If you then flip 🐠 + 🐠 , you’d get 2 points for a match and be back up to 1 total point.

4.6.19 Hint 9

You are allowed to remove all of your code from assignment 1 (since the Required Tasks don’t require you to preserve them).

- You especially will probably want to remove your aspect ratio code until you have the Grid working. But then it would be a great exercise to put it back and if it breaks your application, *go figure out why*.
    - Remember that the order in which modifiers are applied to Views matters in SwiftUI.
    - The slides in Lecture 3 that mention aspectRatio would be a great thing to review if you’re having problems with this.

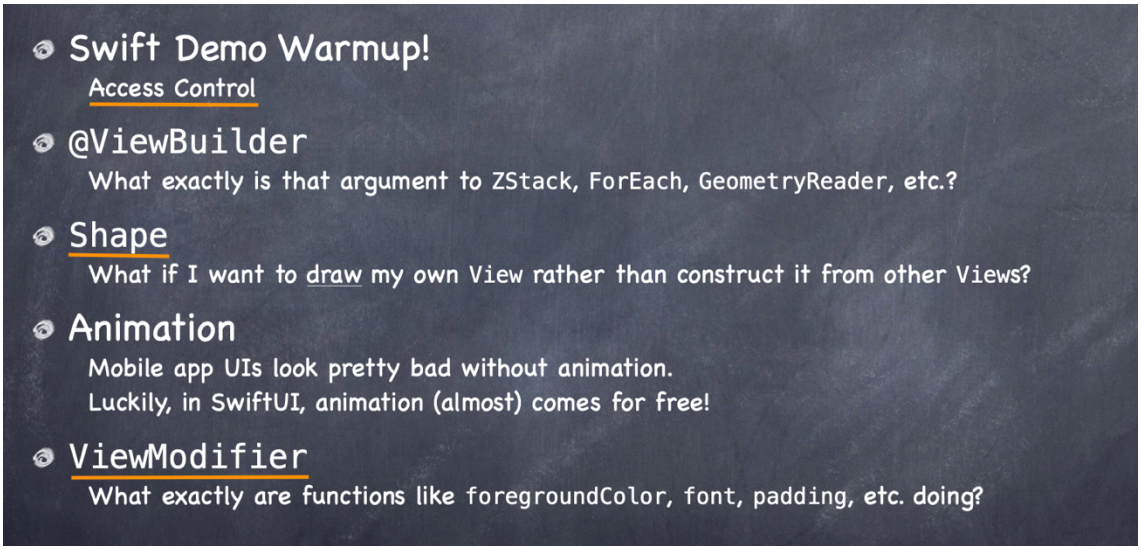
5 Lec05. ViewBuilder + Shape + ViewModifier

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).



Access Control

- Now it's time to apply access control to all Memorize.

Shape

- We will do a demo in Memorize where we add a shape,
  - a little pie that's behind the emoji

ViewModifier

- We're going to dive into exactly what's going on in all that SwiftUI code that you're writing.

5.1 Access control

02:05 The access control that I'm going to do right is something that you're going to have to apply to all your code.

- Following along with Memorize might be a good learning experience.

5.1.1 ViewModel - EmojiMemoryGame

Access control is about controlling the access that different structs have to each other's vars.

```
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)
{ pairIndex in
      return emojis[pairIndex]
    }
  }
```

```
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()`

- By making `model` completely ***private*** in our ViewModel we've force the ViewModel to make the `cards` var and the `choose(card:)` intent to be accessible.

```
struct MemoryGame<CardContent> where CardContent: Equatable {
  private(set) var cards: Array<Card>
...
  struct Card: Identifiable {
    var isFaceUp: Bool = false
    var isMatched: Bool = false
    var content: CardContent
    var id: Int
  }
}
```

`private(set) var cards: Array<Card>`

- We can't really have this be private, but we can use a different access control level which is `set`.
  - This means ***setting*** is *private* but ***reading it is not***.
  - That's exactly what we want in this case.
    - We want the ViewModel to look the Model's cards so they can display them in the UI.
    - However, when it comes to changing the cards, we definitely want that reserved for our self.

```
  private var indexOfTheOneAndOnlyFaceUpCard: Int? {
    get {
      cards.indices.filter {
        cards[$0].isFaceUp
      }.only
    } // get
    set {
      for index in cards.indices {
          cards[index].isFaceUp = index == newValue
      }
    } // set
```

- Access control can also be set on our functions and our own internal computed vars.
- We certainly wouldn't want anyone setting this, that could totally mess us up.
  - And we really don't even want someone looking at it because if they start looking at it, they might depend on it.

```
var isFaceUp: Bool = false
var isMatched: Bool = false
var content: CardContent
var id: Int
```

- Because `cards` array is already *private set*, when someone calls this array they're getting a copy that is read only.
  - So, since there's no way for anyone to get a writeable card, it's perfectly fine for us to leave these vars *non-private*.

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


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

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

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

- We likely do not want the View to be creating memory games.
  - There's no way that views could take responsibility for creating a Model.

5.1.4 View - EmojiMemoryGameView

07:52

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

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

  private func body(for size: CGSize) -> some View {
…
  }

  private let cornerRadius: CGFloat = 10.0
  private let edgeLineWidth: CGFloat = 3
  private func fontSize(for size: CGSize) -> CGFloat {
    min(size.width, size.height) * 0.75
  }
} // CardView
```

`private let cornerRadius: CGFloat = 10.0`

- There's really no reason for anyone to be accessing this drawing constants and function.

`var body: some View {`

- This variable has to be *non-private* because the system is going to call it.

`private func body(for size: CGSize) -> some View {`

- This helper function that we call from inside our geometry reader there's no reason to be public.

`var card: MemoryGame<String>.Card`

- This variable is public because when we create a `CardView`, we're required to give it an initial value.

`@ObservedObject var viewModel: EmojiMemoryGame`

- This has to be also public, because for example when we create our previewer, we're specifying our view model and also in our SceneDelegate, when we're creating the main view of our window of our app

5.1.5 View - Grid

09:22

```
struct Grid<Item, ItemView>: View where Item: Identifiable,
ItemView: View {
  private var items: [Item]
  private 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))
    }
  }

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

  private 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))
  }

}
```

```
private var items: [Item]
private var viewForItem: (Item) -> ItemView
```

- Since we have an *initializer* to initialize them, they don't need to be *public*.

```
struct GridLayout {
  private(set) var size: CGSize
  private(set) var rowCount: Int = 0
  private(set) var columnCount: Int = 0

  init(itemCount: Int,
       nearAspectRatio desiredAspectRatio: Double = 1,
       in size: CGSize) {
…
  } // init

  var itemSize: CGSize {
    if rowCount == 0 || columnCount == 0 {
      return CGSize.zero
    } else {
      return CGSize(
        width: size.width / CGFloat(columnCount),
        height: size.height / CGFloat(rowCount)
      )
    }
  } // itemSize

  func location(ofItemAt index: Int) -> CGPoint {
    if rowCount == 0 || columnCount == 0 {
      return CGPoint.zero
    } else {
      return CGPoint(
        x: (CGFloat(index % columnCount) + 0.5) * itemSize.width,
        y: (CGFloat(index / columnCount) + 0.5) * itemSize.height
      )
    }
  } // location
} // GridLayout
```

```
var itemSize: CGSize {
func location(ofItemAt index: Int) -> CGPoint {
```

- This need this var and func to be public otherwise people wouldn't be able to use them.

```
private var size: CGSize
private var rowCount: Int = 0
private var columnCount: Int = 0
```

- Very important to make these vars private (set) because they are calculated from the initializer.
  - Otherwise someone might think that they can reset the size var.
  - Once you initialize the grid layout, it's fixed forever.
  - We want people to know that they can only read these values.
    - This is a case where access control actually helps people understand how to use certain functionality.

5.1.7 Libraries – Array+Identifiable, Array+Only

```
extension Array where Element: Identifiable {
  func firstIndex(matching: Element) -> Int? {
    for index in 0..<self.count {
      if self[index].id == matching.id {
        return index
      }
    }
    return nil
  }
}
```

```
extension Array {
  var only: Element? {
    count == 1 ? first : nil
  }
}
```

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

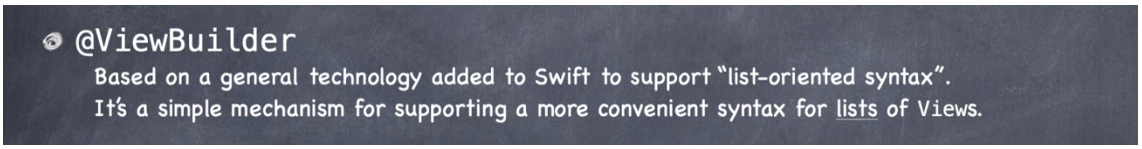
- Obviously, our `firstIndex` func and `only` var , we want it to be *public*.
    - We are adding this functionality to all arrays

5.1.8 Recap

Every time we add new functions, you should definitely be putting *privates* wherever things should be *private*.

5.2 @ViewBuilder

12:03

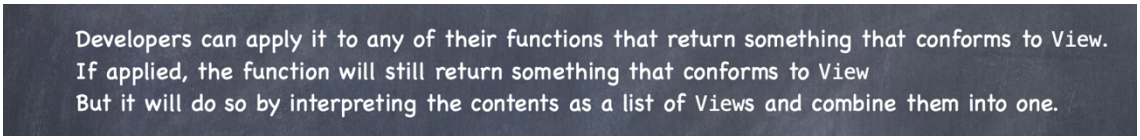


Based on a general technology added to Swift to support "list-oriented syntax".

- There're a lot of things like ***HTML*** that are expressed as a list.
    - A paragraph and then an embedded thing, they're just listed on the page.

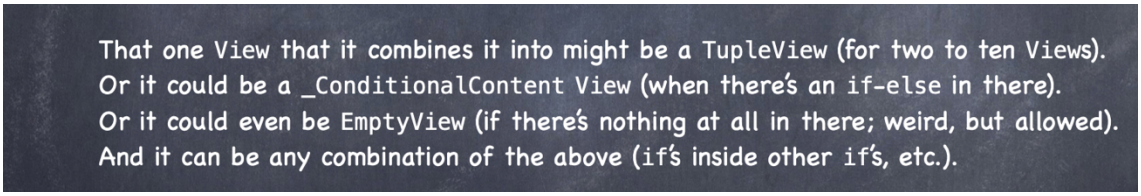
It's a simple mechanism for supporting a more convenient syntax for lists of Views.

- It would be nice in Swift to be able to represent these somewhat native list-oriented syntaxes.
    - Once you have this mechanism then you rapidly realize there're a lot of places where we just want a list of view like the front of our cards.
        - We just want to list the two rounded rectangles and the emoji.
            - View builder lets developers do this and `@ViewBuilder` is the keyword.



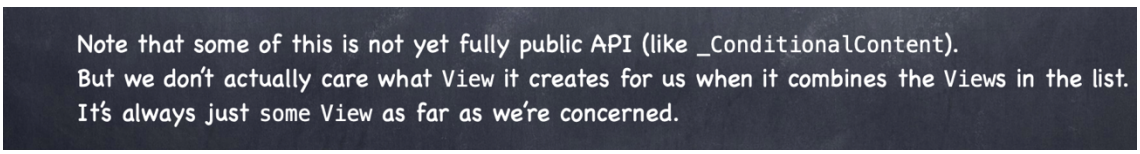
Developer can apply it to any of their functions interpreting the contents as a list of views.

- `@ViewBuilder` can be tagged onto any function that returns some `View`.
    - Then the compiler will interpret what's in the curly braces of that function to be a list of views, instead of just arbitrary code.
- The function that you're tagging with `@ViewBuilder`, it returns *some view* which is a single *view*.
    - So, *view builder* builds that *list of views* into a *single view*.



That one view that it combines it into might be a TupleView (for two to ten views)

- Notice it is limited to 10.
    - It's pretty rare to imagine having more than 10 views just listed straight in a row.
        - You're almost certainly going to be breaking that up with some kind of sub-views like `CardView`s which is the sub-view of our `EmojiGameView`.

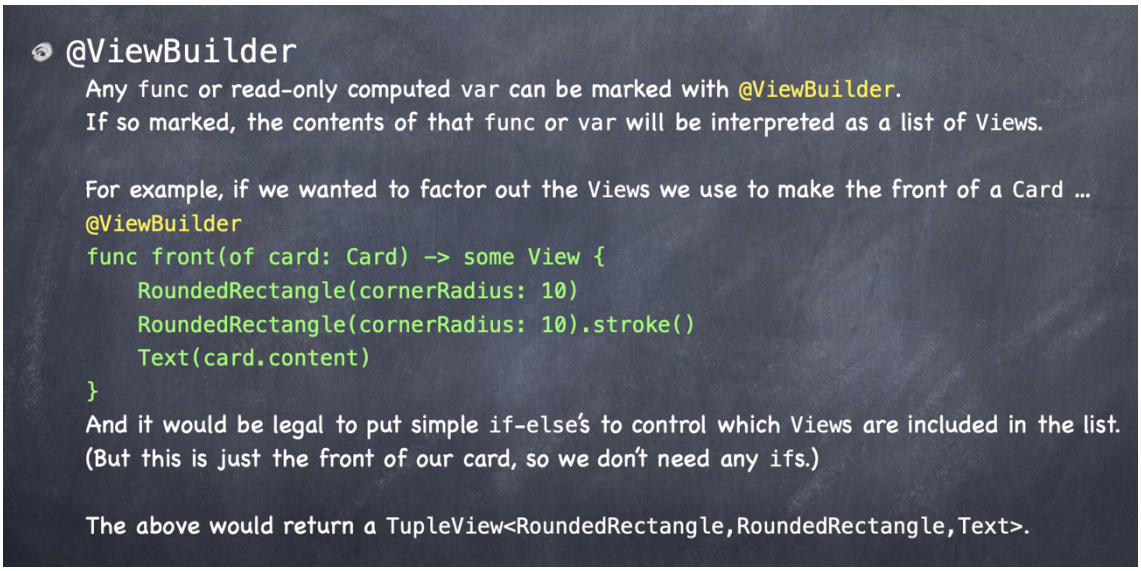


We don't actually care what kind of views these are.

- `TupleViews`, `_ConditionalContent`, whatever.

5.2.1 @ViewBuilder - Example – func front(of:) -> some View

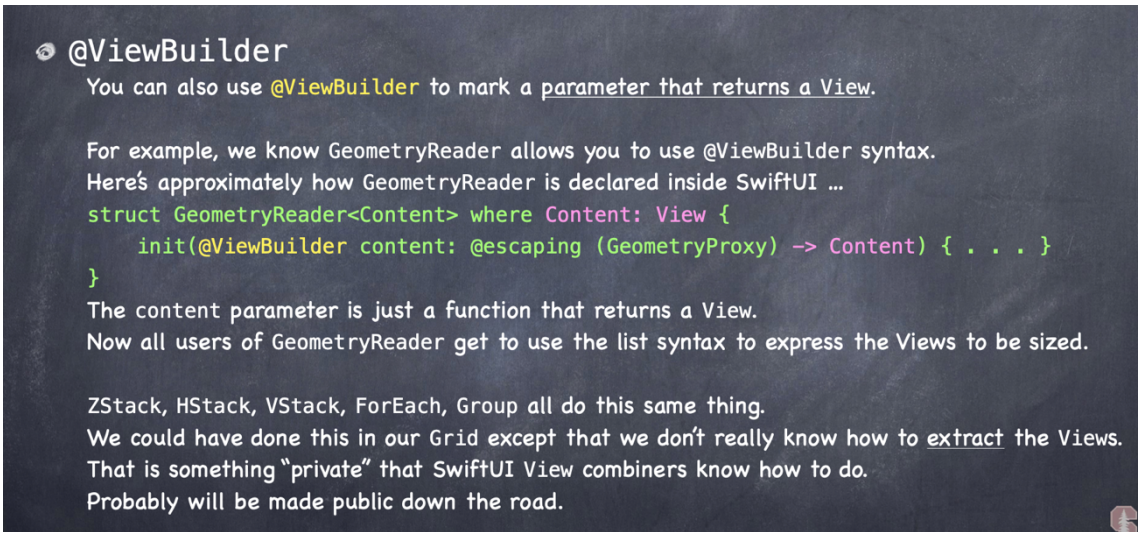
16:00



- In here I would normally have to put it in a group

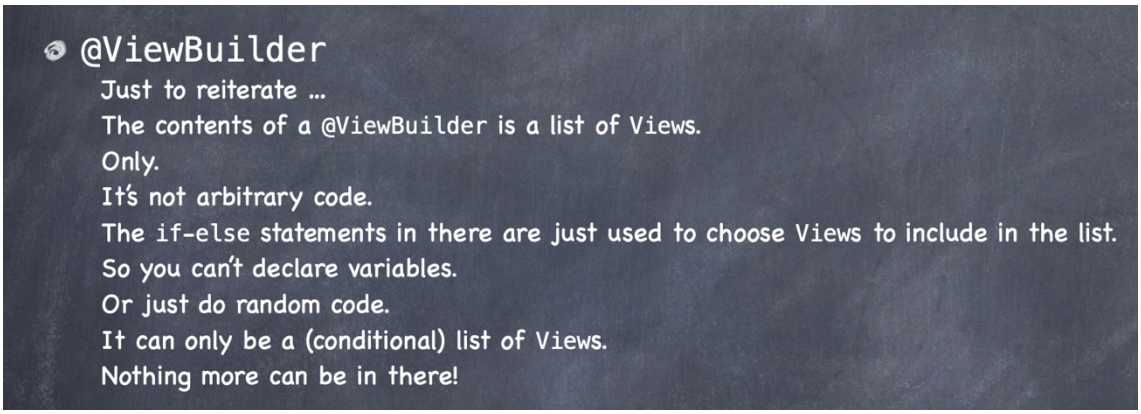
5.2.2 @ViewBuilder – Mark a parameter that returns a view

17:20

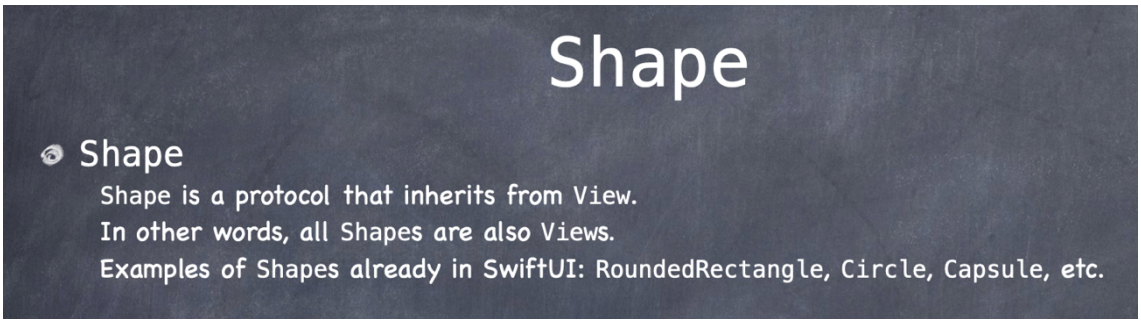


5.2.3 @ViewBuilder - Recap

19:57

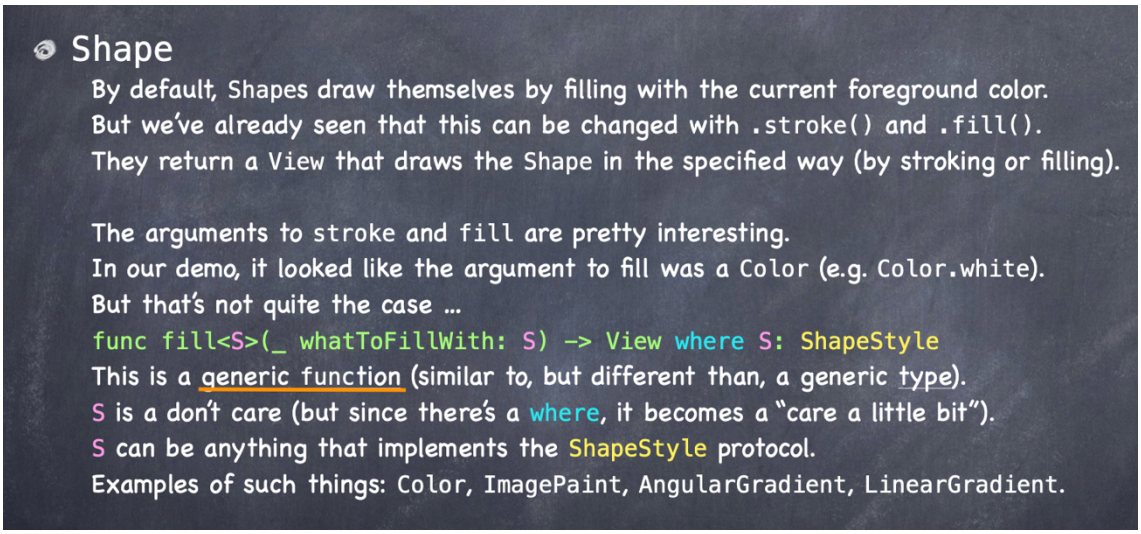


5.3 Shape



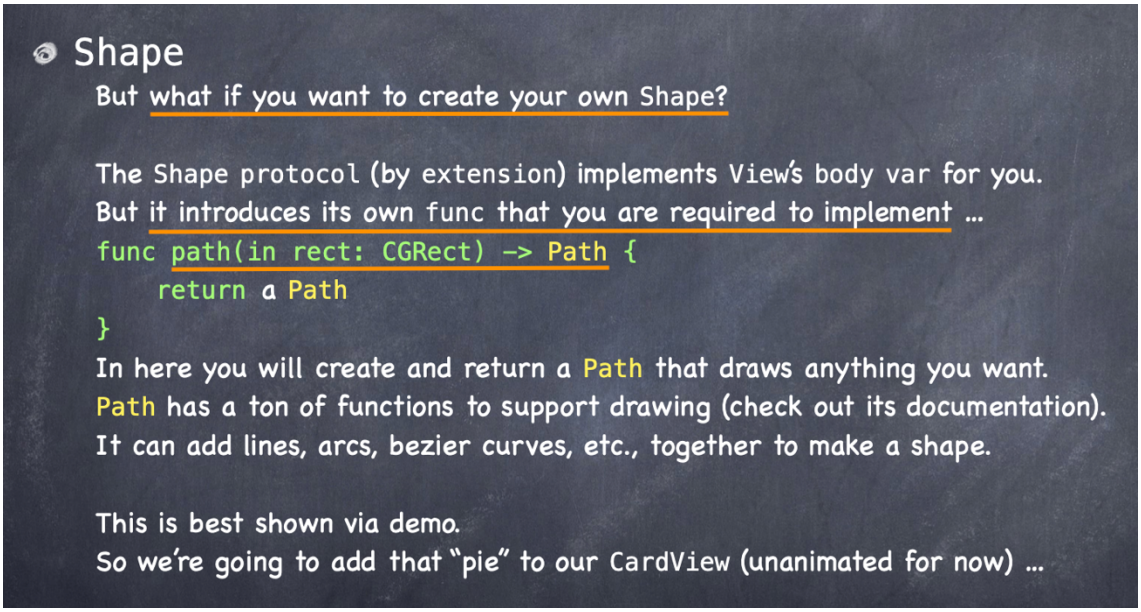
5.3.1 Shape – Generic functions

20:47



5.3.2 Shape – Create your own shape

23:22



5.4 Back to the demo

25:05 I've taken a screenshot of that video of the game in play so that we can see what it is we're shooting for.

We're trying to build this pie chart behind our ghost.

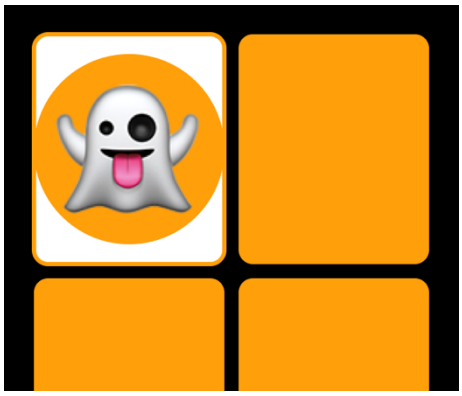
- There's no such shape built in to SwiftUI.

5.4.1 Circle

25:30 Before we start building this custom shape, let's just try and get the circle behind the ghost.

- We know there's a built-in circle.

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

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

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



It's not quite right.

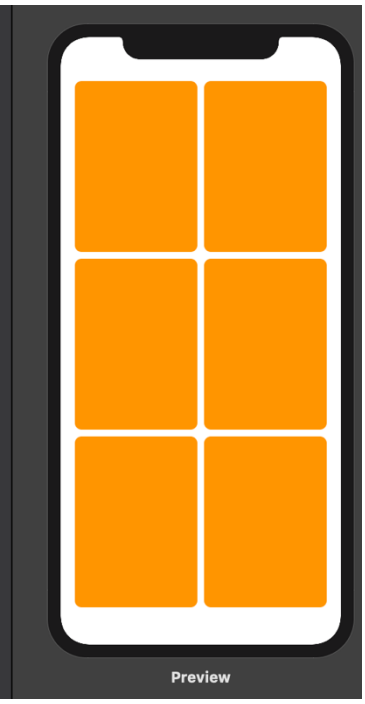
- It's a little too close to the edges, and the color is a little too bright of orange.

5.4.2 Canvas – Show the first card face up

26:40 We can see that in our canvas all the cards are face down.

- I really want at least one of these cards to be face up.

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

//
```




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

`static var previews: some View {`

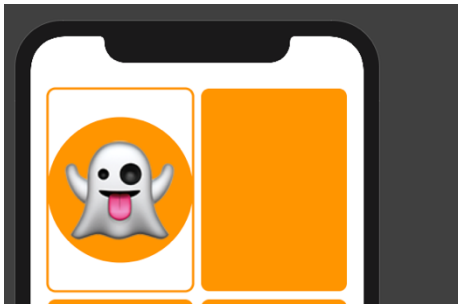
- This is just a *static var* of type `some View` that is returning this whole view to preview our `EmojiMEmoryGameView`.

`EmojiMemoryGameView(viewModel: EmojiMemoryGame())`

- Right now, we're creating an `EmojiMEmoryGameView`, and we're giving this `viewModel` that we just created on the fly.

But there's no reason that we couldn't just do this.

```
struct ContentView_Previews: PreviewProvider {
  static var previews: some View {
    let game = EmojiMemoryGame()
    game.choose(card: game.cards[0])
    return EmojiMemoryGameView(viewModel: game)
  }
}
```



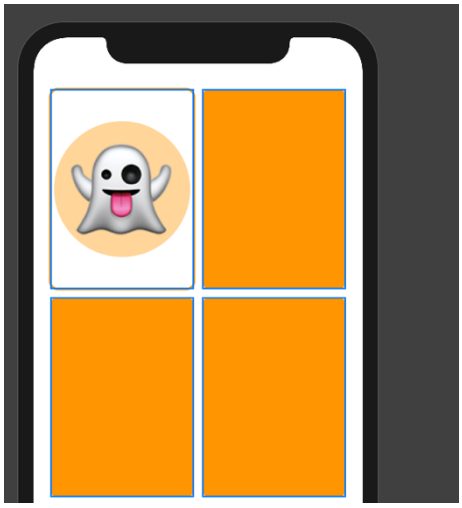
5.4.3 Circle().padding(5).opacity(0.4)

28:14

```
private func body(for size: CGSize) -> some View {
    ZStack {
      if self.card.isFaceUp {
        RoundedRectangle(cornerRadius: cornerRadius)
          .fill(Color.white)
        RoundedRectangle(cornerRadius: cornerRadius)
          .stroke(lineWidth: edgeLineWidth)
        Circle()
          .padding(5)
          .opacity(0.4)
        Text(self.card.content)
      } else {
        if !card.isMatched {
          RoundedRectangle(cornerRadius: cornerRadius)
          .fill()
        }
      }
    } // ZStack
      .font(Font.system(size: fontSize(for: size)))
  }

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

  private func fontSize(for size: CGSize) -> CGFloat {
    min(size.width, size.height) * 0.70
  }
} // CardView
```

`.opacity(0.4)`

- Takes a double between 0, which means completely transparent, to 1, completely opaque.



5.4.4 Struct Pie

29:53 To do our own pie, we just need to replace `Circle` with our own *custom shape*.

File | **New** | **File…** | **iOS** | ***Swift File***

- Save As: ***Pie***

```
import SwiftUI

struct Pie: Shape {

  func path(in rect: CGRect) -> Path {
    var p = Path()

    return p
  }
}
```

`struct Pie: Shape {`

- Conforms to the `Shape` protocol.
    - **Gains**: it's a *view*, can be *filled* and *stroked*.

`func path(in rect: CGRect) -> Path {`

    - **Constraints**: Has to implement this function.
- Returns a ***path*** that we have to create.
    - This *path* is just going to be the edges of what we're drawing.
        - For us it's going to start in the middle and go up, around back to the middle.
            - We're going to build this with functions in `Path` like ***draw a line*** and ***draw an arc***.
            - And once we do, `Shape` we'll take care of all the rest, being able to *fill* it.

`func path(in rect: CGRect) -> Path {`

- What is this `rect` that's passed to us?
    - That is the ***rectangle*** in which we're supposed to fit our shape.
        - Almost all, if not all shapes usually use all the space in the rect that's given to them.
            - Because `Shape` is also a `View` the rect is going to give you here is the space that was offered to it.

5.4.5 Pie - Go to the middle of the rectangle

32:24

```
  func path(in rect: CGRect) -> Path {
    let center = CGPoint(x: rect.midX, y: rect.midY)

    var p = Path()
    p.move(to: center)

    return p
  }
```

`p.move(to: center)`

- I'm just going to go to the middle
- `CG` stands for ***Core Graphics*** .
    - This is the underlying graphics system that all this is built on.

5.4.6 Pie - Starting and ending angle

33:33 I'm going to need a couple of vars.

- One is going to be my ***starting angle***,
  - and the other one is my ***ending angle***.

```
struct Pie: Shape {
  var startAngle: Angle
  var endAngle: Angle

  func path(in rect: CGRect) -> Path {
    let center = CGPoint(x: rect.midX, y: rect.midY)

    var p = Path()
    p.move(to: center)

    return p
  }
```

5.4.7 Pie - Draw a line

34:42 I need to somehow calculate a start position up at the top up here,

- so that I can draw a line.

```
struct Pie: Shape {
  var startAngle: Angle
  var endAngle: Angle

  func path(in rect: CGRect) -> Path {
    let center = CGPoint(x: rect.midX, y: rect.midY)

    let radius = min(rect.width, rect.height) / 2
    let start = CGPoint(
      x: center.x + radius * cos(CGFloat(startAngle.radians)),
      y: center.y +  radius * sin(CGFloat(startAngle.radians))
    )

    var p = Path()
    p.move(to: center)
    p.addLine(to: start)

    return p
  }
}
```

- Drawing the line

36.42 Now we need to go in this big arc.

```
import SwiftUI

struct Pie: Shape {
  var startAngle: Angle
  var endAngle: Angle
  var clockwise: Bool = false

  func path(in rect: CGRect) -> Path {
    let center = CGPoint(x: rect.midX, y: rect.midY)

    let radius = min(rect.width, rect.height) / 2
    let start = CGPoint(
      x: center.x + radius * cos(CGFloat(startAngle.radians)),
      y: center.y +  radius * sin(CGFloat(startAngle.radians))
    )

    var p = Path()
    p.move(to: center)
    p.addLine(to: start)
    p.addArc(
      center: center,
      radius: radius,
      startAngle: startAngle,
      endAngle: endAngle,
      clockwise: clockwise)
    p.addLine(to: center)

    return p
  }
}
```

`p.addArc(`

- Luckily `Path` has a function for arcing as well.
- Let's you create an arc by specifying the center of the arc, the radius of the arc, the start of the angle, the end of the angle, and whether you're going clockwise around the arc or counterclockwise.

`p.addLine(to: center)`

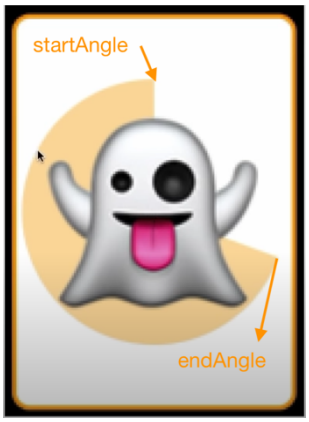
5.4.9 Pie - Draw a line back to the center

38:02 The path needs to come back to the middle.

- So we need a line back to the center.

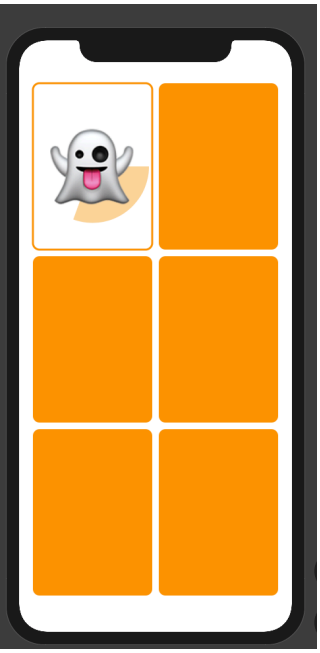
```
import SwiftUI

struct Pie: Shape {
  var startAngle: Angle
  var endAngle: Angle
  var clockwise: Bool = false

  func path(in rect: CGRect) -> Path {
    let center = CGPoint(x: rect.midX, y: rect.midY)

    let radius = min(rect.width, rect.height) / 2
    let start = CGPoint(
      x: center.x + radius * cos(CGFloat(startAngle.radians)),
      y: center.y +  radius * sin(CGFloat(startAngle.radians))
    )

    var p = Path()
    p.move(to: center)
    p.addLine(to: start)
    p.addArc(
      center: center,
      radius: radius,
      startAngle: startAngle,
      endAngle: endAngle,
      clockwise: clockwise)
    p.addLine(to: center)

    return p
  }
}
```

38:30



Where we used `Circle`, we are going to use our `Pie` instead.

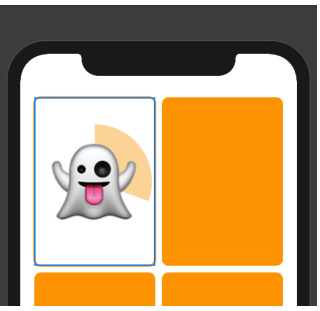
```
private func body(for size: CGSize) -> some View {
  ZStack {
    if self.card.isFaceUp {
      RoundedRectangle(cornerRadius: cornerRadius)
        .fill(Color.white)
      RoundedRectangle(cornerRadius: cornerRadius)
        .stroke(lineWidth: edgeLineWidth)
      Pie(startAngle: Angle.degrees(0),
          endAngle: Angle.degrees(110)
      )
        .padding(5)
        .opacity(0.4)
      Text(self.card.content)
    } else {
      if !card.isMatched {
        RoundedRectangle(cornerRadius: cornerRadius)
        .fill()
      }
    }
  } // ZStack
    .font(Font.system(size: fontSize(for: size)))
}
```



The result is not what we expected.

- The first thing to understand in iOS is that angle zero, zero degrees is not up, instead is out to the right…

```
private func body(for size: CGSize) -> some View {
  ZStack {
    if self.card.isFaceUp {
      RoundedRectangle(cornerRadius: cornerRadius)
        .fill(Color.white)
      RoundedRectangle(cornerRadius: cornerRadius)
        .stroke(lineWidth: edgeLineWidth)
      Pie(startAngle: Angle.degrees(0-90),
          endAngle: Angle.degrees(110-90)
      )
```



…So, If we wanted this to be in the kind of degrees where zero is straight up, we're going to have to subtract 90 degrees from both angles.

40:09 But that didn't really work either. That's still not what we're looking for.

- It looks like it started up, but it went *clockwise* instead of *counterclockwise*.
  - This is happening because in iOS, the drawing coordinate system that you're drawing in has (0, 0) in the upper left and is upside down.
    - It's not ***cartesian coordinates*** that you're used to.
      - So, since this whole thing is upside down, *clockwise* and *counterclockwise* are going the other way as well.

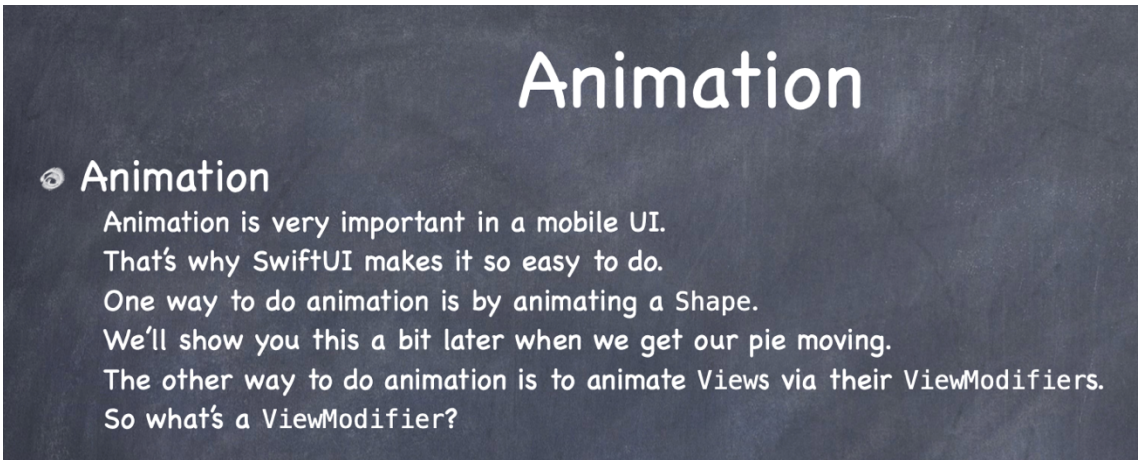

```
struct EmojiMemoryGameView: View {
  @ObservedObject var viewModel: EmojiMemoryGame
…
private func body(for size: CGSize) -> some View {
    ZStack {
      if self.card.isFaceUp {
        RoundedRectangle(cornerRadius: cornerRadius)
          .fill(Color.white)
        RoundedRectangle(cornerRadius: cornerRadius)
          .stroke(lineWidth: edgeLineWidth)
        Pie(startAngle: Angle.degrees(0-90),
            endAngle: Angle.degrees(110-90),
            clockwise: true
        )
          .padding(5)
          .opacity(0.4)
        Text(self.card.content)
      } else {
        if !card.isMatched {
          RoundedRectangle(cornerRadius: cornerRadius)
          .fill()
        }
      }
    } // ZStack
      .font(Font.system(size: fontSize(for: size)))
  }
```

`clockwise: true`

- Even though we're really going the opposite direction.

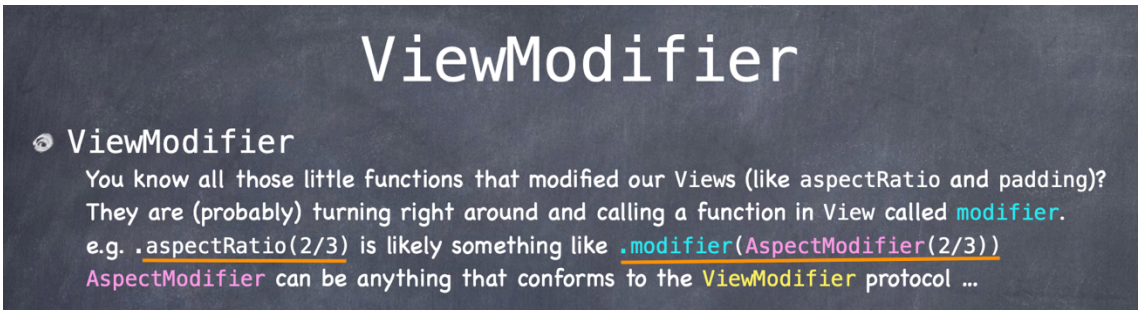
5.5 Animation

41:54



5.6 ViewModifier

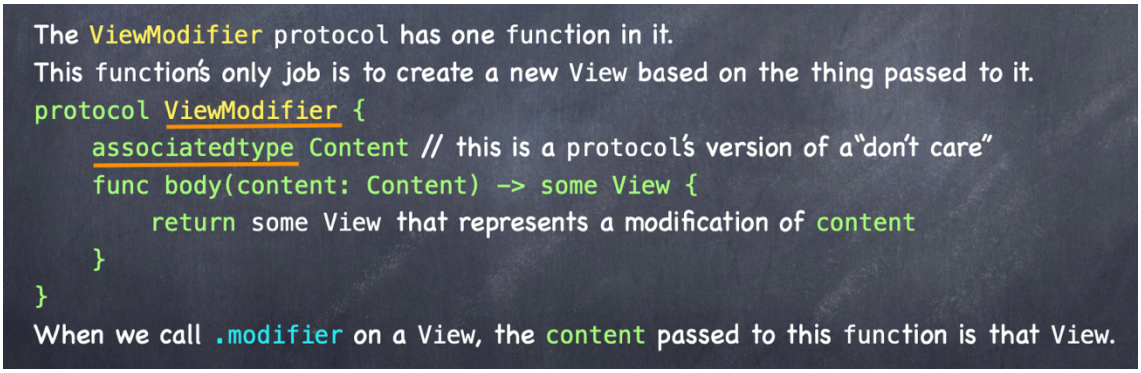
42:31



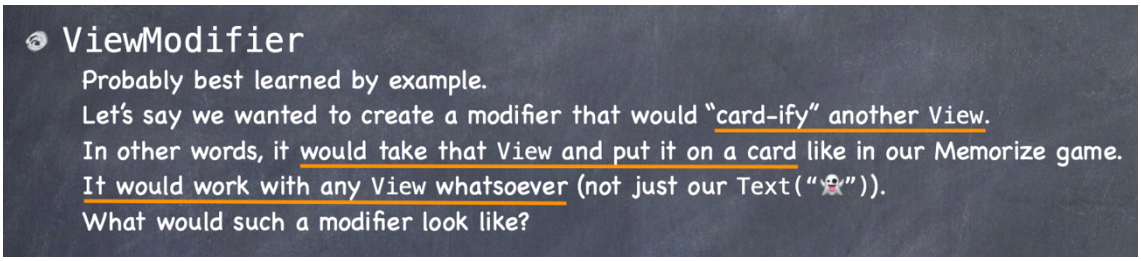
They *modify* a *view* and return a *new view* that is a modified version of the *view* you called them on.

Most of these modifiers are probably implemented by just turning right around and calling a very important function in `View` protocol, added via extension called `modifier`.

- The modifier function takes one argument which is essentially something that implements the `ViewModifier` protocol.

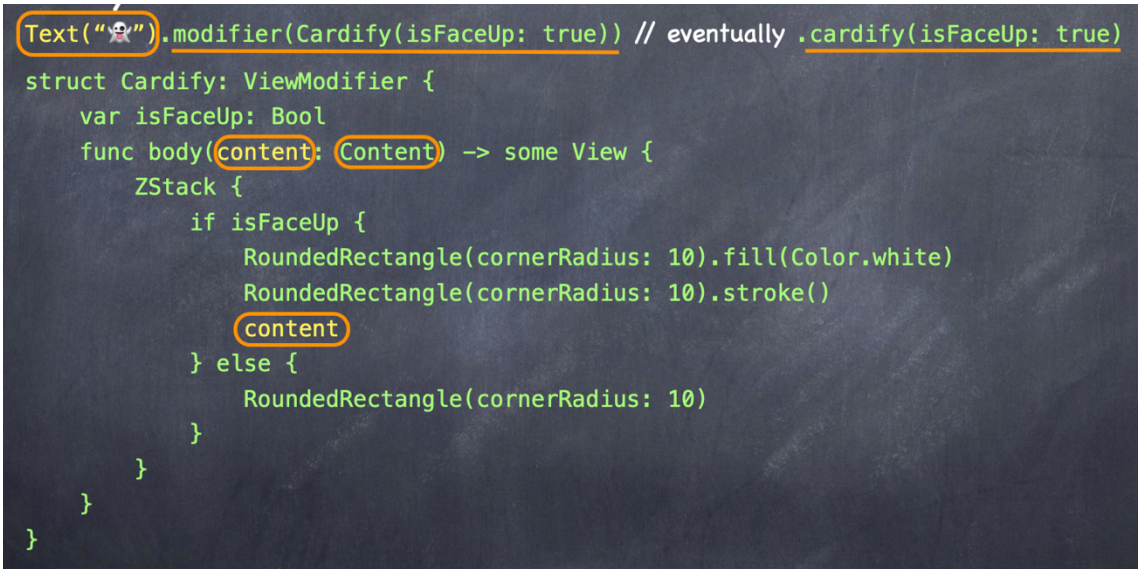


45:30



It would work with any View whatsoever

- In our app we wouldn't necessarily even need a modifier like this because we only have one kind of view that ever gets cardified.
    - But you could imagine an app where you have some cards that have images, emojis or text on them, but they're all having the same card.



`Text("👻").modifier(Cardify(isFaceUp: true)`

- When calling the modifier, the argument is `Cardify` which is a `ViewModifier`.
- I actually want this to say `Text(…).cardify(…)` just like I say `.foregroundColor` or `.padding`. (We'll see how in the next slide.)

`Text("👻").modifier(Cardify(isFaceUp: true)`

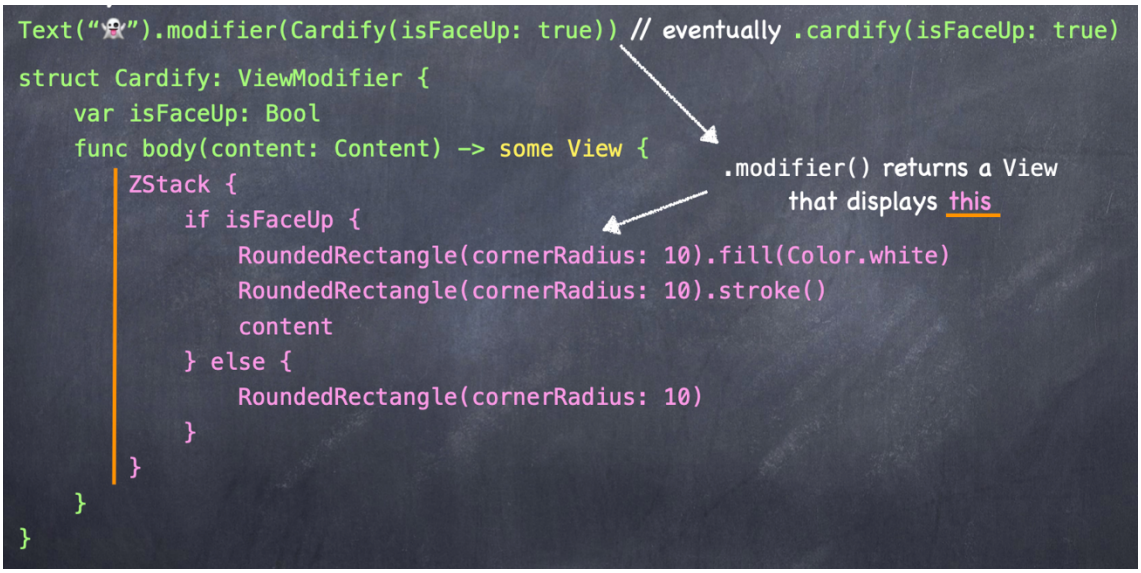
- The modifier takes the text ghost and modifies it into a new view that looks like a card.
- In this case `Text("👻")` is the content
    - I made it yellow here all the places content appears,
        - In `Text("👻")`
        - In body's argument
        - Embedded inside the ZStack.
            - In the same place I would've put that back in my old code.
                - So now it's going to take any view and embed it right there.

`Var isFaceUp: Bool`

- This ViewModifier needs to know whether the card is face up or face down.
    - We know that if you have a var and it's not initialized, then whoever creates this struct needs to specify the value of it.

5.6.2 ViewModifier – Example - body

48:48

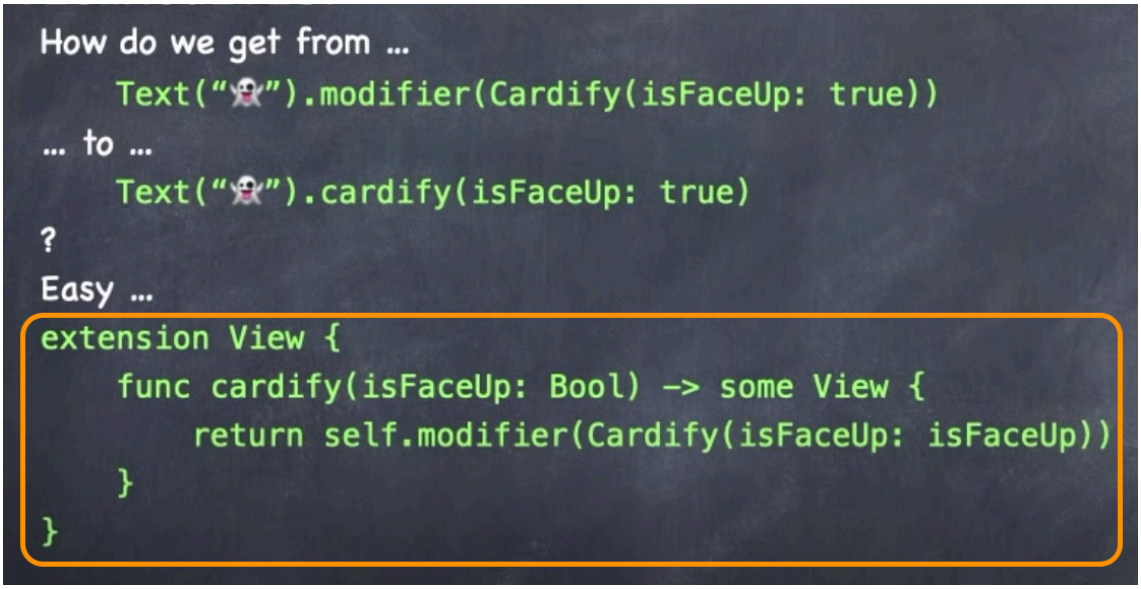


this

- Modifier doesn't directly return this purple code, returns something else that's also a View, but that something else is going to use this purple code to draw what it draws.
    - Since that thing is a View, we can send another modifier to it, and another modifier to that, et cetera,
        - and Swift is very smart about keeping track of all these modifications, so that it can do things like animation.

5.6.3 From .modifier(...) to . cardify(...)

49:43



5.7 Back to demo

50:22 Our goal in this demo is to make any View cardified.

- That means putting this border around when it's face up or just drawing this back when it's face down.

5.7.1 View - .modifier(Cardify(isFaceUp: card.isFaceUp)

Let's imagine what our code would look like if we had this ***cardifier***.

I'm going to take the part of our card right now that's going to get cardified, which is just the *pie* and the *text*.

- We're going to separate them out from the code that actually is doing the cardification.

```
private func body(for size: CGSize) -> some View {
  ZStack {
    if self.card.isFaceUp {
      RoundedRectangle(cornerRadius: cornerRadius)
        .fill(Color.white)
      RoundedRectangle(cornerRadius: cornerRadius)
        .stroke(lineWidth: edgeLineWidth)
      Pie(startAngle: Angle.degrees(0-90),
          endAngle: Angle.degrees(110-90),
          clockwise: true
      )
        .padding(5)
        .opacity(0.4)
      Text(self.card.content)
    } else {
      if !card.isMatched {
        RoundedRectangle(cornerRadius: cornerRadius)
        .fill()
      }
    }
  } // ZStack
    .font(Font.system(size: fontSize(for: size)))
}
```

```
  private func body(for size: CGSize) -> some View {
    ZStack {
      Pie(startAngle: Angle.degrees(0-90),
          endAngle: Angle.degrees(110-90),
          clockwise: true
      )
        .padding(5)
        .opacity(0.4)
      Text(self.card.content)
        .font(Font.system(size: fontSize(for: size)))
    } // ZStack
      .modifier(Cardify(isFaceUp: card.isFaceUp))
//    .cardify(isFaceUp: card.isFaceUp)
  }
```

5.7.2 View – Cardify.swift

52:25 **New** | **File** | **iOS** | ***Swift File***

- Save As: ***Cardify***

```
import SwiftUI

struct Cardify: ViewModifier {
  private let cornerRadius: CGFloat = 10.0
  private let edgeLineWidth: CGFloat = 3

  var isFaceUp: Bool

  func body(content: Content) -> some View {
    ZStack {
      if isFaceUp {
        RoundedRectangle(cornerRadius: cornerRadius)
          .fill(Color.white)
        RoundedRectangle(cornerRadius: cornerRadius)
          .stroke(lineWidth: edgeLineWidth)
        content
      } else {
//        if !card.isMatched {
          RoundedRectangle(cornerRadius: cornerRadius)
            .fill()
//        }
      }
    } // ZStack
  } // body
} // Cardify
```

`func body(content: Content) -> some View {`

- `Content` is a don't care that we get from `ViewModifier` protocol.
  - Protocols specify these don't cares with the associatedtype syntax.

~~`if !card.isMatched {`~~

- Really does not belong in Cardify.
  - Matching has to do with our card game.
  - This Cardify can cardify any view, not just views in our card game.
    - This could put a card face up, face down on any view.
      - So, we want to keep this generic and not tied to the rest of our app.
    - This is a nice reusable thing.
      - We could actually use another game.

5.7.3 View – Cardify.swift – cardify(isfaceUp:)

57:39 We're going to add the cardify function to `View` with an *extension*.

- It really makes the most sense obviously to put it in `Cardify.swift` file

`Cardify.swift`

```
extension View {
  func cardify(isFaceUp: Bool) -> some View {
    self.modifier(Cardify(isFaceUp: isFaceUp))
  }
}
```

`EmojiMemoryGameView.swift`

```
  private func body(for size: CGSize) -> some View {
    ZStack {
      Pie(startAngle: Angle.degrees(0-90),
          endAngle: Angle.degrees(110-90),
          clockwise: true
      )
        .padding(5)
        .opacity(0.4)
      Text(self.card.content)
        .font(Font.system(size: fontSize(for: size)))
    } // ZStack
    .cardify(isFaceUp: card.isFaceUp)
  }
```

5.7.4 Matched cards disappear - ERROR

58:46

```
private func body(for size: CGSize) -> some View {
  if card.isFaceUp || !card.isMatched {
    return ZStack {
      Pie(startAngle: Angle.degrees(0-90),
          endAngle: Angle.degrees(110-90),
          clockwise: true
      )
        .padding(5)
        .opacity(0.4)
      Text(self.card.content)
        .font(Font.system(size: fontSize(for: size)))
    } // ZStack
      .cardify(isFaceUp: card.isFaceUp)
  } // if
} // body  Missing return in a function expected to return 'some View'
```

- We have an `if` with no `else`.
  - We're not returning anything in that case.
    - This is no good, and I'm going to fix this I a kind of novel interesting way, which is I'm jus going to make this into a `ViewBuilder`.

5.7.5 Matched cards disappear – solution @ViewBuilder

59.40

```
@ViewBuilder
  private func body(for size: CGSize) -> some View {
    if card.isFaceUp || !card.isMatched {
      ZStack {
        Pie(startAngle: Angle.degrees(0-90),
            endAngle: Angle.degrees(110-90),
            clockwise: true
        )
          .padding(5)
          .opacity(0.4)
        Text(self.card.content)
          .font(Font.system(size: fontSize(for: size)))
      } // ZStack
        .cardify(isFaceUp: card.isFaceUp)
    } // if
  } // body
```

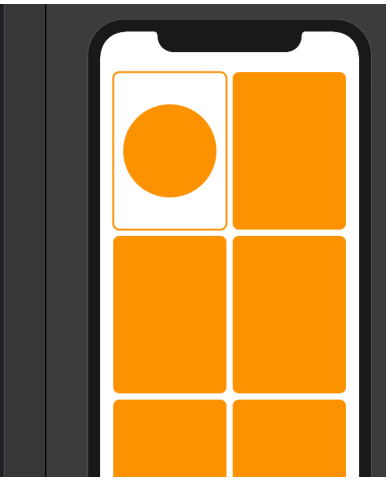
`@ViewBuilder`

- If I turn the `body` function into a `@ViewBuilder`, then the inside of body is interpreted as a ***list of views***.
  - Yes, with *ifs* to decide whether certain views are in or out.
- It's either going to end up being a `ZStack` or an `EmptyView` if the `if` statement is not *true*.

5.7.6 View – Try other views as Circle or Text

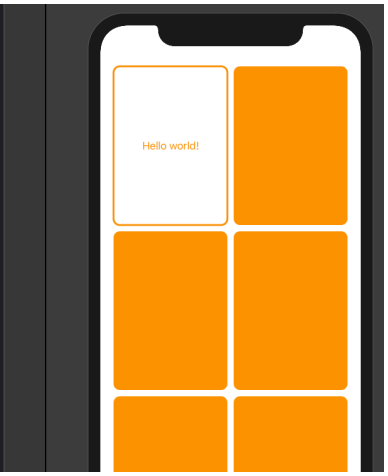
1:00:14 To make sure that our cardify is really working, instead of doing this `ZStack`, let's try some other views.

```
  @ViewBuilder
  private func body(for size: CGSize) -> some View {
    if card.isFaceUp || !card.isMatched {
//      ZStack {
//        Pie(startAngle: Angle.degrees(0-90),
//            endAngle: Angle.degrees(110-90),
//            clockwise: true
//        )
//          .padding(5)
//          .opacity(0.4)
//        Text(self.card.content)
//          .font(Font.system(size: fontSize(for: size)))
//      } // ZStack
      Circle().padding()
        .cardify(isFaceUp: card.isFaceUp)
    } // if
  } // body
```



`Circle().padding().cardify(isFaceUp: card.isFaceUp)`

```
  @ViewBuilder
  private func body(for size: CGSize) -> some View {
    if card.isFaceUp || !card.isMatched {
//      ZStack {
//        Pie(startAngle: Angle.degrees(0-90),
//            endAngle: Angle.degrees(110-90),
//            clockwise: true
//        )
//          .padding(5)
//          .opacity(0.4)
//        Text(self.card.content)
//          .font(Font.system(size: fontSize(for: size)))
//      } // ZStack
      Text("Hello world!")
        .cardify(isFaceUp: card.isFaceUp)
    } // if
  } // body
```




```
Text("Hello world!").cardify(isFaceUp: card.isFaceUp)
```

```
import SwiftUI

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

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

  @ViewBuilder
  private func body(for size: CGSize) -> some View {
    if card.isFaceUp || !card.isMatched {
      ZStack {
        Pie(startAngle: Angle.degrees(0-90),
            endAngle: Angle.degrees(110-90),
            clockwise: true
        )
          .padding(5)
          .opacity(0.4)
        Text(self.card.content)
          .font(Font.system(size: fontSize(for: size)))
      } // ZStack
        .cardify(isFaceUp: card.isFaceUp)
    } // if
  } // body

  private func fontSize(for size: CGSize) -> CGFloat {
    min(size.width, size.height) * 0.70
  }
} // CardView

struct ContentView_Previews: PreviewProvider {
  static var previews: some View {
    let game = EmojiMemoryGame()
    game.choose(card: game.cards[0])
    return EmojiMemoryGameView(viewModel: game)
  }
}
```

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

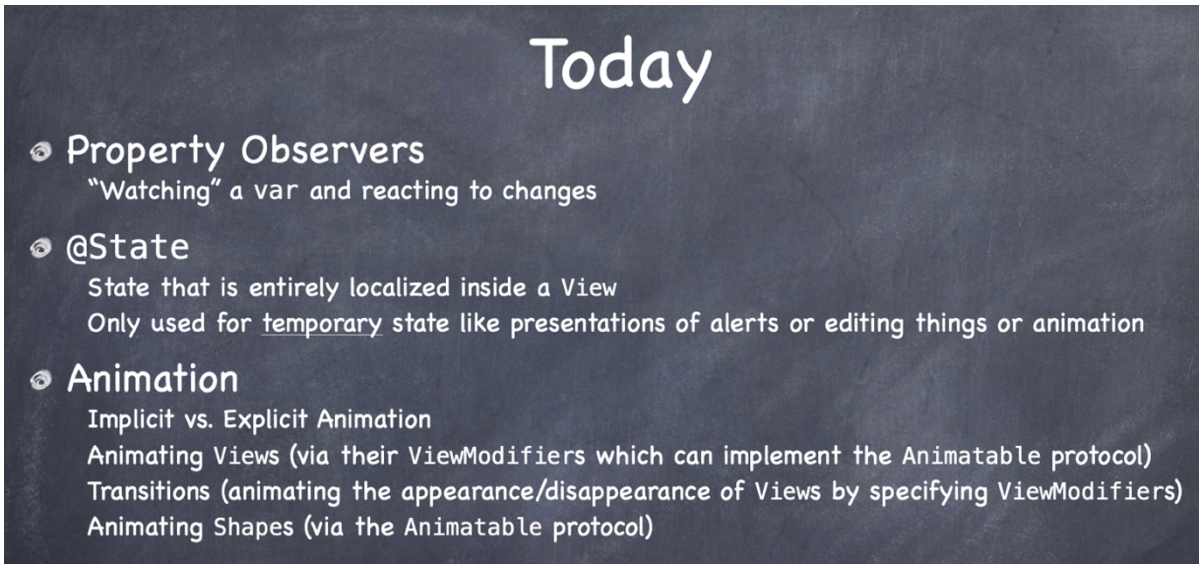
- *implicit* vs. *explicit animations*,
- *transitions*,
- *shape animations*,
- *animating view modifiers* and more.
- These concepts are then installed in the Memorize demo by animating the flipping of cards, creating a new game and giving bonus points for quick matches.

In SwiftUI, any changes to *view modifiers* 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*.

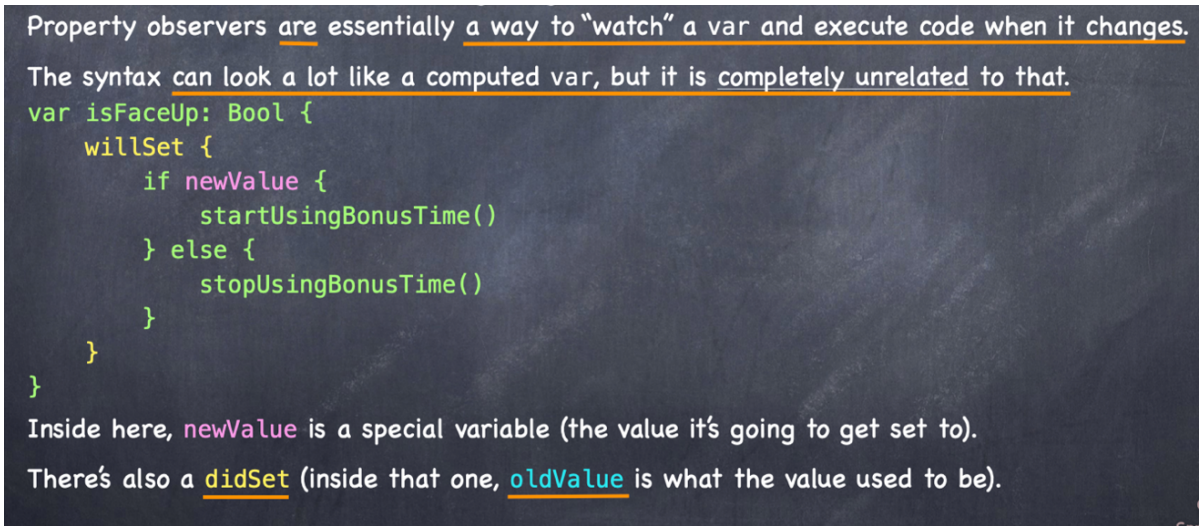
The "arrival and departure" of Views on screen can also be *animated* using the *transition declaration* (which declares which view modifiers to use to draw Views before/after they arrive/depart from the screen).

- In Memorize, the `Cardify` *view modifier* and the `Pie` *shape* are both made *animatable* and *animation* is applied throughout.



6.1 Property Observers

00:34



`var isFaceUp: Bool {`

- This variable
  - has a *property observer* on it.
  - Is stored in memory.

`willSet {`

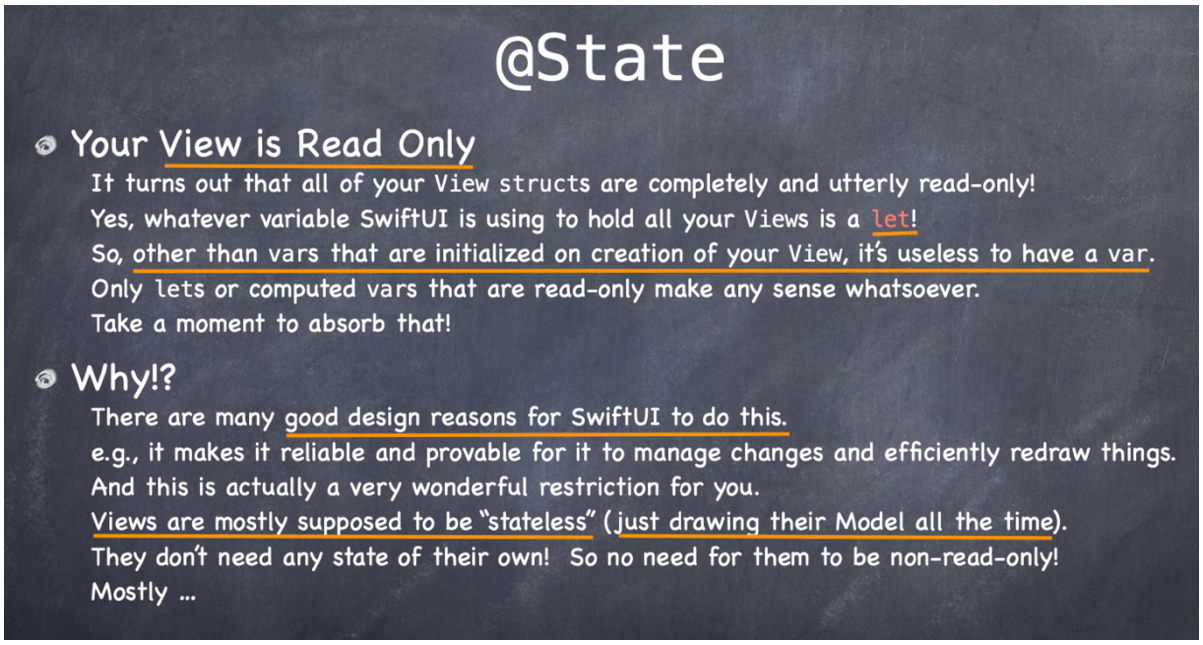
- `willSet` is called ***just before the value is stored***.
- If you don't write the parameter name and parentheses within your implementation,
  - the parameter is made available with a ***default parameter name*** of `newValue`.

`didSet {`

- `disSet` is called ***immediately after the new value is stored***.
- Similarly, if you implement a `didSet` *observer*, it's passed a constant parameter containing the old property value.
  - You can name the parameter or use the ***default parameter name*** of `oldValue`.
  - If you assign a value to a property within its own `didSet` *observer*, the new value that you assign replaces the one that was just set.

6.2 @State

01:48



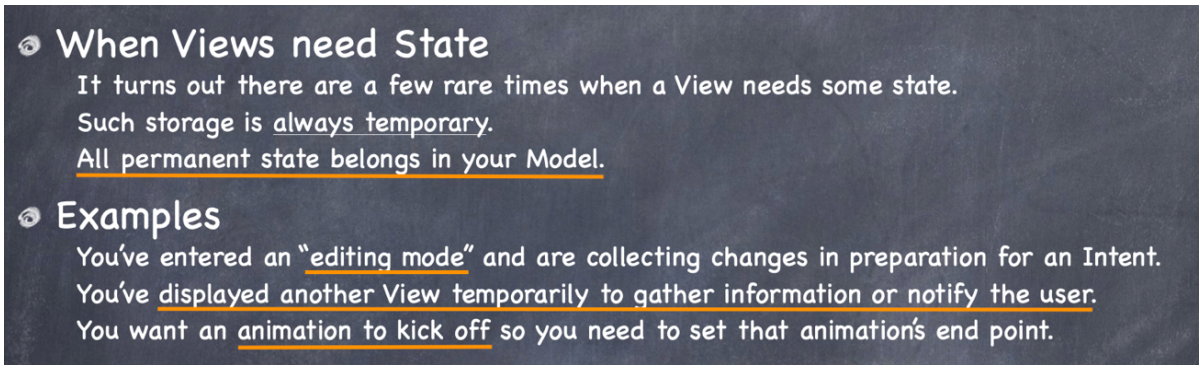
Why!?

- One of the awesome things about *functional programming*, is that it's very clear about *mutability*.
  - There's a huge premium on good design to having things be *immutable*.
    - When things are *immutable*, nobody's changing it behind the scenes or doing something that is messing up the provability that your code actually works.
  - And also, when things are *mutable* or *immutable*, ***when they change, you know they change***, and you can do something about it.
    - That's how we can use these *property observers* on *value types* so effectively.
- In the case of SwiftUI, it wants to know when things are changing.
  - It wants to do the minimum amount of work to replace the view hierarchy with the right views.

In fact, that is what's happening, if `isFaceUp` changes, it makes a new `CardView` that has that.

6.2.1 @State - When views need state

04:20

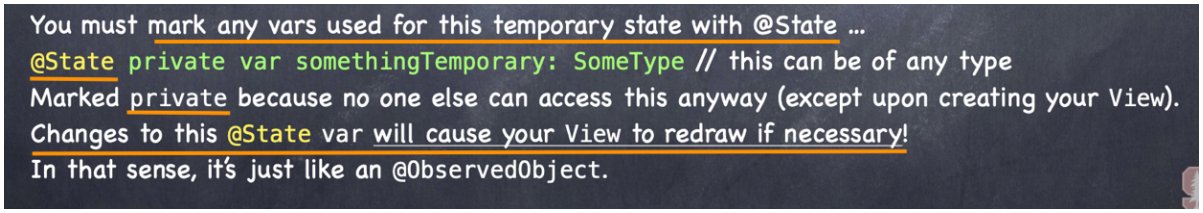


You want an animation to kick off, so you need to set that animation end point.

- *Animations* only ***reflects things in the past***.
    - If you want to have an animation that is kind of going along with the present, you have to have a little variable, which is the future.
        - So, you can set that var to the future and animating towards the future.
        - Hopefully, you're setting the future to be the same thing that's going to eventually be in your permanent state.
            - But you are only using that var during the time the animation is happening.

6.2.2 @State – Property wrapper

05:45

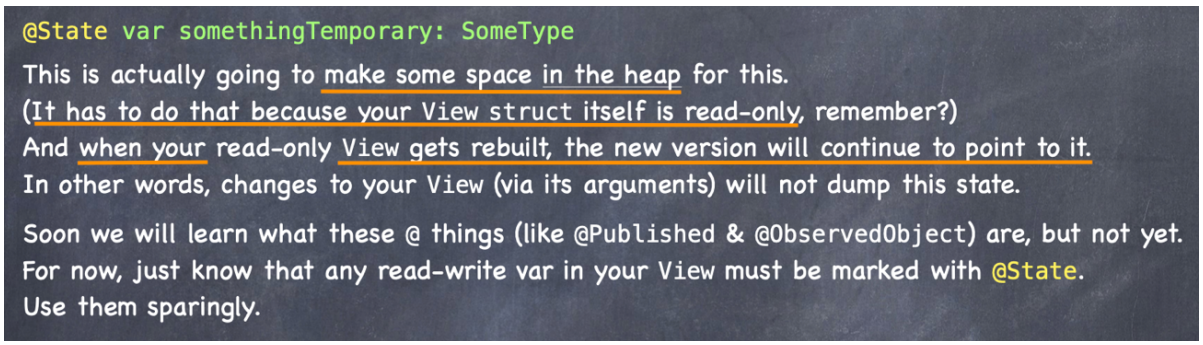


`@State private var somethingTemporary: SomeType`

- We can, in fact, create storage in our read-only *views* and we do it by marking a var that stores information we want with the `@State` ***property wrapper***.

6.2.3 @State – Read-write on the heap

06:58

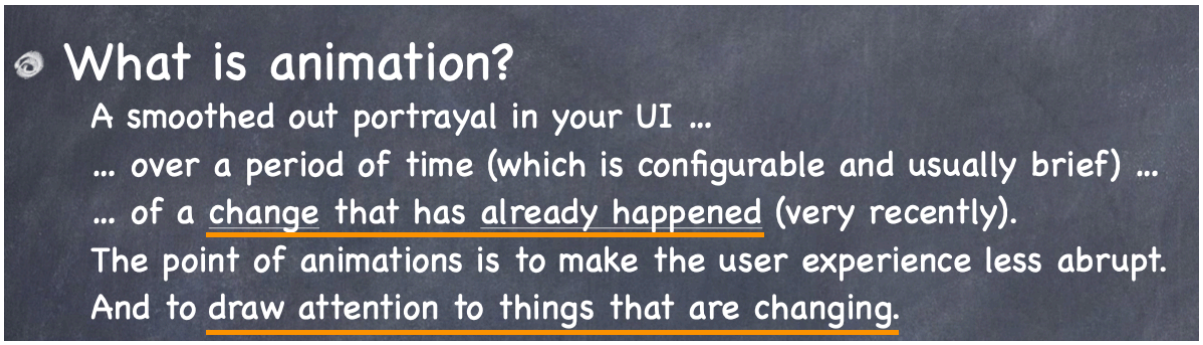


If `isFaceUp` changes, your `CardView` is not going to lose its temporary storage.

- Basically, your temporary storage will stay around even as other things that are causing your *view* to even be completely rebuilt you get to keep your `@State`.

6.3 Animation

08:13

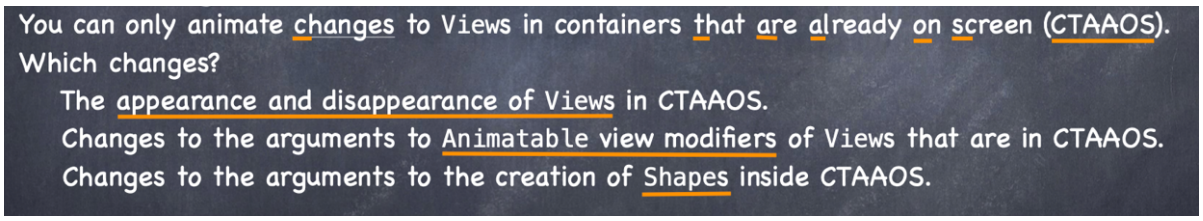


… of a change that has already happened (very recently).

- When the user looks at an *animation*, he's seeing *something* that has already changed in the Model.
  - Something that has already happened, it can't do it any other way.
    - Otherwise, ***all your variables in your Model would have to be constantly changing as the animation went on***, and that's just untenable architecture.
  - First, your Model changes, then your View changes and that change gets animated in front of the user's eyes showing him their very recent past.

6.3.1 Animation – what can get animated?

09:33



Animatable view modifiers

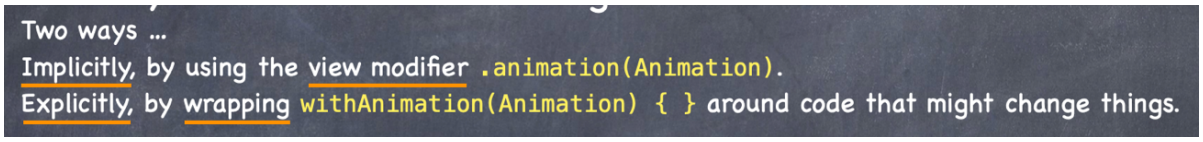
- Like *opacity* or *rotation*.

Creation of shapes

- If you create a *shape*, with certain arguments configured in some way, and then you change those arguments, then they can be animated to go to a new state.

6.3.2 Animation – How do you make an animation "go" ?

10:32



Implicit animation

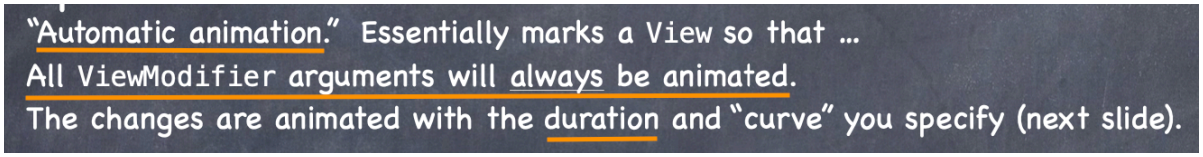
- Where we're going to just mark a *view* and say, whenever one of the modifiers on this view changes, we're going to animate that change.

Explicit animation

- Where we're going to call some code that is going to result in some changes to *view modifiers* or *shapes*, or *views* are going to be coming and going.
    - We're going to wrap that code by calling the `withAnimation(Animation:)` *function*,
        - and inside the curly braces we're going to put the code,
            - and that's going to cause all the things change together in one *concurrent animation*.
- We're saying animate this and then we usually do something like call an ***intent*** in our ViewModel.

6.3.3 Implicit animation

11:52

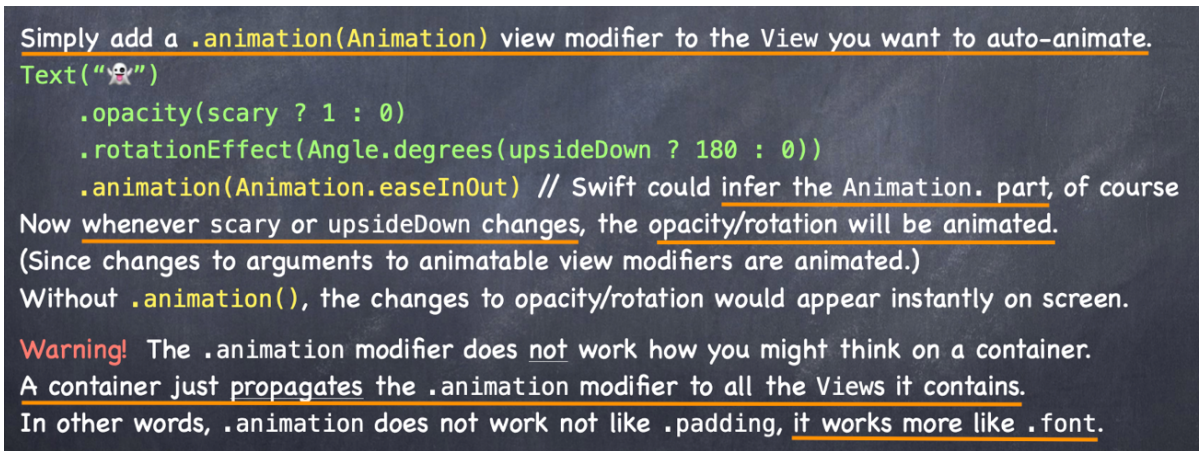


Duration

- How long it takes for them to happen.
    - You do it by calling the `withAnimation(Animation:)` function on any view.

6.3.4 Implicit animation – It works more like .font

12:20

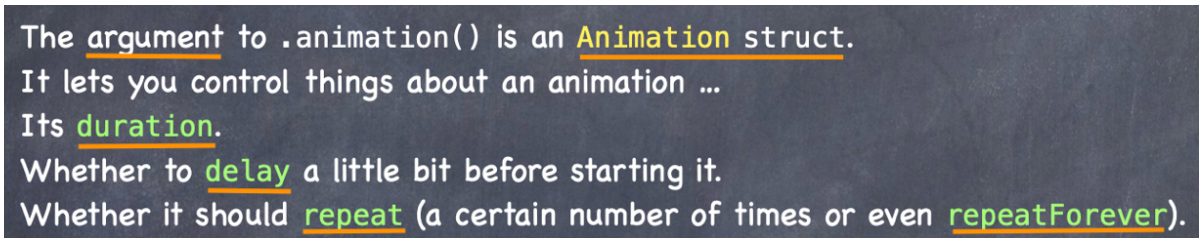


Warning!

- `.animation(Animation:)` on a *container view* does not work how you would generally think.
    - You might imagine, it's just going to animate the whole container like one big, somehow blob of change.
        - It just ***applies*** that ***animation to all the things inside the container***.
    - In other words, `.animation` is not like `.padding` that puts padding around the whole `ZStack` or `VStack`.
        - It's more like `.font` on the `ZStack` where all *texts* in the `ZStack` get that *font*.
- We don't usually put `.animation(Animation:)` on *container views*,
    - They're usually put on ***leaf views*** *or* ***self-contained views***.

6.3.5 Implicit animation – duration, delay, repeat

14:14

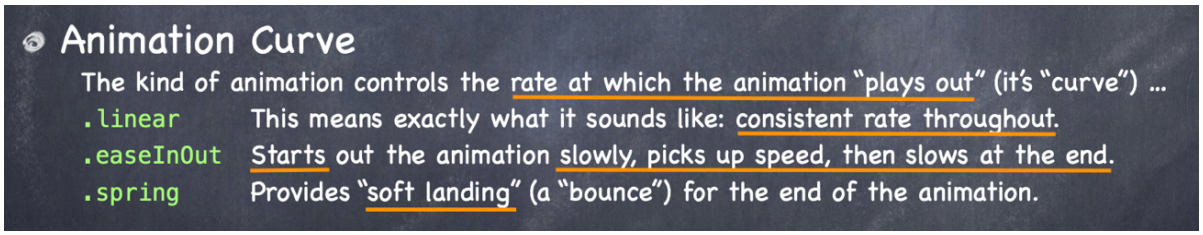


`repeatForever`

- The change, of course, has already been made in the past, but just keep doing the animation.

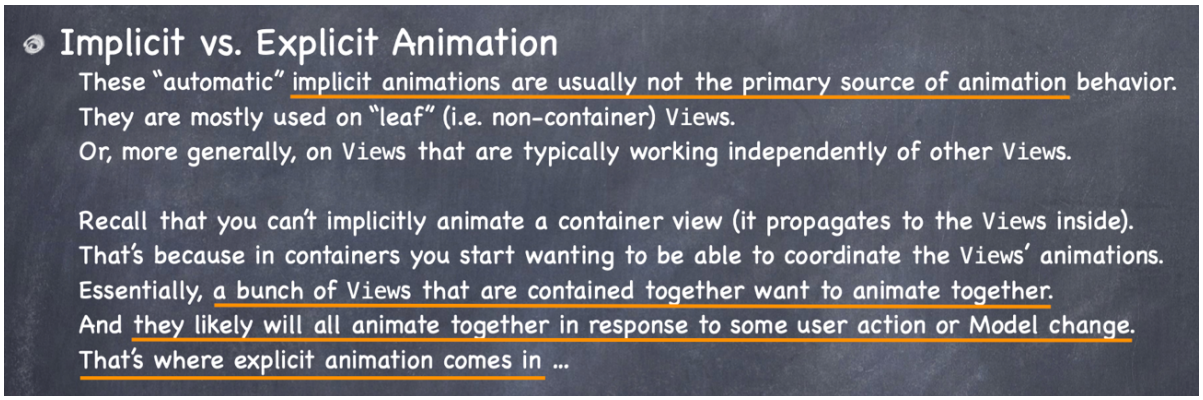
6.3.6 Implicit Animation – Animation curve

14:46



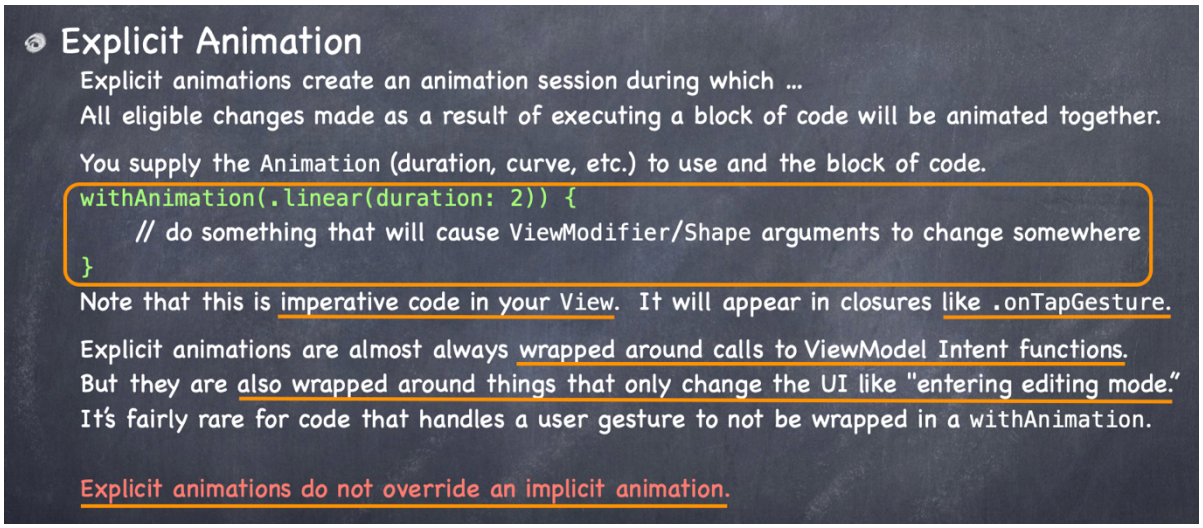
6.3.7 Implicit vs. explicit animation -

16:28



6.3.8 Explicit animation

17:24

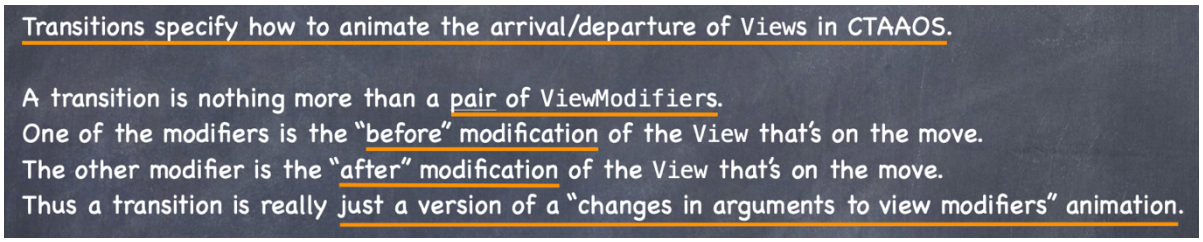


Explicit animations do not override an implicit animation.

- *Implicit animations*
  - are assumed to be on views.
  - They are self-contained.
  - They work independently.
- If there's a *view* with an *implicit animation* attached to it, it's going to be doing that *implicit animation* whenever its things change
  - even if there's an *explicit animation* going on at the same time.

6.3.9 Animation - Transitions

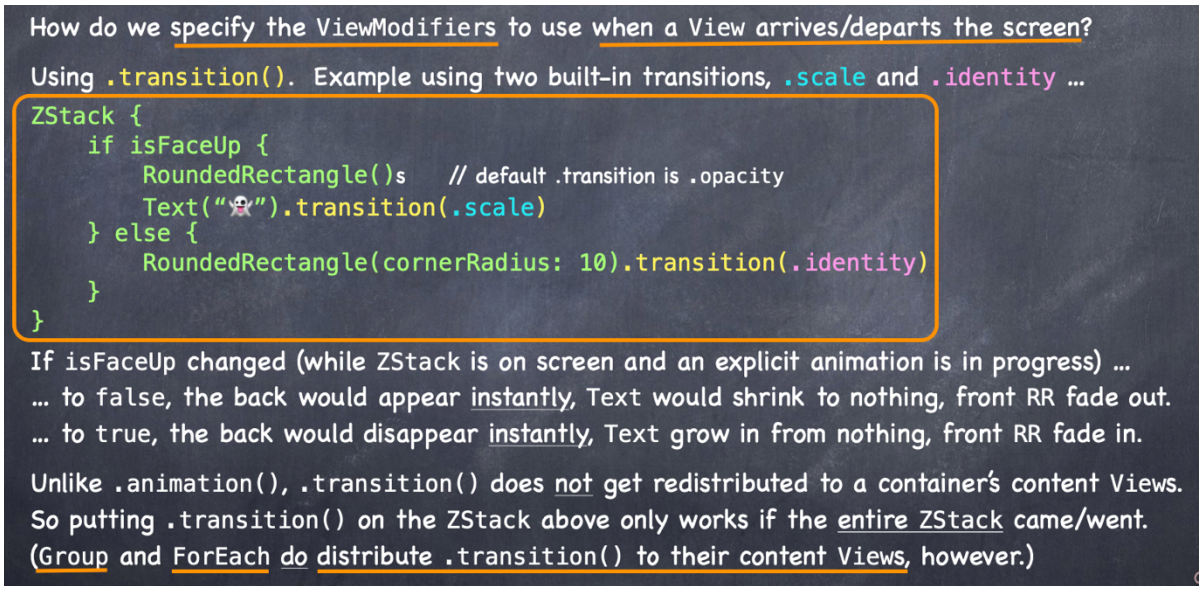
20:47



Transitioning is not really a different kind of *animation*,

- it's just a way of specifying the two *view modifiers* for when *views* appear and disappear.

22:04



`Text("").transition(.scale)`

- The `.scale` *transition* ***zooms*** the view in and out from tiny, zero size up to full size as it goes out or in.

`RoundedRectangle(cornerRadius: 10).transition(.identity)`

- In `.identity` *transition* its *view modifier* does ***nothing***.
  - It just instantly appears and disappears.
    - It is occasionally the case when you're doing an *animation*, and you have *views* coming and going, possibly you might want a *view* to just appear and disappear.

`RoundedRectangle()`

- The ***default*** *transition* is called `.opacity`, which ***fades in*** and ***out***.

`If isFaceUp {`

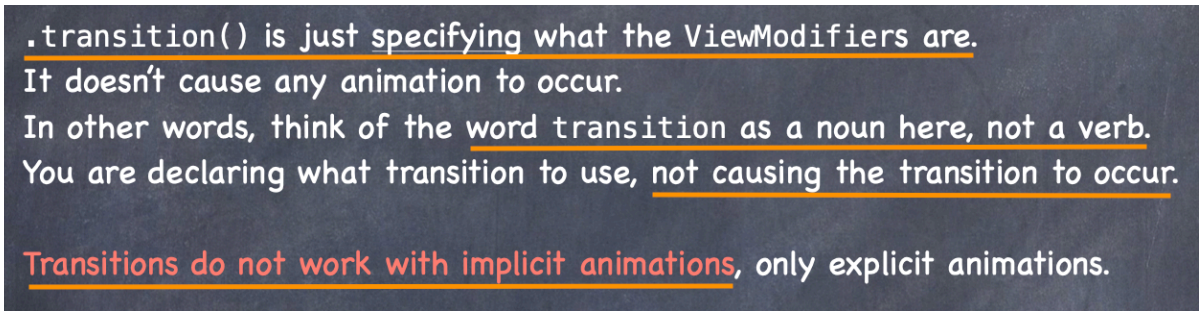
- Inside *view builders* we can use *if-thens* to include or not include the *views*.
  - They appear or disappear on screen.
  - This is probably the number one way for views coming and going.
    - When you have *animations* of the contents of some *view builder* like the `ZStack`, and it's got conditionals in there, you want to think about *transitions*.

`ForEach`

- Another way that views come and go is for example, in a `ForEach`.
  - `ForEach` takes an *array* of *identifiable* things, and makes *views* for them.
    - If that *array* changes, like new elements got added to it, or some of the identifiers got pulled out, it's going to either add new *views* or take some of the *views* it made in the past out of there.
      - Those *views* are going to be coming and going.

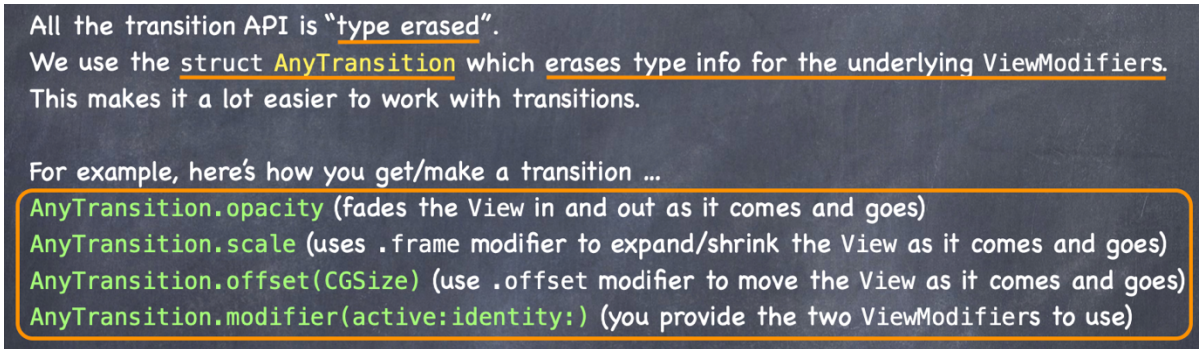
6.3.11 Animation – Transitions – Do not work with implicit animations

29:26



6.3.12 Animation -

30:48



All the transition API is "type erased".

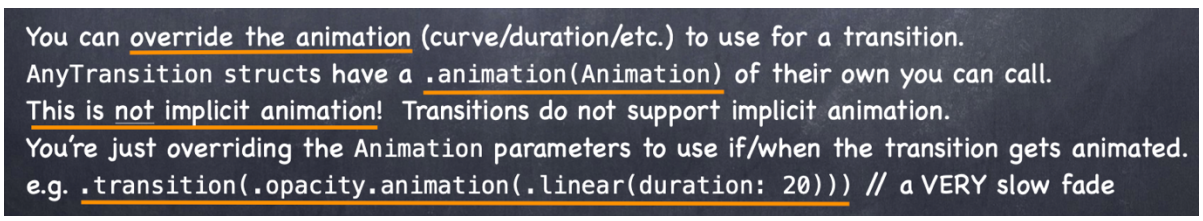
- That means that the real type of a *transition* is going to have ***don't cares*** in there that are the two *view modifiers that you're using*.

Struct AnyTransition

- We don't really lose the *type info*, but we can't see all the details like what kind of *view modifiers* it's using.
    - We do that in Swift on a number of cases. For example, with *views*,
        - there's a *view* called `AnyView` where its initializers will take any kind of *view* no matter how complicated, and return you `AnyView` and ***erase*** all that information.
            - You know nothing about what's inside.

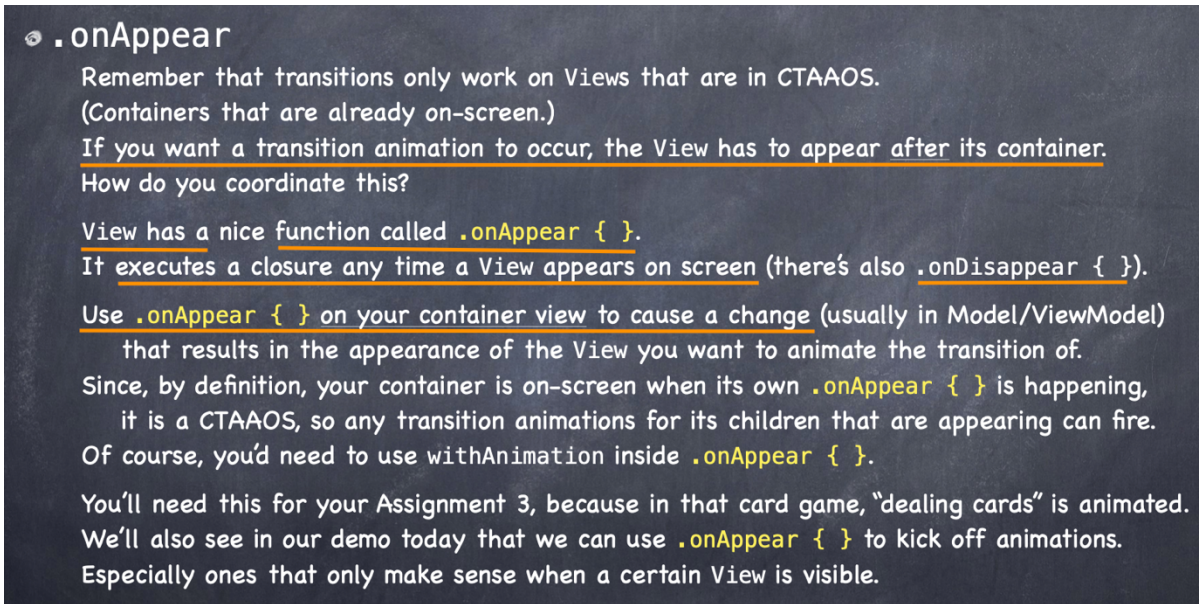
6.3.13 Animation – Transitions – Override the animation

33:26



6.3.14 Animation – Transitions – .onAppear

34:01

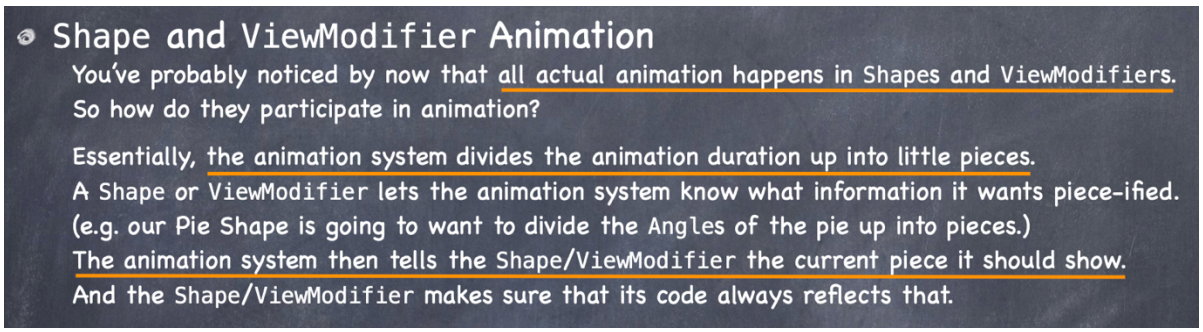


View has a function called `.onAppear { }`

- *Transitions* can be thorny, and a little bit frustrating sometimes when you're first using them, because of this restriction that the container that has the *view* has to already be on screen.
    - There's a great function in `View` for helping with this that is called `.onAppear`.
        - When a *view* appears on screen, then it calls this code.

6.3.15 Animation – Transitions – Shape and ViewModifier animation

36:12

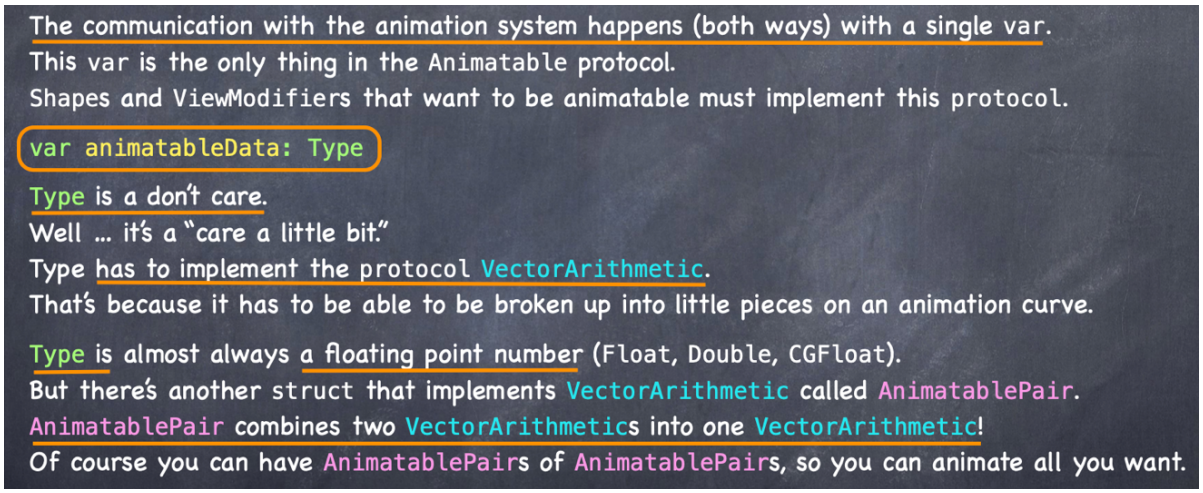


Essentially, the animation system divides the animation up into little pieces, depending on the *curve*.

- Then it just asks all the *shapes* and *view modifiers* that are *animatable*, to draw the pieces at a time.
    - It's just drawing them over and over and over and then piecing it together into ***like a little movie***, which is the ***animation***.

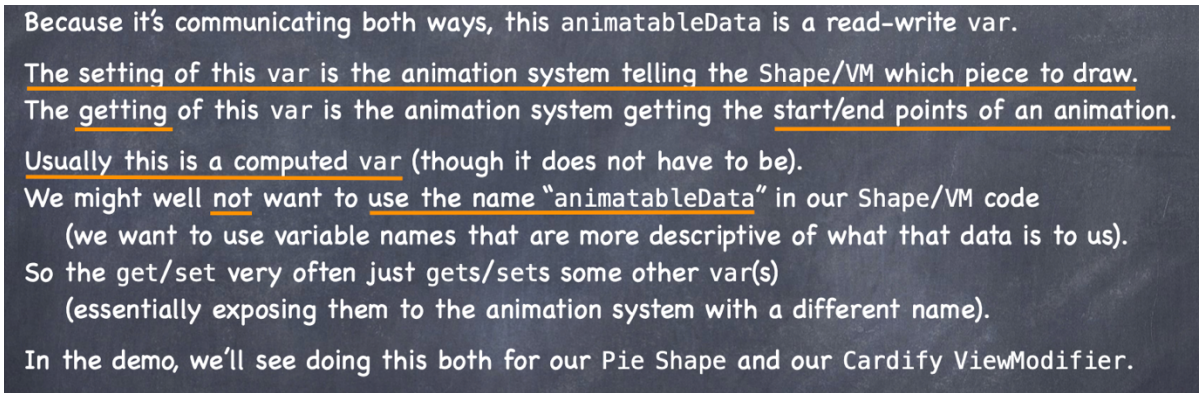
6.3.16 Animation – Transitions – Shape and ViewModifier animation - animatableData

36:57



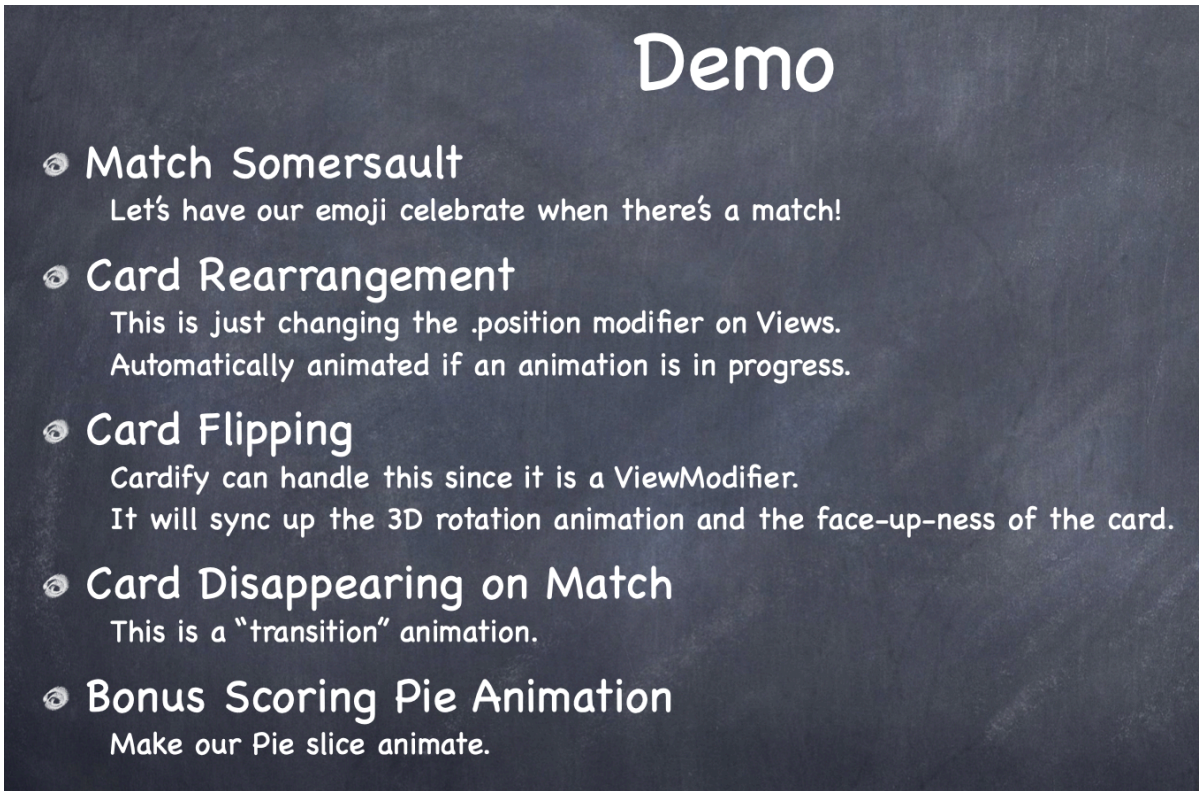
6.3.17 Animation – Transitions – Shape and ViewModifier animation – read-write var

38:36



6.4 Back to demo

39:41



Let's start our *animation* demo with some ***implicit animation***.

- This is an *animation* where we're going to have some very ***self-contained animation*** that always is going to apply no matter what.
    - And it's not really coordinated with a lot of other activity going on in the *animation system*.
- What are we going to do for this is to have our emojis be really excited and celebrate when they get a match by doing a *somersault*.
    - A ***somersault*** is essentially ***rotating the emoji around***.

6.4.1 Match somersault - Modifier - .rotationEffect(Angle.degrees(card.isMatched ? 180 : 0))

There's a *view modifier* called `rotationEffect`.

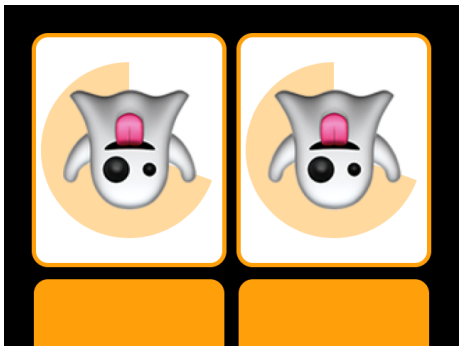
```
func rotationEffect(_ angle: Angle,
                    anchor: UnitPoint = .center) -> some View
```

Rotates this view's rendered output around the specified point.

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

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

@ViewBuilder
  private func body(for size: CGSize) -> some View {
    if card.isFaceUp || !card.isMatched {
      ZStack {
        Pie(startAngle: Angle.degrees(0-90),
            endAngle: Angle.degrees(110-90),
            clockwise: true
        )
          .padding(5)
          .opacity(0.4)
        Text(self.card.content)
          .font(Font.system(size: fontSize(for: size)))
          .rotationEffect(Angle.degrees(card.isMatched ? 180 : 0))
      } // ZStack
        .cardify(isFaceUp: card.isFaceUp)
    } // if
  } // body
```

`.rotationEffect(Angle.degrees(card.isMatched ? 180 : 0))`

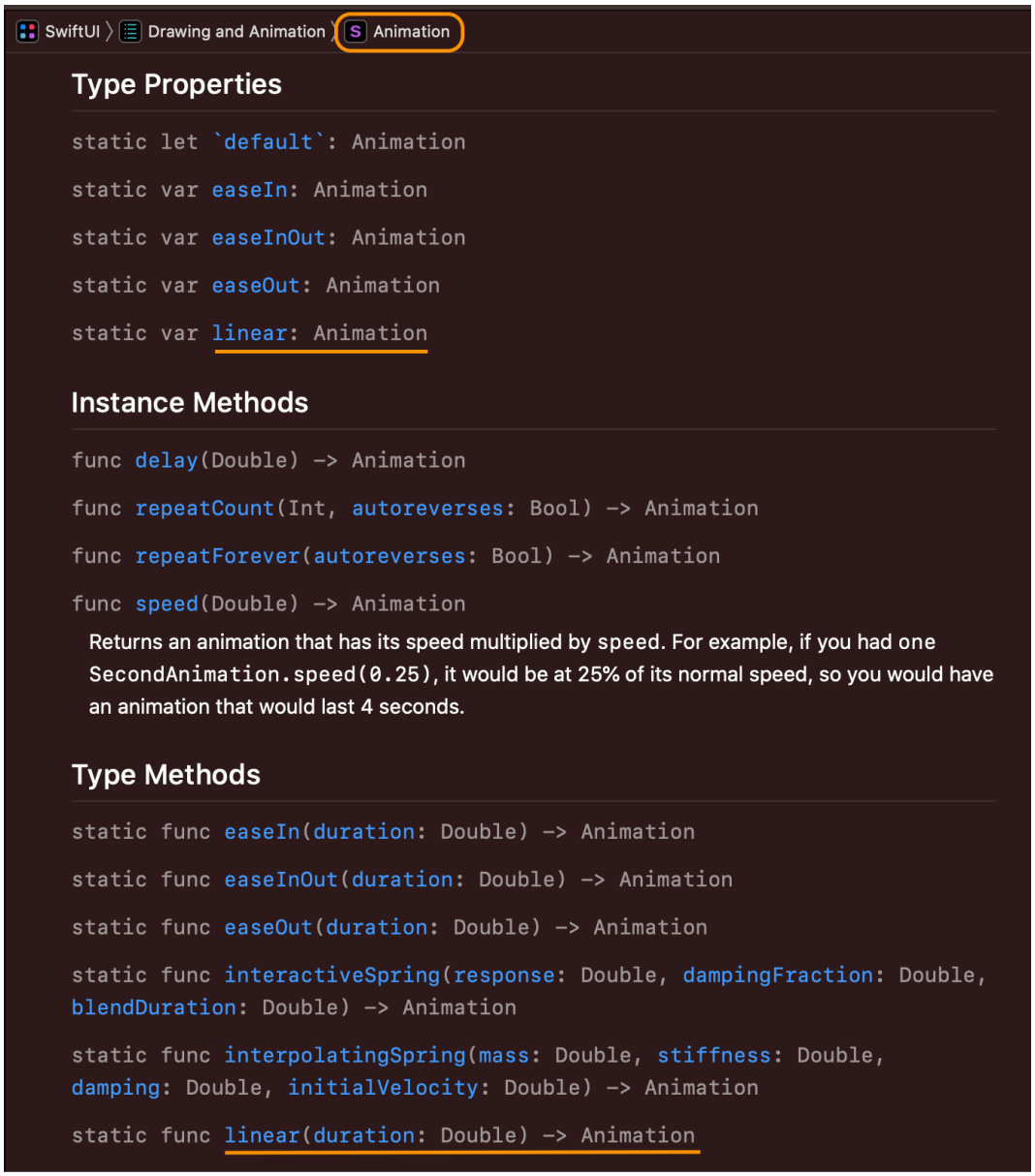
- If the card is matched, then let's start by just having it go upside down, which is a 180-degree rotation.



6.4.2 Match somersault - Implicit animation - .animation(Animation.linear(duration: 1))

41:31 To do a somersault when a card matches, all I need to do is an *implicit animation*.

```
@ViewBuilder
private func body(for size: CGSize) -> some View {
  if card.isFaceUp || !card.isMatched {
    ZStack {
      Pie(startAngle: Angle.degrees(0-90),
          endAngle: Angle.degrees(110-90),
          clockwise: true
      )
        .padding(5)
        .opacity(0.4)
      Text(self.card.content)
        .font(Font.system(size: fontSize(for: size)))
        .rotationEffect(Angle.degrees(card.isMatched ? 180 : 0))
        .animation(Animation.Linear(duration: 1))
    } // ZStack
      .cardify(isFaceUp: card.isFaceUp)
  } // if
} // body
```



```
func animation(_ animation: Animation?) -> some View
```
Applies the given animation to all animatable values within the view.

Run: it is doing the *animation* over a second only in one of the cards.

```
@ViewBuilder
 private func body(for size: CGSize) -> some View {
  if card.isFaceUp || !card.isMatched {
   ZStack {
    Pie(startAngle: Angle.degrees(0-90),
     endAngle: Angle.degrees(110-90),
     clockwise: true
    )
    .padding(5)
    .opacity(0.4)
    Text(self.card.content)
     .font(Font.system(size: fontSize(for: size)))
     .rotationEffect(Angle.degrees(card.isMatched ? 360 : 0))
     .animation(card.isMatched ?
      Animation.linear(duration: 1).repeatForever(autoreverses: false) :
     .default)
     } // ZStack
       .cardify(isFaceUp: card.isFaceUp)
   } // if
  } // body
```

```
.animation(card.isMatched ?
   Animation.linear(duration: 1).repeatForever(autoreverses:
false) :
   .default)
```

- We do not want to be doing this repeat forever.
  - Any time you do a *repeat forever* animation, you should be careful to turn it off when it doesn't apply anymore.
    - If `card.isMatched` we will repeat forever, but otherwise, we're going to go back doing whatever the *default animation* is.

6.4.3 Shuffling cards

45:07 `MemoryGame`

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

    for pairIndex in 0..<numberOfPairsOfCards {
      let content = cardContentFactory(pairIndex)

      cards.append(Card(content: content,
                        id: pairIndex*2))
      cards.append(Card(content: content,
                        id: pairIndex*2+1))
    }
    cards.shuffle()
  }
```

6.4.4 Card rearrangement - Shuffling cards - New game - ViewModel

45:56 New game requires us to have an *intent* in our ViewModel.

`EmojiMemoryGame`

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

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

func resetGame() {
  model = EmojiMemoryGame.createMemoryGame()
}
```

6.4.5 Card rearrangement - Shuffling cards - New game - View

46:38 I'm going to add a new button at the bottom of the screen.

```
struct Button<Label> where Label : View


init(action: @escaping () -> Void,
     @ViewBuilder label: () -> Label)
```

- It also has a *label* which is essentially *any view* you want to be the *label*.

`EmojiMemoryGameView`

```
var body: some View {
  VStack {
    Grid(viewModel.cards) { card in
      CardView(card: card).onTapGesture {
        self.viewModel.choose(card: card)
      }
      .padding(5)
    } // Grid
      .padding()
      .foregroundColor(Color.orange)
    Button(action: {
      self.viewModel.resetGame()
    }) {
      Text("New Game")
    }
  } // VSatck

} // body
} // EmojiMemoryGameView
```

`Button(action: { self.viewModel.resetGame() })`

- Button has an *action*, which is a *closure* to execute when the button gets pressed…

`{ Text("New Game") } }`

- I'm going to have the *label* here be a *text*

6.4.6 Card rearrangement - Explicit animation - withAnimation(.easeInOut)

50:00 We would like the whole screen to be animated.

- We'll do it with ***explicit animation***.

```
func withAnimation<Result>(
  _ animation: Animation? = .default,
  _ body: () throws -> Result) rethrows -> Result
```

Returns the result of recomputing the view's *body* with the provided *animation*.

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

  var body: some View {
    VStack {
      Grid(viewModel.cards) { card in
        CardView(card: card).onTapGesture {
          self.viewModel.choose(card: card)
        }
        .padding(5)
      } // Grid
        .padding()
        .foregroundColor(Color.orange)
      Button(action: {
        withAnimation(Animation.easeInOut) {
          self.viewModel.resetGame()
        }
      }) {
        Text("New game")
      } // Button
    } // VStack
  } // body
} // EmojiMemoryGameView
```

`withAnimation(Animation.easeInOut) {`

- We wrap the *reset of the game* which had a big effect on our Model and changed all our cards.
- We specify the *ease in out animation*.
  - It's not necessary to write the full `Animation.easeInOut` because Swift can infer it.
    - `withAnimation` is expecting an instance of `Animation` as its first argument.
- The second argument is a *closure* that takes no arguments.
  - Inside the *closure* you can put whatever code you want.
    - And this code will be *animated*.

```
      Button(action: {
        withAnimation(Animation.easeInOut(duration: 2)) {
          self.viewModel.resetGame()
        }
      }) {
        Text("New game")
      } // Button
```

`withAnimation(Animation.easeInOut(duration: 2)) {`

- We can change the *duration* to 2 seconds.
    - When doing *animation*, I always recommend to slow things down to see what's going on.

Run: After tapping ***New game*** *button*, you see the cards ***fade out***, and they move to their new position. Why is this happening?

- Because *transition* is by default ***opacity***.
    - What's happening there when we switch that, it's *transitioning* to a new view *and so we're just* ***fading*** *the new one* ***in*** *and* ***fading*** *the old one* ***out***.
        - We don't really want that; we want our cards actually to ***flip over*** when they go from *back to front* and *front to back*.
            - We'll fix that in a few minutes.

6.4.7 Card flipping - Explicit animation - withAnimation(.linear(duration: 2)

52:06 Let's use this same feature of ***explicit animation*** to make it so tapping on the cards is animated, because right now it's very abrupt when you click on a card and things instantly appear

```
var body: some View {
  VStack {
    Grid(viewModel.cards) { card in
      CardView(card: card).onTapGesture {
        withAnimation(.linear(duration: 2)) {
          self.viewModel.choose(card: card)
        }
      }
      .padding(5)
    } // Grid
      .padding()
      .foregroundColor(Color.orange)
    Button(action: {
      withAnimation(Animation.easeInOut(duration: 2)) {
        self.viewModel.resetGame()
      }
    }) {
      Text("New game")
    } // Button
  } // VStack
} // body
```

Notice that when we choose a card, we can see that we're getting ***all cards*** *fading in fading out*.

- *Explicit animation*, ***animate everything*** on the screen
- We want our cards to flip over when they go from back to front and front to back.

6.4.8 Card Disappearing on match - Transitions - Shrink down the cards when disappearing

53:45

```
func transition(_ t: AnyTransition) -> some View
```
Associates a transition with the view.

```
struct AnyTransition
```
A type-erased transition.

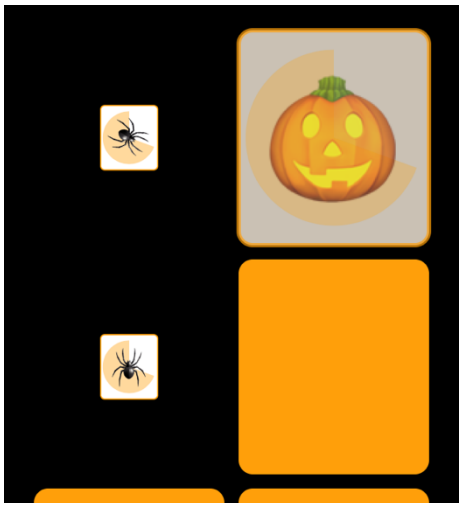
```
@ViewBuilder
 private func body(for size: CGSize) -> some View {
  if card.isFaceUp || !card.isMatched {
   ZStack {
    Pie(startAngle: Angle.degrees(0-90),
     endAngle: Angle.degrees(110-90),
     clockwise: true
    )
    .padding(5)
    .opacity(0.4)
    Text(self.card.content)
     .font(Font.system(size: fontSize(for: size)))
     .rotationEffect(Angle.degrees(card.isMatched ? 360 : 0))
     .animation(card.isMatched ?
      Animation.linear(duration: 1).repeatForever(autoreverses: false) :
      .default)
   } // ZStack
    .cardify(isFaceUp: card.isFaceUp)
    .transition(AnyTransition.scale)
  } // if
 } // body
```

`if card.isFaceUp || !card.isMatched {`

- This *cardified depth stack* view is shown when it's face-up or not matched.
  - Otherwise it ***transitions out*** disappearing from the view and goes away.

`.transition(AnyTransition.scale)`

- `.scale` uses *frame* to make *zoom in*, down to nothingness or out from nothing.



Notice that all other *animations* including the *implicit animation* kept going.

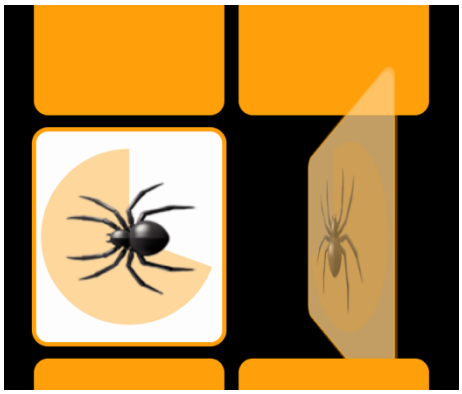
- All *animations* are ***happening at the same time*** working together.
  - This is one of the nicest things about SwiftUI.

6.4.9 Card flipping – rotation3DEffect(_:, axis:)

56:59 SwiftUI has an *animatable view modifier* called a `rotation3DEffect`, which rotates it just like we rotate on a match.

- On matches we rotate in 2D, round and round.

We can also rotate in 3D around with *y-vertical axis*.



```
func rotation3DEffect(
  _ angle: Angle,
  axis: (x: CGFloat, y: CGFloat, z: CGFloat),
  anchor: UnitPoint = .center,
  anchorZ: CGFloat = 0,
  perspective: CGFloat = 1
) -> some View
```

Rotates this view's rendered output in three dimensions around the given axis of rotation.

```
@ViewBuilder
 private func body(for size: CGSize) -> some View {
  if card.isFaceUp || !card.isMatched {
   ZStack {
    Pie(startAngle: Angle.degrees(0-90),
     endAngle: Angle.degrees(110-90),
     clockwise: true
    )
    .padding(5)
    .opacity(0.4)
    Text(self.card.content)
     .font(Font.system(size: fontSize(for: size)))
     .rotationEffect(Angle.degrees(card.isMatched ? 360 : 0))
     .animation(card.isMatched ?
      Animation.linear(duration: 1).repeatForever(autoreverses: false) :
      .default)
   } // ZStack
    .cardify(isFaceUp: card.isFaceUp)
    .transition(AnyTransition.scale)
    .rotation3DEffect(Angle.degrees(card.isFaceUp ? 0 : 180),
                      axis: (x: 0, y: 1, z: 0))
   } // if
  } // body
```

`Angle.degrees(card.isFaceUp ? 0 : 180),`

- You specify how much you want to rotate the card.

`axis: (x: 0, y: 1, z: 0)`

- This *axis* is saying around which *axis* do you want to rotate.
    - For example, (0, 0, 1) would be a ***2D rotation***, because the third argument is in the ***z-axis***.
        - That's the one that points up out at you from the screen.
- What we want is the ***y-axis*** that goes from the top of your screens straight down to the bottom of your screen.

6.4.10 Run – It's rotating, but that's really not what we want

58:52 When we first click the card, ***both the back and the front are visible***,

- one's fading out one's fading in,
    - then by the end, the back is totally faded out and the front has faded in.

There're two ways to make this work.

6.4.11 Card Flipping – First way – Have our own custom transition – Too complicated

That *transition* that is *transitioning* between the back and the front where the back kind of like we're flipping it up, the back is showing for a while until it gets up its edge and then it kind of disappears and then when the front comes on, it starts out on its edge and then kind of rotates down.

- We could definitely write a *view modifier* that does that and then make a *transition* out of it or we're using this sort of half-flip up onto its edge to have the card come in and come out.
    - It's slightly more complicated really, than I think we need to do, because if we remember how *animation* works we know that *view modifiers* are the main things that are doing *animation*.

6.4.12 Card Flipping – Second way – Take our view modifier that rotates itself – Better solution

Take our *view modifier* which draws one card and make it so that it's smart about it rotating itself so that it only shows the front during the first half of *animation* and only shows the back during the second half.

1:00:27

```
import SwiftUI

struct Cardify: ViewModifier {
 private let cornerRadius: CGFloat = 10.0
 private let edgeLineWidth: CGFloat = 3

 var isFaceUp: Bool

 func body(content: Content) -> some View {
  ZStack {
   if isFaceUp {
    RoundedRectangle(cornerRadius: cornerRadius)
     .fill(Color.white)
    RoundedRectangle(cornerRadius: cornerRadius)
     .stroke(lineWidth: edgeLineWidth)
    content
   } else {
    RoundedRectangle(cornerRadius: cornerRadius)
     .fill()
   }
  } // ZStack } // body } // Cardify

extension View {
 func cardify(isFaceUp: Bool) -> some View {
  self.modifier(Cardify(isFaceUp: isFaceUp))
 }
}
```

In other words, in the `Cardify` *struct* when it's rotating, it can rotate itself…

`if isFaceUp {`

- …And as it's doing it, it's going to coordinate what's ***face-up*** with the rotation.
  - First half of the rotation, ***face-up***
  - Second half rotation, ***face-down***.

First thing, we move `rotation3DEffect` *modifier* into our *modifier*.

`EmojiMemoryGameView`

```
@ViewBuilder
  private func body(for size: CGSize) -> some View {
    if card.isFaceUp || !card.isMatched {
      ZStack {
        Pie(startAngle: Angle.degrees(0-90),
            endAngle: Angle.degrees(110-90),
            clockwise: true
        ) …
        Text(self.card.content)…
      } // ZStack
        .cardify(isFaceUp: card.isFaceUp)
        .transition(AnyTransition.scale)
        .rotation3DEffect(Angle.degrees(card.isFaceUp ? 0 : 180),
                          axis: (x: 0, y: 1, z: 0))
    } // if
  } // body
```

Cardify

```
  func body(content: Content) -> some View {
    ZStack {
      if isFaceUp {
        RoundedRectangle(cornerRadius: cornerRadius)
          .fill(Color.white)
        RoundedRectangle(cornerRadius: cornerRadius)
          .stroke(lineWidth: edgeLineWidth)
        content
      } else {
          RoundedRectangle(cornerRadius: cornerRadius)
            .fill()
      }
    } // ZStack
     .rotation3DEffect(Angle.degrees(rotation),
                       axis: (x: 0, y: 1, z: 0))

  } // body
} // Cardify
```

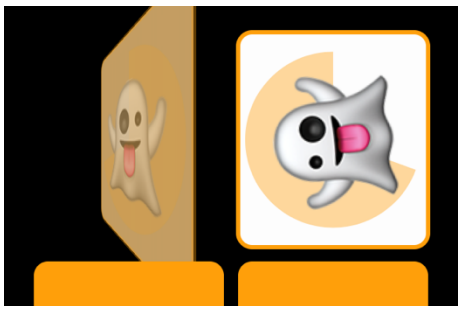
Instead of having the card rotate in a binary sense between 0 and 180, we want to be able to control the entire rotation of it.

- Make some changes to our modifier to be ***based on rotation*** rather than ***face-up***.

```
struct Cardify: ViewModifier {
…
  var rotation: Double

  init(isFaceUp: Bool) {
    rotation = isFaceUp ? 0 : 180
  }

  var isFaceUp: Bool {
    rotation < 90
  }
…
} // Cardify
```

```
  var isFaceUp: Bool { rotation < 90 }
```

- If I'm going to track the `rotation` and *animate* it, then is `isFaceUp` really just becomes a function of the rotation.
- Now I've linked rotation and the face_up fade_down.



This is not enough to make it *animate*. As we can see, it's doing the flip but it's still doing the wrong thing about the *face-up face-down*.

- That's because this *view modifier* is not marked as `Animatable`.
  - SwiftUI thinks this *view modifier* does not know how to *animate*. So, I'm just going to do a normal animation in the `ZStack`.

6.4.14 Card Flipping – Second way - Animatable

1:03:33

```
struct Cardify: ViewModifier {
...
}
```

> `protocol AnimatableModifier`
> A modifier that can create another modifier with animation.
>
> INHERITS FROM
> `Animatable`, `ViewModifier`
>
> `protocol Animatable`
> A type for animating views.
>
> `var animatableData: Self.AnimatableData`
> The data to animate.
> **Required.** Default implementations provided.

We can turn a ***view modifier*** into an ***animatable modifier*** by changing the protocol to `AnimatableModifier`.

```
struct Cardify: ViewModifier, Animatable {
  private let cornerRadius: CGFloat = 10.0
  private let edgeLineWidth: CGFloat = 3

  var rotation: Double

  init(isFaceUp: Bool) {
    rotation = isFaceUp ? 0 : 180
  }

  var animatableData: Double {
    get { return rotation }
    set { rotation = newValue }
  }
```

`var animatableData: Double {`

- This *var* communicates between the *animation system* and our *view modifier* or our *shape*.
  - So, we just need to implement this `animatableData` *computed var*.
- It *animates* the *rotation* of our *view*.
- `Double` is our rotation.

`var rotation: Double`

- I renamed `rotation` to be `animatableData` because this is the name that the *animation systems* is going to look for.

```
struct Cardify: AnimatableModifier {
```

`struct Cardify: AnimatableModifier {`

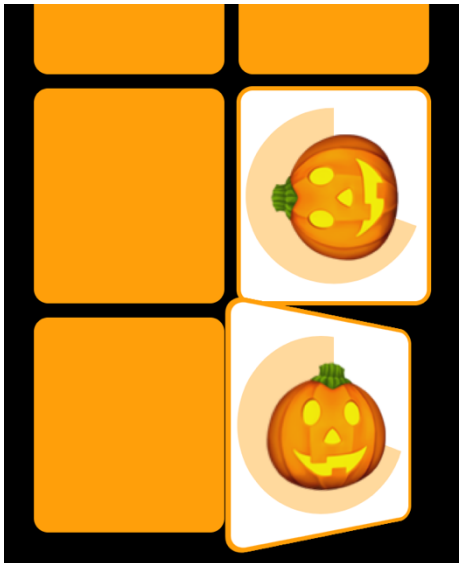
- You need to say that this is an ***animatable modifier*** because this `AnimatableModifier` *protocol* is just `Animatable` and `ViewModifier` together, it also signals to the system, I want to participate in the animation system.

In the ***first half of the flip***, when the card is face-down, it's only showing the back and when it's face-up, it's only showing the front.

- Notice also that there's ***no fading*** anymore.
    - This is because this *view modifier* has taken control of the *animation* and so the *animation system* is no longer trying to reach in here and do this *animation* itself.
        - It assumes that this *view modifier* knows what it's doing.

6.4.15 Card Flipping – Two cards spinning at the same time

1:06:28 Let's go back to the problem we had from the very beginning which is when two cards match.



One spins, and the ***other one not***.

- This doesn't spin because when we touched on the card it matched and it was faced-down at the time, and we switched it to face-up.

`Cardify`

```
  func body(content: Content) -> some View {
    ZStack {
      if isFaceUp {
        RoundedRectangle(cornerRadius: cornerRadius)
          .fill(Color.white)
        RoundedRectangle(cornerRadius: cornerRadius)
          .stroke(lineWidth: edgeLineWidth)
        content
      } else {
          RoundedRectangle(cornerRadius: cornerRadius)
            .fill()
      }
    } // ZStack
    .rotation3DEffect(Angle.degrees(rotation),
                      axis: (x: 0, y: 1, z: 0))
  } // body
} // Cardify
```

`if isFaceUp {`

- So, when it came on screen it was face-up and already matched…

EmojiMemoryGameView

```
@ViewBuilder
 private func body(for size: CGSize) -> some View {
 if card.isFaceUp || !card.isMatched {
  ZStack {
   Pie(startAngle: Angle.degrees(0-90),
    endAngle: Angle.degrees(110-90),
    clockwise: true
   )
    .padding(5)
    .opacity(0.4)
   Text(self.card.content)
    .font(Font.system(size: fontSize(for: size)))
    .rotationEffect(Angle.degrees(card.isMatched ? 360 : 0))
    .animation(card.isMatched ?
      Animation.linear(duration: 1).repeatForever(autoreverses: false) :
      .default)
  } // ZStack
   .cardify(isFaceUp: card.isFaceUp)
   .transition(AnyTransition.scale)
 } // if
} // body
```

`if card.isFaceUp || !card.isMatched {`

- …So, no change happened.
  - `card.isMatched` was already true so there was no need to apply any change.
    - ***Animations only animate changes***.

If we want a match happening, to be animated with the somersault we need the ***text of that card be on screen*** when the match happens.

- But, that's a ***problem for the second card in a match because it's face-down***.
  - We can have the emoji on screen, ***but hidden***.
    - That's another way to deal with having *views* that are ***appearing and disappearing*** instead of having them actually be ***if-then out of existence***.
      - The way we hide things is with something we already know from last time, ***opacity***.

Fully transparent is hidden and ***fully opaque*** is fully visible on screen.

- So, let's use *opacity* to have the back and front of our cards be showing or not, instead of using `if-then` that makes them completely disappear or not.
  - This is a different way of thinking about what's going on in this `ZStack`.

Cardify

```
  func body(content: Content) -> some View {
    ZStack {
      Group {
        RoundedRectangle(cornerRadius: cornerRadius)
          .fill(Color.white)
        RoundedRectangle(cornerRadius: cornerRadius)
          .stroke(lineWidth: edgeLineWidth)
        content
      } // Group
        .opacity(isFaceUp ? 1 : 0)
      RoundedRectangle(cornerRadius: cornerRadius)
        .fill()
        .opacity(isFaceUp ? 0 : 1)
    } // ZStack
      .rotation3DEffect(Angle.degrees(rotation),
                        axis: (x: 0, y: 1, z: 0))
  } // body
} // Cardify
```

```
Group {
…
} // Group
  .opacity(isFaceUp ? 1 : 0)
```

- Instead we *group* these three lines and set their *opacity*.
- If the card is face-up, it's fully opaque, otherwise it's fully transparent.
- Now there're no *ifs* inside the `ZStack`.
  - *Views* are not coming and going anymore…

`content`

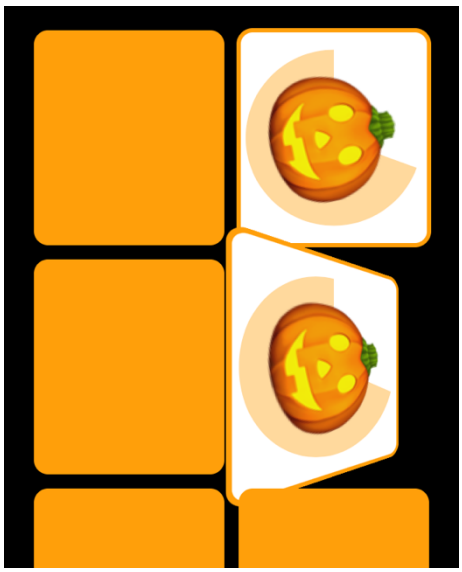
- …This also means that the *content*, *text*, is always on screen even when we're face-down,
  - It's just hidden…

EmojiMemoryGameView

```
@ViewBuilder
 private func body(for size: CGSize) -> some View {
 if card.isFaceUp || !card.isMatched {
  ZStack {
   Pie(startAngle: Angle.degrees(0-90),
    endAngle: Angle.degrees(110-90),
    clockwise: true
   )
    .padding(5)
    .opacity(0.4)
   Text(self.card.content)
    .font(Font.system(size: fontSize(for: size)))
    .rotationEffect(Angle.degrees(card.isMatched ? 360 : 0))
    .animation(card.isMatched ?
      Animation.linear(duration: 1).repeatForever(autoreverses: false) :
      .default)
  } // ZStack
   .cardify(isFaceUp: card.isFaceUp)
   .transition(AnyTransition.scale)
 } // if
} // body
```

`.animation(card.isMatched ?`

- … So, that means that when it gets set to match later this *implicit animation* will be a change until it will get to run.



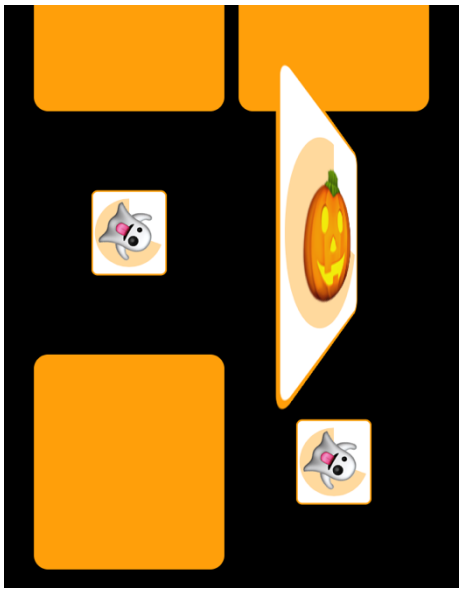
When two cards match, they both are spinning.

- Notice that the ***second card is already spinning*** while flipping over, because it was there hidden, and it was spinning until it became visible.

Animations can be done by *views coming and going* with *transitions*, or controlling it directly on-screen using *opacity*.

- Both of them are perfectly valid ways to go.
- You can decide whether it makes sense or not.
- Later, you're going to see that we actually are going to take advantage of making our little pie *come and go*.
    - When our pie is animating, we want it there as an animating thing, but we're going to put a different pie out there when it's not.
        - there's going to be a huge advantage to ***knowing when it's coming and going***.

So, it's not always the case that you want to use *opacity*, sometimes you want the views coming and going, it just depends whether you're triggering things off that, and whether you want to use *transitions*, or just normal *animations*.



6.4.16 Pie animation Bonus time

1:11:28 The last thing I'm going to do here is, now that we have our *animation*, working pretty well for *flipping* cards, and for having them *disappear*, I'm going to speed them back up again and then we're going to work on the *animation* of this little pie.

Whenever a card ***flips up***, we want it to ***start counting down*** and when it ***flips back down***, we ***stop counting*** .

- When it ***matches***, then we ***stop counting***.
- If it totally disappears, you don't get as many points as if you do it before it disappears.

To do this, we have to do a couple of things.

- One is we're going to have to kind of ***enhance our Model*** to know ***how much bonus time is left***.

6.4.17 Pie animation – Code tracking cards coming up and down

1:13:39 I have actually put some code that is tracking every time the card comes up and down…

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

var bonusTimeLimit: TimeInterval = 6 // zero is "no bonus available"

 // Last time this card was turned face up (and still face up)
 var lastFaceUpDate: Date?

 // The acumulated time this card has been face up in the past.
 var pastFaceUpTime: TimeInterval = 0;

 // How long this card has ever been faced up.
 private var faceUpTime: TimeInterval {
  if let lastFaceUpDate = self.lastFaceUpDate {
   return pastFaceUpTime +
    Date().timeIntervalSince(lastFaceUpDate)
  } else {
   return pastFaceUpTime
  }
 }

var bonusTimeRemaining: TimeInterval {
  max(0, bonusTimeLimit - faceUpTime)
 }

 // Percentage of the bonus time remaining
 var bonusRemaining: Double {
 (bonusTimeLimit > 0 && bonusTimeRemaining > 0) ?
bonusTimeRemaining/bonusTimeLimit : 0
 }

 // Whether the card was matched during the bonus time period.
 var hasEarnedBonus: Bool {
  isMatched && bonusTimeRemaining > 0
 }

 // Whether currently face up, unmatched and not yet used bonus
 var isConsumingBonusTime: Bool {
  isFaceUp && !isMatched && bonusTimeRemaining > 0
 }

 // Called when the card transition to face up state
 private mutating func startUsingBonusTime() {
  if isConsumingBonusTime, lastFaceUpDate == nil {
   lastFaceUpDate = Date()
  }
 }

 private mutating func stopUsingBonusTime() {
  pastFaceUpTime = faceUpTime
  self.lastFaceUpDate = nil
 }
} // struct Card
```

- … or gets mapped it tracking the time used
  - `private var faceUpTime: TimeInterval {`
- and then it answers questions like
  - how much time is remaining
    - `var bonusTimeRemaining: TimeInterval {`
  - or what percentage of the time is remaining
    - `var bonusRemaining: Double {`
  - and we can learn whether we earn the bonus
    - `var hasEarnedBonus: Bool {`
- and start using the bonus time
  - `private mutating func startUsingBonusTime() {`
- and stop using the bonus time.
  - `private mutating func stopUsingBonusTime()`

6.4.18 Pie animation – Model – isFaceUp property observer

1:13:58 I'm going to make sure that I ***call these functions when a card goes face-up or face-down*** in my Model and also when they match.

- To do this let's use ***property observers*** to call these functions.

```
  struct Card: Identifiable {
    var isFaceUp: Bool = false {
      didSet {
        if isFaceUp {
          startUsingBonusTime()
        } else {
          stopUsingBonusTime()
        }
      }
    }
```

`didSet {`

- Every time `isFaceUp` changes, I'm going to use `didSet` to start or stop using bonus time.
  - This is more reliable that trying to look at all the times I say `isFaceUp` is true or false,
    - and try to also call `startUsingBonustime()`. I might make a mistake and forget it.

```
mutating func choose(_ card: 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
 }
…
}
```

6.4.19 Pie animation – Model - animatableData

1:15:03

```
    var isMatched: Bool = false {
      didSet {
        stopUsingBonusTime()
      }
    }
```

`didSet {`

- The use of *property observers* is really a powerful way to sync up what's going on inside your code.

Now our Model knows how much time is remaining

6.4.20 Pie animation – View component Pie- Reflect the Model states with animations

1:15:31 For our card pie to *animate*, we have to enhance our *shape* to do *animation*.

```
struct Pie: Shape, Animatable {
```

- `Shape` *protocol* already inherits `Animatable` *protocol*.

We want to animate both, the start and the end angle.

- We use `AnimatablePair` to animate two things at once.

```
struct AnimatablePair<First, Second> where
       First : VectorArithmetic,
       Second : VectorArithmetic
```

A pair of animatable values, which is itself animatable.

```
struct Pie: Shape {
  var startAngle: Angle
  var endAngle: Angle
  var clockwise: Bool = false

  var animatableData: AnimatablePair<Double, Double> {
    get {
      AnimatablePair(startAngle.radians, endAngle.radians)
    }

    set {
      startAngle = Angle.radians(newValue.first)
      endAngle = Angle.radians(newValue.second)
    }
  }
```

```
var startAngle: Angle
var endAngle: Angle
```

- Thanks to connecting up two of our vars to the `AnimatableData` *var*, we're going to be able to redraw over and over during *animation* the start and end angle.
- This *animation system* is elegant because with ***just one var***, `AnimatableData`, as being the ***only entry point in both directions*** we are able to animate our pie.

6.4.21 Pie animation – View – Put all together

1:18:53

```
@ViewBuilder
 private func body(for size: CGSize) -> some View {
 if card.isFaceUp || !card.isMatched {
  ZStack {
   Pie(startAngle: Angle.degrees(0-90),
    endAngle: Angle.degrees(110-90),
    clockwise: true
   )
    .padding(5).opacity(0.4)
   Text(self.card.content)
    .font(Font.system(size: fontSize(for: size)))
    .rotationEffect(Angle.degrees(card.isMatched ? 360 : 0))
    .animation(card.isMatched ?
      Animation.linear(duration: 1).repeatForever(autoreverses: false) :
      .default)
  } // ZStack
   .cardify(isFaceUp: card.isFaceUp)
   .transition(AnyTransition.scale)
 } // if
} // body
```

`startAngle: Angle.degrees(0-90)`

- We leave the start angle always ***straight up***, zero degrees.

```
  ZStack {
    Pie(startAngle: Angle.degrees(0-90),
        endAngle: Angle.degrees(-card.bonusRemaining*360-90),
        clockwise: true
  )
```

`endAngle: Angle.degrees(-card.bonusRemaining*360-90)`

- We vary the end angle depending on ***how much time I have remaining***.
  - I'm going backwards because this pie is negatively going down to zero.

It's actually showing us the time remaining, but it ***didn't*** actually ***animate*** it.

6.4.22 Pie animation – View – Animate the second angle going from where it is now, around to zero

1:20:17 There's a bit of a challenge to animate the pie,

- because what this is really animating is the angle on the right going from where it is now, around to zero.

I told you that *animation* only shows you things that have already happened. But when the card appears, the ***clock*** starts going, it ***hasn't gotten to zero yet***.

- How do I start an *animation* that's going to have the second angle go to zero when zero hasn't happened yet?
  - That's a little bit of a conundrum that is going to prevent us from using the *bonus remaining* directly from the Model.
    - The Model is not constantly changing but ***we still have to be in sync with the Model***.

If I ask the Model what's the bonus percentage remaining (`card.bonusRemaining`),

- it will tell me the right answer it always does because that's the Model job.
  - However, the Model is not constantly changing.
    - It's doing its job but it does it in a way that it's not causing our UI to change.
  - We just can't use this directly from the Model, although we still have to be in sync with the Model

```
func onAppear(perform action: (() -> Void)? = nil) -> some View
```
Adds an action to perform when the view appears.

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

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

 @State private var animatedBonusRemaining: Double = 0
 private func startBonusTimeAnimation() {
  animatedBonusRemaining = card.bonusRemaining
   withAnimation(.linear(duration: card.bonusTimeRemaining)) {
   animatedBonusRemaining = 0
  }
 }

 @ViewBuilder
 private func body(for size: CGSize) -> some View {
  if card.isFaceUp || !card.isMatched {
   ZStack {
    Group {
     if card.isConsumingBonusTime {
      Pie(startAngle: Angle.degrees(0-90),
          endAngle: Angle.degrees(-animatedBonusRemaining*360-90),
          clockwise: true)
      .padding(5).opacity(0.4)
```

1.21:34

```
      Text(self.card.content)
       .font(Font.system(size: fontSize(for: size)))
       .rotationEffect(Angle.degrees(card.isMatched ? 360 : 0))
       .animation(card.isMatched ?
       Animation.linear(duration: 1).repeatForever(autoreverses: false) :
       .default)
     } else  {
        Pie(startAngle: Angle.degrees(0-90),
            endAngle: Angle.degrees(-card.bonusRemaining*360-90),
            clockwise: true)
     }
    }
   } // ZStack
    .cardify(isFaceUp: card.isFaceUp)
        .transition(AnyTransition.scale)
  } // if card.isFaceUp || !card.isMatched
 } // body
```

```
@State private var animatedBonusRemaining: Double = 0
…
Pie(startAngle: Angle.degrees(0-90),
    endAngle: Angle.degrees(-animatedBonusRemaining*360-90),
    clockwise: true
)
```

- I'm going to animate the *end angle* using my own temporary `animatedBonusRemaining` *var*

Somehow I have to make `animatedBonusRemaining` be the right values to cause the animation to happen.

- The first thing I have to do is get it to be synced up with the Model.
    - This will happen every single time the *pie view* comes on screen.

I'm going to make it only be on screen if my card *is consuming bonus time*.

- This is a function that calls a closure anytime the Pie view appears on screen.