NOTES

Course

CS193p – Spring 2020
Developing Apps for iOS

Professor

Paul Hegarty

University

Stanford

Version: 1.0
Date: May 28, 2020

72. The 5 Step Approach to solve any programming problem

1 Lec01. Course Logistics and intro to SwiftUI

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

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

1.1 Create a new SwiftUI project

Single View Application | ***Memorize***

1.2 ContentView.swift

19:07 `ContentView.swift`

- This is a Swift file.

```
import SwiftUI

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

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

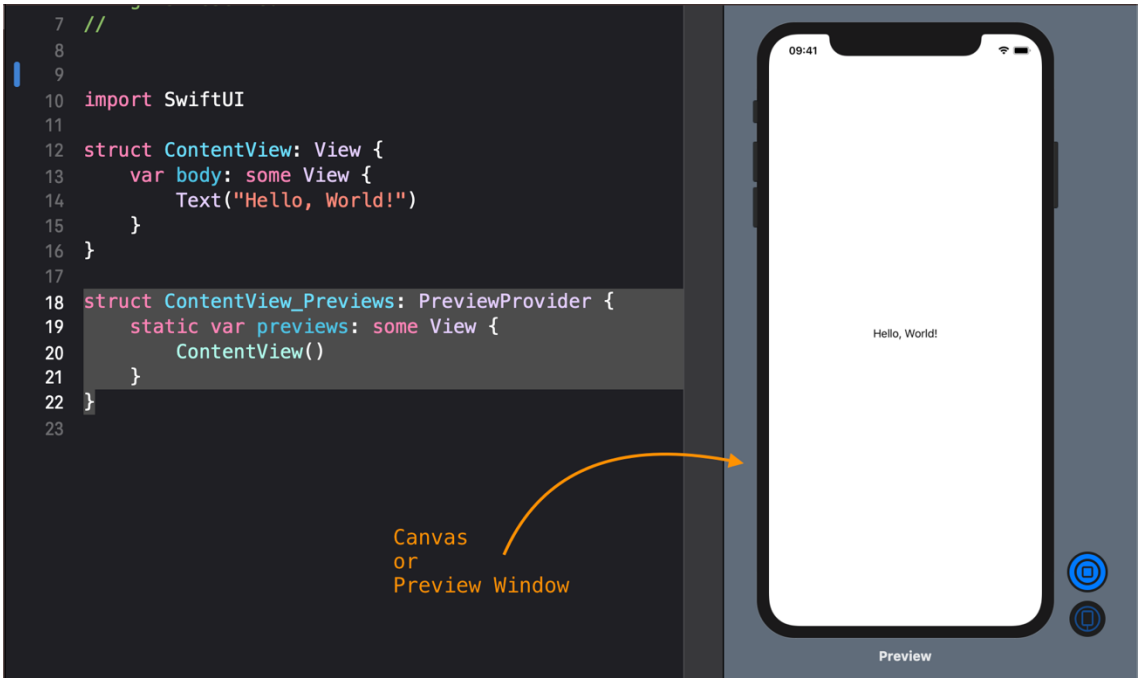
`struct ContentView_Previews: PreviewProvider {`

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

1.2.1 Canvas

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

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



1.2.2 import SwiftUI

21:10 `import SwiftUI`

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

1.2.3 Struct

```
import SwiftUI

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

`struct ContentView`

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

1.2.4 struct ContentView: View

`struct ContentView: View {`

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

1.2.5 Text

`Text("Hello, World!")`

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

1.2.6 var body: some View {

```
import SwiftUI

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

`var`

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

`body`

- It's the name of the property

`:some View {`

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

`:some`

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

You could kind of think of views as legos

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

`{ … }`

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

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

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

`Text("Hello There, World!"`

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

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

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

```
struct Text: View {

}
```

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

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

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

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

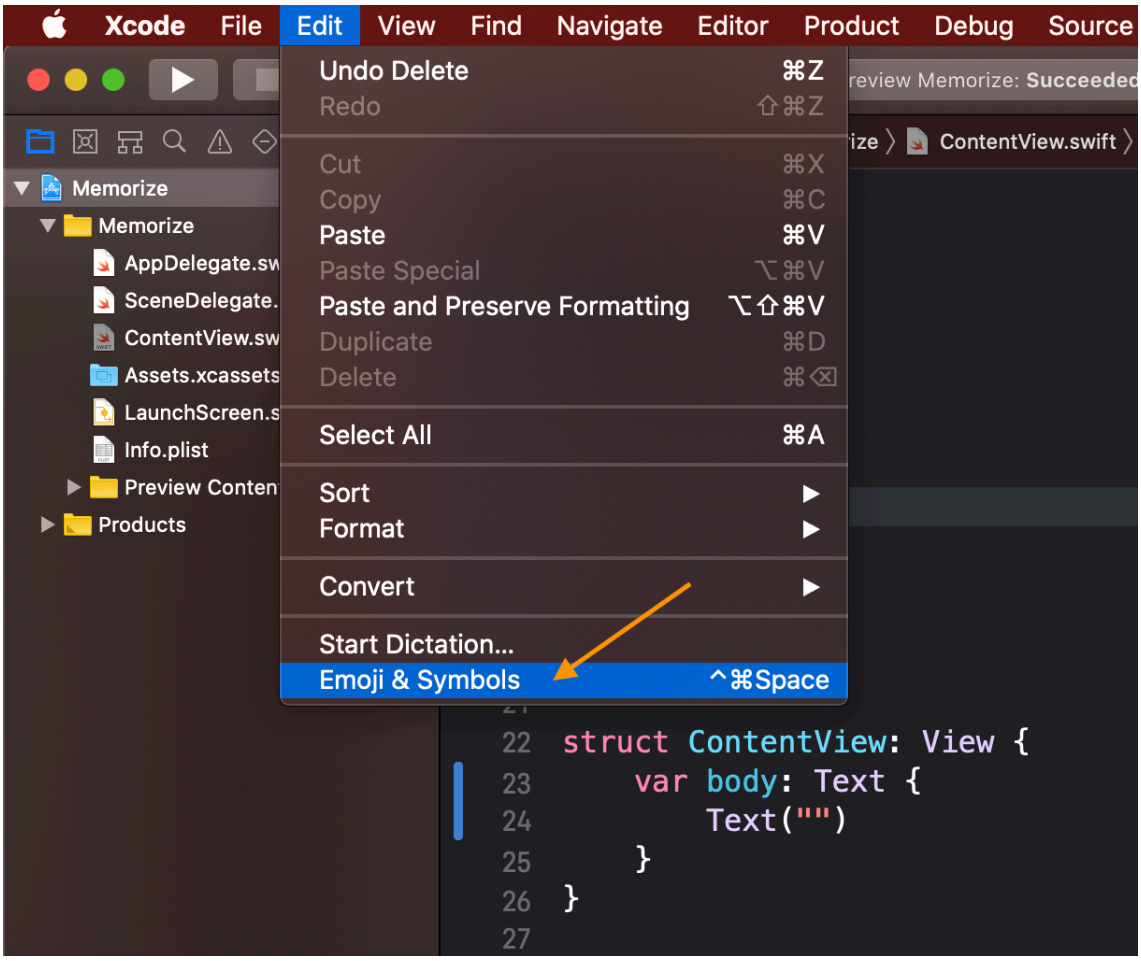
1.3 Build a UI that looks a little bit more like a card.

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

1.3.1 Emoji

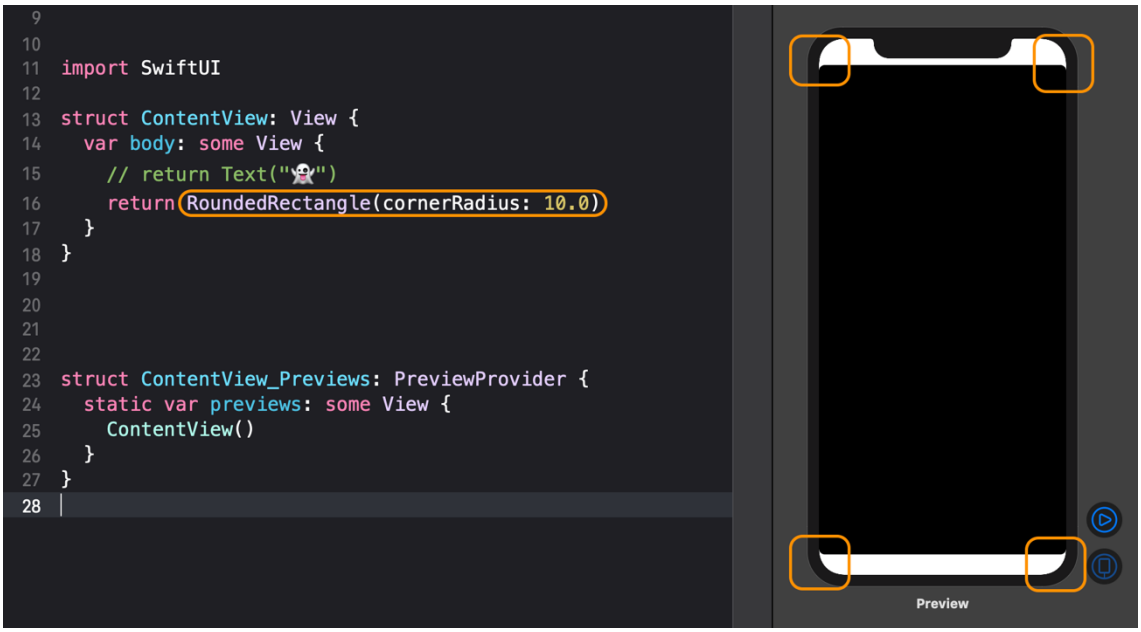
That's going to be really easy, because an emoji is just a piece of text.

- Edit | ***Emoji & Symbols***




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

1.3.2 Rounded rectangle



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

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

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

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

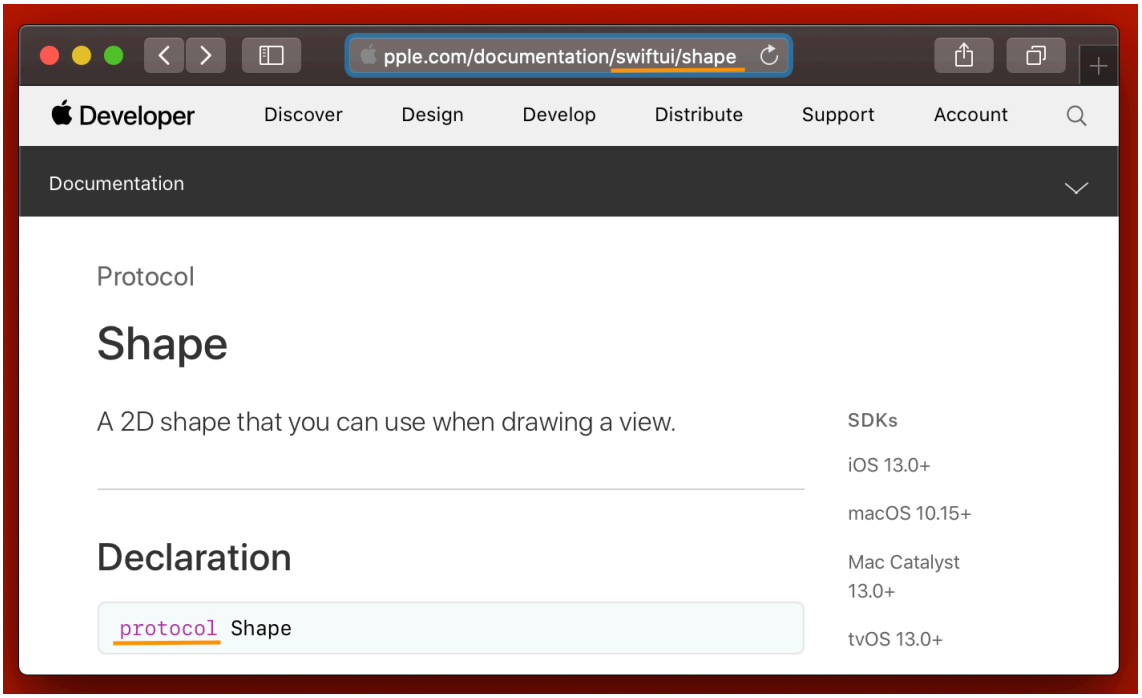
1.3.4 Views, Shapes and .stroke()

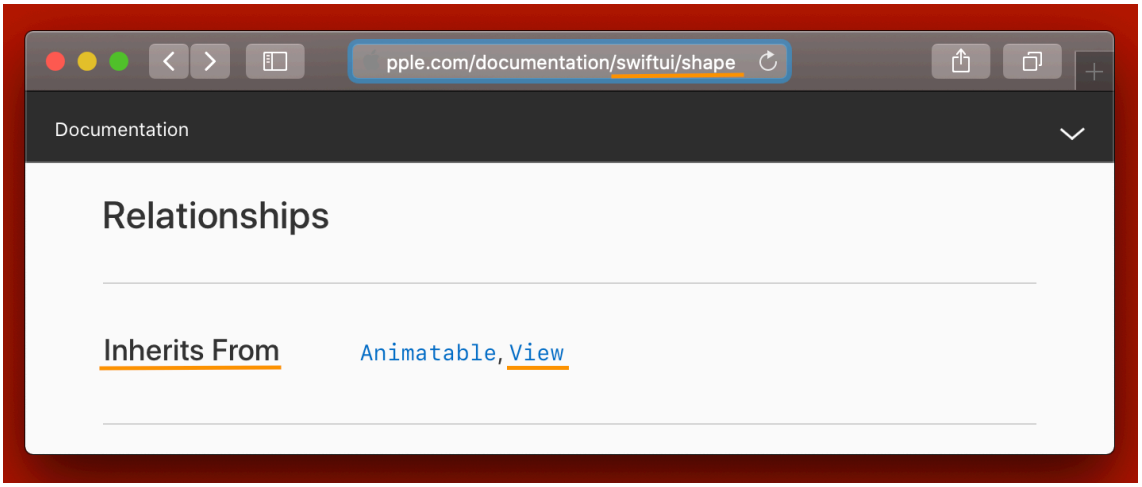
39:33 We call a function on `RoundRectangle` called `stroke()`.

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

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

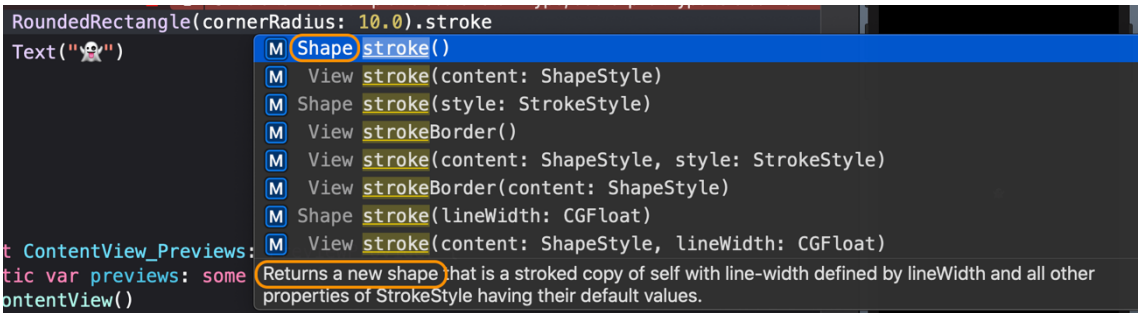
- Other ***shapes*** are `Circle` or `Capsule`.





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

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



DECLARATION

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

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

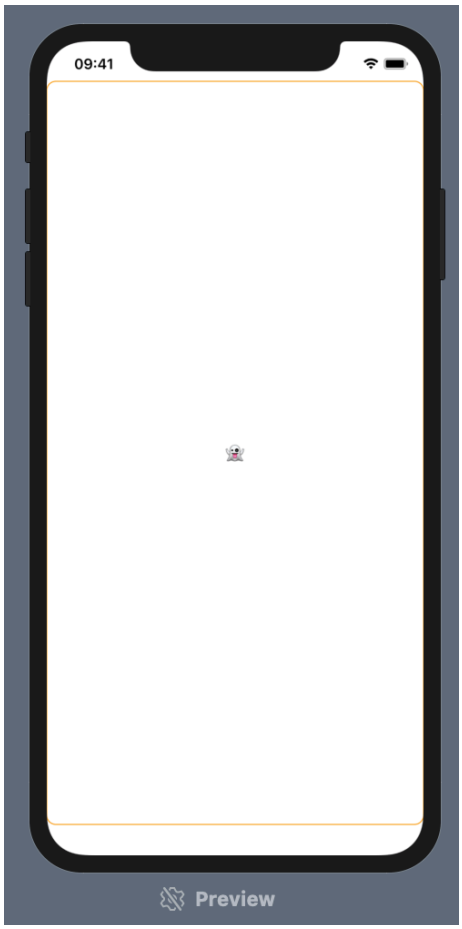
```
RoundedRectangle(cornerRadius: 10.0).stroke()
```

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

1.3.5 Orange foreground color

41:17

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



1.3.6 Padding

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

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

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

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

1.3.7 Fill

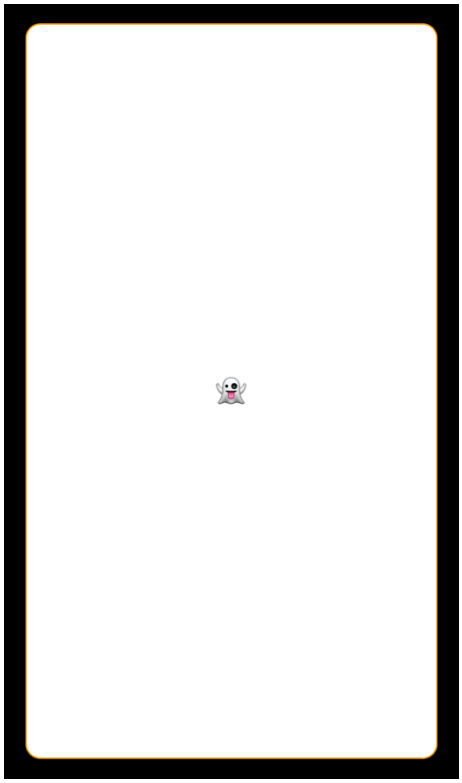
Unfortunately, there's no function you can send to a RoundedRectangle that says stroke it with one color and then fill it with another color.

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

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

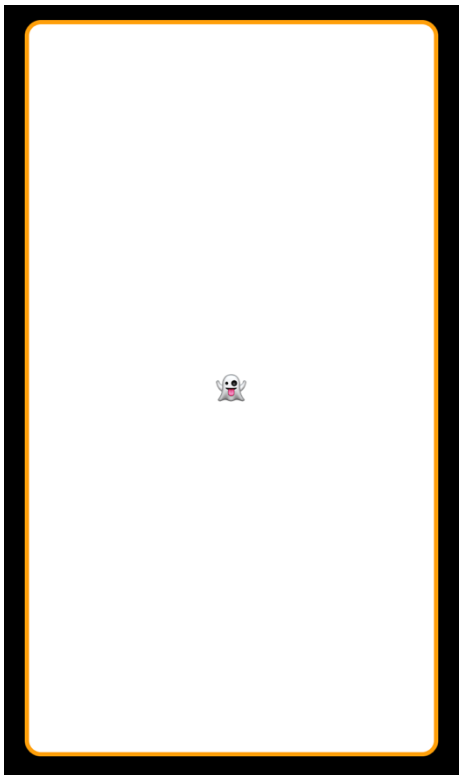
`RoundedRectangle(cornerRadius: 10.0).fill(Color.white)`

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

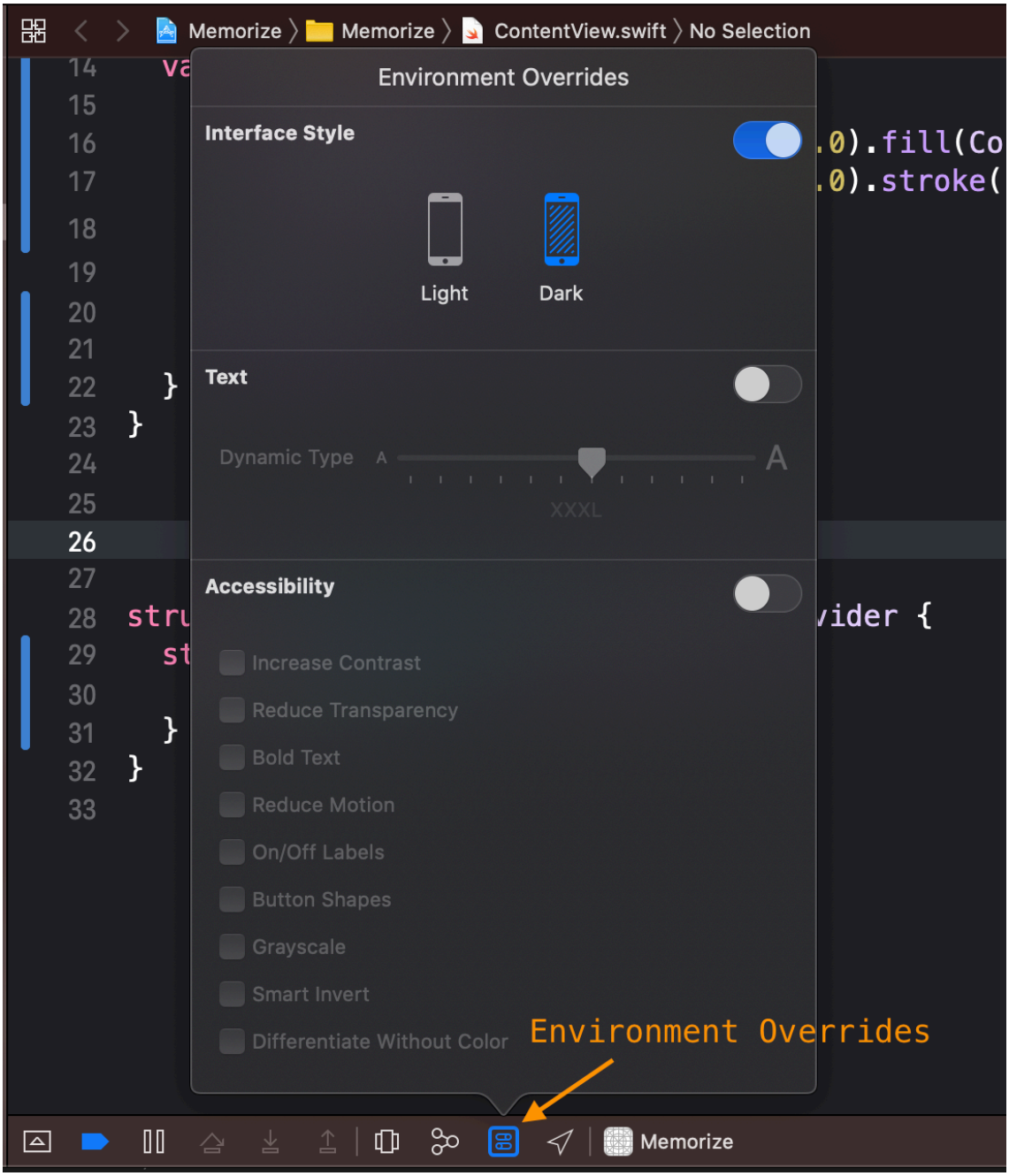


1.3.8 stroke(linewidth: 3)

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



1.3.9 Dark mode in simulator



1.3.10 font(Font.largeTitle)

47:44

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

The font can actually be put on the `ZStack`

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

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

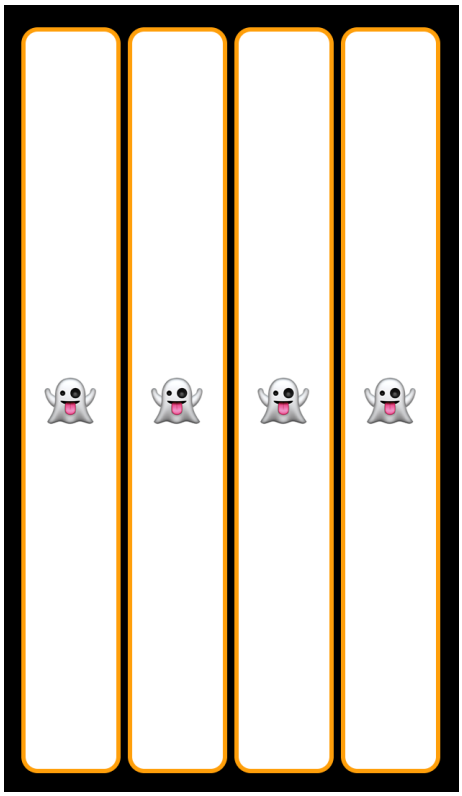
1.4 Multiple cards

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

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

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

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



1.4.1 #TBC

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

1.4.2

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

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

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

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

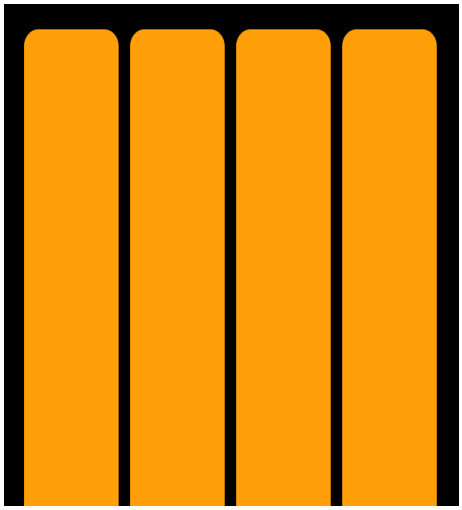
- to make our code look even simpler and nicer.

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

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

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

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

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



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

- The card is either face up or face down.

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


struct CardView: View {
  var isFaceUp: Bool

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

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

2 Lec02. MVVM and the Swift Type System

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

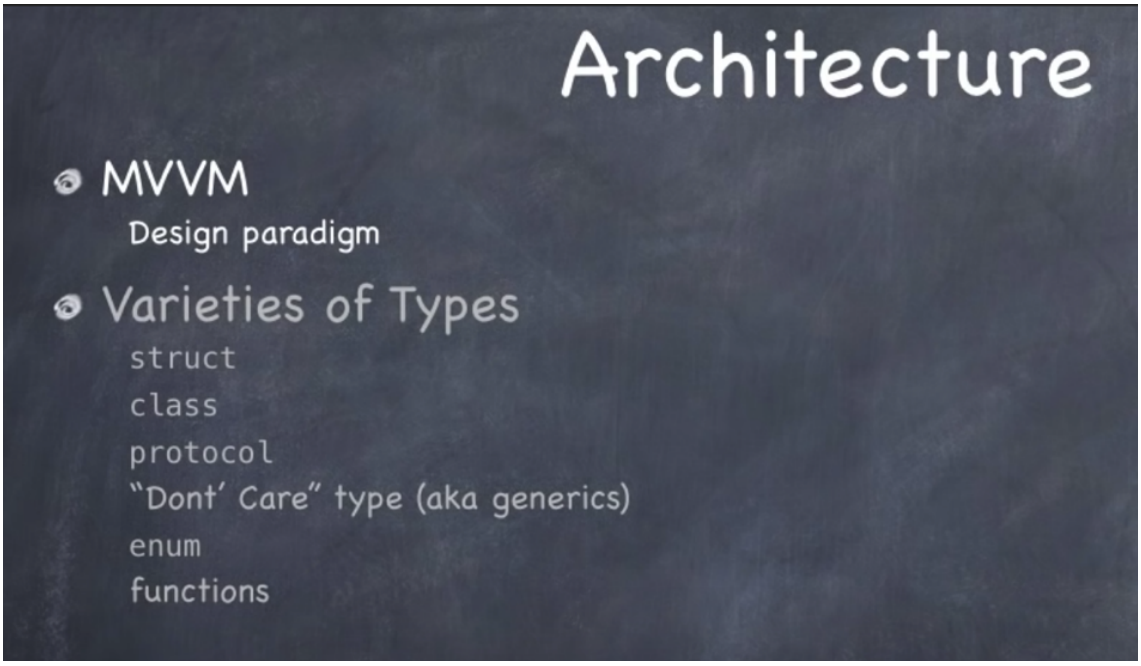
- In addition a key concept in the Swift Programming Language, its type system, is explained. The Memorize demonstration continues, incorporating *MVVM*.

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

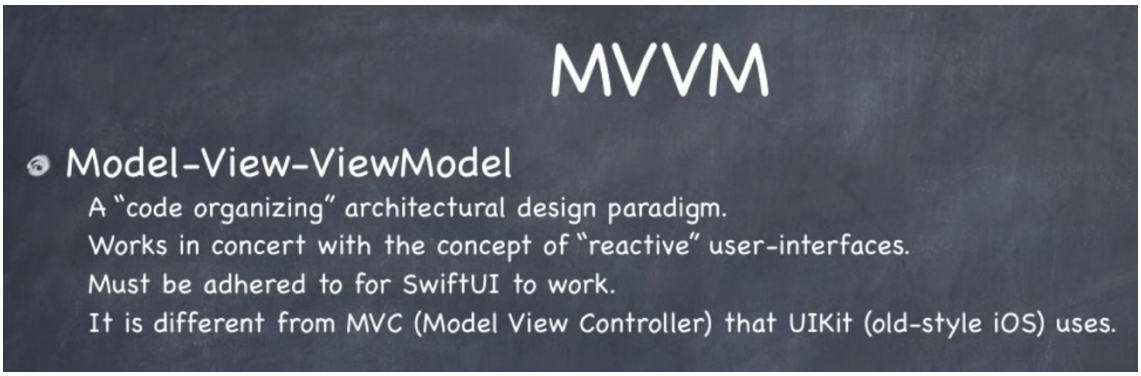
- This lecture explains what that is and then demonstrates how it works in our demonstration application.
- SwiftUI development happens entirely in the programming language Swift.
    - Swift is unique in its support of most modern language features, including both ***object-oriented programming*** and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

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

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



2.1 MVVM (Model-View-ViewModel)



A "code organizing" architectural design paradigm

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

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

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

2.1.1 MVVM - Model

01:18

UI Independent

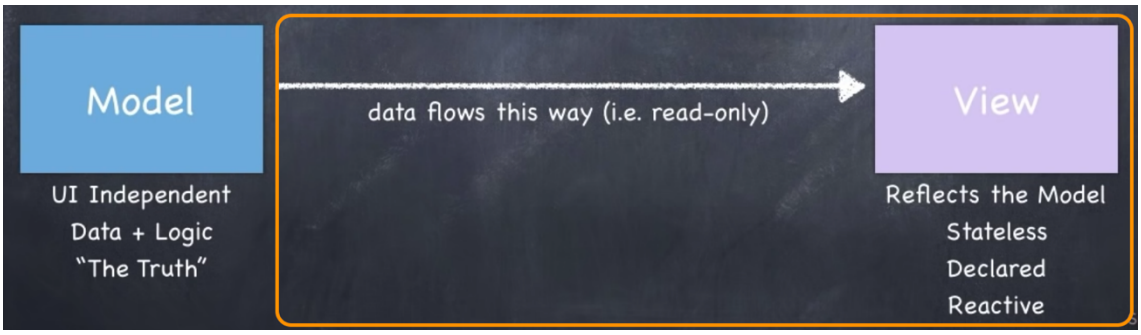
- The Model doesn't import `SwiftUI`, for example.

Data + Logic

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

"The truth"

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



Reflects the Model

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

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

Stateless

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

Declared

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

why is the imperative model so bad for UI?
The main reason has to do with ***time***.

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

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

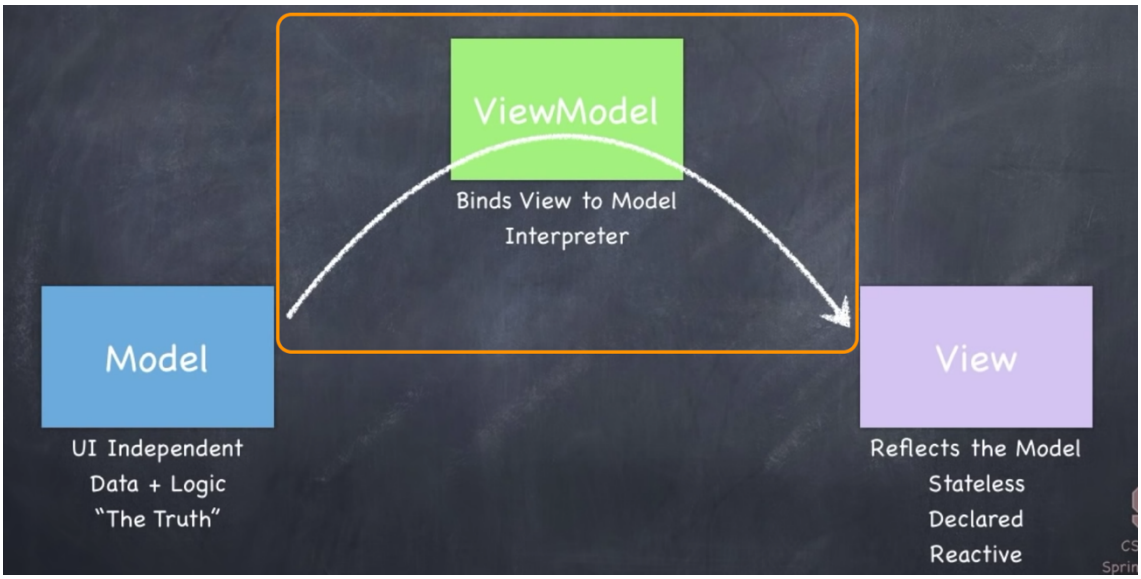
- Because our views are structs that are actually read-only by default,
  - no one is allowed to call a function that would change it.
    - It's not even possible to do. So that way, you can be sure that this view is always going to look like exactly what you see in the code.

Reactive

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

2.1.3 MVVM - ViewModel

07:30



Binds View to Model

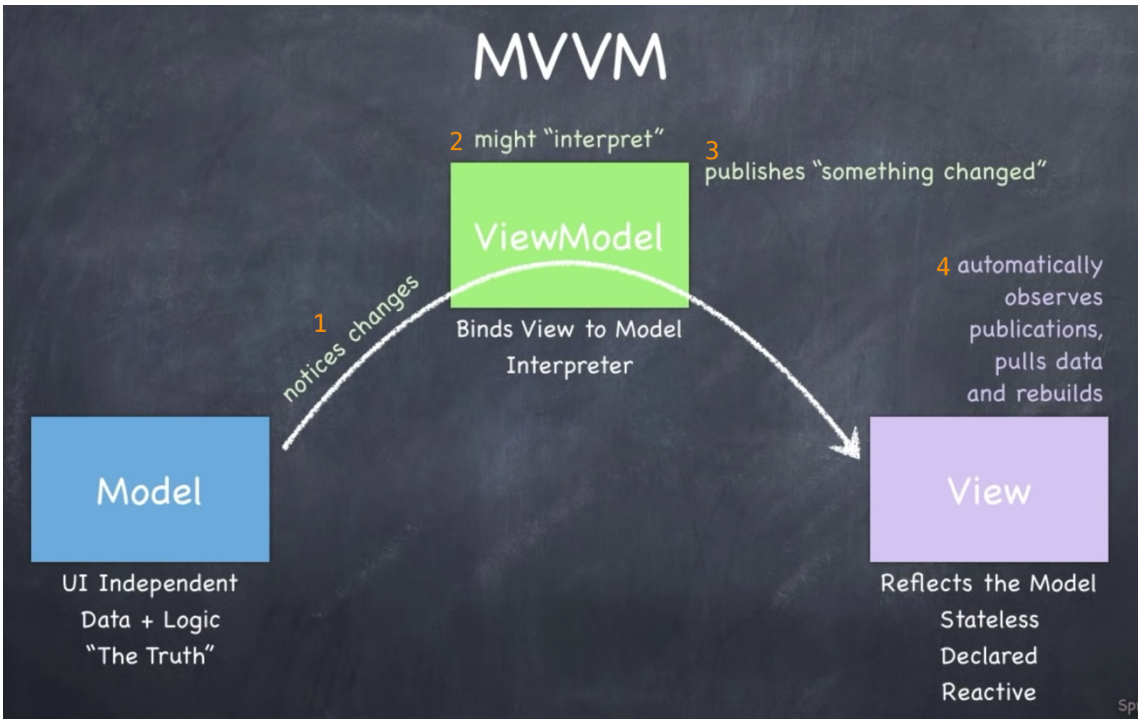
- The ViewModel's job is to ***bind*** the *View* to the ***Model***.
  - When the Model changes, the View get reflected.

Interpreter

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

2.1.4 MVVM - How does MVVM work?

08:48



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

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

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

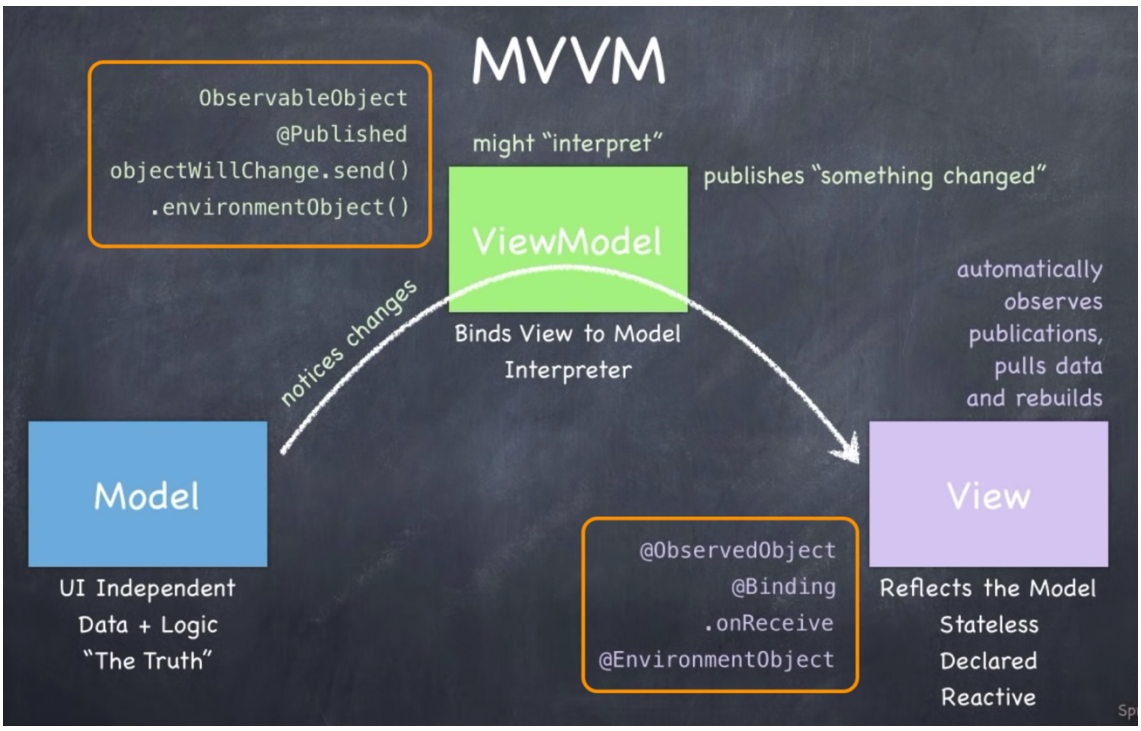
4. automatically observes publications, pulls data and rebuilds.
   - In fact, the View ***subscribes*** to that ***publication***, and when it sees that something has changed, it goes back to the ViewModel and asks,
     - what's the current state of the world?
       - I'm going to draw myself to match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

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

2.1.5 MVVM – Swift syntax

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

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

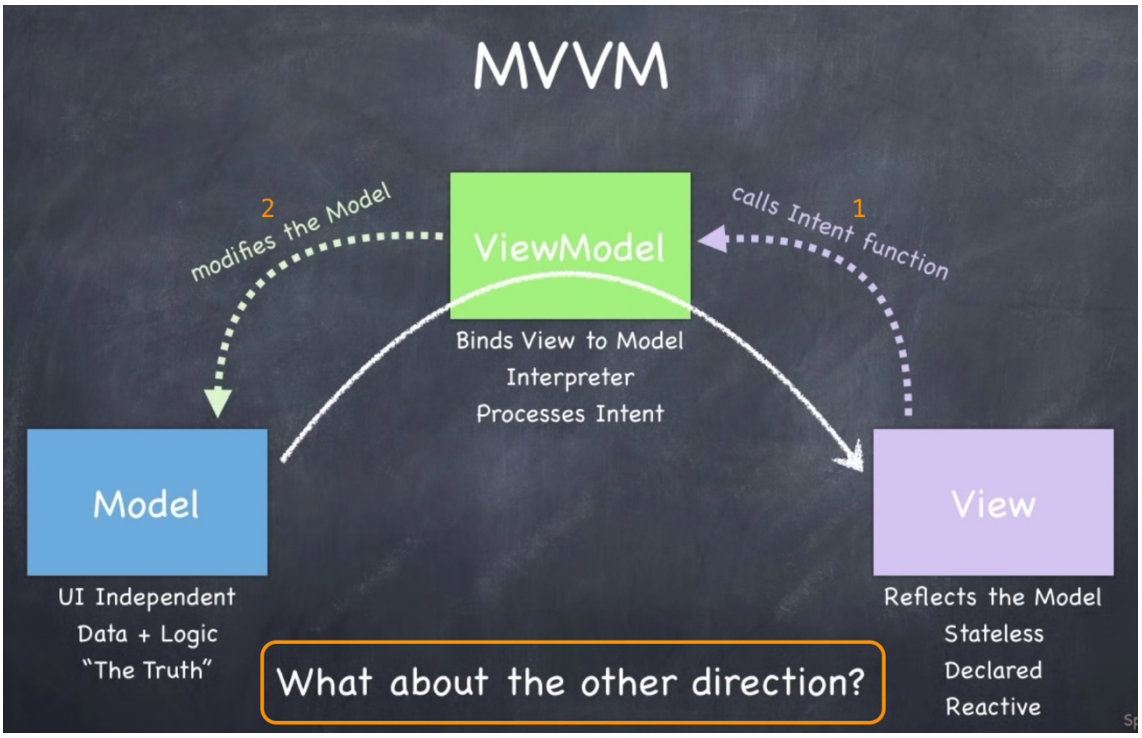


2.1.6 MVVM – What if the View wants to change the Model?

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

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

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

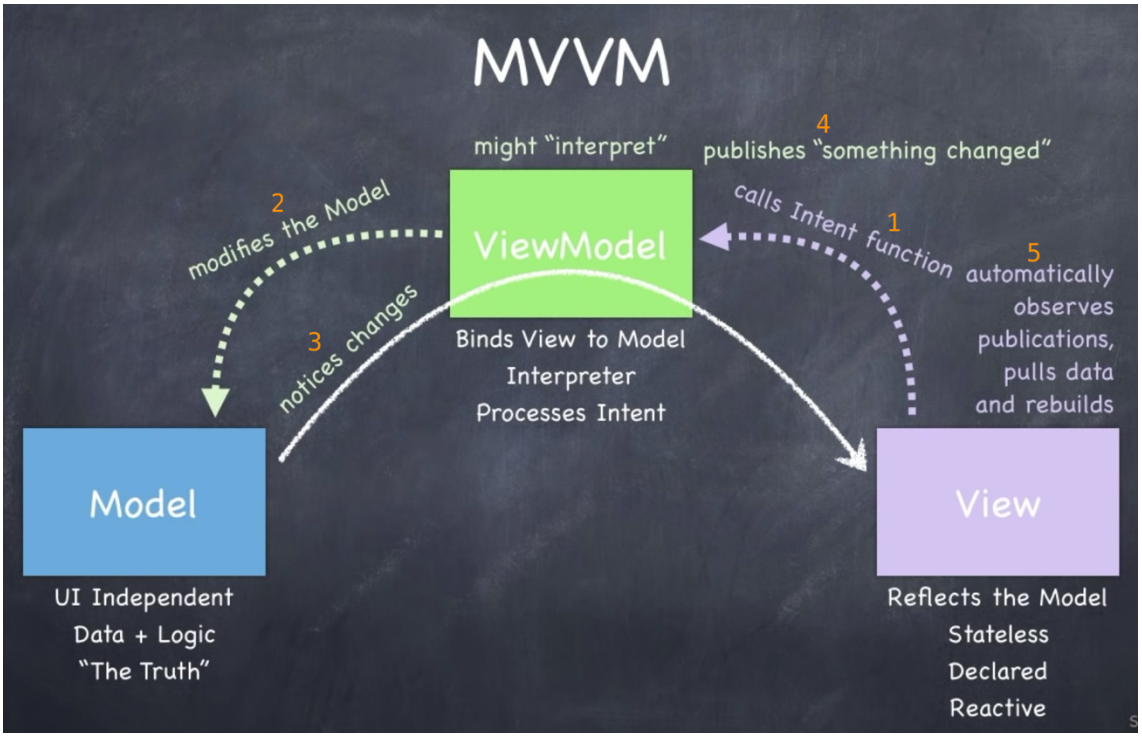


1. --- > calls Intent function

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

2. --- > modifies the Model

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



3. --- > notices changes

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

4. publishes "something changed"

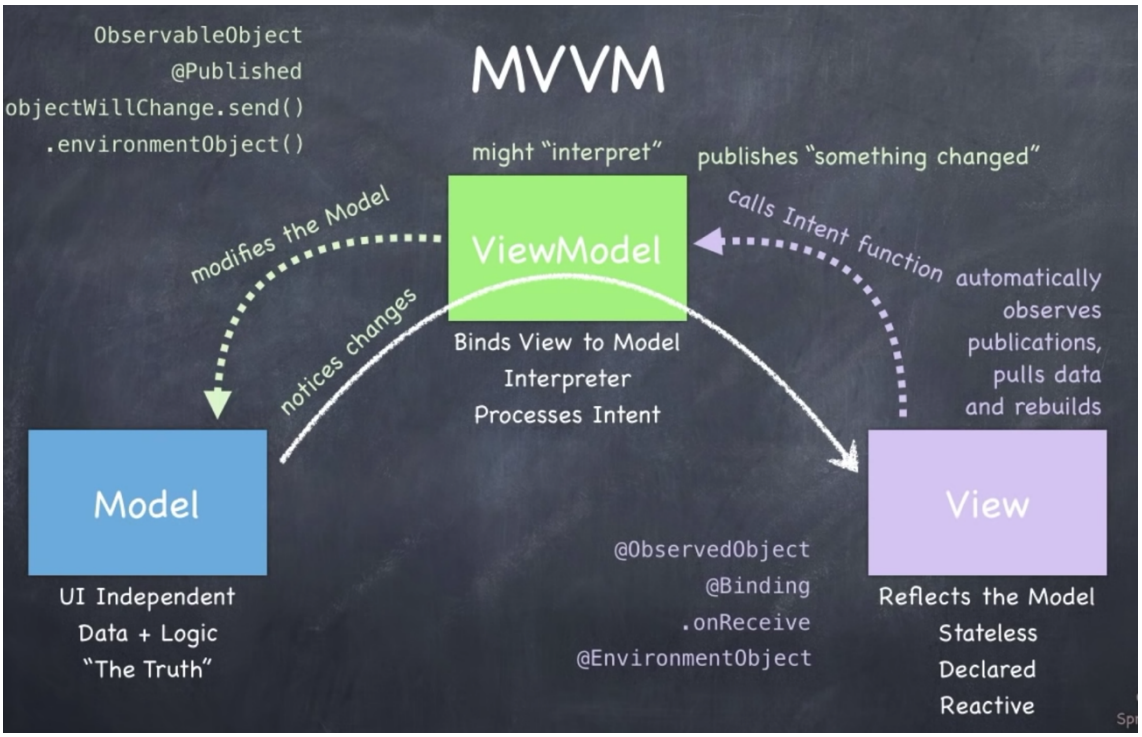
- it publishes something changed

5. automatically observes publications, pulls data and rebuilds

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

2.1.7 MVVM – This is it

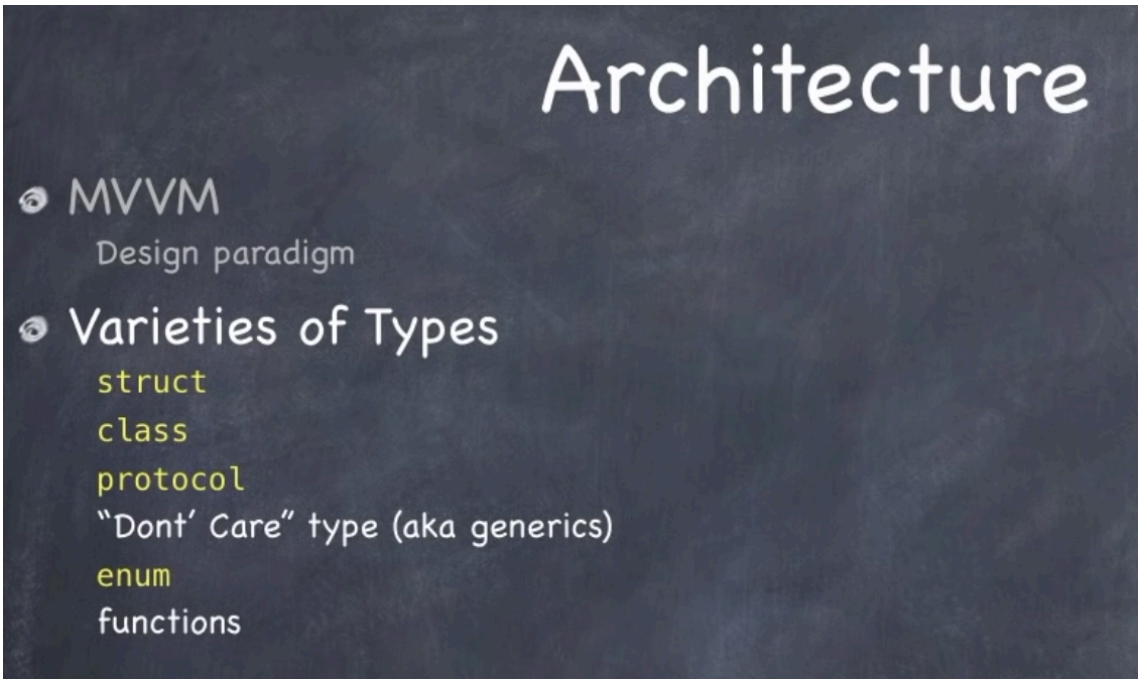
This whole picture is the MVVM architecture with its Swift keywords.



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

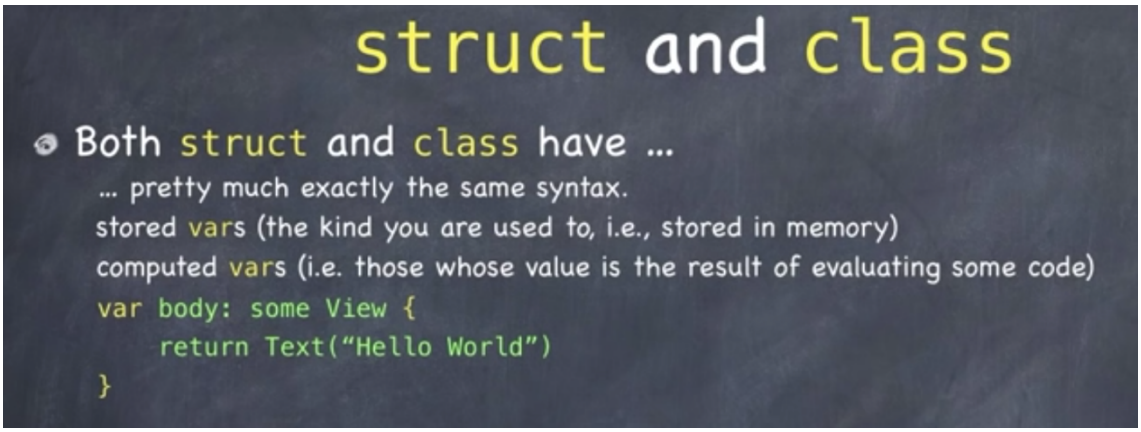
2.2 Types

14:55



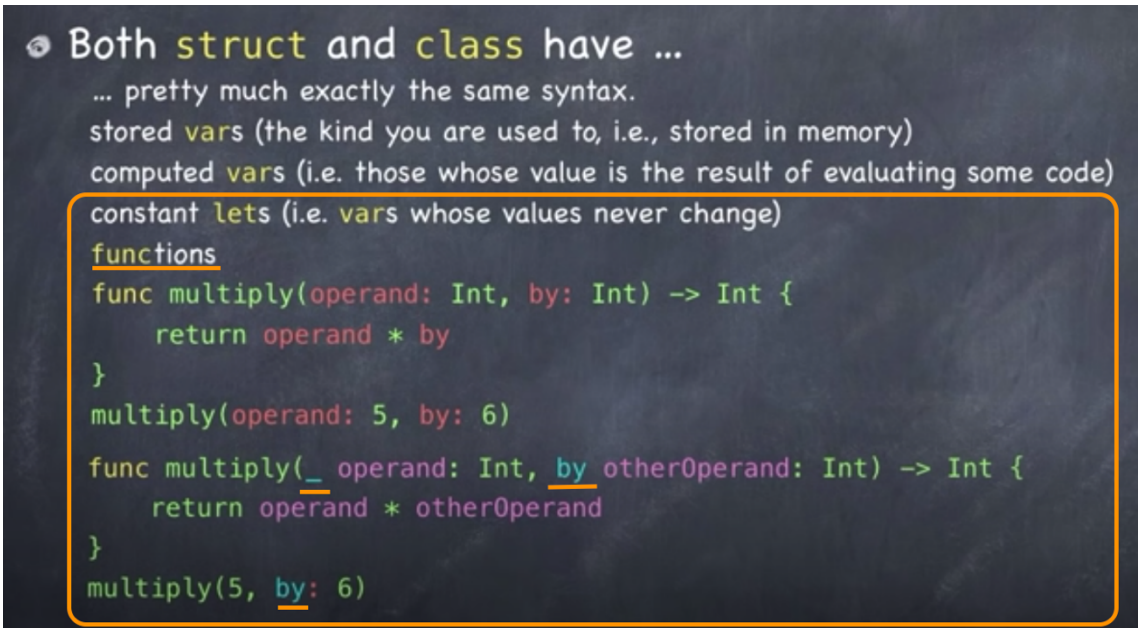
2.2.1 Structs and classes – Stored vars, computed vars

15:38



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

2.2.2 Structs and classes – Constant lets and functions

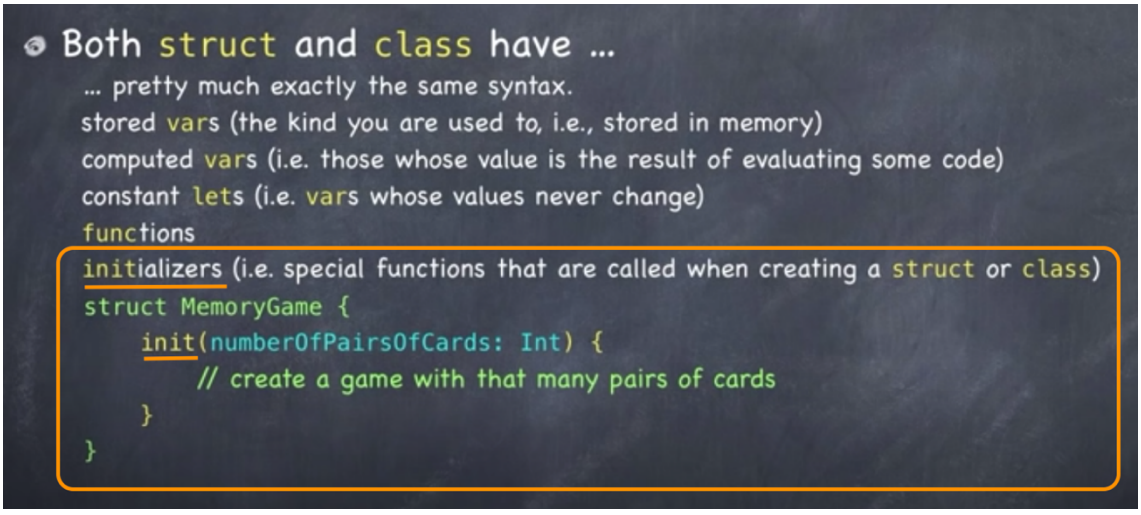


The arguments have labels, actually each parameter can have two labels.

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

2.2.3 Structs and classes – Initializers

18:40

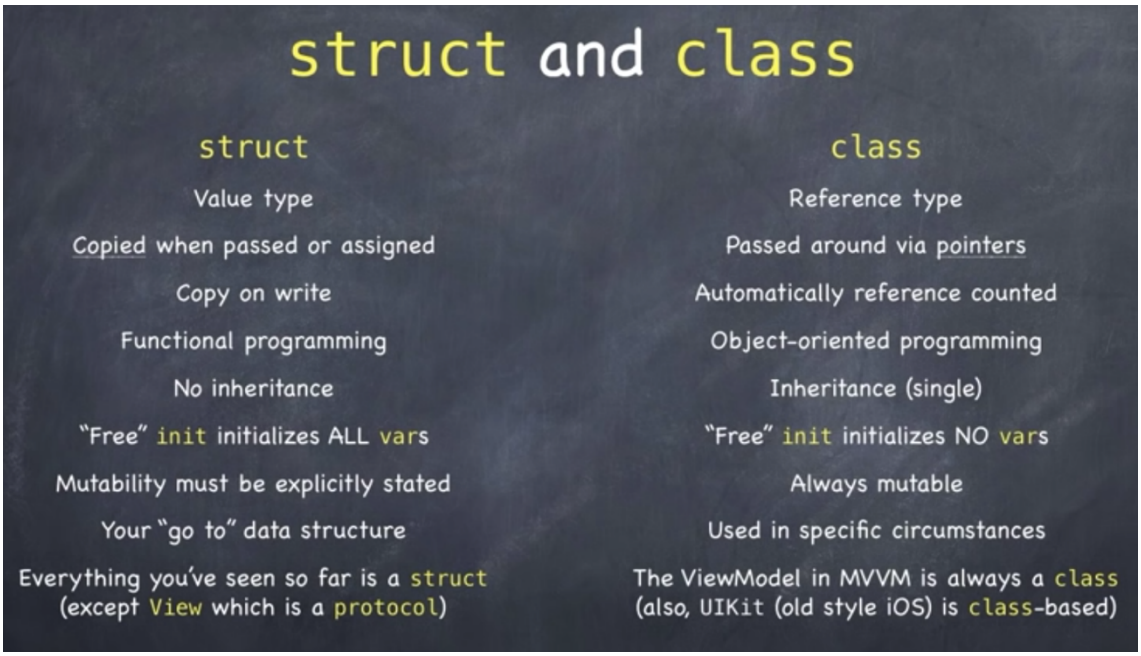


Put an `init(…)` function inside the struct which takes that as an argument.

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

2.2.4 What's the difference between structs and classes?

19:56



Passed around via pointers

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

Copied when passed or assigned

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

Automatically reference counted

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

Most things that you see are structs.

- Arrays, dictionaries, ints, bools, doubles

Functional programming

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

Object-oriented programming

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

28:15

Generics

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

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

- How Array uses a "don't care" type
  - Array's declaration looks something like this ...

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

  - The type of the argument to append is Element. A "don't care" type.
  - Array's implementation of append knows nothing about that argument and it does not care.
  - Element is not any known struct or class or protocol, it's just a placeholder for a type.
  - The code for using an Array looks something like this ...

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

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

- Note that Array has to let the world know the names of all of its "don't care" types in its API.
- It does this with the < > notation on its struct declaration Array<Element> above.
- That's how users of Array know that they have to say what type Element actually is.

```
var a = Array<Int>()
```

- It is perfectly legal to have multiple "don't care" types in the above (e.g. <Element, Foo>)

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

Functions as Types

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

You can declare a variable (or parameter to a func or whatever) to be of type "function".
The syntax for this includes the types of the arguments and return value.
You can do this anywhere any other type is allowed.

Examples ...

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

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

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

Example ...

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

This is a var called operation.
It is of type "function that takes a Double and returns a Double".

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

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

operation = square // just assigning a value to the operation var, nothing more
let result1 = operation(4) // result1 would equal 16
```

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

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

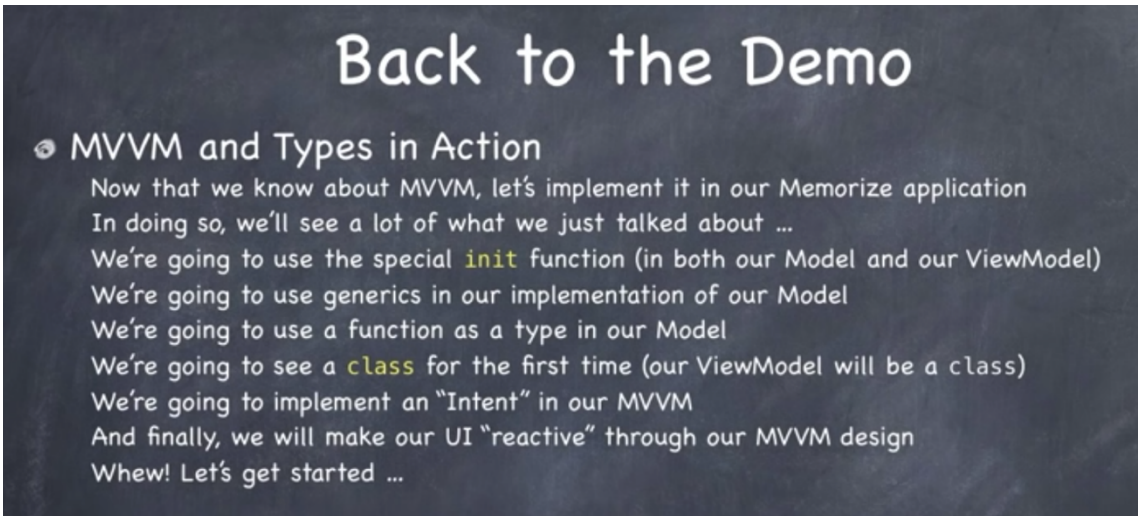
- Closures

It's so common to pass functions around that we are very often "inlining" them.
We call such an inlined function a "closure" and there's special language support for it.
We'll cover this in the demo and again later in the quarter.

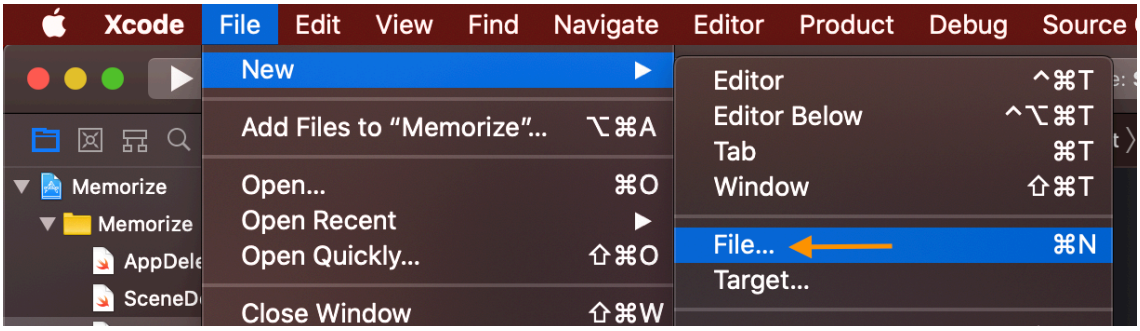
Remember that we are mostly doing "functional programming" in SwiftUI.
As the very name implies, "functions as types" is a very important concept in Swift. Very.

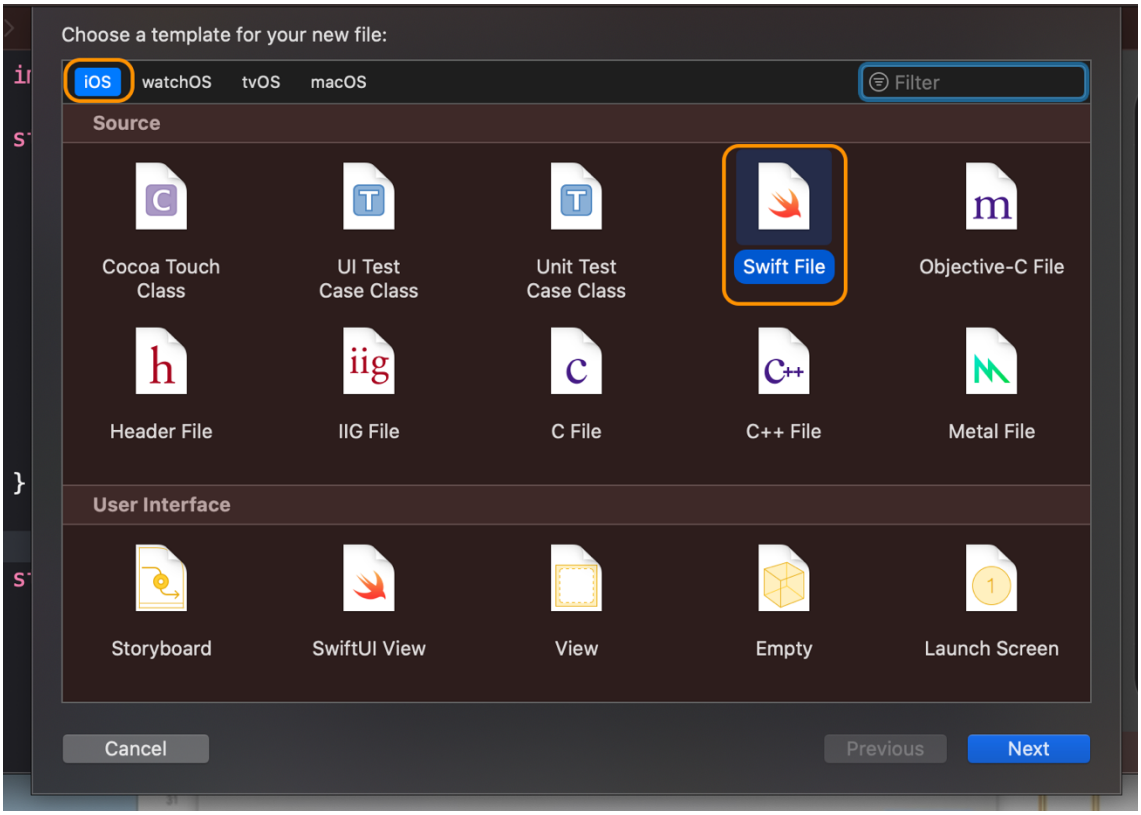
2.3 Back to the demo

36:40



2.3.1 Model – Create MemoryGame.swift





Save as: ***MemoryGame***

2.3.2 Model – MemoryGame.Card struct

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

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

```
import Foundation

struct MemoryGame {
  var cards: Array<Card>


  struct Card {

  }
}
```

`struct Card {`

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

2.3.3 Model – choose(card:) function

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

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

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


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


  struct Card {

  }
}
```

2.3.4 Model – isFaceUp and isMatched variables

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

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

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

2.3.5 Model – MemoryGame<CardContent>

44:37

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

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

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

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

2.3.6 ViewModel – Create EmojiMemoryGame.swift

46:27

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

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

Save As: ***EmojiMemoryGame***

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

2.3.7 ViewModel - Import Foundation

47:14

```
import Foundation
```

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

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

2.3.8 ViewModel – Create a property to access the Model

```
import Foundation

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

`var model: MemoryGame`

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

`var model: MemoryGame<String>`

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

`class EmojiMemoryGame {`

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

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

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

2.3.10 ViewModel - Intents

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

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

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

func choose(card: Card){

}
```

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

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

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

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

`model.choose(card: card)`

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

2.3.11 ViewModel – Intents -

56:51

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

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

  // MARK: - Intent(s)

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

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

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

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

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

2.3.12 ViewModel – Access to the Model

57:48

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

  // MARK: - Access to the Model
  //

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

  // MARK: - Intent(s)

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

`model.cards`

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

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

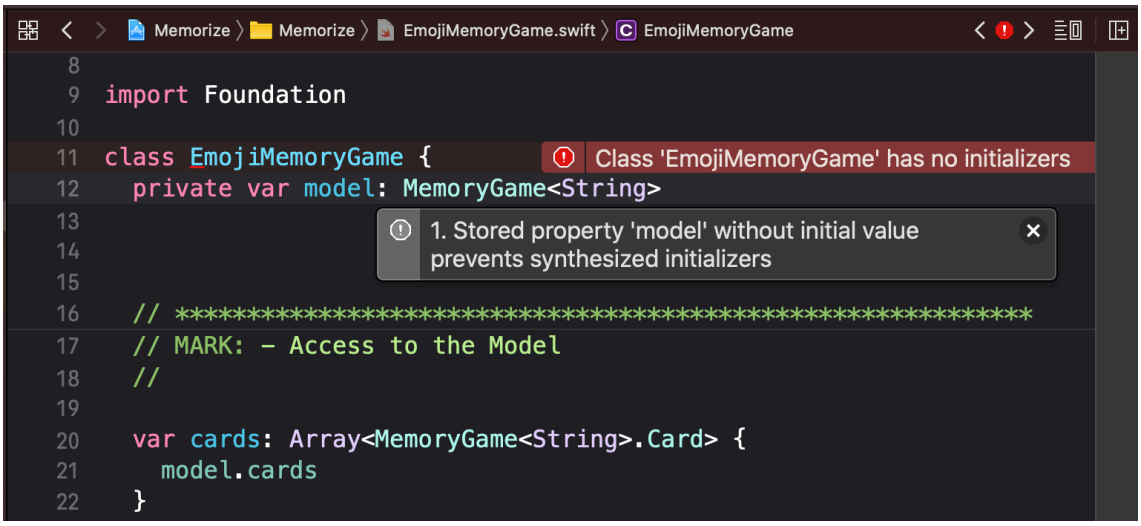
- I prefer this little more closed non-glass door, kind of approach to things, but occasionally, it makes sense to do that `private(set)` and let people have a glass door and see the Model, but you're always going to want to have the functions that play the role of intents.

`// MARK: - Intent(s)`

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

2.3.13 ViewModel – Class EmojiMemoryGame has no initializers

59:48

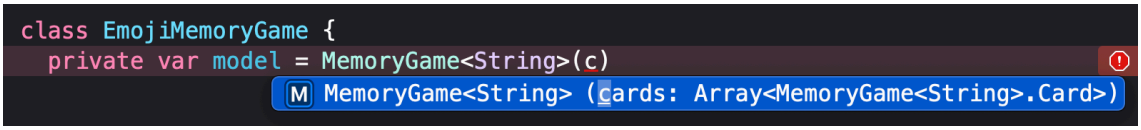



```
struct MemoryGame<CardContent> {
  var cards: Array<Card> // <- Not good place to initialize the cards
```

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

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

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




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

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

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

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

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

1:02:56

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

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


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

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

`init(numberOfPairsOfCards: Int) {`

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

`cards = Array<Card>()`

- We create an empty array of cards.

```
struct MemoryGame<CardContent> {
  …
  init(numberOfPairsOfCards: Int) {
    cards = Array<Card>()

    for pairIndex in 0..<numberOfPairsOfCards {
      cards.append(Card(isFaceUp: false,
                        isMatched: false,
                        content: CardContent))
      cards.append(Card(isFaceUp: false,
                        isMatched: false,
                        content: CardContent))
    }
  }
```

`for pairIndex in 0..<numberOfPairsOfCards {`

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

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

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

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

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

```
struct MemoryGame<CardContent> {
  var cards: Array<Card> // <- Not good place to initialize the cards
  …
  init(numberOfPairsOfCards: Int,
       cardContentFactory: (Int) -> CardContent) {
    cards = Array<Card>()

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

      cards.append(Card(isFaceUp: false,
                        isMatched: false,
                        content: CardContent))
      cards.append(Card(isFaceUp: false,
                        isMatched: false,
                        content: CardContent))
    }
  }
```

`cardContentFactory: (Int) -> CardContent`

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

`let content = cardContentFactory(pairIndex)`

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

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

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

`cardContentFactory: (Int) -> String`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`cardContentFactory: (pairIndex) {`

- We don't even really need the parentheses surrounding `pairIndex`.

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

`return "😀"`

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

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

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

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

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

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

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

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

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

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

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

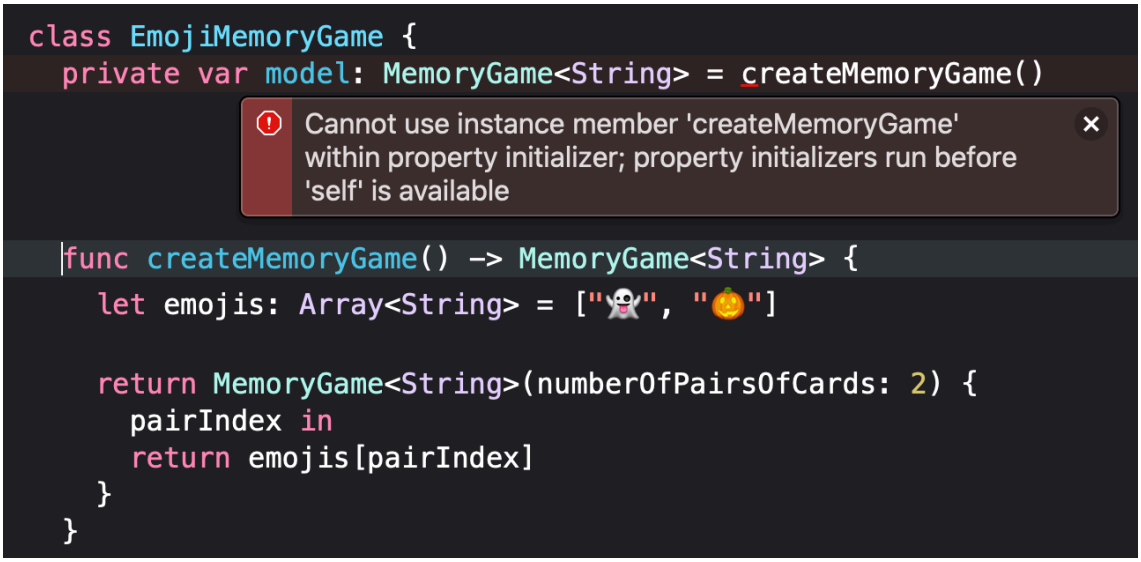
`let emojis: Array<String> = ["👻", "🎃"]`

- This is a constant array.

`return emojis[pairIndex]`

- My little card factory is just going to return emojis

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



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

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

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

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

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

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

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

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

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

```
import SwiftUI

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

struct CardView: View {
  var isFaceUp: Bool

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

`var isFaceUp: Bool`

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

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

```
import SwiftUI

struct ContentView: View {

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

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

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

`CardView(card: ...)`

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

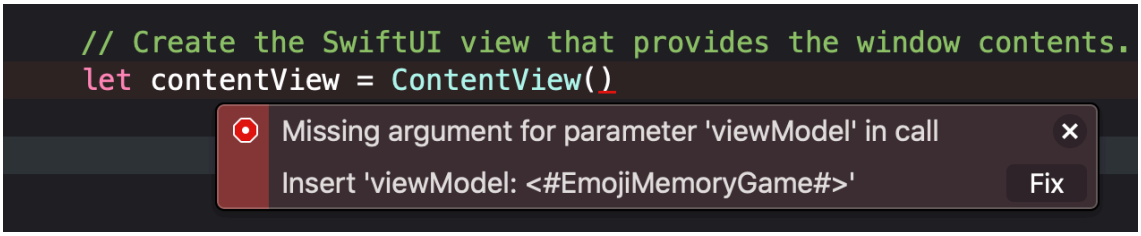
We're going to have to find some way to provide it with a card.

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

```
import SwiftUI

struct ContentView: View {
  var viewModel: EmojiMemoryGame

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

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

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

`var viewModel: EmojiMemoryGame`

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



```
import UIKit
import SwiftUI

class SceneDelegate: UIResponder, UIWindowSceneDelegate {

  var window: UIWindow?


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

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

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

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

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

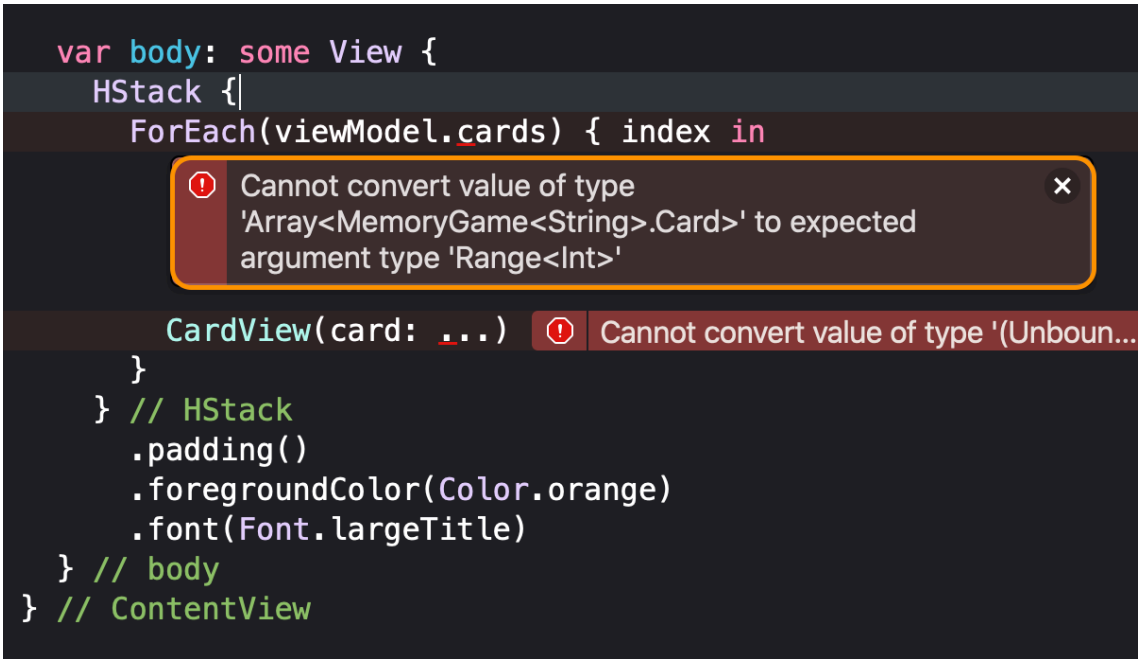
```
struct ContentView: View {
  var viewModel: EmojiMemoryGame

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

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

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

I told you this could be any iteratable thing.

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



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

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

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

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

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

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

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

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

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

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

    for pairIndex in 0..<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 = false
    var isMatched: Bool = false
    var content: CardContent
    var id: Int
  }
}
```

`struct Card: Identifiable {`

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

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

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

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

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

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

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

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

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

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

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

`CardView(card: card)`

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

2.3.25 Summary

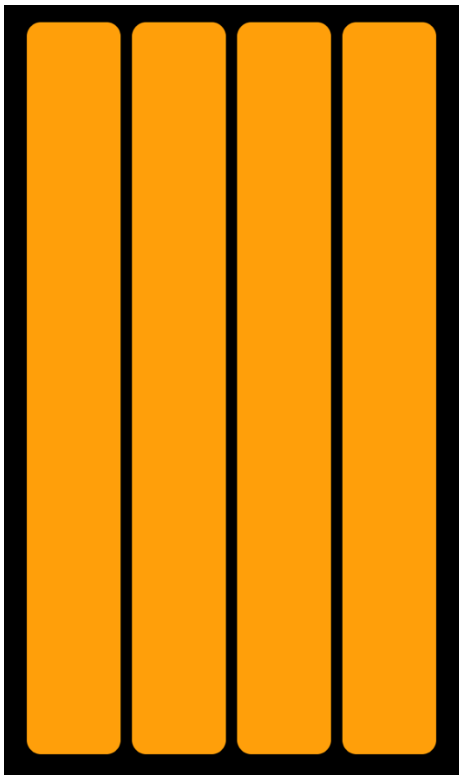
1:36.19 And that's it

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

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

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

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

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

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

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

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



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

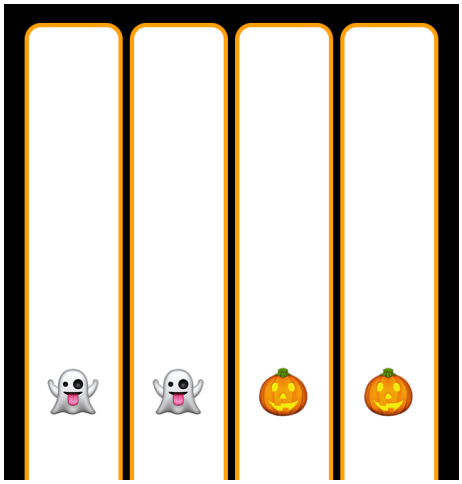
```
class EmojiMemoryGame {
  private var model: MemoryGame<String> =
    EmojiMemoryGame.createMemoryGame()


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

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

And in MemoryGame we have all our cards start facedown.

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

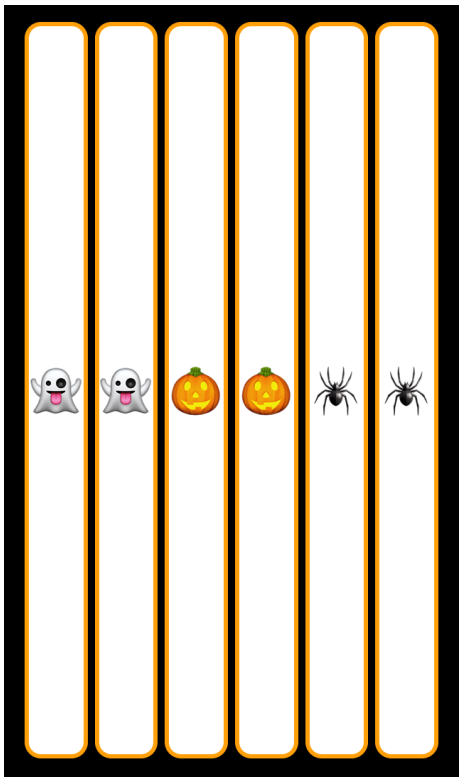
Let's change in our Model to be true

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



Now, we add another emoji.

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


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

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



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

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

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


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

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

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

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

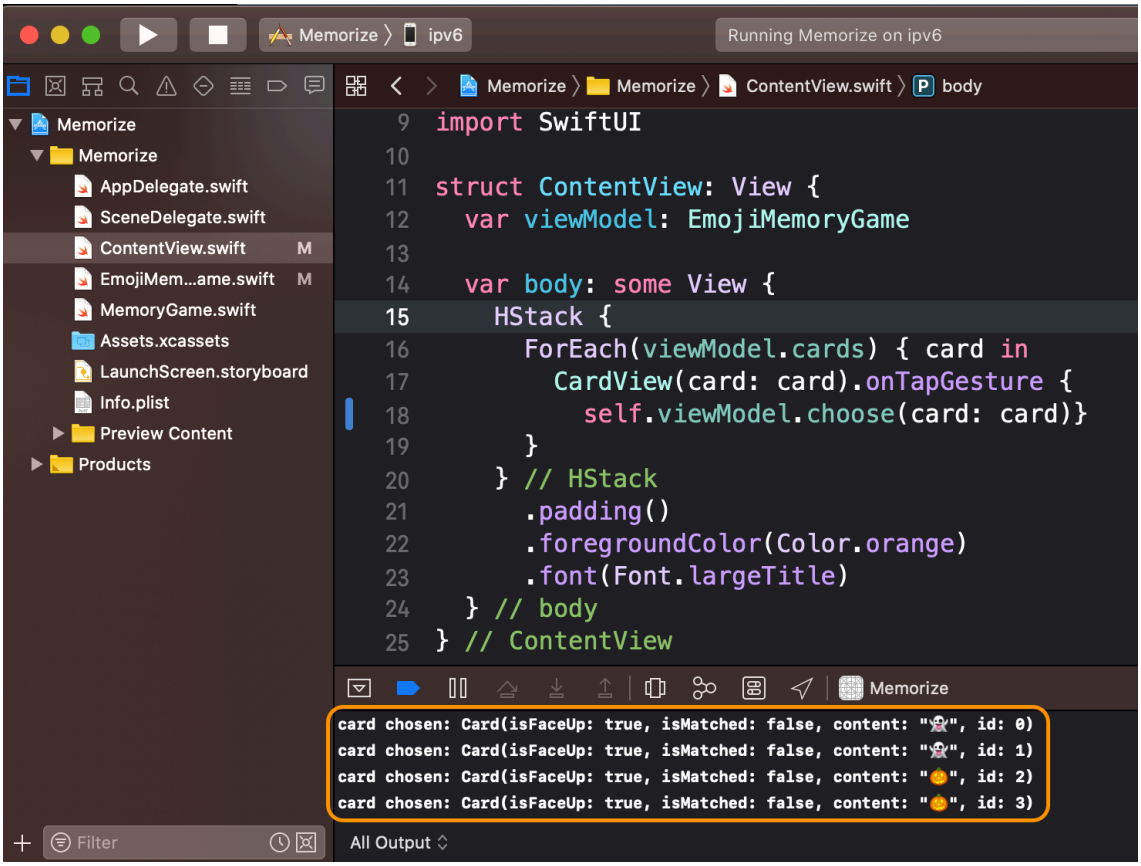
`self.viewModel.choose(card: card)`

- This is an intent to choose a card

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

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

2.3.28 Run




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

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

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

    for pairIndex in 0..<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
  }
}
```

2.4 Assignment I: Memorize

1:42:56

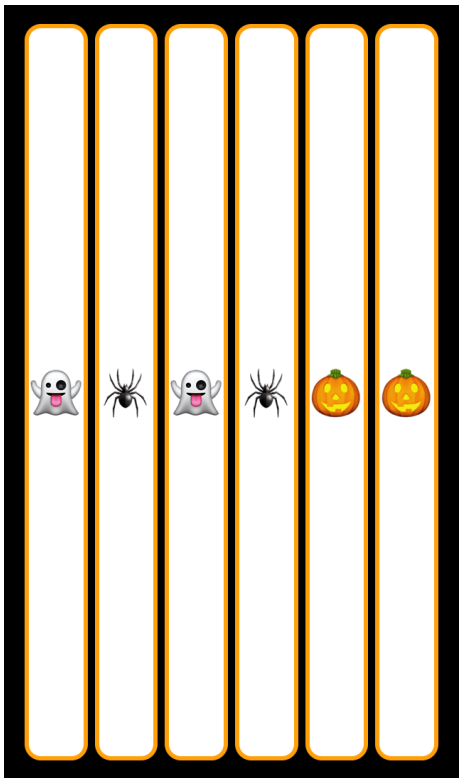
2.4.1 Required Task 1

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

2.4.2 Required Task 2

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

- Shuffle the cards.



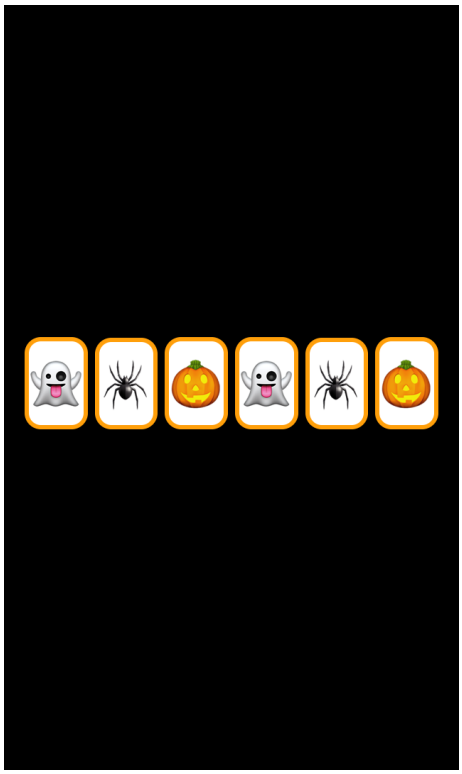
DECLARATION

```
mutating func shuffle()
```

2.4.3 Required Task 3

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

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

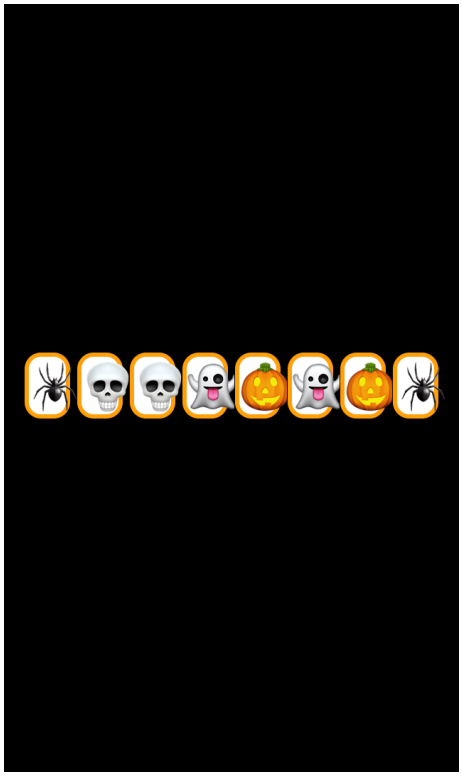


DECLARATION

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

2.4.4 Required Task 4

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



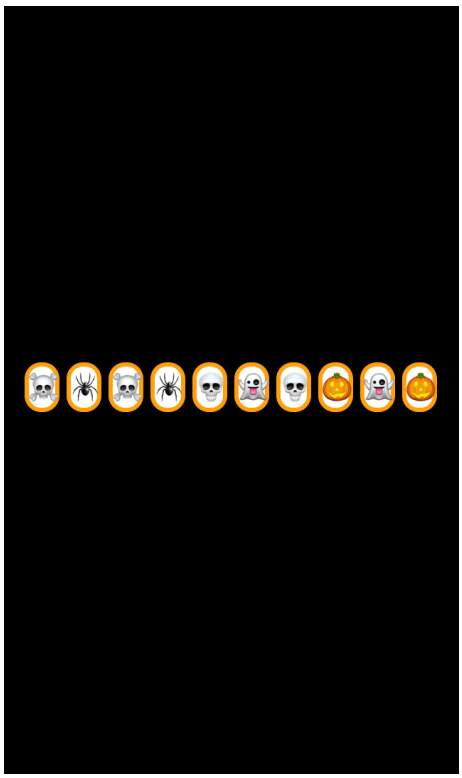
`Int.random(in:)`

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

2.4.5 Required Task 5

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

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



2.4.6 Required Task 6

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

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

2.4.7 Hints

1. Economy is valuable in coding. The easiest way to ensure a bug-free line of code is not to write that line of code at all. Required Tasks 2, 3 and 4 (and possibly even 5) can each be done with an addition or change to **a single line of code** (this is a Hint, not a Required Task, so you are in no way required to do that).
2. We haven't implemented the "reactive" part of SwiftUI yet (i.e. when the View automatically updates every time the Model changes), so you'll be re-launching your application over and over in the simulator (and observing it in the Preview pane) as you test your code.
3. Shuffling the cards might be easier than you think.
    - Be sure to familiarize yourself with the documentation for Array.
4. Be sure to do your shuffling in the proper place in your MVVM.
    - Is shuffling a UI thing (i.e. View or ViewModel) or a Model thing?
5. Another area of the documentation you will want to become very familiar with is the documentation for the `View` *protocol*.
    - There's a lot there (View is very powerful), and you are of course not required to understand every function in `View` (yet), but at least scanning/searching through it will both give you an idea of what's there and likely be helpful in solving your aspect ratio problem.
6. The function `Int.random(in:ClosedRange<Int>)` can generate a random integer in any range,
    - for example, `let random = Int.random(in: 15...75)` would generate a random integer between 15 and 75 (inclusive).
7. You'll definitely need to look at the documentation for `Font`.
    - You are welcome to use one of the other built-in fonts listed there (i.e. you don't have to call any of Font's functions like system or custom).
    - Just pick whichever built-in gives you the size you want since the font does not affect what an emoji looks like (other than size).
8. Required Task 5 can also be implemented in a single line of code.
    - But you'll likely need to use Swift's "ternary" operator (e.g. `let x = truthTest ? foo : bar`) to do so.
    - It is very common to use ternary operators to affect the arguments to modifiers of Views.
9. One thing we are not addressing in this assignment is that our emoji font looks **too small** in landscape. We'll be fixing this in a much more powerful way next week.

3 Lec03.

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

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

3.1.1

3.1.2

3.1.3

3.1.4

3.1.5

3.1.6

3.1.7

3.1.8

3.1.9

3.1.10