John Larkin
12/6/20
Developing Apps for iOS Spring 2020
Class Notes

Developing Apps for iOS Spring 2020

Table of Contents

Lecture 1: Course Logistics and Introduction to SwiftUI

- SwiftUI is a functional programming language
- ContentView : View this is NOT object oriented, just means that ContentView is going to function like a View
- Swift is a strongly typed language
- .foregroundColor can be called basically on every view
- https://www.youtube.com/watch?v=jbtqIBpUG7g&list=PLpGHT1n4-mAtTj9oywMWoBx0dCGd51_yG&index=14

Lecture 2: MVVM and the Swift Type System

- MVVM
  - Model-View-ViewModel

  - Design paradigm
  - Code organizing architectural design paradigm
  - Model
    - Data and logic
    - The truth
  - View
    - Declarative
    - Just going to look at the model and go from there
    - Different than imperative (imperial) – emperor goes around ruling. Imperative says put that button there and etc. Imperial is not great because you have to keep track of this other dimension – time.
    - All the code to draw your UI is all right in front of you.
    - Reactive programming
    - Swift syntax:
      - ObservableObject
      - @Published
      - objectWillChange.send()
      - .environemntObject()
  - ViewModel
    - Binds the View to the Model
    - Interpreter
    - Hopefully not a lot of code
    - Our model is just going to be a little struct… but could be a SQL database. Could be making HTTP calls
    - Up to the ViewModel to notice changes and the publishes
    - Automatically observes publications, pulls data and rebuilds
    - The View subscribes
    - Swift Syntax
      - @ObservedObject
      - @Binding
      - .onReceive
      - @EnvironmentObject
  - Model -> ViewModel (on model changes) -> View
  - Another related architecture Model View Intent
    - Some user intent : in our example, choosing a card
    - View would call an Intent function – makes it very clear what the User can do to modify the model
- Varieties of Types
  - Struct
  - Class
  - Protocol

    - Don’t care types (generics)
    - Enum
    - Functions (yes they’re types in swift)
- Struct and class
    - Syntax is basically the same
    - Stored vars (stored in memory)
    - Computed vars (value is the result of evaluating some code)
    - Constant lets (vars whos values never change)
    - Also functions
        - Functions can have two different labels – one for exposed vs one for internal usage
        - _ means don’t have an external one
    - Initializers
        - Init(arguments : Int) {
        -   // create a game with that many pairs of cards
        - }

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

- Generics
    - Sometimes we don’t really care about the types
    - Swift is very strongly typed
    - Array: contains a bunch of things and it doesn’t care what type they are
    - This gives us to Generics
    - Struct Array<Element> {
        - Func append(_ element: Element) { }
    - }
- Functions
    - (int, int) -> Bool // takes two ints and returns a bool
    - (double) -> Void
    - () -> Array<String>

  - So this is totally legal
  - Var foo: (double) -> Void
  - Var operation: (Double) -> Double
  - **Note that we don't use argument labels) when executing function types**
- Functions as Types
  - Closures
    - Often inling them
    - Capturing local variables and such

Extra Reading

- **A Swift Tour**
  - To create an empty array or dictionary, use the initializer syntax.
    - let emptyArray = [String]()
    - let emptyDictionary = [String: Float]()
  - If type information can be inferred, you can write an empty array as [] and an empty dictionary as [:]-for example, when you set a new value for a variable or pass an argument to a function.
    - shoppingList = []
    - occupations = [:]

Lecture 3: Reactive UI + Protocols + Layout

- protocol
  - this is basically an interface
- extension
  - you can basically use this to add things to structs and classes
  - you can even make it implement a protocol
- protocols are important
  - it's what functional programming is all about
  - formalizing how data strucutres in our application function
  - we focus on the functionality
  - hide the implementation details
  - it's the promise of encapsulation from OOP but taken to a higher level
- generics and protocols
  - these are extremely powerful
  - protocol Greatness {
    - func isGreaterThan(other: Self) -> bool
  - }
  - Self means the actual type of the thing implementing this protocol
  - Extension Array where Element : Greatness {
    - Var greatest: Element {
      - Return the greatest by calling isGreaterThan on each Element

        - }
    - }
- Very powerful foundation for designing things
- Functional programming does require some mastery through experience
- Layout
    - How si the space on-screen apportioned to the Views
        - Amazingly simple
        - Container Views “offer” space to the Views inside them
        - Views then choose what size they want to be
        - Container views then position views inside of them
    - Container Views
        - Stacks (HStack,VStack) divide upthe space that is offered to them and then offer that to the views
        - Offers space to the least flexible subveiws
        - Inflexible? -> **Image**
        - Another one: Text
        - Very flexible view: RoundedRectangle
        - It’s size is removed from the space available and then goes to the next least flexible Views
    - HStakc and VStack
        - Work with any views
        - Spacer(minLength: CGFloat)
        - Always takes all the space offered to it
        - Draws nothing
        - minLength -> defaults to the most likely spacing you’d want on a given platform
        - Divider()
        - Draws a dividng line cross-wise to the way the stack is laying out
        - For an example in an HStack draws a vertical lin
        - Min space needed to fit the line
    - HStack and VStack
        - Stack’s choice of who to offer space to next can be overridden with .layoutPriority(Double)
        - layoutPriority trumps “least flexible”
        - Important text above will get the space it wants first
    - HStack and VStack
        - Alignment is key
        - VStack(alignment: .leading) { … }
        - HStack(alignment: .firstTextBaseline)
        - You can use custom alignments as well
    - Modifiers

- GeometryReader
    - Wrap this GeometryReader View around what would normally appear in your View's body
    - Var body: View {
        - GeometryReader { geometry in …}
    - }
    - Geometry parameter is a GeometryProxy
    - struct GeometryProxy {
        - var size : CGSize
        - func frame(in: CoordinateSpace) -> CGRect
        - var safeAreaInsets : EdgeInsets
    - }
    - Size var is the amt of space that is being offered to us by our container
    - Now we can for example pick a font size appropriate to that sized space
    - GeometryReader itself always accepts all the space offered to it
- SafeArea
    - When a VIEW is offered space, that space does not include "safe areas"
    - Most obvious "safe area" is the notch of an iphone X
    - ZStack { … }.edgesIgnoringSafeArea([.top]) // draw in "safe area" on top edge"
- Contianer
    - How exactly do contain Views "offer" space to the Views they contain?
    - With the modifier .frame(…)
    - This .frame modifier has a lot of args
    - Once a view chooses the size, set position with .position (the center of the subview)
    - You can also .offset the view
    - We're going to use frame and position to build our grid

Lecture 4: Grid + enum + Optionals

- Grid
    - Generics iwth
- Varieties of Types
    - Gonna hammer the enums
- Optional
    - Extremely important type in Swift (it's an enum)
- Enums
    - Another variety of data structure
    - **Enum is a value type**
    - Each associated value can have associated value
    - Switch and case

  - If you don’t want to do anything in a given case, use break
  - Switch must handle ALL POSSIBLE CASES
    - You can use default easily
  - Methods (yes) properties no
  - If you have a function in an enum, you can switch on self and just have the case .item type
  - Can extends with **CaseIterable** you can extend
- Most important enum! **Optional**
  - Essentially looks like this:
  - enum Optional<T>
    - case non
    - case some(T)
  - You can see that it can only have two values: is set (some) or not set (none)
  - var hello: String?
  - Always basically starts out with an implicit nil
  - Access the associated value by force with **!**
  - Can also do this safely
  - If let safehello = hello {
    - Print(safehello)
  - } else {
    - // do something else
  - }
  - **??** optional defaulting -> null coalescing operator

Lecture 5: ViewBuilder + Shape + ViewModifier

- Swift demo warmup
- @ViewBuilder
  - General tech added to Swift to support “list-oriented syntax”
  - More convenient syntax for lists of Views
  - Devs can apply it to any of their functions that return something that conforms to view
  - Combines this list of views into one
  - @ViewBuilder
    - Func front(of card: Card) -> some View {
      - RoundedRectangle(CornerRadius: 10)
      - RoundedRectangle(CornerRadius: 10).stroke()
      - Text(card.content)
    - }
  - Can also use @ViewBuilder to mark a parameter that returns a View

        - Struct GeometryReader<Content> where Content : View {
            - Init(@ViewBuilder content: @escaping (GeometryProxy) -> Content) {…}
        - }
    - Can't declare variables or just do random code
    - Can only be a view
- Shape
    - Shape is a protocol that inherits from View
    - All shapes are views
    - They draw themselves by filling with the current foreground color
    - Can call .stroke() and .fill()
    - func fill<S> (_ whatToFillWith: S) -> View where S : ShapeStyle
    - **Generic function**
    - S can be anything that implements the ShapeStyle protocol
    - Protocl introduces this func that you are required to implement
    - func path(in rect: CGRect)
        - return a path that draws anything you want. Tons of functions to support drawing
- Animation
    - Animation is very important
    - One way to do animation is by animating a shape
    - Views are animated from ViewModifiers
- ViewModifier
    - AspectRatio and padding for example
    - Probably turning right around and calling a function called .modifier
    - **Associatedtype Content**
    - Text("ghost").modifier(Cardify(isFaceUp: true))
    - Struct Cardify: ViewModifier {
        - Var isFaceUp: Bool
        - Func body(content: Content) -> some View {
            - Zstack {
            - }
    - }
    - Extension View {
        - Func cardify(isFaceUp: bool) -> some View {
            - Return self.modifier
        - }
    - }

Lecture 6: Animation

- Property Observers

    - Essentially a way to watch a var and execute code when it changes
    - Syntax can look a lot like a computer var
- @State
    - Your view is **Read Only**
    - Turns out that all of your View structs are completely and utterly read-only
    - Whatever var is holding all your views
    - Why?
        - Reliable and provable for it to manage changes and efficiently re-draw things
        - Views are mostly supposed to be **stateless** (just drawing their model all the time)
    - When Views need State?
        - Sometimes this is true
        - Such storage is always temporary
    - Examples
        - Entered an editing mode and collecting changes in prep for an Intent
        - You've displayed another temp view to gather info
        - Animation to kick off so you need to set that animation's end point
    - @State
        - Must mark any vars used for this temp state with @State
        - @State private var somethingTemporary: SomeType
        - Marked private because no one else can access this anyway
        - Changes to this @State var **will cause your View to redraw if Necessary**
        - In that sense, it's like an @ObservableObject
        - When View needs State
        - Going to make some space in the heap for this
        - View struct itself is read-only remember
- Animation
    - A smoother out portrayal in your UI
        - Over a period of time (which is configurable and brief)
        - Of a change that has already happened
        - Point of animations is to make the user experience less abrupt
        - To draw attention to things that are changing
    - What can get animated?
        - Changes to the Views in containers that are already on screen CTAAOS
        - Which changes?
            - Appearance and disappearance of Views
            - Changes to args to Animatable view modifiers of Views that are in CTAAOS
            - Changes to the args of creation of Shapes inside CTAAOS
    - How do you make an animation go?

  - Impliclty by using the view modifier .animation(Animation)
  - Explicitly withAnimation(Animation) {}
- Implicit Animation
  - “Automatic animation”
  - Has duration and curve you specify
  - **Warning:** .animation modifier does not work how you might think on a container. Just propagates the .animation modifier to all the Views it contains. In other words, .animatino does not work like .padding, more like .font
- Animation
  - Argument to .animation() is an Animation struct
  - Duration, delay, repeat, it’s curve
- Animation Curve
  - .linear
    - Means exactly what it sounds like – consistent rate throughout
  - .easeInOut
    - Starts out the animation slowly, picks up the speed, then slows at the end
  - .spring
    - Provides a “soft landing” a “bounce”
- Implicit vs Explicit Animation
  - “automatic” implicit animations are usually not the primary source of animation behavior
- Explicit Animation
  - Create an animation session
  - Supply the animation (duration, curve, etc.)
  - withAnimation(.linear(duration: 2)) {
  - }
  - More imperative
- **Explicit animations do not override an implicit animation**
- Transitions
  - Specify how to animation arrival / departure of Views in CTAAOS
  - Transition is nothing more than a pair of ViewModifiers
  - One of the mods is the before
  - Other is the after
  - Using .transition()
  - ZStack {
    - If isFaceUp {
      - RoundedRectangle()
      - Text(“emoji”).transition(.sclae)
    - } else {

        - RoundedRectangle(cornerRadius: 10).transition(.identity)
      - }
    - }
    - If isFaceUp changed
      - To false, the **back** would appear instantly. Text would shrink to nthing, front RRfade out.
      - To true, the back would disappear instantly, Text grow in from nothing, front RR fade inw
    - Default .transition is .opactiy
    - .transition gets redistributed to a container's content Views
    - All the transition API is type erased
    - Makes it a lot easier to work with
    - We use the struct **AnyTransition** which erases type info for underlying ViewModifieras
  - .onAppear
    - Remember that transitions only work on Views that are in CTAAOS
    - Executes a closure any time a View appearas on screen
    - Use .onAppear on your container view to cause a change that results in the appearance of the view you want to animate the transition
  - Shape and ViewModifier animation
    - Communication between is just one single var
    - AnimatableData: Type
    - Type is a don't care
    - Well it's a car a little bit
    - Type is almost always a floating point number
    - animatableData is a read-write var
    - Setting of this var is the animation system telling the Shape/VM which piece to draw
    - Getting of this var is the animation getting the start/end points of an animation
    - Usually this is a computer var
    - Get/set often just gets/sets some other vars
- Demo
  - Match somersault
  - Card rearrangement
  - Card flipping
  - Card disappearing
  - Bonus scoring pie animation

Lecture 7: Multithreading EmojiArt

- This lecture

John Larkin
12/6/20
Developing Apps for iOS Spring 2020
Class Notes

  - Colors and Images
    - Color vs UIColor
    - Image vs UIImage
  - Multithreaded Programming
    - Ensuring that my app is never “frozen”
  - EmojiArt Demo
    - Review MVVM
    - ScrollView
    - Fileprivate
    - Drag and Drop
    - UIImage
    - Multithreading
- Color vs UIColor
  - Color
    - This can play different roles
    - Color-specifier, ShapeStyle, View etc
  - UIColor
    - Used to manipulate colors
    - Also has many built in colors than Color, including system-related colors
    - You can get the RGBA values from a UIColor
- Image vs UIImage
  - Image
    - Primarily serves as a View
    - Access images in your Assets.xcassets by name
    - Tons of system images Image(systemName:)
    - **Need to download system images in the SF Symbols app (developer.appl.com/design)**
    - While you’re there, study the **Human Interface Guidelines, a must for AppStore submissions**
    - **You can controle the size with .imageSclae() View Modifier**
    - System images are also very useful as masks
  - UIImage
    - Creating / manipulating images
- Multithreading
  - **Never** okay for an app to be unresponsive
  - Threads
    - Let you specify a thread of execution
    - Appear to be executing their code simultaneously
    - They might in face be a multi core computer
    - You as a programmer can’t tell the difference
  - Queues

  - Challenge is to make a multithreaded code authorable, readable, and understandable
  - Queue is just a bunch of blocks of code, lined up, waiting for a thread to execute them. Only concerned with queues.
- Queues and closures
  - Specify the blocks of code waiting using closures
- MainQueue
  - Most important queue is called the **main queue**
  - Queue that has all the blcoks of code that might muck with the UI
  - **Must use this queue if we want to do something with the UI**
- Background Queues
  - Long-lived, non-UI tasks
- GCD
  - **Base API for doing all this is called GCD (Grand Central Dispatch)**
  - **Has a number of different functions in it, two fundamental taks:**
    - Getting access
    - Pushing block of code ona queue
- Creating a Queue
  - **DispatchQueue.main // queue where all UI code must be posted**
  - **DispatchQueue.global(qos: QoS)** // non-UI queue with a certain quality of service
    - **Qos one of the following:**
      - .userInteractive // do this fast, UI depends on it
      - .userInitiaed // user just asked to do this, so dit now
      - .utility // needs to happen, but user didn’t ask for it
      - .background // clean up
- Pushing Closure onto Queue
  - Queue.async
  - Queue.sync
  - Second one above blocks waiting for that closure to be picked off.
  - We almost always use .async, always remember that .async will execute that closure “sometime later”
  - Also functions to have the queue wait for a delay interval before executing
- Nesting
  - Beauty of this API is when you end up nesting:
  - DispatchQueue(global: .userInitiated).async {
    - Do something that will take a long time
    - Can’t update the main UI once this is done so we can just nest it
    - DispatchQueue.main.async {
    - }

    - }
    - Makes async code looks synchronous
  - Asynchronous API
    - You will do **DispatchQueue.main.async {}** often when programming asynchronously
    - There's a lot of higher iOS APIs
- Demo

Lecture 8: Gestures JSON

- UserDefaults
  - Lightweight persistent store
- Gestures
  - Getting input from the user into your app
- Persistence
  - Storing data permanently
    - Numerous ways to make data persist in iOS
    - Filesystem (FileManager)
    - Sql database (CoreData for OOP access or even direct SQL calls)
    - iCloud (interoperates with both)
    - CloudKit (a database in the cloud)
    - Many third part options as well
    - Simplest: UserDefaults
  - UserDefaults
    - Persistent dictionary (want to use it for small lightweight things)
- UserDefaults
  - Data Types in UserDefaults
    - Old
    - Predates
  - Property Lists
    - UserDefaults can only store what is called a PropertyList
    - This is not a protocol or a struct or anything like that
    - Simply a concept
    - Combination of String, Int, Bool, floating point, Date, Data, Array or Dictionary
    - Powerful way to do this is using the Codeable protocol in Swift
    - Codable converts structs in Data objects
  - Any type
    - API for UserDefaults is a strnage because it is pre-Swift
    - Doesn't like Any but supports this for backwards compatibility
    - Going to try to ignore this
  - Using UserDefaults

        - Need an instance
        - Let defaults = UserDefaults.standard
    - Storing Data
        - defaults.set (object, forKey: “someKey”) // object must be a PropertyList
        - defaults.setDouble(37.5, forKey: “double”)
- Gestures
    - Getting input from the user
        - Powerful primitives for recognizing gesturs by the user’s fingers
        - Called multitouch
    - Making your Views Recognize Gestures
        - Cuase your view
        - myView.gesture(theGesture) // theGesture must implement the Gesture protocol
    - Creating a Gesture
        - Usually the gesture will be created by some func or computer var you create
        - Var theGesture : some Gesture {
            - Return TapGesture(count: 2)
        - }
        - This is a double tap
        - SwiftUI will recognize the TapGesture, but it won’t do anything yet
    - Handling the recognition of a Discrete Gesture
        - So how do we “do something” about a recognized gesture?
        - TapGesture is a discrete gesture
        - Happens all at once
        - .onEnded { /* doSomething *? }
        - That’s it you just pass it a closure that says what to do
    - Non Discrete Gesture
        - Handle the gesture while it is in the process of happening (fingers are moving)
        - Examples: DragGesture, MagnificationGesture, RotationGesture
        - LongPressGesture can also be treated as non-discrete (fingers down and up)
        - Var theGesture : some Gesture {
            - DragGesture(…)
                - .onEnded { value in /* do something 8? }
        - }
        - Value tells you the state of the Drag Gesture when it ended
        - What that value is varies from gesture to gesture
        - DragGesture, struct with things like start and end location
        - MagnificationGesture, scale of magnification

- RotationGesture, angle of the rotation
- You’ll *also* get a chance to do something while its happening
- **Mark this with**
- @GestureState var myGestureState : MyGestureStateType = <startingValue>
- Can be a variable of any type
- **var will always return to starting value when gesture ends**

- Handling Non-Discrete Gestures
  - var theGesture : some Gesture {
    - DragGesture(…)
      - .updating($myGestureState) { value, myGestureState, transaction in
        - myGestureState = /* related to value */
      - }
      - .onEnded { value in /* do something */ }
  - }
  - .updating – will cause closure you pass to it to be called when the fingers move
  - Note the $ in front of your GestureState var
  - Value arg is the same as with .onEnded
  - myGestureState arg is essentially your Gesture state
  - ***Cannot change your GestureState except inside this closure. You should not be setting it at any other time.***
  - Transaction… kinda for advanced interaction
- Handling Non – Discrete Gestures
  - **Summary**
  - Collect any info you need to draw your View during the gesture into a GestureState
  - Add .updating to your gesture
  - In .updating, use the value that is passed to you to update your GestureState
  - Understand that when the gesture ends, your GEstureState will reset

Lecture 9: Data Flow

- Today:
  - Property Wrappers
  - Publishers (very light)
  - Demo
- Property Wrappers
  - Background
    - All of these @something statements are property wrappers

  - A property wrapper is actually a structe
  - Encapsulate some template behavior
  - Examples:
    - Making a var live in the heap (@State)
    - Making a var publish its changes (@Published)
    - Causing a view to redraw when a published change is detected (@ObservedObject)
  - Property wrapper feature is really like syntactic sugar to make these structs easy to create/use
- Property Wrapper Syntatic Sugar
  - @Published var emojiArt : EmojiArt = EmojiArt()
  - Is really just
  - struct Published {
    - var wrappedValue : EmojiArt
  - }
  - Var _emojiArt : Publsihed = Published(wrappedValue : EmojiArt())
  - Var emojiArt : EmojiArt {
  - Get { _emojiArt.wrappedValue }
  - Set { _emojiArt.wrappedValue = newValue}
  - There's another var inside Property Wrapper structs
  - Can access this var using $emojiArt
  - Its type is up to the PropertyWrapper Published's is a Publisher<EmojiArt, Never>
- Why do we do this?
  - Wrapper struct does something on set / get of wrapped value
  - @Published -> publishes it through projectedValue ($emojiArt). Also does an invoke objectWillChange.send() in its enclosing ObservableObject
- @State
  - Wrapped value is anything
  - Stores the wrappedValue in the heap
  - When it changes, invalidates the View
  - Projected value : a binding (to that value in the heap)
  - Binding -> a way to connect two value types together
- @ObservedObject
  - Wrapped value is anything that implements the observableObject protocol (ViewModels)
  - What is does: invalidates the View when wrappedValue does objectWillChange.send()
  - Projectd value: a binding ot that vars of the wrapped value
  - If either changes, the other gets updated
- @Binding

  - wrappedValue: value that is bound to something else
  - What it does: gets/sets the value of the wrapped value from some other source
  - What it does: when the bound-to value changes, it invalidates the View
  - Projected value: a Binding (self; i.e. the binding itself)
- **Bindings**
  - Use them allllll over the place
  - ***One of the most important things in the MVVM model***
  - Bindings are all about having a single source of the truth
  - We don’t ever want to have state stored in our ViewModel and also state in our View
  - ***We only want one source of truth***
  - Bindings create connections between variables that link them
- Where do we use Bindings?
  - Sharing @State with other Views
  - Struct myView : View {
    - @State var myString = “Hello”
    - Var body : View {
      - OtherView(sharedText : $myString)
    - }
  - }
  - Struct otherView : View {
    - @Binding var sharedText : String
    - Var body : View {
      - Text(sharedText)
    - }
  - }
  - OtherView’s body is a text whose string is <u>always</u> the value of myString in myView
  - OtherView’s sharedText is boudn to myString
- This is fundamental to understanding how data flow works
- Binding to a constant value
  - This is fine
  - Binding.constant(value)
- Computed Binding
- Another prop wrapper
- @EnvironmentObject
  - Same as @ObservedObject, but passed to a View in a different way
  - Let myView = MyView.environmentObject(theViewModel)
  - Vs
  - Let myView = MyView(viewModel: the viewModel)

- Inside the View
- @EnvironmentObject var viewModel : ViewModelClass
- @ObservedObject var viewModel : ViewModelClass
- Biggest difference?
    - Environment objecsts are visible to all views in your body (except modally presented ones).
    - So its sometimes used whena number of views are going to share the same view model. Can only use one EnvironmentObject wrapper per observed type
- WrappedValue : ObservableObject obtained via .environmentObject() sent ot the View
- What it does: invalidates the View when wrappedValue does objectWillChange.send()
- Project value: a Binding

- @Environment
    - Unrelated to @EnvironmentObject
    - Property wrappers can have more variables
    - Can pass values to set these other vars using ()
        - E.g. @Environment(\.colorScheme) var colorScheme
    - It's a KEYPATH
    - It specifies which instance variable to look at in an EnvironmentValues struct
    - wrappedValue's type is internal to the Environment Property Wrapper
    - Its type will depend on which key ppath you're asking for
    - ColorScheme is an enum with values .dark and .light
    - wrappedValue: value of some var in EnvironmentValues
    - Projected value i.e. none

- Publisher
    - "light" explanation
    - What is a publisher?
        - An object that emits values and possibly a failure object if it fails while doing so
        - Output – type of thing this Publisher publishes
        - Failure – type of thing it communicates if it fails while trying to publish
        - If the publisher does not deal with erros, the Failure can be Never
    - You listen to Publishers!!
        - Also can transform it values on the fly
        - Sometimes massage
    - Listening (subscribing) to a publisher
        - Simply execute a closure whenever a Pubsliher publishes
        - Cancellable = myPublisher.sink (

      - receiveCompletion : {result in … }
      - receiveValue : { thingThePublisherPubslihes in … }
    -)
    - Note that **.sink** returns a cancellable
      - The purpose here is
      - You can send .cancel() to it to stop listening to that publisher
      - **It keeps the .sink subscriber alive**
      - **Always keep this var somewhere that will stick around as long as you want the .sink to**
      - **This is really used to decide how long to listen to the publisher**
  - Listening (subscribing) to a publisher
    - A view can listen to a publisher too
    - .onReceive(publisher) { thingThenPublisherPublishes in … }
    - .onReceive will be super useful
  - Where do they come from?
    - $ in front of vars marked @Publsihed
    - URLSession's dataTaskPublsiher (publishes the data obtained from a URL)
    - Timer's published (every:)
    - NotificationCenter's publisher(for:) publishes notifications when system events happen
  - Other stuff we can do with a Publsiher
    - Just a couple examples in this lesson
- Demo
  - Publishers and Binding

Lecture 10: Navigation + TextField

- Demo all lecture
- EmojiArt is going to have multiple view models
- Demo topics
  - .sheet
  - .popover
  - TextField
  - Form
  - Constraints and Gains via Grid enhancement
  - Dismisisng modally presented Views via Binding
  - Multiple MVVMs in a single app
  - Hashable and Equatable
  - NavigationView and NavigationLink and .navigationBarTitle
  - Alerts
  - Deleting from a ForEach with .onDelete

- EditButton
- EditMode @Environment variable (a @binding)
- Setting @Environment variable
- .zIndex

- **This is a big lecture**
- **This is where we can have multiple ViewModels**

Lecture 11: Picker

- FlightAware API is basically what we're querying
- Code seems to already be generated / in the Demo Code downloadable
- Largely see the commented code for class notes!! Not that much to take just processing what he's saying

Lecture 12: Core Data

- Core Data
  - Object-oriented database
- SQL vs OOP
  - Programming SQL is very different than what we're doing in Swift
  - We're going to get the best of both worlds using Core Data framework
  - Does actual storing using SQL
  - Don't even need to know SQL
- Core Data
  - Map
    - Heart of core data is MAP
    - Map between objects/var and the tables and rows of a relational database
    - **Xcode has a built in editor for this map**
    - Lets us graphically create relationships
  - Then what?
    - Xcode generates classes behind the scenes
    - This seems very very similar to Entity Framework…
  - Features:
    - Creating objects
    - Changing values
    - Saving
    - Fetching on criteria
    - Optimistic locking, undo management (LOTS OF STUFF AT HIGHER LEVELS)
  - SwiftUI Integration
    - Objects we create in the db are ObservableObjects

- A very powerful property wrapper FetchRequest which fetches objects for us
- Standing query

  - The Setup
    - Start by clicking the “Core Data” button when you create a New Project
    - Creates a blank map
    - Little bit of code to your AppDelegate to create the store
    - First line gets a window onto the db
    - Window is: NSManagedObjectContext
    - Second line passess that context into the environment of your SwiftUI views
  - The Code
    - We’ll cover all this in the demo, but here are some highlights
    - Envrionment(\.managedObjectContext) var context
    - Let flight = Flight(context: context)
    - Flight.aircraft = “B737”
    - Let request = NSFetchReuqest<Flight>(entityName: “Flight”)
    - Request.predicate = **NSPredicate**(format: “arrival < %@ and origin = %a”, Date(), ksjc)
    - Request.sortDescriptors = [NSSortDescriptor(key: “ident”, ascending: true)]
    - **We want to do the sorting on the database side**
    - How do we ask our db to go fetch?
      - Let flights = try? Context.fetch(request)
  - SwiftUI
    - @ObservedObject var flight: Flight
    - These ObservedObjects don’t seem to automatically objectWillChange.send()
    - @FetchRequest(entity:sortDescriptors:predicate☺ var flights: FetchedResults<Flight>
    - **What’s cool here: it is continuously trying to do this fetch. Standing query. SwiftUI is always going to be reflecting the database.**
- Often easiest to just copy and paste your code into a new project with core data
- Demo
  - AppDelegate
  - NSPersistentCloudKitCOntainer
  - Context -> passed via the environment to all of our views
  - When we use a sheet or popover, you need to pass the environment over
  - Optional in the .xcdatamodelId really means that it’s optional in the database
  - You hold down the control key to drag and create relationships
- Once again, a lot of the comments are in the actual code

Lecture 13: Persistence

- Persistence
    - Storing stuff between application launches
        - UserDefaults
        - Codable/JSON
        - UIDocument (UIKit feature worth mentioning)
        - Core Data
        - Cloud Kit
        - File System
- Persistence
    - UserDefaults
        - Simple. Limited. Small
    - Codable/JSON
        - Clean way to turn almost any structure into an interoperable / storable format
    - UIDocument
        - Integrates the Files app and user perceived documents inot your application. This is really the way to do things when you have a true doc. UIKit compatibility code is required. Not covered in the class but good.
    - Core Data
        - Powerful. Object oriented. Elegant Swift UI integration.
    - Cloud Kit
        - Storing data into a database in the cloud (i.e. on the network)
        - That data thus appears on all the user's devices
        - Also has its own networked UserDefaults like thing
        - Plays nicely with Core Data
        - Going to be slide heavy…. So should probably practice this on my own time at some point
    - FileManager/URL/Data
        - Storing things in the unix file system that underlies iOS
- Cloud Kit
    - A database in the cloud
    - Simple to use but with very basic database operations
    - Requires thoughtful programming
    - **THIS IS ASYNCHRONOUS PROGRAMMING**
    - REQURES A LOT OF THOUGHT
    - **Important Components**
        - Record Type – like a class or struct
        - Fields – like vars in a class or struct
        - Record – an instance of a record type

        - Reference – a pointer to another Record. Doing relationships can be tricky. Not a full relational database
        - Database – a place where Records are stored
        - Zone – sub area of a database
        - Container – a collection of databases
        - Query – a Database search
        - Subscription – a standing Query which sends push notifications when changes occur
    - **Must enable iCloud in your Project Settings**
        - **Under capabilities tab, turn on iCloud (On/Off switch)**
        - **Then choose CloudKit from the services**
    - CloudKitDashboard
    - **Dynamic Schema Creation**
        - But you don’t have to create your schema in dashboard
        - Create it organically by simply creating and storing things in the database
        - When you store a record with a new, never before seen Record Type, it’ll create that type
        - Or if you add a field or record, it’ll auto create a field for that in dev
        - Only works in Development
    - **Example**
        - What it looks like to create a record in a database
        - let db = CKContainer.default.public/shared/privateCloudDatabase
        - Public… well publically posted data… could be possible
        - Shared… is invitation only. People can share data. Cool way to share things
        - let tweet = CKRecord(“Tweet”)
        - tweet[“text”] = “140 characters of joy”
        - tweet[“tweeter”] = CKReference(record: tweeter, action: .deleteSelf)
            - The action is basically like what happens if the tweet gets deleted
        - db.save
            - This is an async function
    - **Standing Queries**
        - One of the coolest features is the ability to send push notifcations on changes
        - If you’re interested, check out the UserNotifications framework
- File System
    - Background
        - Your application sees iOS file system like a normal unix filesystem
        - Starts at /
        - Applications sandbox is isolated and read / write
        - Why sandbox?

      - Security (no one else can damage this)
      - Privacy (no other apps can view that data)
      - Cleanup (when you delete an app, everything is has ever written goes with it)
    - What's in the sandbox?
      - Application dir – executables, .jpgs, etc; not writeable
      - Documents dir – permanent storage created by and always visible to the user
      - Application Support dir – permanent storage **not** seen directly by the user
      - Caches dir – Store temporary files here (NOT backed up)
      - Other directories
  - File Manager
    - Getting a path to these special sandbox directories
    - FileManager is what you use to find out about what's in the file system
    - let url : URL = FileManager.default.url (
      - for directory: FileManager.SearchPathDirectory.documentDirectory,
      - in domainMask: .userDomainMask,
      - appropriateFor: nil
      - create: true
    - Base URL
      - appendingPathComponent
      - appendingPathExtension
    - Finding out about what's at the other end of a url
      - isFileURL
      - resourceValues
  - Data
    - Reading binary data from a URL
    - Init(contentsOf: URL, options: Data.ReadingOptions) throws
    - Options are almost always []
    - Notice that this function throws
    - Writing binary Data to a URL
- Demo:
  - Going to store EmojiArt Documents in the file system (which will be more reasonable than user defaults...
  - Switching to code

John Larkin
12/6/20
Developing Apps for iOS Spring 2020
Class Notes

Lecture 14: UIKit Integration

- Integrating with UIKit
    - Not every feature from UIKit was transported into SwiftUI
    - Some good apis
- UIKit Integration
    - Views are not as *elegant*
        - No MVVM either, MVC instead
        - MVC, views are grouped together and controlled by a controller
        - This Controller is the granularity at which you present views on screen
        - In other words, UIKit's .sheet, .popover, and NavigationLink destination equivalents don't present a veiew, they present a controller (which in turn controls views)
    - Integration
        - So there must be two points of integration for wiftUI to UIKit
        - UIViewRepresentable and a UIViewControllerRepresentable
    - Delegation
        - UIKit is based on OO tech
        - Delegation! Not as reactive
        - Objects have delegate that they delegate functionality to
        - Delegate var is constrained via a protocol
    - Representables
        - UIViewRepresentable and UIViewCOntrollerRepresentable are SwiftUI Views
        - 5 main components
            - Function which creates UIKit thing in question (view or controller)
                - Func makeUIView{Controller}(context: Context) -> view / controller
            - Function which updates the UIKit thing when appropriate (bindings change, etc)
                - Func updateUIView{Controller}(view/controller, context: Context)
            - A coordinator object which handles any delegate activity that goes on
                - func makeCoordinator() -> Coordinator
            - a context (contains the coordinator, swiftui's env, animation transaction)
                - Passed into the methods above
            - A "Tear down" phase if you need to clean up when the view or controller disappears
- Demos
    - Choose Destination Airport from a Map

    - No Map in SwiftUI, so let’s use the one from UIKit
      - Demo of integrating a UIView into SwiftUI
  - Set our EmojiArt background from the Camera
    - No Camera API in SwiftUI either, there’s on in UIKit
    - Demo of integrating a UIViewController in SwiftUI
    - Focus on the integration here, not the pretty bad feature we’re adding to EmojiArt
    - You’d probably want to store the actual image data in your model
- Code to look at:
  - Enroute L14
  - EmojiArt L14
- Very very cool demo with MkMapView in the MapKit package

Class complete!! Woohoo

Final project: Going to build my own app. Will be its own thing.