IS 543 Fall 2023

Mobile Platform Development

Swift Functions

Reactive SwiftUI

Swift Protocol, Extensions

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

Today

- Swift Fundamentals
  - Functions
  - Reactive SwiftUI
  - `protocol`
  - `extension`

Fun Facts

- Which of the following is NOT on Prof. Liddle's shelf in his office?

  Pig-on-a-motorcycle cookie jar
  Raiders logo bear with XXL on the shirt
  Scoutmaster Build-a-Bear, complete with sunglasses and hat
  Metallic Mer-lion statue
  American Girl doll, made up to look like my wife
  Mao wristwatch that ceased working shortly after purchase in Tiananmen Square
  Beekeeper Barbie, in its original box
  None of the above — they're ALL on his shelf

How would you know? Office hours

IS ANNUAL ALUMNI & STUDENT

TAILGATE

BYU COUGARS VS UC BEARCATS

29 SEPTEMBER 2023

5 PM - 7:30 PM

NORTH TNRB PARKING LOT

VOLUNTEER HERE

- PHOTOS WITH COSMO @ 5:30 PM - 6 PM
- ROCK WALL, BOUNCE HOUSE, GAMES & PRIZES
- FREE BABY COUGAR TAIL
- WAFFLE LOVE & J DAWGS FOOD TRUCKS
  (FOOD AVAILABLE FOR PURCHASE)

ALL ALUMNI & STUDENTS INVITED TO ATTEND

REGISTER HERE

Tuesday's Devotional

- What did you learn from the messages of Pres. & Sister Reese?
  I'm confident you found something worthwhile and uplifting

  Thoughts on being comfortable or uncomfortable

  God loves you
  He will direct your paths to places where you can serve, to people whom you can love, to situations in which you can bless others and they can bless you
  Our lives are connected
  Are there any coincidences?

HW 3 Debrief

- What did you learn from HW 3?
  Any particular observations you want to share?
  Any lingering questions we should answer?

Functions as Types

- Functions are first-class types in Swift

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

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

  You can do this anywhere any other type is allowed

  Examples:

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

  All of the above are just types, no different than Bool or View or Array<Int>, all are types

```
var f: (Double) -> Void  // f's type: function that takes a Double, returns nothing

func doSomething(what: () -> Bool) // what's type: function, takes nothing, returns Bool
```

Functions as Types

- Functions are first-class types in Swift

  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 of type (Double) -> Double
  let result2 = operation(4) // result2 would be 2
  ```

  We'll soon see an example of using a function type for a parameter to a function in our demo

Functions as Types

- Closures

  It's so common to pass functions around that we very often "inline" 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 semester

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

In-Class Exercise

- You can do this in groups or by yourself

  Write a function named `calculate` that receives two parameters: one named `description` that is a `String` and another named `operation` whose type is a function that receives `Double`s named `operand1` and `operand2` and returns a `Double`

  In the body of the function `calculate`, invoke `operation` with test values (two different numbers) you specify and store its result in a variable

  Have `calculate` print the `description` and the result of `operation`

  Try these test cases:

```
calculate(description: "Multiply", operation: { (o1, o2) -> Double in o1 * o2 })
calculate(description: "Divide", operation: { (o1, o2) -> Double in o1 / o2 })
calculate(description: "Power", operation: { (o1, o2) -> Double in pow(o1, o2) })
```

My Solution

- You could try this in an Xcode playground:

```
func calculate(operation: (Double, Double) -> Double, description: String) {
    var result = operation(5, 3)

    print("\(description)(5, 3): \(result)")
}

calculate(description: "Multiply", operation: { (o1, o2) -> Double in o1 * o2 })
calculate(description: "Divide", operation: { (o1, o2) -> Double in o1 / o2 })
calculate(description: "Power", operation: { (o1, o2) -> Double in pow(o1, o2) })
```

These are closures, or anonymous inline functions

- The result printed on the console:

```
Multiply(5, 3): 15.0
Divide(5, 3): 1.6666666666666667
Power(5, 3): 125.0
```

Trailing Closure Syntax

- When the last parameter to a function is a closure (a function type), we can pull the parameter out of the parentheses and drop the label
  This is called "trailing closure syntax"

  Instead of this:

```
calculate(description: "Multiply", operation: { (o1, o2) -> Double in o1 * o2 })
```

  We can write this:

```
calculate(description: "Multiply") { (o1, o2) -> Double in
    o1 * o2
}
```

Inference in Swift

- Because Swift knows the types of the parameters, we can simplify this expression even further. We drop everything the compiler can infer.

Instead of this:

```
calculate(description: "Multiply") { (o1: Double, o2: Double) -> Double in
    return o1 * o2
}
```

We can write this:

```
calculate(description: "Multiply") { o1, o2 in
    o1 * o2
}
```

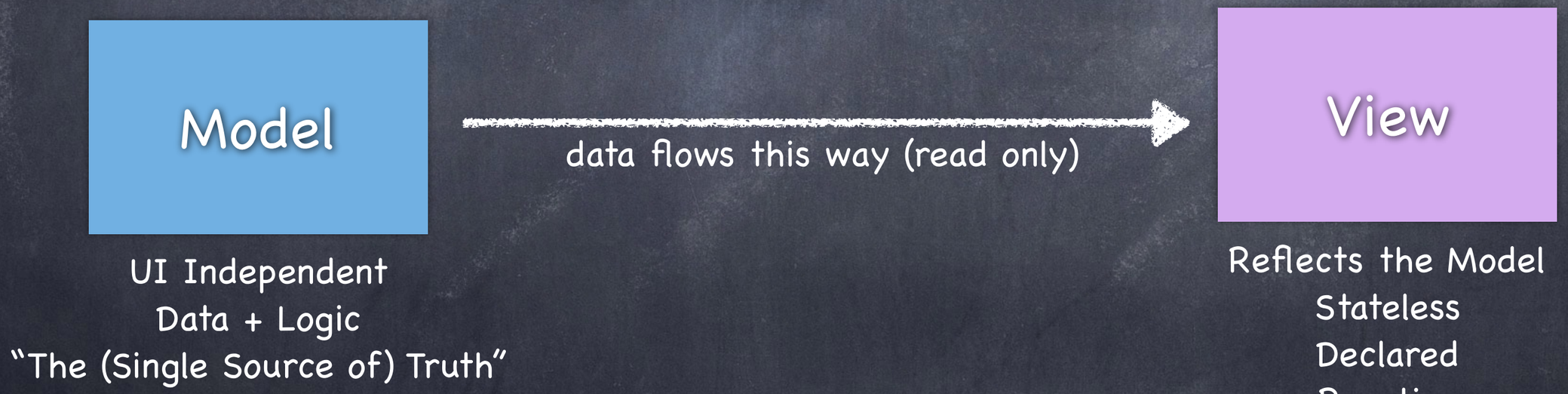
MVVM

Model

UI Independent
Data + Logic
"The (Single Source of) Truth"

View

MVVM


Model
data flows this way (read only)
View
UI Independent
Data + Logic
"The (Single Source of) Truth"
Reflects the Model
Stateless
Declared
Reactive

MVVM

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

might "interpret"

ViewModel

publishes "something changed"

notices changes

Binds View to Model
Interpreter

automatically
observes publications,
pulls data,
and rebuilds

Model

UI Independent
Data + Logic
"The (Single Source of) Truth"

View

Reflects the Model
Stateless
Declared
Reactive

MVVM

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

might "interpret"

publishes "something changed"

ViewModel

Binds View to Model
Interpreter

notices changes

automatically
observes publications,
pulls data,
and rebuilds

Model

UI Independent
Data + Logic
"The (Single Source of) Truth"

@ObservedObject
@Binding
.onReceive
@EnvironmentObject

View

Reflects the Model
Stateless
Declared
Reactive

MVVM

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

might "interpret"

publishes "something changed"

calls "intent" function

modifies the Model

notices changes

ViewModel

Binds View to Model
Interpreter
Processes Intent

automatically observes publications, pulls data, and rebuilds

Model

UI Independent
Data + Logic
"The Truth"

@ObservedObject
@Binding
.onReceive
@EnvironmentObject

View

Reflects the Model
Stateless
Declared
Reactive

What about the other direction?

Reactive SwiftUI

- MVVM in a nutshell

  The Model holds data your app wants to manage — it's the "what" of your app
  The Model might live in your code as a property of the View Model
  Be sure to mark the ViewModel class with `@Observable`
  In the View, have a property that references the ViewModel
  Any code that uses the View must supply a ViewModel

  `@Observable` takes care of all the hard work: when the ViewModel publishes a new value, the View re-renders

  More about this later
  Let's see the foundation for a simple pushup tracker app

Swift Types

- There are several varieties of types in Swift:
  - `struct`
  - `class`
  - Generics (sometimes called "Don't Care" types)
  - Functions (`func`)
  - `protocol`
  - `enum`

protocol

- A protocol is sort of a "stripped-down" struct/class

  It has functions and vars, but no implementation (or storage)

  Declaring a protocol looks very similar to struct or class (just without implementation)

```
protocol Moveable {
    func move(by: Int)
    var hasMoved: Bool { get }
    var distanceFromStart: Int { get set }
}
```

  No implementation

  The { } on the vars just say whether it's read only or a var whose value can also be set

  This idea is similar in many ways to the concept of an "abstract class"

protocol

- A protocol is sort of a "stripped-down" struct/class

  It has functions and vars, but no implementation (or storage)

  Declaring a protocol looks very similar to struct or class (just without implementation)

```
protocol Moveable {
    func move(by: Int)
    var hasMoved: Bool { get }
    var distanceFromStart: Int { get set }
}
```

Now any other type can claim to implement Moveable

```
struct PortableThing: Moveable {
    // must implement move(by:), hasMoved, and distanceFromStart here
}
```

protocol

- A protocol is sort of a "stripped-down" struct/class

  It has functions and vars, but no implementation (or storage)

  Declaring a protocol looks very similar to struct or class (just without implementation)

```
protocol Moveable {
    func move(by: Int)
    var hasMoved: Bool { get }
    var distanceFromStart: Int { get set }
}
```

And "protocol inheritance" is also allowed:

```
protocol Vehicle: Moveable {
    var passengerCount: Int { get set }
}

class Car: Vehicle {
    // must implement move(by:), hasMoved, distanceFromStart and passengerCount here
}
```

protocol

- A protocol is sort of a "stripped-down" struct/class

  It has functions and vars, but no implementation (or storage)

  Declaring a protocol looks very similar to struct or class (just without implementation)

```
protocol Moveable {
    func move(by: Int)
    var hasMoved: Bool { get }
    var distanceFromStart: Int { get set }
}
```

  And you can claim to implement multiple protocols:

```
class Car: Vehicle, Impoundable, Leasable {
    // must implement move(by:), hasMoved, distanceFromStart, and passengerCount here
    // and must implement any funcs/vars in Impoundable and Leasable too
}
```

protocol

- A protocol is a type

  Most protocols can be used anywhere any other type can be used

  For example, it can be the type of an argument to a function or of any variable:

```
var m: Moveable
var car: Car = Car(type: "Tesla")
var portable: PortableThing = PortableThing()  // we defined this struct previously
m = car      // perfectly legal
m = portable // perfectly legal
```

  Even though Car is a completely different class or struct than PortableThing, both implement the Moveable protocol and so can be assigned to a var of type Moveable

  Note, though, that this will not work:

```
portable = car // NOT legal
```

  The var portable is of type PortableThing (not Moveable) and a Car is not a PortableThing

  Swift enforces the type of the var at all times

protocol extension

- Adding protocol implementation

One way to think about protocols is constrains and gains

```
struct Tesla: Vehicle {
    // Tesla is constrained to have to implement everything in Vehicle
    // but gains all the capabilities a Vehicle has too
}
```

But how does a Vehicle "gain capabilities" if it has no implementation?

You can add implementations to a protocol using an extension to the protocol:

```
extension Vehicle {
    func registerWithDMV() { /* implementation here */ }
}
```

Now Teslas (and all other Vehicles) can be registered with the DMV

Adding extensions to protocols is at the heart of functional programming in Swift

The protocol View an excellent example of this

protocol extension

- Adding protocol implementation

  You can even add "default implementations" of the protocol's own funcs/vars

```
protocol Moveable {
    func move(by: Int)
    var hasMoved: Bool { get }
    var distanceFromStart: Int { get set }
}
extension Moveable {
    var hasMoved: Bool { return distanceFromStart > 0 }
}
struct ChessPiece: Moveable {
    // only need to implement move(by:) and distanceFromStart here
    // don't have to implement hasMoved because there's a default implementation
    // would be allowed to implement hasMoved here if we wanted to, though
}
```

extension

- Adding code to a struct or class via an extension

  Of course, you can use an extension to add things to structs and classes too:

```
struct Boat {
    ...
}
extension Boat {
    func sailAroundTheWorld() { /* implementation */ }
}
```

  You can even make something conform to a protocol purely via an extension:

```
extension Boat: Moveable {
    // implement move(by:) and distanceFromStart here
}
```

  Now Boat conforms to the Moveable protocol (!)

protocol

- Why protocols?

  Why do all this protocol stuff?
  It's a way for types (structs/classes/other protocols) to say what they are capable of
  And also for other code to demand certain behavior out of another type

  But neither side has to reveal what sort of struct or class they are
  This is core to "functional programming"

  Even when we talk about vars in the context of protocols, we don't define how they're stored
  We focus on the functionality and hide the implementation details behind it
  It's the promise of encapsulation from OOP but taken a step further

  And this is even more powerful when we combine it with generics...

Generics and protocols

- How do these things work together?

  When we combine these don't care types with constrains and gains, we get superpowers

  If we had a protocol like this:

```
protocol Greatness {
    func isGreaterThan(other: Self) -> Bool
}
```

  (BTW, the type Self means "the actual type of the thing implementing this protocol")

  … then we could add an extension to Array like this:

```
extension Array where Element: Greatness {
    var greatest: Element {
        // Loop through all the Elements, which (in this extension)
        // we know each implements the Greatness protocol,
        // figure out which one is greatest by calling isGreaterThan(other:)
        // on them and then return the greatest
    }
}
```

Generics and protocols

- Mercy!

  You might be shuddering right about now (sorry!)
  Are you thinking, "how am I going to design systems using generics/protocols?!"

  This is, indeed, a powerful foundation for designing things
  But functional programming does require some mastery that only comes with experience

  The good news: you can do a lot in SwiftUI without having to master functional programming
  But the more you use it, the more you'll want to really understand it
  That's why I explain to you up front what's going on

  No one expects you to now be able to add extensions to protocols with generics!
  But eventually you will be able to
  And in the meantime, you'll have at least some idea how SwiftUI is built

Coming Up...

- More Swift, SwiftUI, Xcode discussion & demo
  We'll do more with the Pushup Tracker app demo
  And I'll show you how to implement a simple Concentration Game app, complete with animation of cards flipping over

Homework 4 for Tuesday

- HW 4 due BEFORE class Tue. Sep. 19

1) Read the Swift API Design Guidelines document carefully and completely.
   Tell me about a couple of things you learned while doing this reading.

2) Follow the video I posted on YouTube to construct a simple MVVM pushup tracker app
   If you only have Xcode 14, it won't run without some changes and that's okay
   Type up the code regardless of whether you can make it run; Xcode 15 will release soon

3) Build a SwiftUI interface for a 4-function calculator app. You could try nesting `HStack`s and `VStack`s to build the interface, or you could try learning about `LazyVGrid`. You don't need to program the full functionality of the calculator (unless you really want to). But create a beautiful user interface. Use color, fonts, and spacing thoughtfully. Pay attention to the details.

To turn in this assignment, upload to Learning Suite a single PDF with your comments on the reading assignment and also a screenshot of your completed UI (I don't need to see your code).

Additional Reading

- Suggested sections of the Swift Language Guide to study with HW 4:
  Extensions
  Protocols
  Generics

- Also, you'll need to study the iOS Human Interface Guidelines soon:
  https://developer.apple.com/design/human-interface-guidelines/

  We'll focus only on iOS, not macOS, watchOS, tvOS

  The "HIG" is industry-leading material
  A lot of the value of the Apple ecosystem is captured in the philosophy described here