IS 543 Fall 2023

Mobile Platform Development

Swift Fundamentals

Types: Struct Class Generics Functions

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
  - `struct` vs. `class`
  - Generic types (or "generics" for short)
  - Functions

Fun Facts

- Which of the following is Professor Liddle's favorite ride?

A. BMW i8
B. Yamaha VStar 1100
C. Tesla Roadster
D. 2021 16" MacBook Pro, M1 Pro, 32GB RAM, 2TB SSD

HW 2 Debrief

- How was HW 2 for you?
  Any particular observations you want to share?
  Any lingering questions we should answer?
  Would anyone like to show us your solution?

MVVM

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

might ... pret"

Vi ... el

publishes "something changed"

calls "intent" function

modifies the Model

automatically observes publications, pulls data, and rebuilds

Don't worry, we'll get back to this later

Model

UI Independent
Data + Logic
"The Truth"

dObject
inding
ceive
@L ... ject

View

Reflects the Model
Stateless
Declared
Reactive

W ... bout the other di ... n?

Swift

- New language to learn!
  A completely new language, similar to some you have & haven't heard of (Go, Rust, JavaScript, ...)
  Integrates well with Objective-C and iOS
  Designed for "defensive" programming
  Swift is strongly typed and "type safe" — you can't pass an `Int` where a `String` is expected
- Several interesting new concepts
  E.g. type inference, optionals, tuple type, closures, nested functions
- This is just your first glimpse of this language!

Swift Types

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

struct and class

- Both struct and class have ...
  ... pretty much the same syntax
  Stored vars (the kind you're used to, stored in memory, called properties or instance variables)
  `var isFaceUp: Bool`

struct and class

- Both struct and class have …

  … pretty much the same syntax

  Stored vars (the kind you're used to, stored in memory, called properties or instance variables)

  Computed vars (those whose value is the result of executing/evaluating some code, called computed properties)

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

struct and class

- Both struct and class have ...

  ... pretty much the same syntax

  Stored vars (the kind you're used to, stored in memory, called properties or instance variables)

  Computed vars (those whose value is the result of executing/evaluating some code, called computed properties)

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

```
let defaultColor = Color.orange
...
CardView().foregroundColor(defaultColor)
```

struct and class

- Both struct and class have …

  … pretty much the same syntax

  Stored vars (the kind you're used to, stored in memory, called properties or instance variables)

  Computed vars (those whose value is the result of executing/evaluating some code, called computed properties)

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

  functions

```
func multiply(operand: Int, by: Int) -> Int {
    return operand * by
}
multiply(operand: 5, by: 6)

func multiply(_ operand: Int, by otherOperand: Int) -> Int {
    return operand * otherOperand
}
multiply(5, by: 6)
```

struct and class

- Both struct and class have …

  … pretty much the same syntax

  Stored vars (the kind you're used to, stored in memory, called properties or instance variables)

  Computed vars (those whose value is the result of executing/evaluating some code, called computed properties)

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

  functions

  initializers (special functions called when creating a struct or class instance)

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

struct and class

- Both struct and class have ...

  ... pretty much the same syntax

  Stored vars (the kind you're used to, stored in memory, called properties or instance variables)

  Computed vars (those whose value is the result of executing/evaluating some code, called computed properties)

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

  functions

  initializers (special functions called when creating a struct or class instance)

  So what's the difference between struct and class?

struct vs. class

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 specific circumstances
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 is class-based)

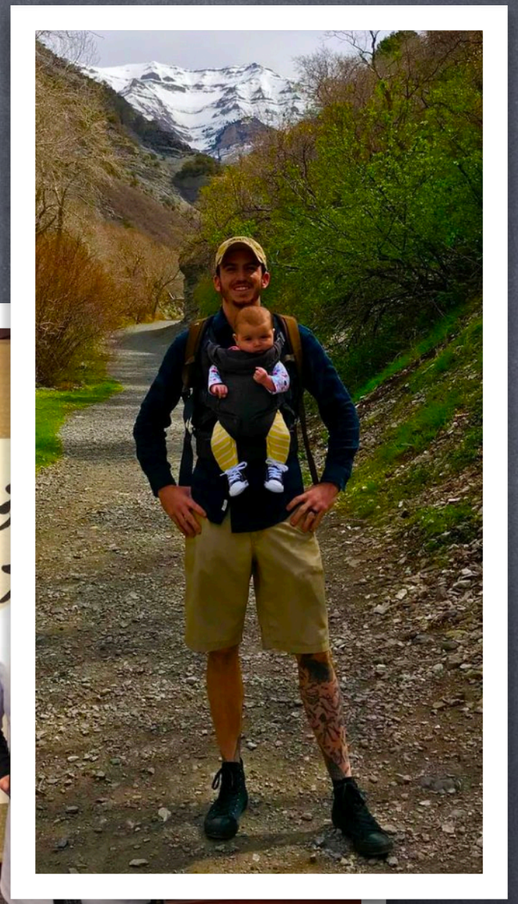
Pushups for Kyle

- Kyle Lamoreaux, 1989-2019
- Goal: 100,000 by the Marine Birthday (Nov. 10)

  Honor Kyle
  Promote the combination of physical and mental health
  Show love, compassion, patience, and support for those around you
  Encourage people in your circle to lean on you when life is tough

  Could you do:
  10, 25, or 50 pushups
  for 10 or 20 days

  On Facebook:
  https://bit.ly/PushupsForKyle



Sketching an App

- It's common to start by sketching the "storyboard" for an app
  - You can use a tool to do it on the computer
  - Or pencil and paper
  - Some people like using a whiteboard
- What features would we want in an app to track Pushups for Kyle?
- Could we sketch a storyboard?

Memory Usage

- Three "segments" of memory
  Code segment
  Stack
  Heap
- Code segment holds compiled machine code
- Stack holds "frames"
  Frame contains parameters passed to functions/methods, return address, local variables
  "Stack overflow" occurs when we have called too many nested methods and there's no more room for additional frames (a common logic error in a recursive program)
- Heap holds allocated objects
  Usually the largest memory segment
  This is what Java manages when it performs garbage collection — it's cleaning up the heap

Generics

- Sometimes we just <u>don't care</u>

  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 don't use variables 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"

Generics

- Common "don't care" example: Array

  An Array contains a bunch of things and it doesn't care what type they are!
  (What — really?? Yes, my friends, yes indeed.)

  But inside Array's code, it has to use variables for the things it contains
  They need types (Swift is a strongly typed language)
  And it needs types for the arguments to Array functions that do things like adding items to it

  Enter our new friend: generics

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>)

Generics

- How Array uses a "don't care" type

  Array's declaration looks something like this:

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

  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 a generic type (e.g. <Element, Foo>)

Generics

- Type Parameter

  I will often refer to these types like `Element` in `Array` as "don't care" types
  But the actual name is "Type Parameter"

  Other languages you may know (e.g. Java) have generics too
  However, Swift combines this with protocols to take it all to the next level
  We'll talk about that next time

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 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
```

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

Homework 3 for Thursday

- HW 3 due BEFORE class Thu. Sep. 8
  - See description on Learning Suite
  - Work on designing an app you'd like to use to track workouts

Additional Reading

- You need to start reading the Swift Language Guide too
  I've given enough homework for Thursday, so I won't measure your reading by giving homework points for it
- Suggested sections of the Swift Language Guide to study now:
  Closures
  Structures and Classes
  Properties
  Methods
- Also, you'll need to study the Swift API Guidelines at some point
  https://swift.org/documentation/api-design-guidelines/

Today's Devotional

- President and Sister Reese speaking today
  I urge you all to participate (it's part of the BYU experience, and you'll find it uplifting)