Recap

More Swift & Xcode Auto Layout

Session 102

Alex Telek
Clarence Ji
KCL Tech iOS Engineers

iTunes U

itunesu.kcl.tech

Optionals

Optionals

```
let numberOfLegs = [“ant” : 6 , “snake” : 0]
```

Optionals

```
let numberOfLegs = [“ant” : 6 , “snake” : 0]

let possibleLeg: Int? = numberOfLegs[“frog”]
```

Optionals

```
let numberOfLegs = [“ant” : 6 , “snake” : 0]

let possibleLeg: Int? = numberOfLegs[“frog”]


if let countLeg = possibleLeg {

  print(“A frog has \(countLeg) legs”)

} else {

  print(“Frog wasn’t found”)
}
```

Tuples

Tuples

```
(3.79, 2.23, 5.65)      //(Double, Double, Double)
```

Tuples

```
(3.79, 2.23, 5.65)        //(Double, Double, Double)

(404, “Not found”)        //(Int, String)
```

Tuples

```
(3.79, 2.23, 5.65)        //(Double, Double, Double)

(404, “Not found”)        //(Int, String)

(2, “banana”, 3.21)       //(Int, String, Double)
```

Closures

Closures

```
func methodWithClosure(closure: (String) -> ()) {

    closure(“In a closure”)
}
```

Closures

```
func methodWithClosure(closure: (String) -> ()) {

    closure("In a closure")
}

methodWithClosure { (returnString) -> () in

    print(returnString)
}
```

Enums

Enums

An enumerations defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code

Enums

An enumerations defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code

```
enum CompassPoint {
    case North
    case South
    case East
    case West
}
```

Enums

An enumerations defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code

```
enum CompassPoint {
   case North
   case South
   case East
   case West
}

let direction = .North
```

Error Handling - Guard - Defer

Error Handling - Guard - Defer

```
fun loadData() throws {
   // Do something…

   throw MyError.Failed
}
```

Error Handling - Guard - Defer

```
fun loadData() throws {
   // Do something…

   throw MyError.Failed
}
```

```
fun refresh() throws {
   defer {
      print(“Clean up”)
   }

   guard isInternetAvailable() else {
      throw MyError. NoInternetConnection
   }

   do {
      try loadData()
   } catch {
      print(error)
   }
}
```

Extensions

Extensions

```
extension String {
    func toArray() -> Array<Character> {
      return Array(self.characters)
    }
}
```

Extensions

```
extension String {
    func toArray() -> Array<Character> {
      return Array(self.characters)
    }
}

“Hello there”.toArray
```

MVC in iOS

Session 103

Alex Telek
Clarence Ji
KCL Tech iOS Engineers

Classes and Structures

Classes and Structures

- Define properties to store values

Classes and Structures

- Define properties to store values
- Define methods to provide functionality

Classes and Structures

- Define properties to store values
- Define methods to provide functionality
- Define constructors to set up their initial state

Classes and Structures

- Define properties to store values
- Define methods to provide functionality
- Define constructors to set up their initial state
- Be extended to expand their functionality beyond default

Classes

Classes

- Inheritance - enables one class to inherit the characteristics of another

Classes

- Inheritance - enables one class to inherit the characteristics of another
- Deinitializers enable an instance of a class to free up any resources it has assigned

Classes

- Inheritance - enables one class to inherit the characteristics of another
- Deinitializers enable an instance of a class to free up any resources it has assigned
- Reference counting allows more than one reference to a class instance - Structures are copied

Class

Class

```
class VideoMode {
    var resolution = Resolution()
    var frameRate = 0.0
    var name: String?

    func display() { … }
}
```

Class

```
class VideoMode {
    var resolution = Resolution()
    var frameRate = 0.0
    var name: String?

    func display() { … }
}

let someVideoMode = VideoMode()
```

Class - Accessing properties and methods

Class - Accessing properties and methods

```
let someVideoMode = VideoMode()
```

Class - Accessing properties and methods

```
let someVideoMode = VideoMode()

someVideoMode.resolution = Resolution(width: 1920, height: 1080)
```

Class - Initializers

Class - Initializers

```
class VideoMode {

    init() { … }
    init?() { … }
    required init() { … }
    init(resolution: Resolution, frameRate: Double) { … }

    convenience init() {

        self.init(Resolution(), frameRate: 25.0)
    }
}
```

Class - Initializers

```
class VideoMode {

    init() { … }
    init?() { … }
    required init() { … }
    init(resolution: Resolution, frameRate: Double) { … }

    convenience init() {

        self.init(Resolution(), frameRate: 25.0)
    }
}

let someVideoMode = VideoMode(Resolution(), frameRate: 25.0)
```

Class - Methods

Class - Methods

```
class VideoMode {

    func display() { … }

    class func minimize() { … }
}
```

Class - Methods

```
class VideoMode {

    func display() { … }

    class func minimize() { … }
}

someVideoMode.display()
```

Class - Methods

```
class VideoMode {

    func display() { … }

    class func minimize() { … }
}

someVideoMode.display()

VideoMode.minimize()
```

Structure

Structure

```
struct Resolution {
    var width = 0
    var height = 0

    mutating func crop() { … }
}
```

Structure

```
struct Resolution {
    var width = 0
    var height = 0

    mutating func crop() { … }
}

let hd = Resolution()
```

Structure

```
struct Resolution {
    var width = 0
    var height = 0

    mutating func crop() { … }
}


let hd = Resolution()

let hd = Resolution(width: 1920, height: 1080)
```

Structure - Accessing properties

Structure - Accessing properties

```
let hd = Resolution()
```

Structure - Accessing properties

```
let hd = Resolution()

hd.width = 1920
```

Structure - Accessing properties

```
let hd = Resolution()

hd.width = 1920

print("The width is \(hd.width)")
```

Choosing Between Classes and Structures

Structure	Class
instances are always passed by value	passed by reference
encapsulate a few simple data	
no inheritance needed	

Inheritance

Inheritance

```
class Vehicle {

    var numberOfWheels = 0

    var description: String {
       return “\(numberOfWheels) wheel(s)”
    }
}
```

Inheritance

```
class Vehicle {

    var numberOfWheels = 0

    var description: String {
       return “\(numberOfWheels) wheel(s)”
    }
}
```

```
class Bicycle: Vehicle {

    override init() {

       super.init()
       numberOfWheels = 2
    }
}
```

Protocols

Protocols

A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. It can be adopted by a class, structure or enumeration.

Protocols

A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. It can be adopted by a class, structure or enumeration.

```
protocol RandomNumberGenerator {

    var lastRandomNumber: Double { get }

    func random() -> Double

    mutating func changeSomething()
}
```

Protocols

Protocols

```
class Game: RandomNumberGenerator, OtherProtocol {

    var lastRandomNumber: Double

    func random() -> Double { … }

    mutating func changeSomething() { … }
}
```

Delegation

Delegation

Delegation is a design patter that enables a class or structure to hand off some of its responsibilities to an instance of another type.

Delegation

Delegation is a design patter that enables a class or structure to hand off some of its responsibilities to an instance of another type.

```
protocol DiceGame {

    func play()
}
```

Delegation

Delegation is a design patter that enables a class or structure to hand off some of its responsibilities to an instance of another type.

```
protocol DiceGame {

    func play()
}

protocol DiceGameDelegate {

    func gameDidStart(game: DiceGame)
    func gameDidEnd(game DiceGame)
}
```

Delegation

Delegation

```
class SnakesAndLadders: DiceGame {

    // extra attributes and methods

    var delegate: DiceGameDelegate?

    func play() {

        delegate?.gameDidStart(self)

        // Play game

        delegate?.gameDidEnd(self)
    }
}
```

Delegation

```
class SnakesAndLadders: DiceGame {

    // extra attributes and methods

    var delegate: DiceGameDelegate?

    func play() {

      delegate?.gameDidStart(self)

      // Play game

      delegate?.gameDidEnd(self)
    }
}
```

```
class DiceGameTracker: DiceGameDelegate {

    myGame.delegate = self

    func gameDidStart() { … }
    func gameDidEnd() { … }
}
```

Video

Stanford - CS193p

Project 2: Basic iOS 9 app

Project 2: Basic iOS 9 app

Let’s build a simple iOS 9 application (MVC in iOS - Project 2)


KCL TECH SOCIETY