### ENUMS

```swift
enum CompassPoint {
    case north
    case south
    case east
    case west
}

// rawValues
enum CompassPoint : String {
    case north
    case south
    case east
    case west
}
print(CompassPoint.north.rawValue)  // “north”
```
ENUMS

```swift
// rawValues
enum CompassPoint : String, CaseIterable {
    case north
    case south
    case east
    case west
}
print(CompassPoint.allCases.count) // "4"
for dir in CompassPoint.allCases {
    print(dir)
}
```

MORE ENUMS

- Associated Data! Each case can have associated state.

```swift
class Motorcycle {
    enum ObnoxiousCycle {
        case harley
        case canam
    }
```
MORE ENUMS

• Setting enum values:
  • Must specify associated values
  • Type inference works on one side or the other, not both.

```swift
let mine = Motorcycle.tourer(numCylinders: 4)
let yours : Motorcycle = .crotchRocket("I will soon die",
  topSpeed: 164)
let theirs = Motorcycle.cruiser  // fail
```

MORE ENUMS

• Associated data can be ignored when reading

```swift
let theHonda = Motorcycle.tourer(numCylinders: 4)

switch (theHonda) {
  case .cruiser:
    print("Get your motor running... ")
  case .tourer:
    print("Head out on the highway...")
  case .crotchRocket:
    print("Looking for adventure... ")
}
```
MORE ENUMS

- Associated values can be read, renamed etc in switch:
  - cruiser type renamed to `noisemaker`
  - crotchRocket’s unnamed data labeled `catchPhrase`

```swift
switch theHonda {
    case .tourer(let numCylinders):
        print("\(numCylinders) cylinders is better than none")
    case .cruiser(let noisemaker):
        print("My \(noisemaker) is louder than yours")
    case .crotchRocket(let catchPhrase, let speedGoal):
        print("zzzzz0000mmm:\(catchPhrase) at \(speedGoal)"")
}
```

PROTOCOLS

- defines functionality, API
  - class, struct, enum
  - can’t allocate object of protocol
  - can create variables to hold objects that conform

- What used for in Swift?
  - allowing generic collection classes
  - multiple inheritance
  - allows *delegation* between View and Controller (soon)
PROTOCOLS

A protocol is a type:

- Any class/struct/enum implementing a protocol must:
  - include specified properties (stored / computed orthogonal)
  - include specified methods
- But:
  - `@objc` can be used to mark methods as optional
  - any class implementing opt prot inherits from NSObject

Often used for delegation.

---

PROTOCOLS

Defining a protocol:
```swift
protocol MyProtocol : InheritedProtocol {
    var stringVar : String { get set }
    func method1(arg1: String, arg2: String) -> Bool
    mutating func method2(addin: Int)
}
```

Implementing a protocol:
```swift
class MyClass: MySuper, MyProtocol {
    var stringVar : String = "hello"
    func method1(arg1: String, arg2: String) -> Bool {
        return arg1 == arg2
    }
    func method2(addin: Int) {
        stringVar = String(repeating: stringVar, count: addin)
    }
    func setStr(str: String) { stringVar = str }
}
```

Can limit to classes (‘class’), mutating for structs, etc.
PROTOCOLS

```swift
protocol Shape {
    func area() -> Double
}

struct Circle: Shape {
    var radius: Double
    func area() -> Double {
        return 3.14 * radius * radius
    }
    init(rad: Double) { radius = rad }
}

class Square: Shape {
    var side = 3.5
    func area() -> Double {
        return side * side
    }
}

var aShape: Shape = Square()
print(aShape.area())

aShape = Circle(rad: 1.84)
print(aShape.area())

switch aShape {
    case is Circle:
        print("It's a circle")
    case is Square:
        print("It's a square")
    default:
        print("something else")
}

if var circle = aShape as? Circle {
    circle.radius = 16
}
```

PROTOCOL EXAMPLE

```swift
protocol DiceGame {
    var dice: Dice { get } // optional chaining
    func play() -> Int
}

protocol DiceGameDelegate: AnyObject { // "AnyObject" means class-only
    func gameDidStart(_ game: DiceGame)
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int)
    func gameDidEnd(_ game: DiceGame)
}

class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: [Int]
    init() {
        board = Array(repeating: 0, count: finalSquare + 1)
        board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
    }
    weak var delegate: DiceGameDelegate? // property type defined by a protocol
    func play() {
        square = 0
        delegate?.gameDidStart(self) // optional chaining
        gameLoop: while square != finalSquare {
            let diceRoll = dice.roll()
            delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
            switch square + diceRoll {
                case finalSquare:
                    break gameLoop
                case let newSquare where newSquare > finalSquare:
                    continue gameLoop
                default:
                    square += diceRoll
                    square += board[square]
            }
        }
        delegate?.gameDidEnd(self)
    }
}
```

Game might be handing off responsibility for tracking progress to the delegate.
MORE PROTOCOLS

Dictionaries require keys to be *unique*
- hashes are most likely unique
- equality testing

Dictionary keys required to implement *Hashable*:

```swift
protocol Hashable: Equatable {
    var hashValue: Int { get }
}
protocol Equatable {
    static func ==(lhs: Self, rhs: Self) -> Bool
}
```

CASTING

- Casting for:
  - casting from *superclass* to *subclass*
  - object to *protocol* it implements

```swift
class Vehicle {
    var numWheels : Int = 4
}
class Car : Vehicle {
    var maker: String = "Honda"
}
var mover : Vehicle = Car()
// checking the type
if mover is Car {
}
if let wheels = mover as? Car {
    print(wheels.maker, wheels.numWheels)
}
Honda 4
ERROR HANDLING

• propagates upwards to caller
  • by calling function, also throwing

• handle using do-catch
  • convert to optional value
  • assert that it just won’t happen

• errors are values that correspond to Error protocol
  • empty

```swift
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

ERROR HANDLING

• you will know because:

```swift
func foo() throws -> Int { …
```

• Calls to these funcs must be caught:

```swift
do {
    try func foo()
} catch let error {
    // something that knows the Error protocol
}
```

• If you are certain they won’t err, you can force:

```swift
try! foo()  // crash on throw…
```

• Or conditionally try, getting an optional:

```swift
let bar = try? foo()  // bar is Int?
```
ERROR HANDLING

- works well w/ enum associated values

```swift
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

... throw VendingMachineError.insufficientFunds(coinsNeeded: 5)

EXTENSIONS

Extensions add new functionality to existing struct/class/enum/protocol, even if you don’t have original source.

- Extensions can:
  - add computed properties
  - define new methods
  - provide new initializers
  - define subscripts
  - make existing type conform to protocol
EXTENSIONS

```swift
struct Circle : Shape {
    var radius : Double
    func area() -> Double { return 3.14 * radius * radius }
    init(rad: Double) { radius = rad }
}

extension Circle {
    mutating func changeRadius(newr: Double) {
        radius = newr
    }
}

extension Int {
    func square() -> Int { return self * self }
    subscript(digitIndex: Int) -> Int {
        var decimalBase = 1
        for _ in 0..<digitIndex {
            decimalBase *= 10
        }
        return (self / decimalBase) % 10
    }
}

print(4.square())
print(746381295[3])
```

NSATTRIBUTEDSTRING

- String w/ attributes (a dictionary) per character
  - keys are fonts, colors, drawing instructions
- characters often share a dictionary
  - even the entire string
- Can use `NSAttributedString` on:
  - UIButtons
  - UILabels
  - while drawing
  - etc
NSATTRIBUTEDSTRING

Creating a NSAttributedString:

```swift
let attr: [NSAttributedString.Key:Any] = [
    .strokeColor : UIColor.green,
    .strokeWidth : 5 // neg num means fill, pos outline
]
let attrStr = NSAttributedString(string: "Hello!", attributes: attr)
button.setAttributedTitle(attrStr, for: UIControl.State.normal)
```

From Objective-C, be careful when mult dictionaries per string

VESTIGES OF OBJECTIVE-C

- Any
  - can hold any type of object
- AnyObject
  - any class

Swift is strongly typed, so you can use directly.

Need a concrete type...
VESTIGES OF OBJECTIVE-C

• Where is it used?
  
  let attributes: [NSAttributedStringKey: Any] = …

• Attributes can be different things: UIFont, UIColor, etc
  
  func prepare(for segue: UIStoryboardSegue, sender: Any?)

• sender (UIButton ..) caused the “segue”

• optional because sender need not be specified

Might use enum with associated data in Swift.

VESTIGES OF OBJECTIVE-C

• How to use a var of type Any?
  
  Must be converted, as don’t know what it is…

  Conversion with Swift's as? keyword, makes optional.

  let foo: Any = [0, 1, 2]
  print(foo.count())
  if let bar = foo as? [Int] { …

  Can also check it can be converted with is.

  if foo is [Int] { …
VESTIGES OF OBJECTIVE-C

```swift
var things = [Any]()
    things.append(0)
    things.append(0.0)
    things.append(42)
    things.append(3.14159)
    things.append("hello")
    things.append((3.0, 5.0))
    puts("\(things")

for thing in things {
    switch thing {
        case 0 as Int:
            print("zero as an Int")
        case 0 as Double:
            print("zero as a Double")
        case let someInt as Int:
            print("an integer value of \(someInt)")
        case let someDouble as Double where someDouble > 0:
            print("a positive double value of \(someDouble)")
        case is Double:
            print("some other double value that I don't want to print")
        case let someString as String:
            print("a string value of \"\(someString)\"")
        case let (x, y) as (Double, Double):
            print("an (x, y) point at \(x), \(y)")
        case let stringConverter as (String) -> String:
            print(stringConverter("Michael"))
        default:
            print("something else")
    }
}
```

VIEW CONTROLLER LIFECYCLE

- Why do we care?
  - easiest way to install hooks to your code
- Creation
  - from storyboard (nib, coder init)
- And then:
  - segue preparation
  - setting outlets
  - appearing and disappearing
  - setting bounds
  - geometry changes
CREATION

- from .xib (XML version of a .nib)
  - a way to create re-usable custom view
  - called either programmatically or through code
- directly through code
- through storyboard

Rare.

```swift
override func awakeFromNib() {
    super.awakeFromNib()
    // very early
}
```

VIEWDidLoad

- Primary place to do app-specific setup
  - view is in place
    - good time to customize from model
  - start model
  - setup interactions across the MVC
- Bear in mind
  - outlets are set
  - bounds are not set
  - called only once
**viewWillAppear**

- right before view appears on screen
- update views to reflect changes since last seen
  - DB changes
  - running times
  - background progress
- called
  - possibly many times, even consecutively
  - *view does not always appear*

**viewDidAppear**

- view is on-screen
- UI tasks are now allowable
  - animations
  - GPS
  - timers
  - network fetches
- Why all this here instead of ViewWillAppear?
  - willAppear does not always lead to didAppear
  - don’t want to waste cycles
**VIEW WILL DISAPPEAR**

- view still on-screen
  - but about to go
- wind things down
  - animations
  - GPS
  - timers
  - network fetches (maybe)

**VIEW DID DISAPPEAR**

- it’s gone
  - possibly de-allocate large, memory-intensive data

- used only rarely
VIEW WILL LAYOUT SUB VIEWS

- Top-level bounds about to change
  - might change representation of some views

- Caveats
  - doesn’t mean bounds actually will change
  - can be called often

- ViewDidLayoutSubviews()
**VIEWWILLTRANSITION**

- Auto-rotation!

```swift
override func viewWillTransition(to size: CGSize,
  with coordinator: UIViewControllerTransitionCoordinator)
```

- **size** gives you new bounds
- animations
  - rotate done automatically
  - might interfere with an animation you have
  - alongsideTransition:

```swift
func animate(alongsideTransition animation: ((UIViewControllerTransitionCoordinatorContext) -> Void)?,
  completion: ((UIViewControllerTransitionCoordinatorContext) -> Void)? = nil) -> Bool
```

**LOWMEMORY**

- Happens rarely, but
  - called when your app starts taking up more and more memory
  - might lead to your app being killed

- Most devices have huge amount of memory
  - you probably have a memory leak (memory cycle)
SUMMARY

- startup
- awakeFromNib (if from storyboard)
- segue prepare()
- outlets set
- viewDidLoad
- bounds set
- view will/did appear/disappear()

SEE THEM IN ACTION...

https://gitlab.cs.umd.edu/keleher/iosstudents/blob/master/VCLLoggingViewController.swift