Swift

Mobile Application Development in iOS

School of EECS

Washington State University

Instructor: Larry Holder

Why Swift

Pros

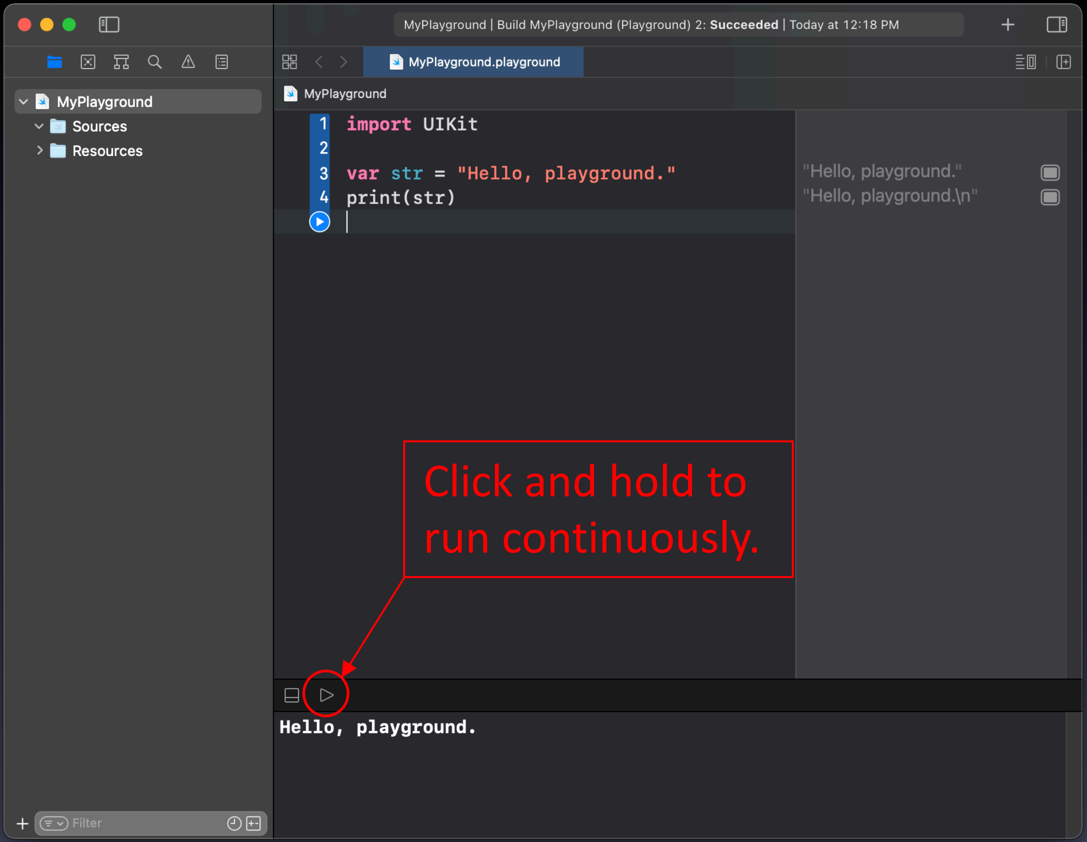
- Recommended for all iOS, macOS, watchOS, and tvOS app development
- Designed by Apple, but now open source
- Available for Windows and Linux
- Faster than Python, and even C++ in some cases
- Can call C/C++ code

Cons

- Not as many developers
- Not as many packages
- Cross-platform support is weak
- Weirdness to maintain compatibility to Objective-C

Xcode Playgrounds

- File → New → Playground


MyPlayground | Build MyPlayground (Playground) 2: Succeeded | Today at 12:18 PM
MyPlayground.playground
MyPlayground
Sources
Resources
Filter

```
import UIKit

var str = "Hello, playground."
print(str)
```

```
"Hello, playground."
"Hello, playground.\n"
```

Click and hold to run continuously.

```
Hello, playground.
```

Good Swift Tutorials

- Swift Tour (version 5)
  - docs.swift.org/swift-book/GuidedTour/GuidedTour.html
  - Includes Playground for all code
- CodingWithChris.com (version 5)
  - codewithchris.com/learn-swift

Caveat

- Assume proficiency in some object-oriented language (e.g., C++, Java, Python)

Constants, Variables & Types

- Constants (let) vs. variables (var)
- Types (Swift tries to guess)
  - Basic types: Bool, Int, Float, Double, String
  - Collection types: Array, Set, Dictionary
  - Tuples: (x1, x2, …)

Constants, Variables & Types

```
let three = 3          // Constant
var four: Int = 4      // Variable

var myarr = [String]()            // Array of strings
var myset = Set<String>()         // Set of strings
var mydict = [String: Int]()      // Dictionary

var shoppingList = ["coffee": three, "candy": four]
shoppingList["milk"] = 2
for (item, amount) in shoppingList { // tuple
    print("\(item): \(amount)")
}
```

Optionals ? and Unwrapping !

- Optional variable ? can be nil or hold a value

```
var possibleStr: String? = "Hello" // optional type
print(possibleStr) // outputs "Optional("Hello")", warning
var forcedStr: String = possibleStr! // unwrapping
print(forcedStr) // outputs "Hello"
let assumedStr: String! = "Hello" // implicitly unwrapped
let implicitStr: String = assumedStr // no need for !
var str2: String? // optional type, set to nil
print(str2!) // error
```

- Optional binding

```
if let tempStr = possibleStr { // true if non-nil
    print(tempStr)
} else {
    print("string empty")
}
```

Range Operators

- Range operators (a...b, a..<b)

```
let count = 5
for index in 1...count {
    print("\(index)")    // 1 2 3 4 5
}

for index in 0..<count {
    print("\(index)")    // 0 1 2 3 4
}
```

Functions

```
func fahrenheitToCelsius (temp: Float) -> Float {
    let tempC = (temp - 32.0) * 5.0 / 9.0
    return tempC
}

func printCelsius (temp tempF: Float) {
    let tempC = fahrenheitToCelsius(temp: tempF)
    print("\(tempF) F = \(tempC) C")
}

func printF2CTable (_ low: Int = 0, _ high: Int = 100) {
    for temp in low...high {
        printCelsius(temp: Float(temp))
    }
}
printF2CTable()
printF2CTable(70)
printF2CTable(70,80)
```

Function Types

```
func addTwoInts (_ a: Int, _ b: Int) -> Int {
    return a + b
}

var mathFunction: (Int, Int) -> Int = addTwoInts

print("Result: \(mathFunction(2, 3))") // prints “Result: 5”

func printMathResult (_ mathFunction: (Int, Int) -> Int,
                      _ a: Int, _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}

printMathResult(addTwoInts, 3, 5) // prints “Result: 8”
```

Closures

- Self-contained block of code
- Can capture references to variables in context
- General form:

```
{  (parameters) -> return-type in
        statements
}
```

Closures (cont.)

```
var names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]

func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}

var reversedNames = names.sorted (by: backward)

reversedNames = names.sorted (by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

Closures: Capturing Values

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}

let incrementByTen = makeIncrementer(forIncrement: 10)

incrementByTen() // returns a value of 10
incrementByTen() // returns a value of 20
incrementByTen() // returns a value of 30
```

Escaping Closures

- Closure passed to function, but called after function returns

```
var completionHandlers: [() -> Void] = []

func addCompletionHandler (handler: @escaping () -> Void) {
    completionHandlers.append(handler)
}

func printHello() {
    print("Hello")
}
addCompletionHandler(handler: printHello)

for handler in completionHandlers {
    handler()
}
```

Enumerations

```
enum Direction {
    case up // does not imply .up = 0
    case left
    case down
    case right
}

var playerDirection = Direction.right
playerDirection = .up  // type inference

func turnLeft (direction: Direction) -> Direction {
    var newDirection: Direction
    switch direction {
        case .up: newDirection = .left    // no break
        case .left: newDirection = .down
        case .down: newDirection = .right
        case .right: newDirection = .up
    }
    return newDirection
}
```

Enumerations (cont.)

```
func facingLeftOrRight (direction: Direction) -> Bool {
    switch direction {
        case .left, .right: return true
        default: return false
    }
}
```

- Raw values

```
enum Direction2: Int {
    case up = 0, left, down, right    // now they're Int's
}
Direction2.left.rawValue        // equals 1

enum Direction3: String {
    case up, left, down, right     // now they're String's
}
Direction3.left.rawValue        // equals "left"
```

Classes

```
class Player {
    var direction: Direction
    var speed: Float
    var inventory: [String]?   // initialized to nil

    // init required to set uninitialized variables
    init (speed: Float, direction: Direction) {
        self.speed = speed
        self.direction = direction
    }

    func energize() {
        speed += 1.0
    }
}

var player = Player(speed: 1.0, direction: .right)
```

Classes (cont.)

```
class FlyingPlayer : Player {
    var altitude: Float

    init (speed: Float, direction: Direction, altitude: Float) {
        self.altitude = altitude
        super.init (speed: speed, direction: direction)
    }

    override func energize() {
        super.energize()
        altitude += 1.0
    }
}

var flyingPlayer = FlyingPlayer(speed: 1.0, direction: .right,
altitude: 1.0)
```

Must initialize all non-optional child properties before initializing parent.

Class vs. Struct

- Classes passed by reference
- Structs passed by value

```
class Foo1 {
    var x : Int = 1
}
func changeX (foo : Foo1) {
    foo.x = 2
}

var foo1 = Foo1()
changeX(foo: foo1)
foo1.x     // equals 2
```

```
struct Foo2 {
    var x : Int = 1
}

func changeX (foo: Foo2) {
    foo.x = 2 // error
    var tmpFoo: Foo2 = foo
    tmpFoo.x = 2
}

var foo2 = Foo2()
changeX(foo: foo2)
foo2.x     // equals 1
```

Optional Chaining

```
var myPlayer = Player(speed: 1.0, direction: .right)

let firstItem = myPlayer.inventory.first // error
let firstItem = myPlayer.inventory!.first// error
let firstItem = myPlayer.inventory?.first// nil (OC)
myPlayer.inventory?.append("potion")      // nil (OC: no effect)
type(of: firstItem)                       // Optional<String>

if let item = myPlayer.inventory?.first {
    print("item = \(item)")           // nothing printed (OC)
}

myPlayer.inventory = []                   // array initialized
myPlayer.inventory?.append("potion")      // "potion" added
let item = myPlayer.inventory?.first      // "potion"

if let item = myPlayer.inventory?.first {
    print("item = \(item)")           // "item = potion"
}
```

Error Handling

- Do-try-throw-catch error handling

```
enum myError: Error {
    case good
    case bad
    case fatal
}

func throwsError () throws {
    throw myError.fatal
}
```

```
func testError () {
    do {
        try throwsError()
        print("no error")
    } catch myError.fatal {
        print("fatal")
    } catch {
        print("good or bad")
    }
}
```

Error Handling

- try?: returns nil if error thrown
- try!: assumes no error thrown (or error)
- guard <condition> else {throw or return}
  - Preferred to: if not <condition> {throw or return}

```
if let result = try? throwsError() {
    print("no error: result = \(result)")
}
let forcedResult = try! throwsError()
let amount = 1
guard (amount > 0) else {
    throw myError.bad
}
```

Type Casting

- Regular type casting

```
let x = 10
let xstr = String(x)   // "10", xstr of type String
let xstr2 = "\(x)"     // "10"
let ystr = "100"
let y = Int(ystr)      // 100, y of type Optional<Int>
var arrayOfAnything: [Any]
var arrayOfAnyClassInstances: [AnyObject]
```

- Downcasting (as?, as!)

```
var playerArray = [Player]()
playerArray.append(flyingPlayer)
playerArray.append(player)
var fp : FlyingPlayer!
fp = playerArray[0] as? FlyingPlayer     // fp = flyingPlayer
fp = playerArray[1] as? FlyingPlayer     // fp = nil
fp = playerArray[1] as! FlyingPlayer     // error
```

Protocol

- Required properties and methods
  - Although some can be optional
- Adopted by class, struct or enum type
- Said to "conform" to protocol

```
protocol MyFunProtocol {
    func isFun() -> Bool
}

class MyFunClass1: MyFunProtocol {
    func isFun() -> Bool {
        return true
    }
}
```

Delegate

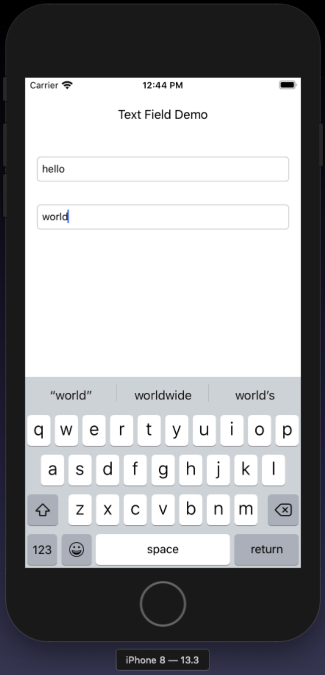
- Object that responds to events from another object
- Defines protocol that must be followed
- Delegation
  - Class defines delegate property set to delegate object
  - Class calls delegate methods in response to events

Delegation: Example

```
protocol MyFunDelegate {  // Functions a delegate class must provide
     func isFun() -> Bool
}

class MyFunDelegateClass: MyFunDelegate {  // one possible delegate class
     func isFun() -> Bool {
          return true
     }
}

class MyFunClass2 {
     var delegate: MyFunDelegate?  // Can be set to any class instance
                                   // conforming to MyFunDelegate protocol

     func fun() -> Bool {
          return delegate?.isFun() ?? false
     }
}

var myFunClass2 = MyFunClass2()
var myFunClassDelegate = MyFunDelegateClass()
myFunClass2.delegate = myFunClassDelegate
myFunClass2.fun()     // returns true
```

Delegate Example: UITextField

- View elements that generate multiple different actions use delegates (rather than IBAction)
- UITextFieldDelegate
  - textFieldDidBeginEditing
  - textFieldDidEndEditing
  - textFieldShouldReturn
- More
  - developer.apple.com/documentation/uikit/uitextfielddelegate



UITextField Delegate

```
class ViewController: UIViewController, UITextFieldDelegate {
    @IBOutlet weak var myTextField: UITextField!

    override func viewDidLoad() {
        super.viewDidLoad()
        myTextField.delegate = self
    }

    func textFieldDidEndEditing(_ textField: UITextField) {
        print("Message: \(textField.text!)")
    }

    func textFieldShouldReturn(_ textField: UITextField) -> Bool {
        textField.resignFirstResponder()  // remove keyboard on Return
        return false   // do default behavior (i.e., nothing)? No
    }
}
```

Resources

- Swift
  - swift.org
  - developer.apple.com/swift
- UITextField
  - developer.apple.com/documentation/uikit/uitextfield
  - developer.apple.com/documentation/uikit/uitextfielddelegate