SWIFT

PROGRAMMING LANGUAGE

by Matteo Battaglio

@ Commit University – 18/03/2015

HELLO WORLD

- Matteo Battaglio
- iOS dev & partner @ Codermine
- #pragma mark co-founder

#PRAGMA MARK

fb.com/groups/pragmamark

www.pragmaconference.com

WHO ARE YOU?

Slides and Sample Code @ GitHub:

https://github.com/madbat/SwiftWorkshop

TABLE OF CONTENTS

HISTORY

LANGUAGE FATHER

Chris Lattner

LANGUAGE BIRTH

I started work on the Swift Programming Language in July of 2010. I implemented much of the basic language structure, with only a few people knowing of its existence. A few other (amazing) people started contributing in earnest late in 2011, and it became a major focus for the Apple Developer Tools group in July 2013.

– Chris Lattner

LANGUAGE INSPIRATION

The Swift Programming Language greatly benefited from the experiences hard-won by many other languages in the field, drawing ideas from Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, and far too many others to list.

– Chris Lattner

PRINCIPLES

SWIFT PROGRAMMING LANGUAGE

imperative, functional, object oriented, multi-paradigm,
static, strong typed, type safe, inferred,
general purpose, compiled, fast,
modern, elegant, clean,
funny, happy,

SYNTAX

Basic Syntax

Functions & Closures

Data Types & Instances

Extensions, Protocols & Generics

BASIC SYNTAX

- Constants & Variables
- Numbers & Booleans
- Tuples & Optionals
- Operators & Loops
- Conditionals

CONSTANTS & VARIABLES

TYPE INFERENCE

```
let constant = 1 // readonly, cannot be re-assigned
constant = 2 // ❌ ERROR!!!

var variable = 1 // readwrite, can be re-assigned
variable = 2 // OK

// Type inference multiple variables
var variable1 = 1, variable2 = 2, variable3 = 3
```

Prefer 'let' over 'var' whenever possible

CONSTANTS & VARIABLES

TYPE ANNOTATIONS

```
let constant = 1
let constant: Int = 1 // no need for : Int

var variable: Int
variable = 1

// Type annotations multiple variables
var double1, double2, double3: Double
```

CONSTANTS & VARIABLES

UNICODE CHARACTERS IN CONSTANT & VARIABLE NAMES

```
let 🍀🍀🍀🍀 = 4
var 🌸🌸🌸🌸🌸 = 5
```

KILLER APPLICATION

NUMBERS

INT, UINT, FLOAT & DOUBLE

```
var magic: Int = 42 // decimal

0b101010 // binary
0o52     // octal
0x2A     // hexadecimal

let pi: Double = 3.14 // 64-bit (default)
let pi: Float  = 3.14 // 32-bit

000009.90              // padded
1_000_000.000_000_1    // underscores
```

BOOLEAN

TRUE & FALSE

```
let enabled: Bool = true // obj-c YES
let hidden = false       // obj-c NO
```

TYPE ALIASES

DEFINE AN ALTERNATIVE NAME FOR AN EXISTING TYPE

```
typealias SmallIntAlias = UInt16
let min = SmallIntAlias.min // exactly like original UInt16.min
```

TUPLES

LIGHTWEIGHT, TEMPORARY CONTAINERS FOR MULTIPLE VALUES

```
let complex = (1.0, -2.0)  // Compound Type: (Double, Double)
let (real, imag) = complex // Decompose
let (real, _) = complex    // Underscores ignore value

// Access by index
let real = complex.0
let imag = complex.1

// Name elements
let complex = (real: 1.0, imag: -2.0)
let real = complex.real
```

OPTIONALS

AN OPTIONAL VALUE EITHER CONTAINS A VALUE OR NIL

```
var optionalInt: Int? = 42
optionalInt = nil // to indicate that the value is missing

var optionalDouble: Double? // automatically sets to nil

// Check if nil
optionalDouble == nil
optionalDouble != nil
```

OPTIONALS

FORCE UNWRAP & OPTIONAL BINDING

```
// Force unwrap
let optionalInt: Int? = 42
let definitelyInt = optionalInt! // throws runtime error if nil

// Optional binding
if let definitelyInt = optionalInt {
  // runs if optionalInt is not nil and sets definitelyInt: Int
}
```

OPTIONALS

IMPLICITLY UNWRAPPED OPTIONALS

```
// Used mainly for class initialization
var assumedInt: Int! // set to nil
assumedInt = 42

var implicitInt: Int = assumedInt // do not need an exclamation mark
assumedInt = nil
implicitInt = assumedInt // ❌ RUNTIME ERROR!!!
```

OPERATORS

COMMON OPERATORS 1/2

```
// Modulo Operator
3 % 2 // 1

// Increment and Decrement Operator
i++; ++i; i--; --i

// Compound Assignment Operator
i += 1 // i = i + 1

// Logical Operator
a || b && c // equivalent to a || (b && c)
```

OPERATORS

COMMON OPERATORS 2/2

```
// Ternary Conditional Operator
(a == b ? 1 /* if true */ : 0 /* if false */)

// Nil Coalescing Operator
a ?? b // a != nil ? a! : b

// Closed Range Operator
0...2 // 0, 1, 2

// Half Open Range Operator
0..<2 // 0, 1
```

LOOPS ◎

FOR [IN]

```
// old school c-style for loop
for var index = 0; index < 2; index++ { /* use index */ }

// new iterator-style for-in loop
for value in 0..<2 { /* use value */ }
```

LOOPS ◎

[DO] WHILE

```
// while evaluates its expression at the top of the loop
while x > 2 { /* */ }

// do-while evaluates its expression at the bottom of the loop
do { /* executed at least once */ } while x > 2
```

CONDITIONALS

IF-ELSE

```
let temperature = 40

var feverish: Bool
if temperature > 37 {
    feverish = true
} else {
    feverish = false
}
```

CONDITIONALS

SWITCH

```
switch x { // break by default
case value1:
  /* ... */
case value2, value3:
  fallthrough  // to not break
default:
  /* ... */
} // switch must be exhaustive
```

CONDITIONALS

SWITCH RANGE MATCHING

```
switch count {
case 0:
  /* ... */
case 1...9:
  /* ... */
default:
  /* ... */
}
```

CONDITIONALS ⎇

SWITCH TUPLE MATCHING

```
switch point {
case (0, 0):
  /* ... */
case (_, 0):
  /* ... */
case (let x, 1): // value binding
  /* use x value */
default:
  /* ... */
}
```

CONDITIONALS

SWITCH WHERE

```
switch point {
case let (x, y) where x == y:
   /* use x value */
case let (x, y):
   /* use x and y values */
} // switch is exhaustive without default:
```

PARENTHESES & BRACES IN ◎ & ⎇

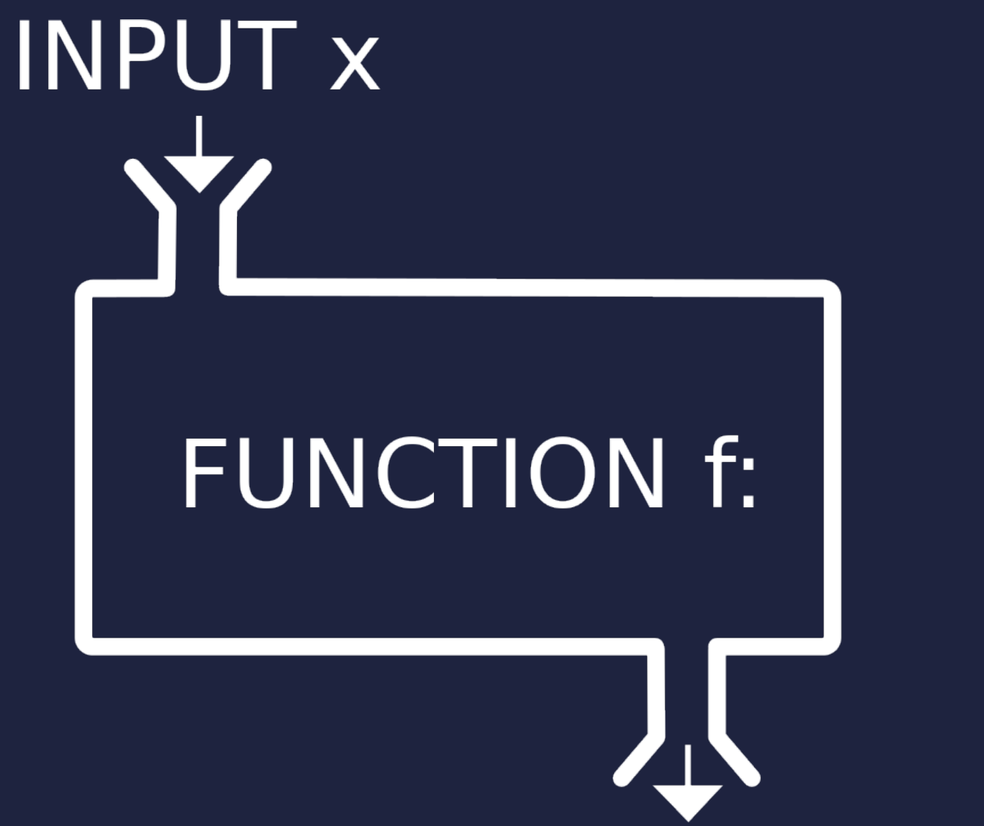
(Parentheses) are optional and by convention are often omitted

{Braces} are always required, even in one statement-long bodies[1]

```
if  temperature > 37  { feverish = true } // OK
if (temperature > 37) { feverish = true } // OK
if (temperature > 37)   feverish = true   // ❌ ERROR!!!
```

[1] Do you remember the infamous "goto fail;" Apple's SSL bug?

FUNCTIONS & CLOSURES



FUNCTIONS

FIRST-CLASS FUNCTION

- assign a function to variable
- pass function as argument to another function
- return a function from a function
- functional programming tools: map, reduce, flatMap, ...

FUNCTIONS

DECLARATION & CALL

```
// With parameters and return value
func foo(parameter1: Type1, parameter1: Type2) -> ReturnType { /* function body */ }
foo(argument1, argument2) // call

// Without parameters and return value
func bar() -> Void { /* function body */ }
func baz() -> () { /* function body */ }
func quz() { /* function body */ }
quz() // call
```

FUNCTIONS

EXTERNAL, LOCAL & DEFAULT PARAMETERS

```
// external and local parameter names
func foo(externalParameterName localParameterName: Type) { /* body use localParameterName */ }
foo(externalParameterName: argument) // call must use externalParameterName label

// # = shorthand for external parameter names
func bar(#parameterName: Type) { /* body use parameterName */ }
bar(parameterName: argument) // call must use parameterName label

// default parameter values
func baz(parameter1: Type1, parameterWithDefault: Int = 42) { /* ... */ }
baz(argument1) // call can omit value for parameterWithDefault

// automatic external parameter name for default parameters, as if declared #parameterWithDefault
baz(argument1, parameterWithDefault: 10)
```

FUNCTIONS

VARIADIC PARAMETERS

```
func foo(parameter: Int...) {
  /* parameter */  // use parameter as an array with type [Int]
}

foo(1, 2, 3, 4, 5) // call with multiple arguments
```

FUNCTIONS

VARIABLE PARAMETERS

```
// let creates a local immutable copy of parameter - let is the default and can be omitted
// var creates a local mutable copy of parameter - mutating it can have a surprising behaviour
func foo(let costantParameter: Int, var variableParameter: Int) {
  costantParameter  += 1 // ❌ ERROR!!!
  variableParameter += 2 // OK
}

// inout parameters are passed by reference
func bar(inout parameter: Int) { parameter = 42 }

var x = 0 // cannot be declared with 'let'
bar(&x)   // need to prepend & to the argument
x         // = 42
```

FUNCTIONS

FUNCTION TYPE

```
// The Type of addOne function is (Int) -> Int
func addOne(n: Int) -> Int { return n + 1 }

// Function as parameter
func foo(functionParameter: (Int) -> Int) { /* */ }
foo(addOne)

// Function as return type
func bar() -> (Int) -> Int {
  func addTwo(n: Int) -> Int { return n + 2 }
  return addTwo
}
```

FUNCTIONS

CURRIED FUNCTIONS

```
// Rewrite a function that takes multiple parameters as
// an equivalent function that takes a single parameter and returns a function

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

func addTwoIntsCurried (a: Int) -> (Int) -> Int {
  func addTheOtherInt(b: Int) -> Int { return a + b }
  return addTheOtherInt
}
// equivalent to:
func addTwoIntsCurried (a: Int)(b: Int) -> Int { return a + b }

addTwoInts(1, 2)         // = 3
addTwoIntsCurried(1)(2) // = 3
```

CLOSURES

MEANING & SYNTAX

Closures are blocks of functionality that can be passed around

```
{ (parameter1: Type1, parameter2: Type2) -> ReturnType in
  / * ... */
}
```

CLOSURES

SORTING WITHOUT CLOSURES

```
func sorted(array: [Int], algorithm: (Int, Int) -> Bool) -> [Int] { /* ... */ }

let array = [1, 2, 3]
func backwards(i1: Int, i2: Int) { return i1 > i2}

var reversed = sorted(array, backwards)
```

CLOSURES

SORTING WITH CLOSURES 1/2

```
// Fully declared closure
reversed = sorted(array, { (i1: Int, i2: Int) -> Bool in return i1 > i2 } )

// Infer closure type
reversed = sorted(array, { (i1, i2) in return i1 > i2 } )

// Implicit returns
reversed = sorted(array, { (i1, i2) in i1 > i2 } )
```

CLOSURES

SORTING WITH CLOSURES 2/2

```
// Shorthand argument names
reversed = sorted(array, { $0 > $1 } )

// Operator functions
reversed = sorted(array, >)

// Trailing closure: outside of () only if it's function’s final argument
reversed = sorted(array) { $0 > $1 }

// Optional parentheses
array.map({ $0 + 1 })
array.map { $0 + 1 } // equivalent
```

CLOSURES

CAPTURING VALUES

Closures can capture and store references to any constants and variables from the context in which they are defined.

```
func makeRepeaterWithRepeatedValue(valueToRepeat: Int) -> () -> [Int] {
  var capturedArray = []
  func repeater() -> [Int] {            // capture valueToRepeat
    capturedArray.append(valueToRepeat) // capture capturedArray
    return capturedArray
  }
  return repeater
}
let repeater3 = makeRepeaterWithRepeatedValue(3)
repeater3() // [3]
repeater3() // [3,3]
```

CLOSURES

@AUTOCLOSURE WRAPS FUNCTION ARGUMENTS IN EXPLICIT CLOSURE[2]

```
// You can apply the @autoclosure attribute to a function type that
// has a parameter type of () and that returns the type of an expression
func simpleAssert(condition: () -> Bool, message: String) { if !condition() { println(message) } }

// An autoclosure function captures an implicit closure over the specified expression,
// instead of the expression itself.
func simpleAssert(condition: @autoclosure () -> Bool, message: String)  { if !condition() { println(message) } }
simpleAssert(3 % 2 == 0, "3 isn't an even number.")

// By taking the right side of the expression as an auto-closure,
// Swift provides proper lazy evaluation of that subexpression
func &&(lhs: BooleanType, rhs: @autoclosure () -> BooleanType) -> Bool { return lhs.boolValue ? rhs().boolValue : false }
```

[2] see Swift Blog Post: Building assert() in Swift

FUNCTIONS VS CLOSURES

- **Global functions:**
  named closures / do not capture any values
- **Nested functions:**
  named closures / capture values from enclosing function
- **Closure expressions:**
  unnamed closures / capture values from their surrounding context

ADVANCED OPERATORS

```
// Prefix Operator Functions
prefix func -(vector: Vector) -> Vector { /* return the opposite Vector */ }
let negativeVector = -positiveVector

// Infix Operator Functions
func +(left: Vector, right: Vector) -> Vector { /* return the Vector sum */ }
func +=(inout left: Vector, right: Vector) {  /* set left to the Vector sum */ }
var originalVector = /* a Vector */; var anotherVector = /* another Vector */;
originalVector += anotherVector

// Custom Operators
prefix operator +++ {} // Custom Infix Operators can specify Precedence and Associativity
prefix func +++(inout vector: Vector) -> Vector { vector += vector; return vector; }
```

DATA TYPES & INSTANCES

- Value & Reference Types
- Methods, Properties & Subscript
- Inheritance & Initialization
- Automatic Reference Counting
- Type Casting & Access Control

VALUE & REFERENCE TYPES

ENUMERATIONS & STRUCTURES VS CLASSES

- Enumerations & Structures are passed by Value
- Classes are passed by Reference

Enumerations & Structures are always copied when they are passed around in the code, and do not use reference counting.

VALUE & REFERENCE TYPES

COMMON CAPABILITIES OF ENUMERATIONS, STRUCTURES & CLASSES:

- Define properties, methods and subscripts
- Define initializers to set up their initial state
- Be extended to expand their functionality
- Conform to protocols to provide standard functionality

VALUE & REFERENCE TYPES

ADDITIONAL CAPABILITIES OF CLASSES:

- Inheritance enables one class to inherit from another.
- Type casting enables to check the type of an object at runtime.
- Deinitializers enable an object to free up any resources.
- Reference counting allows more than one reference to an object.

ENUMERATIONS

AN ENUMERATION DEFINES A COMMON TYPE FOR A GROUP OF ITEMS

```
enum CellState {
  case Alive
  case Dead
  case Error
}
let state1 = CellState.Alive;
let state2 : CellState = .Dead // if known type, can drop enumeration name

switch state1 {
  case .Alive:
    /* ... */
  case .Dead:
    /* ... */
  default:
    /* ... */
}
```

ENUMERATIONS

AN ENUMERATION ITEM CAN HAVE AN ASSOCIATED VALUE

```
enum CellState {
  case Alive
  case Dead
  case Error(Int) // associated value can also be a tuple of values
}
let errorState = CellState.Error(-1);

// extract associated value as constant or variable
switch errorState {
  case .Alive:
    /* ... */
  case .Dead:
    /* ... */
  case .Error(let errorCode): // == case let .Error(errorCode)):
    /* use errorCode */
}
```

ENUMERATIONS

OPTIONAL IS AN ENUMERATION WITH ASSOCIATED VALUE[3]

```
enum OptionalInt {
  case None
  case Some(Int)
}

let maybeInt: OptionalInt = .None // let maybeInt: Int? = nil
let maybeInt: OptionalInt = .Some(42) // let maybeInt: Int? = 42
```

[3] indeed the Swift Library's Optional use Generics (see below)

ENUMERATIONS

AN ENUMERATION ITEM CAN HAVE A RAW VALUE

```
enum CellState {
  case Alive = 1
  case Dead  = 2
  case Error = 3
}

enum CellState: Int { // Specify the Item Raw Value Type
  case Alive = 1, Dead, Error // Int auto-increment
}

let stateValue = CellState.Error.rawValue // 3
let aliveState: CellState? = CellState(rawValue: 1) // CellState.Alive
```

STRUCTURES

COPY VALUE ON ASSIGNMENT OR WHEN PASSED INTO FUNCTION

```
// Structures are value types
struct CellPoint {
  var x = 0.0
  var y = 0.0
}

// Structures are always copied when they are passed around
var a = CellPoint(x: 1.0, y: 2.0); var b = a; b.x = 3.0;
a.x // 1.0 - a not changed
```

STRUCTURES

MANY SWIFT LIBRARY'S BASE TYPES ARE STRUCTURES

- String
- Character
- Array
- Dictionary

STRUCTURES

STRINGS & CHARACTERS 1/2

```
// String Declaration
var emptyString = "" // == var emptyString = String()
emptyString.isEmpty // true

let constString = "Hello"

// Character Declaration
var character: Character = "p"
for character in "Hello" { // "H", "e", "l", "l", "o" }

// Declare a String as var in order to modify it
var growString = "a"; growString += "b"; growString.append("c")
countElements(growString) // 3
```

STRUCTURES

STRINGS & CHARACTERS 2/2

```
// String Interpolation
let multiplier = 2
let message = "\(multiplier) times 2.5 is \(Double(multiplier) * 2.5)"

// Print a message in the std output
println(message)

// Comparing Strings
let str1 = "Hello"; let str2 = "Hello";
str1 == str2 // true

str1.hasPrefix("Hel") // true
str2.hasSuffix("llo") // true
```

STRUCTURES

ARRAYS

```
// Array Declaration
var emptyArray = [Int]() // Array<Int>()
var emptyArray : [Int] = []
var array = [Int](count: 2, repeatedValue: 0)
var array = [25, 20, 16]

// Array Access: subscript out of bounds generate crash
array[1]    // 20
array[1000] // ❌ RUNTIME ERROR!!!

array.count   // 3
array.isEmpty // false

// Array Iteration
for value in array { /* use value */ }
for (index, value) in enumerate(array) { /* use index and value */ }
```

STRUCTURES

VARIABLE ARRAYS

```
var variableArray = ["A"]

variableArray = ["X"]                       // ["X"]
variableArray[0] = "A"                      // ["A"]
variableArray.append("B");                  // ["A", "B"]
variableArray += ["C", "D"]                 // ["A", "B", "C", "D"]

variableArray.insert("Z", atIndex: 4)  // ["A", "B", "C", "D", "Z"]
let a = variableArray.removeAtIndex(1) // ["A", "C", "D", "Z"]
variableArray[1...2] = ["M"]           // ["A", "M", "Z"]
let l = variableArray.removeLast()     // ["A", "M"]
```

STRUCTURES

CONSTANT ARRAYS

```
let constantArray = ["A"]

constantArray = ["X"]              // ❌ ERROR!!!
constantArray[0] = "Y"             // ❌ ERROR!!!
constantArray.append("B");         // ❌ ERROR!!!
constantArray.removeAtIndex(0) // ❌ ERROR!!!
```

STRUCTURES

DICTIONARIES

```
// Dictionary Declaration
var emptyDictionary = [String, Int]() // Dictionary<String, Int>()
var emptyDictionary : [String, Int] = [:] // key : value
var dictionary = ["First" : 25, "Second" : 20, "Third" : 16]

// Dictionary Access: subscript return Optional
let second: Int? = dictionary["Second"] // .Some(20)
let last: Int? = dictionary["Last"]     // nil

dictionary.count   // 3
dictionary.isEmpty // false

// Dictionary Iteration
for (key, value) in dictionary { /* use key and value */ }
dictionary.keys   // [String]
dictionary.values // [Int]
```

STRUCTURES

VARIABLE DICTIONARIES

```
var variableDictionary = ["First" : 25]

variableDictionary = ["Second" : 20]     // ["Second" : 20]
variableDictionary["Third"] = 16         // ["Second" : 20, "Third" : 16]

let oldKeyValue = variableDictionary.updateValue(18, forKey: "Second")
// oldValue => 20                        // ["Second" : 18, "Third" : 16]

// removes key
variableDictionary["Second"] = nil       // ["Third" : 16]
let removedValue = variableDictionary.removeValueForKey("Third")
// removedValue => 25                    // [:]
```

STRUCTURES

CONSTANT DICTIONARIES

```
let constantDictionary = ["First" : 25, "Second" : 20, "Third" : 16]

constantDictionary = ["Last" : 0] // ❌ ERROR!!!
constantDictionary["Third"] = 15  // ❌ ERROR!!!
constantDictionary["Forth"] = 21  // ❌ ERROR!!!
constantDictionary["First"] = nil // ❌ ERROR!!!
```

CLASSES

COPY REFERENCE ON ASSIGNMENT OR WHEN PASSED INTO FUNCTION

```
// Classes are reference types
class Person {
  var name: String?
  var age: Int = 0
}

// Class reference (not object) are copied when they are passed around
var matteo = Person()
let madbat = matteo
madbat.name = "MadBat"
matteo.name // "MadBat"

// Compare references using === and !==
matteo === madbat // true
```

PROPERTIES

STORED INSTANCE PROPERTIES

```
class Cell {
  let name: String          // constant stored property
  var position: CellPoint? // variable stored property
  var score: Int = 10       // variable stored property with default value

  lazy var spa = Spa()      // lazy stored property (only var)
  // lazy: a property whose initial value is not calculated until the first time it is used
}

struct CellPoint {
  var x = 0.0               // variable stored property with default value
  var y = 0.0               // variable stored property with default value
}

// Enumerations do not have instance stored properties
```

PROPERTIES

COMPUTED INSTANCE PROPERTIES 1/2

```
struct Rect {
  var origin = Point()
  var size = Size()
  var center: Point {
    get {
      let centerX = origin.x + (size.width / 2)
      let centerY = origin.y + (size.height / 2)
      return Point(x: centerX, y: centerY)
    }
    set(newCenter) {
      origin.x = newCenter.x - (size.width / 2)
      origin.y = newCenter.y - (size.height / 2)
    }
  }
}

// Also available for Enumerations and Classes
```

PROPERTIES

COMPUTED INSTANCE PROPERTIES 2/2

```
struct Rect {
  var origin = Point()
  var size = Size()
  var center: Point {
    get {
      let centerX = origin.x + (size.width / 2)
      let centerY = origin.y + (size.height / 2)
      return Point(x: centerX, y: centerY)
    }
    set { // shorthand 'newValue' equivalent
      origin.x = newValue.x - (size.width / 2)
      origin.y = newValue.y - (size.height / 2)
    }
  }
  // readonly can omit 'get' keyword
  var area: Double { return size.width * size.height }
}
```

PROPERTIES

TYPE PROPERTIES: SHARED AMONG INSTANCES

```
// Stored Type Properties: available only for Enumerations and Structures
struct Path {
  // You must always give stored type properties a default value
  static var maxLength = 1000
}

// Computed Type Properties: available for Enumerations, Structures and Classes
struct Path { static var maxLength: Int { return Int.max / 2 } /* Structures use 'static' keyword */ }
class Cell  { class  var maxNum:    Int { return Int.max / 5 } /* Classes    use 'class'  keyword */ }
}
```

PROPERTIES

PROPERTY ACCESS

```
let length = Path.maxLength // Type Property applies on a Type

let matteo = Person()
let name = matteo.name    // Instance Property applies on an Instance
```

PROPERTIES

PROPERTY OBSERVERS[4]

```
struct CellRoute {
  var duration: Int {
    willSet(newDurationValue) { /* ... */ }
    didSet(oldDurationValue)  { /* ... */ }
  }
  // Using default parameter names 'newValue' and 'oldValue'
  var cost: Double {
    willSet { /* use newValue */ }
    didSet  { /* use oldValue */ }
  }
}
```

[4] You can add property observers to any stored properties you define, apart from lazy stored properties.
You can also add property observers to any inherited property (whether stored or computed) by overriding the property within a subclass.

METHODS

AVAILABLE FOR ENUMERATIONS, STRUCTURES & CLASSES

```
class Cell {
  // type method
  class func dispatchAll() { /* ... */ } // use 'static' for Structures

  // instance methos
  func moveTo(destination: CellPoint, withSteps: Int) {
    // first argument treated as local name (require explicit #)
    // rest of arguments treated as external and local name (have implicit #)
    // to omit implicit external name requirement use an undescore _
  }
}

var cell = Cell()
cell.moveTo(center, withSteps: 10)
cell.moveTo(center, 10) // ❌ ERROR!!!

Cell.dispatchAll()
```

METHODS

MUTATING METHODS FOR ENUMERATIONS

```
// Methods that change internal state, must be marked as 'mutating'

enum Toggle {
  case On, Off
  mutating func toggle() {
    switch self {
    case On:
      self = Off
    case Off:
      self = On
    }
  }
}
```

METHODS

MUTATING METHODS FOR STRUCTURES

```
// Methods that change internal state, must be marked as 'mutating'

struct CellPoint {
  var x = 0.0, y = 0.0
  mutating func moveByX(deltaX: Double, y deltaY: Double) {
    x += deltaX
    y += deltaY
  }
  // equivalent assigning to self
  mutating func moveByX(deltaX: Double, y deltaY: Double) {
    self = Point(x: x + deltaX, y: y + deltaY)
  }
}

var point = CellPoint(x: 3.0, y: 4.0)
point.moveByX(1.0, y: 2.0)
```

SUBSCRIPTS

ACCESS INTERNAL MEMBERS BY [ ] SYNTAX

```
class CellContainer {
  var cells : [Cell] = []

  // readwrite
  subscript(index1:Int, index2: Double) -> Cell {
    get { /* return internal member at specified indexes */  }
    set(newValue) { /* save internal member at specified indexes */   }
  }

  // readonly: no need the 'get' keyword
  subscript(index1: Int, index2: Double) -> Cell { /* return internal member at specified indexes */  }
    // getter code
  }
}

var container = CellContainer()
var cell = container[0][1.0]
```

INHERITANCE

AVAILABLE ONLY FOR CLASSES

```
// class Subclass: Superclass
class Car: Vehicle {

  // override property
  override var formattedName: String {
    return "[car] " + name
  }

  // override method
  override func order() {
    // can call super.order()
  }
}
```

INHERITANCE

FINAL CLASSES, PROPERTIES & METHODS

```
// Final class cannot be subclassed
final class Car : Vehicle { /* ... */ }
class Jeep : Car // ❌ ERROR!!!

class Bicycle : Vehicle  {
  // final property cannot be overridden
  final var chainLength: Double

  // final method cannot be overridden
  final func pedal() { /* ... */ }
}
```

INITIALIZATION

SETTING INITIAL VALUES FOR STORED PROPERTIES

```
class Car {
  // Classes and Structures must set all of their stored properties
  // to an appropriate initial value by the time an instance is created

  let model: String // stored properties must be initialized in the init
  let wheels = 4    // stored properties with default property value are initialized before the init

  //Initializers are declared with 'init' keyword
  init() {
    model = "Supercar"
  }
}

// Initializers are called to create a new instance of a particular type with Type() syntax
let car = Car()
car.model // "Supercar"
```

INITIALIZATION

INITIALIZATION PARAMETERS

```
class Car {
  let model: String // You can set the value of a constant property at any point during initialization

  // An automatic external name (same as the local name) is provided for every parameter in an initializer
  init(model: String) {  // equivalent to: init(model model: String)
    self.model = model // use self. to distinguish properties from parameters
  }

  // alternatively
  init (fromModel model: String) { /* ... */} // override the automatic external
  init (_         model: String) { /* ... */} // omit automatic external name using an underscore _
}

// Initializers are called with external parameters
let car = Car(model: "Golf")
car.model // "Golf"
```

INITIALIZATION

FAILABLE INITIALIZERS FOR ENUMERATIONS, STRUCTURES & CLASSES

```
class Car {
  // For classes only: failable initializer can trigger a failure
  // only after all stored properties have been set
  let wheels: Int! // set to nil by default

  init?(wheels: Int) {
    if weels > 4 { return nil }
    self.wheels = wheels
  }
}
var car: Car? = Car(wheels: 10) // nil

if let realCar = Car(wheels: 4) {
  println("The car has \(car.wheels) wheels")
}
```

INITIALIZATION

DEFAULT INITIALIZER

```
// Default Initializer are provided for class with no superclass or structure
// if these conditions are all valid:
// 1) all properties have default values
// 2) the type does not provide at least one initializer itself

class Car {
  var model = "Supercar"
  let wheels = 4
}

let car = Car()
car.model // "Supercar"
```

INITIALIZATION

MEMBERWISE INITIALIZERS FOR STRUCTURES

```
// Structure types automatically receive a memberwise initializer
// if they do not define any of their own custom initializers

struct Wheel {
    var radius = 0.0
    var thickness = 0.0
}

let Wheel = Wheel(radius: 10.0, thickness: 1.0)
```

INITIALIZATION

DESIGNATED & CONVENIENCE INITIALIZERS FOR CLASSES

```
class Car {
  var model: String

  init (model: String) { /* ... */} // Designated  initializers
                                    // are the primary initializers for a class

  convenience init () { /* ... */}  // Convenience initializers
                                    // are secondary, supporting initializers for a class
}

var golf = Car(model: "Golf")
var supercar = Car()
```

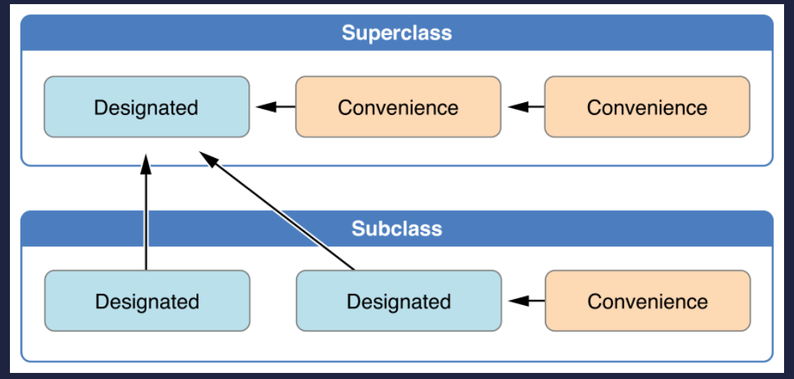
INITIALIZATION

INITIALIZER DELEGATION FOR CLASSES

```
// Rule 1: A designated initializer must call a designated initializer from its immediate superclass.
// Rule 2: A convenience initializer must call another initializer from the same class.
// Rule 3: A convenience initializer must ultimately call a designated initializer.

class Vehicle {
  var wheels: Int
  init (wheels: Int) {  self.wheels = wheels  }
}

class Car: Vehicle {
  var model: String
  init (model: String) { super.init(wheels: 4); self.model = model }
  convenience init ()   { self.init(model: "Supercar") }
}
```

INITIALIZATION

INITIALIZER DELEGATION FOR CLASSES



Designated initializers must always delegate up.
Convenience initializers must always delegate across.”

INITIALIZATION

TWO-PHASE INITIALIZATION FOR CLASSES

```
init() {

  // The First Phase
  // each stored property is assigned an initial value by the class that introduced it.

  // The Second Phase
  // each class is given the opportunity to customize its stored properties further
  // before the new instance is considered ready for use.

}
```

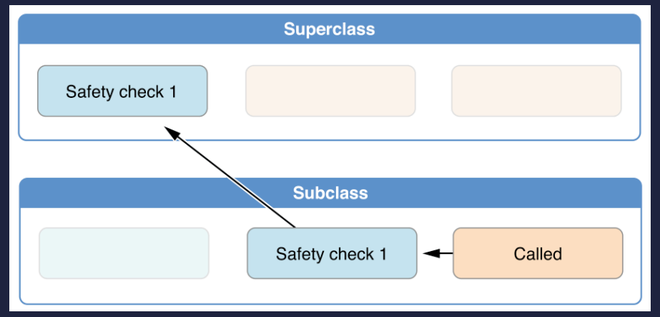
INITIALIZATION

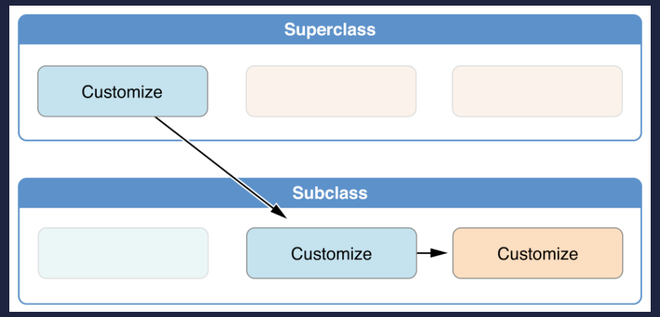
SAFETY CHECKS FOR CLASSES

```
init() {

  // Safety check 1
  // A designated initializer must ensure that all of the properties introduced by its class
  // are initialized before it delegates up to a superclass initializer.

  // Safety check 2
  // A designated initializer must delegate up to a superclass initializer
  // before assigning a value to an inherited property.

  // Safety check 3
  // A convenience initializer must delegate to another initializer
  // before assigning a value to any property (including properties defined by the same class).

  // Safety check 4
  // An initializer cannot call any instance methods, read the values of any instance properties,
  // or refer to self as a value until after the first phase of initialization is complete.”

}
```

INITIALIZATION

PHASE 1: BOTTOM UP & PHASE 2: TOP DOWN





INITIALIZATION

INITIALIZER INHERITANCE & OVERRIDING FOR CLASSES

```
class Vehicle {
  var wheels = 0
}
let vehicle = Vehicle() // vehicle.wheels == 0

// Subclasses do not inherit their superclass initializers by default
class Car: Vehicle {
  override init () { // explicit override
    super.init();
    wheels = 4
  }
}
let car = Car() // car.wheels == 4
```

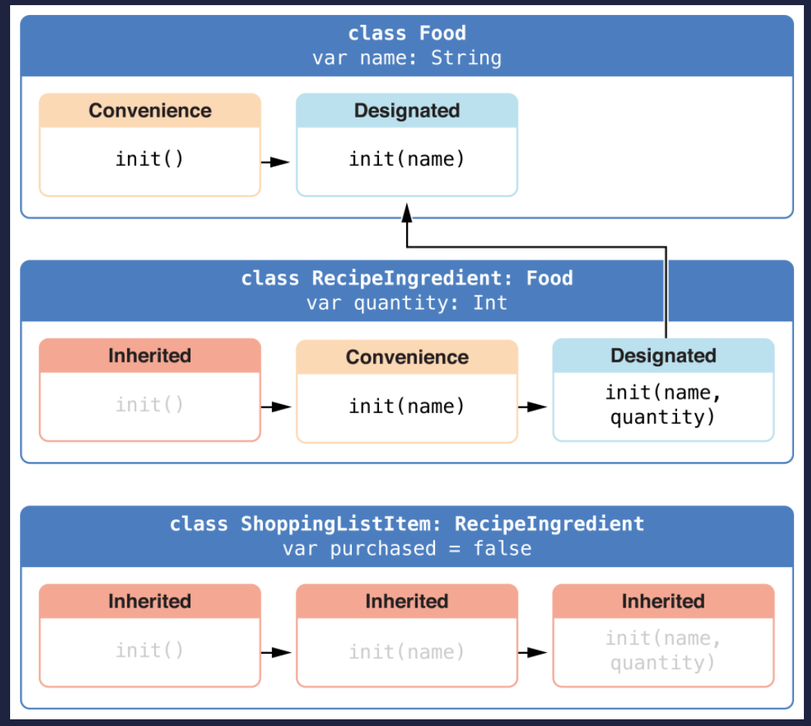
INITIALIZATION

AUTOMATIC INITIALIZER INHERITANCE FOR CLASSES

```
class Car: Vehicle {
  // Assuming that you provide default values for any new properties you introduce in a subclass
  var model = "Supercar"

  // Rule 1
  // If your subclass doesn’t define any designated initializers,
  // then it automatically inherits all of its superclass designated initializers.

  // Rule 2
  // If your subclass provides an implementation of all of its superclass designated initializers
  // (either by inheriting them as per rule 1, or by providing a custom implementation as part of its definition)
  // then it automatically inherits all of the superclass convenience initializers.
}
let car = Car() // car.model == "Supercar"
```

INITIALIZATION

AUTOMATIC INITIALIZER INHERITANCE FOR CLASSES



INITIALIZATION

REQUIRED INITIALIZERS FOR CLASSES

```
class Vehicle {
  required init() { /* ... */ }
}

class Car: Vehicle {
  // You do not write the override modifier
  // when overriding a required designated initializer
  required init () {
    super.init();
  }
}
```

INITIALIZATION

DEINITIALIZER FOR CLASSES

```
class Car {
  // A deinitializer is called immediately
  // before a class instance is deallocated
  deinit {
    /* free any external resources */
  }
}

var car: Car? = Car()
car = nil // as a side effect call deinit
```

AUTOMATIC REFERENCE COUNTING

ARC IN ACTION

```
// ARC automatically frees up the memory used by class instances
// when those instances are no longer needed (reference count == 0)

class Car { /* ... */ }

var reference1: Car?
var reference2: Car?
var reference3: Car?

reference1 = Car()      // reference count == 1
reference2 = reference1 // reference count == 2
reference3 = reference1 // reference count == 3

reference1 = nil        // reference count == 2
reference2 = nil        // reference count == 1
reference3 = nil        // reference count == 0

// no more reference to Car object => ARC dealloc the object
```

AUTOMATIC REFERENCE COUNTING

REFERENCE CYCLES BETWEEN CLASS INSTANCES

```
class Car { var tenant: Person }
class Person { var car: Car }

car.person = person
person.car = car

// weak and unowned resolve strong reference cycles between Class Instances
```

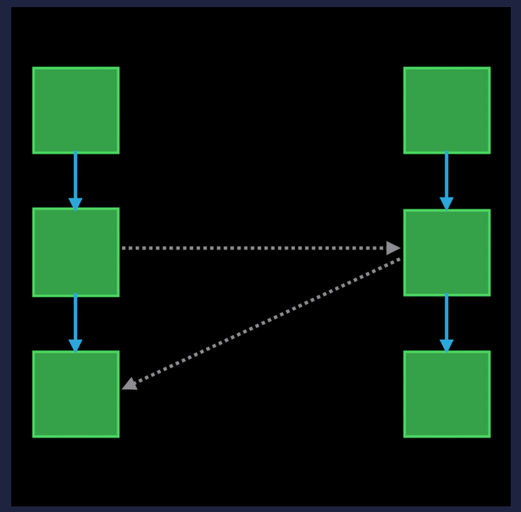
AUTOMATIC REFERENCE COUNTING

WEAK REFERENCES

```
// Use a weak reference whenever it is valid for that reference to become nil at some point during its lifetime
class Person {
    let name: String
    init(name: String) { self.name = name }
    var car: Car?
    deinit { println("\(name) is being deinitialized") }
}
class Car {
    let wheels: Int
    init(wheels: Int) { self.wheels = wheels }
    weak var tenant: Person?
    deinit { println("Car #\(wheels) is being deinitialized") }
}
```

AUTOMATIC REFERENCE COUNTING

WEAK REFERENCES



Use weak references among objects with independent lifetimes

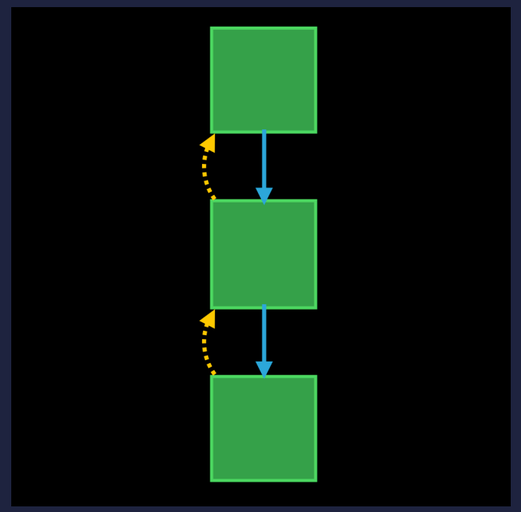
AUTOMATIC REFERENCE COUNTING

UNOWNED REFERENCES

```
// Use an unowned reference when you know that the reference will never be nil once it has been set during initialization.
class Country {
    let name: String
    let capitalCity: City!
    init(name: String, capitalName: String) {
        self.name = name
        self.capitalCity = City(name: capitalName, country: self)
    }
}
class City {
    let name: String
    unowned let country: Country
    init(name: String, country: Country) {
        self.name = name
        self.country = country
    }
}
```

AUTOMATIC REFERENCE COUNTING

UNOWNED REFERENCES



Use unowned references from owned objects with the same lifetime

AUTOMATIC REFERENCE COUNTING

REFERENCE CYCLES FOR CLOSURES

```
// Capture lists capture the value at the time of closure's definition
var i = 1
var returnWithCaptureList    = { [i] in i } // captures value of i
var returnWithoutCaptureList = {        i } // do not capture value of i

i = 2
returnWithCaptureList()    // 1
returnWithoutCaptureList() // 2

class MyClass {
  // Use capture lists to create weak or unowned references
  lazy var someClosure1: () -> String = { [unowned self] () -> String in "closure body" }
  // can infer types of closure parameters
  lazy var someClosure2: () -> String = { [unowned self]              in "closure body" }
}
```

TYPE CASTING

TYPE CHECKING & TYPE DOWNCASTING

```
class Car: Vehicle { /* ... */ }
class Bicycle: Vehicle { /* ... */ }
let vehicles = [Car(), Bicycle()]

// Type checking
let firstVehicle = vehicles[0]
firstVehicle is Car // true
firstVehicle is Bicycle // false

// Type downcasting
let firstCar     = firstVehicle as? Car     // firstCar: Car?
let firstCar     = firstVehicle as  Car     // firstCar: Car
let firstBicycle = firstVehicle as? Bicycle // nil: Car?
let firstBicycle = firstVehicle as  Bicycle // ❌ RUNTIME ERROR!!!
```

TYPE CASTING

SWITCH TYPE CASTING

```
for thing in things {
  switch thing {
  case 0 as Int:
    /* ... */ // thing is 0: Int
  case 0 as Double:
    /* ... */ // thing is 0.0: Double
  case let someInt as Int:
    /* ... */
  case is Double:
    /* ... */
  default:
    /* ... */
  }
}
```

NESTED TYPES

AVAILABLE FOR ENUMERATIONS, STRUCTURES & CLASSES

```
struct Song {  // struct ⎫
  class Note {            // nested class ⎫
    enum Pitch: Int {                     // nested enumeration
      case A = 1, B, C, D, E, F, G
    }
    var pitch: Pitch = .C
    var length: Double = 0.0
  }
  var notes: [Note]
}

Song.Note.Pitch.C.rawValue // 3
```

OPTIONAL CHAINING

QUERYING & CALLING MEMBERS ON AN OPTIONAL THAT MIGHT BE NIL

```
class Car {
  var model: String

  init(model: String) { self.model = model }
  convenience init()  { self.init(model: "Supercar") }

  func jump() -> String? { if model == "Supercar" { return "🚀⚡" } else { return nil } }
}

// Return type of chaining is always optional
var car1: Car? = nil;                car1?.model // nil: String?
var car2: Car? = Car();              car2?.model // "Supercar": String?
var car3: Car? = Car(model: "Golf"); car3?.model // "Golf": String?

// Chain on optional return value
car1?.jump()?.hasPrefix("🚀") // type Bool?
```

ACCESS CONTROL

MODULES & FILES

A module is a single unit of code distribution
(a framework or an application)

A source file is a single Swift source code file within a module
(can contain definitions for multiple types, functions, and so on)

ACCESS CONTROL

PUBLIC, INTERNAL, PRIVATE

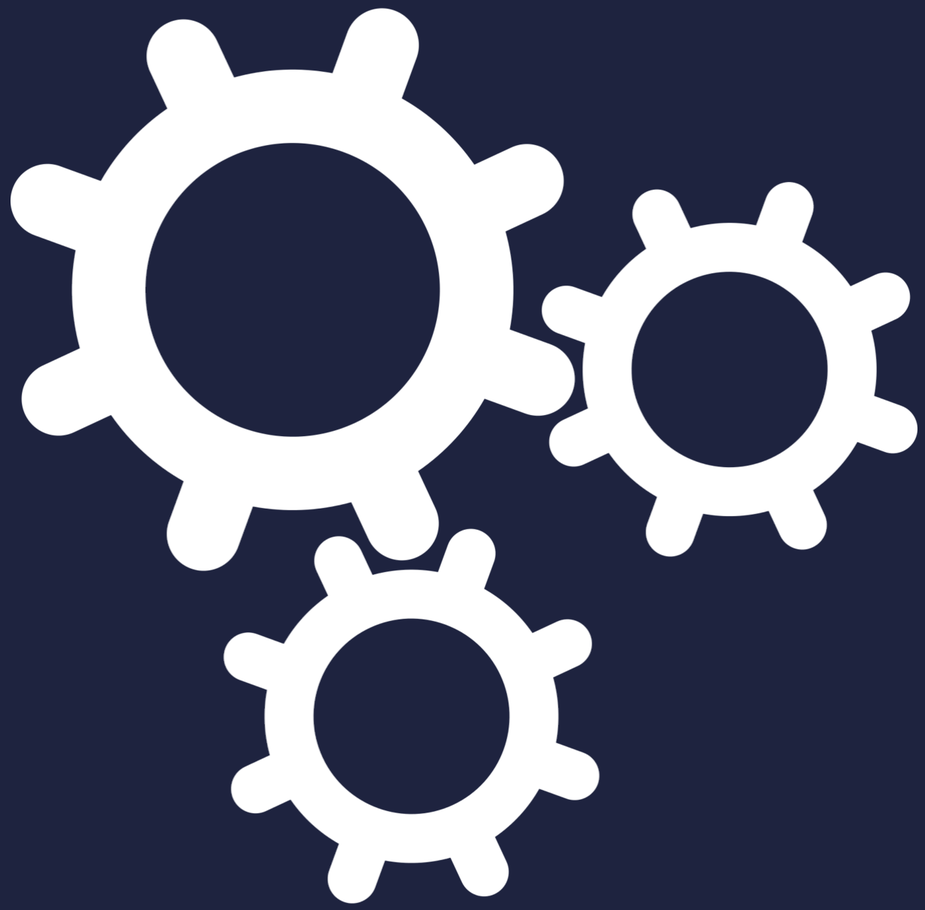
```
public
// enables entities to be used within any source file from their defining module,
// and also in a source file from another module that imports the defining module

internal
// enables entities to be used within any source file from their defining module,
// but not in any source file outside of that module

private
// restricts the use of an entity to its own defining source file.
```

ACCESS CONTROL

MEMBERS ACCESS MODIFIERS RESTRICT TYPE ACCESS LEVEL

```
public class SomePublicClass {
  public var somePublicProperty
  var someInternalProperty // default is internal
  private func somePrivateMethod() {}
}

// internal class
class SomeInternalClass {
  var someInternalProperty // default is internal
  private func somePrivateMethod() {}
}

private class SomePrivateClass {
  var somePrivateProperty // everything is private!
  func somePrivateMethod() {}
}
```

EXTENSIONS, PROTOCOLS & GENERICS



EXTENSIONS

AVAILABLE FOR ENUMERATIONS, STRUCTURES & CLASSES

```
extension SomeType: SomeProtocol {                  // Extensions can make an existing type conform to a protocol

  var stored = 1 // ❌ ERROR!!!                     // Extensions CANNOT add stored properties!

  var computed: String { /* ... */ }                // Extensions can add computed properties (type and instance)

  func method() { /* ... */ }                       // Extensions can add methods (type and instance)

  subscript(i: Int) -> String { /* ... */ }         // Extensions can add subscripts

  init(parameter: Type) { /* ... */ }               // Extensions can add initializers

  enum SomeEnum { /* ... */ }                       // Extensions can add nested types
}
```

EXTENSIONS

EXTENSIONS CAN EXTEND ALSO SWIFT LIBRARY TYPES

```
extension Int {
    func times(task: () -> ()) {
        for i in 0 ..< self {
            task()
        }
    }
}

3.times({ println("Developer! ") }) // Developer! Developer! Developer!

// see also: http://en.wikipedia.org/wiki/Developer!_Developer!_Developer! 😱
```

PROTOCOLS

AVAILABLE FOR ENUMERATIONS, STRUCTURES & CLASSES

```
protocol SomeProtocol {                          // Protocols define a blueprint of requirements that suit a functionality

  var instanceProperty: Type { get set }         // Protocols can require instance properties (stored or computed)

  class var typeProperty: Type { get set } // Protocols can require type properties (stored or computed)
                                           // └ Always use 'class', even if later used by value type as 'static'

  func someMethod()                              // Protocols can require instance methods

  mutating func someMutatingMethod()             // Protocols can require instance mutanting methods

  class func someTypeMethod()                    // Protocols can require type methods

  init()                                         // Protocols can require initializers
}
```

PROTOCOLS

CONFORMING TO A PROTOCOL & PROTOCOL USED AS A TYPE

```
// Conforming to a protocol
class SomeClass: SomeProtocol {

  // init requirements have 'required' modifier
  required init() { /* ... */ } // ('required' not needed if class is final)

  // Other requirements do not have 'required' modifier
  var someReadWriteProperty: Type
  ...
}

// Protocol used as type
let thing:   SomeProtocol  = SomeClass()
let things: [SomeProtocol] = [SomeClass(), SomeClass()]
```

PROTOCOLS

DELEGATION

```
protocol GameDelegate {
  func didPlay(game: Game)
}

class Game {
  var delegate: GameDelegate? // Optional delegate

  func play() {
    /* play */
    delegate?.didPlay(self)   // use Optional Chaining
  }
}
```

PROTOCOLS

PROTOCOL INHERITANCE

```
// A protocol can inherit one or more other protocols
// and can add further requirements on top of the requirements it inherits

protocol BaseProtocol    { func foo() }
protocol AnotherProtocol { func bar() }

// InheritingProtocol requires functions: foo, bar and baz
protocol InheritingProtocol: BaseProtocol, AnotherProtocol { func baz () }

class SomeClass: InheritingProtocol {
  func foo() { /* ... */ }
  func bar() { /* ... */ }
  func baz() { /* ... */ }
}
```

PROTOCOLS

CLASS-ONLY PROTOCOLS

```
// can only be used by classes

// prefix protocols conformance list with 'class' keyword
protocol SomeClassProtocol: class, AnotherProtocol { /* ... */ }

class  SomeClass:  SomeClassProtocol { /* ... */ } // OK

struct SomeStruct: SomeClassProtocol { /* ... */ } // ❌ ERROR!!!
```

PROTOCOLS

CHECKING PROTOCOL CONFORMANCE

```
// Need @objc attribute because under the hood use Objective-C runtime to check protocol conformance
@objc protocol SomeProtocol { /* ... */ }

class SomeClass: SomeSuperclass, SomeProtocol, AnotherProtocol { /* ... */ }
var instance = SomeClass()

instance is  SomeProtocol    // true
instance is  UnknownProtocol // false

instance as? SomeProtocol    // instance: SomeProtocol?
instance as? UnknownProtocol // nil:      SomeProtocol?

instance as  SomeProtocol    // instance: SomeProtocol
instance as  UnknownProtocol // ❌ RUNTIME ERROR!!!
```

PROTOCOLS

OPTIONAL PROTOCOL REQUIREMENTS

```
// Need @objc attribute because under the hood use Objective-C runtime to implement optional protocol
@objc protocol GameDelegate {
  optional func didPlay(game: Game) // Optional method requirement
}

class Game {
  var delegate: GameDelegate?  // Optional delegate

  func play() {
    /* play */
    delegate?.didPlay?(self)   // use Optional Chaining
  }
}
```

PROTOCOLS

ANY & ANYOBJECT

```
// Any: any instance of any type
let instance: Any = { (x: Int) in x }
let closure: Int -> Int = instance as Int -> Int

// AnyObject: any object of a class type
let instance: AnyObject = Car()
let car:      Car       = x as Car

let cars: [AnyObject] = [Car(), Car(), Car()] // see NSArray
for car in cars as [Car] { /* use car: Car */ }
```

PROTOCOLS

EQUATABLE & HASHABLE

```
// Instances of conforming types can be compared
// for value equality using operators '==' and '!='
protocol Equatable {
    func ==(lhs: Self, rhs: Self) -> Bool
}

// Instances of conforming types provide an integer 'hashValue'
// and can be used as Dictionary keys
protocol Hashable : Equatable {
    var hashValue: Int { get }
}
```

GENERICS

FUNCTIONS TYPES CAN BE GENERIC

- Generics avoid duplication and expresses its intent in the abstract
- Much of the Swift Library is built with generics (Array, Dictionary)
- Compiler can optimize by creating specific versions for some cases
  - Type information is available at runtime

GENERICS

PROBLEM THAT GENERICS SOLVE

```
// Specific Functions
func swapTwoStrings(inout a: String, inout b: String) { let temporaryA = a; a = b; b = temporaryA }
func swapTwoDoubles(inout a: Double, inout b: Double) { let temporaryA = a; a = b; b = temporaryA }

// Generic Function
func swapTwoValues<T>(inout a: T, inout b: T)         { let temporaryA = a; a = b; b = temporaryA }

var someInt = 1, anotherInt = 2
swapTwoValues(&someInt, &anotherInt)                   // T == Int
// someInt == 2, anotherInt == 1

var someString = "hello", anotherString = "world"
swapTwoValues(&someString, &anotherString)             // T == String
// someString == "world", anotherString == "hello"
```

GENERICS

GENERIC FUNCTIONS

```
func append<T>(inout array: [T], item: T) {
  array.append(item)
}
var someArray = [1]
append(&someArray, 2)

// T: SomeClass    -> T must be subclass of SomeClass
// T: SomeProtocol -> T must conform to SomeProtocol

func isEqual<T: Equatable>(a: T, b: T) -> Bool {
  return a == b
}
```

GENERICS

GENERIC TYPES

```
class Queue<T> {
  var items = [T]()

  func enqueue(item: T) { items.append(item) }
  func dequeue() -> T   { return items.removeAtIndex(0) }
}

extension Queue { // don't specifiy T in extensions

  func peekNext() -> T? { return items.isEmpty ? nil : items[0] }
}
```

GENERICS

ASSOCIATED TYPES

```
protocol SomeProtocol {
  typealias ItemType                    // Define the Associated Type
  func operate(item: ItemType)          // a placeholder name to a type used in a protocol
}

class SomeClass: SomeProtocol {
  typealias ItemType = Int              // Specify the actual Associated Type
  func operate(item: Int) { /* ... */ }
}

class SomeClass: SomeProtocol {
  // typealias ItemType = Int           // Infer the actual Associated Type
  func operate(item: Int) { /* ... */ }
}
```

GENERICS

WHERE CLAUSES ON TYPE CONSTRAINTS

```
protocol Container {
    typealias ItemType
}

func allItemsMatch<
  C1: Container, C2: Container
  where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>
    (someContainer: C1, anotherContainer: C2) -> Bool {

    if someContainer.count != anotherContainer.count { return false }

    for i in 0..<someContainer.count { if someContainer[i] != anotherContainer[i] { return false } }

    return true // all items match, so return true
  }
}
```

TOOLS

COMPILER

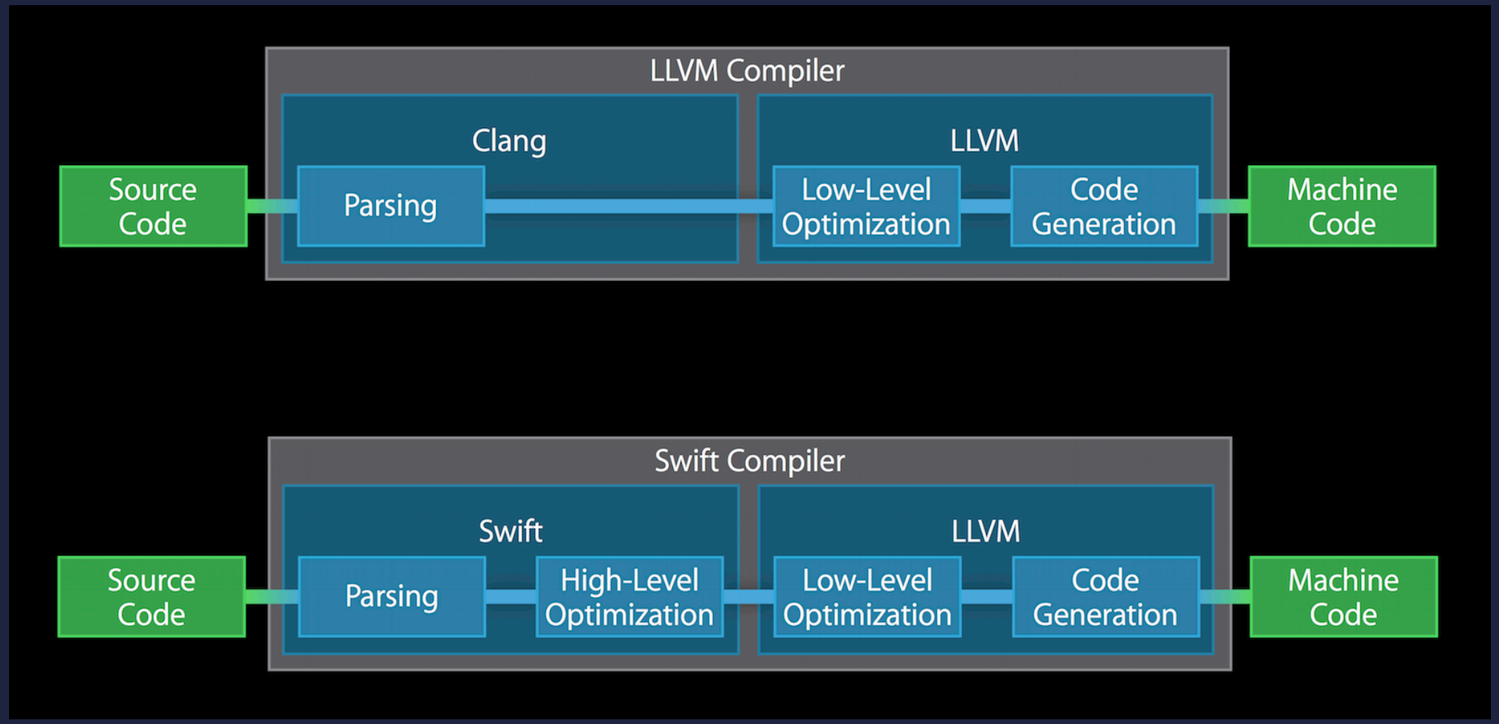
LLVM COMPILER INFRASTRUCTURE

- Clang knows absolutely nothing about Swift
- Swift compiler talks to clang through XPC[5]

[5] XPC = OS X interprocess communication technology

COMPILER

COMPILER ARCHITECTURE

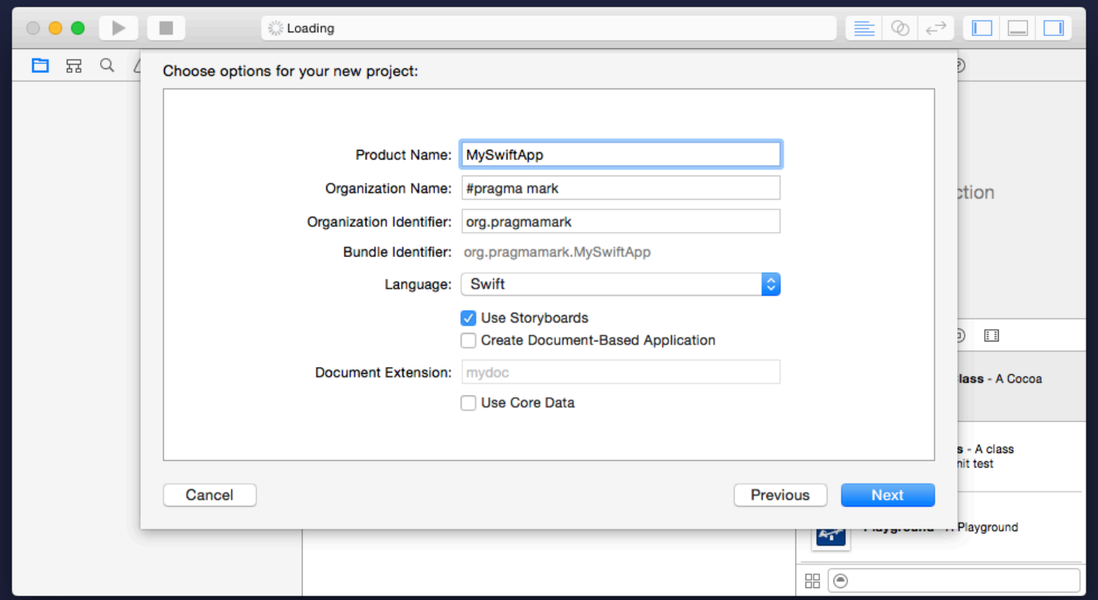


RUNTIME

- Under the hood Swift objects are actually Objective-C objects
  with implicit root class ‘SwiftObject’
- Just like C++, Swift methods are listed in a vtable
  unless marked as @objc or :NSObject*
- Swift's runtime and libraries are not (yet) included in the OS
  so must be included in each app

XCODE

THE INTEGRATED DEVELOPMENT ENVIRONMENT



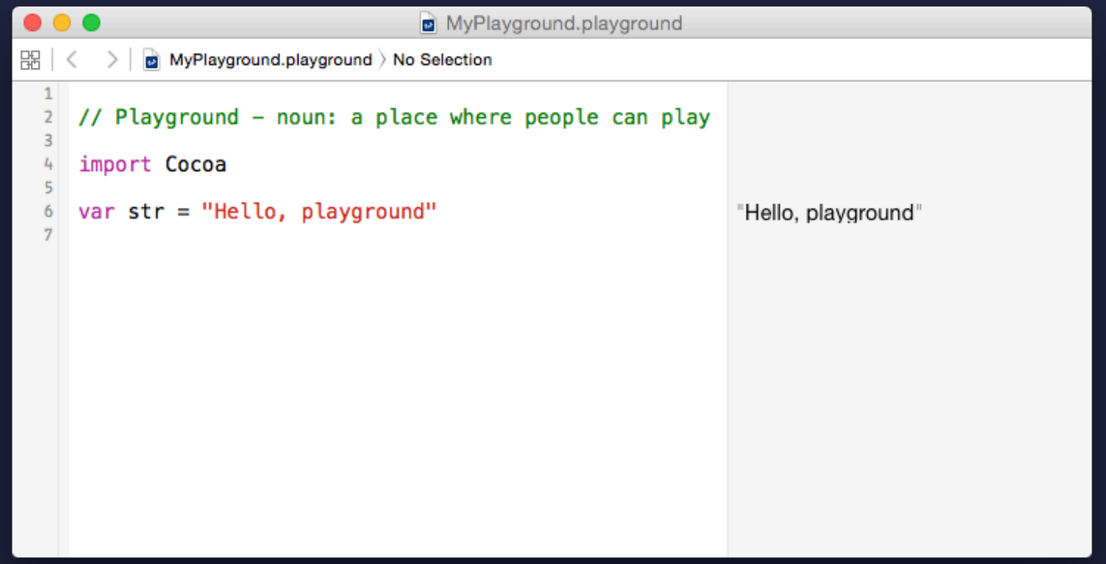
REPL

THE READ-EVAL-PRINT LOOP

```
xcrun swift # launches REPL
xcrun -i 'file.swift' # executes script
```

PLAYGROUND

THE INTERACTIVE ENVIRONMENT



PLAYGROUND

The Xcode Playgrounds feature and REPL were a personal passion of mine, to make programming more interactive and approachable. The Xcode and LLDB teams have done a phenomenal job turning crazy ideas into something truly great. Playgrounds were heavily influenced by Bret Victor's ideas, by Light Table and by many other interactive systems.

– Chris Lattner

PRACTICE

INTEROPERABILITY

- Bridging: from Objective-C to Swift & from Swift to Objective-C
  - Call CoreFoundation (C language) types directly
  - C++ is not allowed: should be wrapped in Objective-C

INTEROPERABILITY

FROM OBJECTIVE-C TO SWIFT

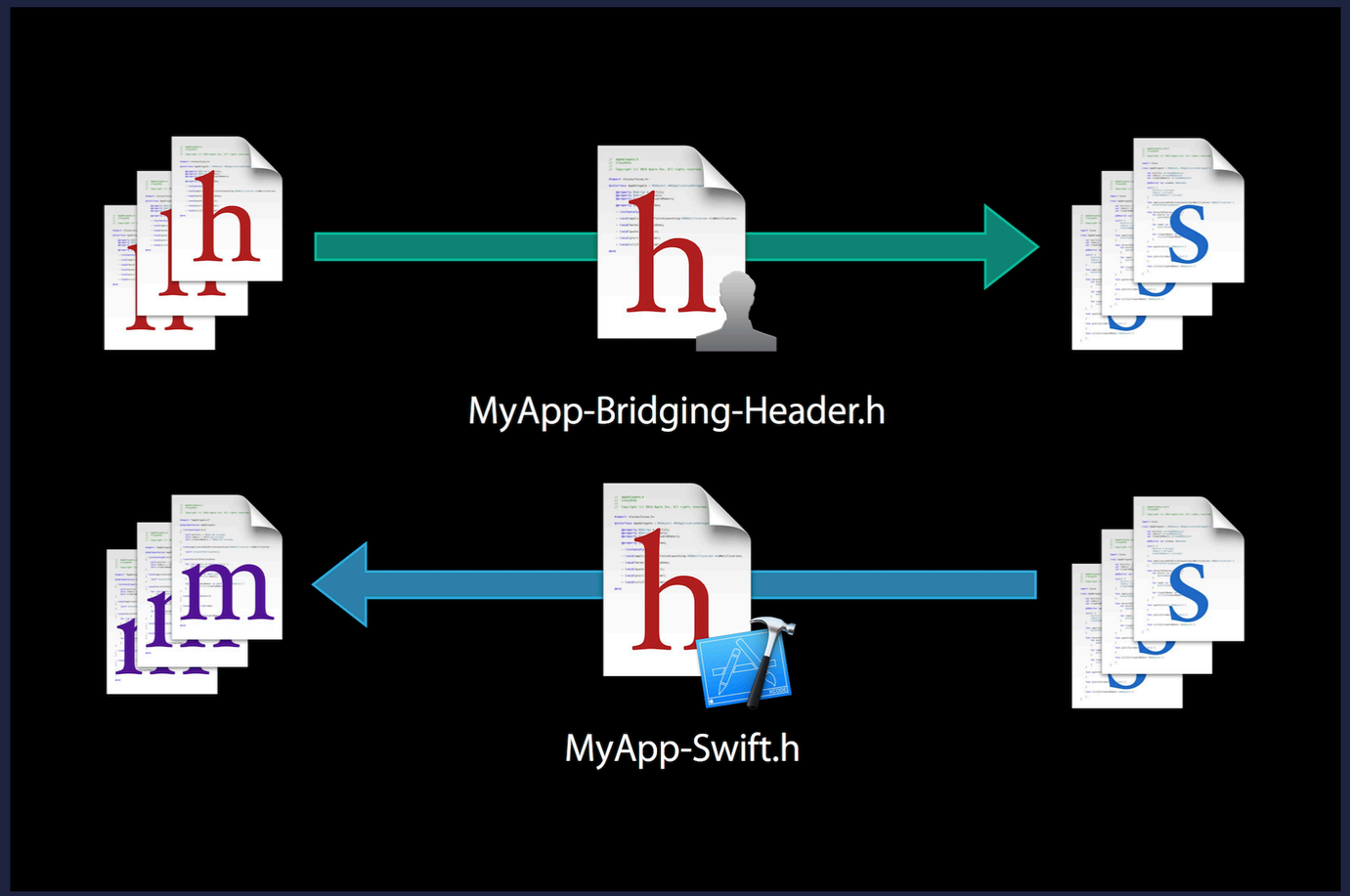
- All Objective-C code available in Swift
- All standard frameworks and all custom libraries available in Swift
- Use Interfaces headers in MyApp-Bridging-Header.h

INTEROPERABILITY

FROM SWIFT TO OBJECTIVE-C

- Not all Swift code available in Objective-C (no Generics, no...)
  - Subclassing Swift classes not allowed in Objective-C
- Mark Swift Classes as Objective-C compatible with `@objc`
- Use automatic generated Swift module header in MyApp-Swift.h

INTEROPERABILITY



LET'S PLAY WITH SWIFT

- Swift in playground
- Swift distinctive features

LET'S SEE IT IN ACTION 👉

FINAL THOUGHTS

SWIFT VS ... ?

- Swift vs Scala
- Swift vs Rust
- Swift vs C#

~~Swift is Objective-C without the C~~

OPEN SOURCE SWIFT?

1. Open source Swift compiler
2. Open source Swift runtime
3. Open source Swift standard library

Objective-C is 30 years old and they still haven't done #3

OPEN SOURCE SWIFT?

Guys, feel free to make up your own dragons if you want, but your speculation is just that: speculation. We literally have not even discussed this yet, because we have a ton of work to do [...] You can imagine that many of us want it to be open source and part of llvm, but the discussion hasn't happened yet, and won't for some time.

– Chris Lattner @ llvmdev

WHAT IS SWIFT MISSING?

[SWIFT 1.2 IN XCODE 6.3B]

- Compiler attributes and Preprocessor
- Class Variables, Exceptions, KVO, KVC and Reflection[6]

[6] Though it's not documented in the Swift Standard Library – and is subject to change – Swift has a reflection API:

```
let mirror = reflect(instance) // see Reflectable Protocol
```

SWIFT IN FLUX

https://github.com/ksm/SwiftInFlux

This document is an attempt to gather the Swift features that are still in flux and likely to change.

WHEN TO USE SWIFT?

- New apps (iOS 7, iOS 8, OS X 10.10)
- Personal projects
- Scripts

SWIFT IN PRODUCTION?

- Companies are doing it
- Freelances are doing it
- Many of us are doing it
- But be careful if you do it!

A few apps that were built using Swift @ apple.com

REFERENCES

RESOURCES

- Official Swift website
- Apple's Swift Blog
- Chris Lattner
- The Swift programming language
- WWDC Videos

PRESENTATIONS 1/2

- Swift by Denis Lebedev
- Swift 101 by Axel Rivera
- I Love Swift by Konstantin Koval
- Swiftroduction by Łukasz Kuczborski
- Functional Swift by Chris Eidhof

PRESENTATIONS 2/2

- Solving Problems the Swift Way by Ash Furrow
- Swift Enums, Pattern Matching, and Generics by Austin Zheng
- Swift and Objective-C: Best Friends Forever? by Jonathan Blocksom
- Swift for JavaScript Developers by JP Simard
- Swift for Rubyists by JP Simard

PROJECTS

- Github Trending Repositories
- Swift-Playgrounds
- SwiftInFlux
- Carthage
- Alamofire

BLOGS 1/2

- NSHipster
- Russ Bishop
- Erica Sadun
- Natasha The Robot
- Airspeed Velocity

BLOGS 2/2

- Rob Napier
- David Owens
- So So Swift
- We ❤ Swift
- Swift SubReddit

ARTICLES 1/2

- Exploring Swift Memory Layout by Mike Ash
- Swift Language Highlights by Matt Galloway
- Running Swift script from the command line
- Understanding Optionals in Swift
- Segues in Swift

ARTICLES 2/2

- Design Patterns in Swift
- Custom Operations in Swift
- Instance Methods are Curried Functions in Swift
- Swift: Is it ready for prime time?
- Why Rubyist Will Love Swift

TUTORIALS

- iOS8 Day by Day by Sam Davies
- An Absolute Beginner's Guide to Swift by Amit Bijlani
- Swift Tutorial: A Quick Start by Ray Wenderlich
- Learn Swift Build Your First iOS Game by Stan Idesis
- Swift Essential Training by Simon Allardice

CODING STYLE GUIDES

- Swift Style Guide by Github
- Swift Style Guide by Ray Wenderlich

MISCELLANEOUS

- Phoenix: Open Source Swift
- Swift Developer Weekly
- This Week in Swift
- Balloons Playground
- Swift Toolbox

THANKS

see you @ https://fb.com/groups/pragmamark

Matteo Battaglio
twitter: @m4dbat

QUESTIONS?