Swift

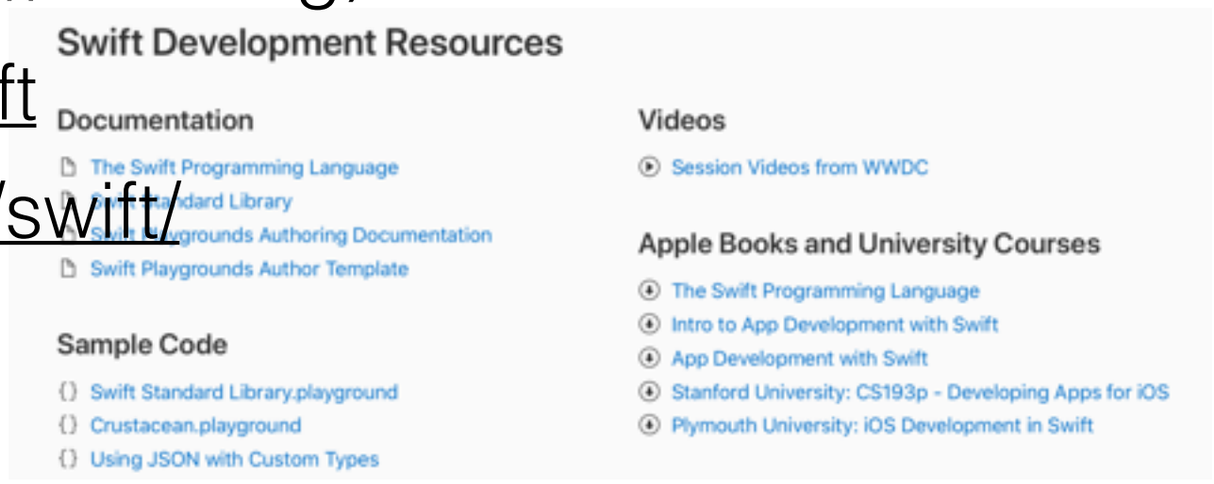
谷方明
fmgu2002@sina.com

Swift 简介

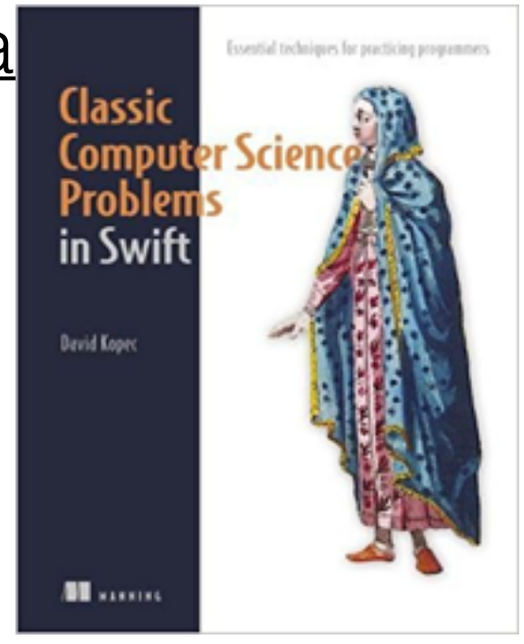
- Swift是Apple公司2014 发布一种新的编程语言
  - 最初用于编写 iOS 和 macOS 等应用，兼容Object-C.
  - 现已成为一门独立的语言，支持跨平台使用（ubuntu, windows)
- Swift语言的主要设计目标
  - 简洁高效
  - 既满足工业标准又充满表现力和趣味的脚本语言
- 简史
  - 2015开源，2016年3.0，2017年4.0，2019年5.0，**最新5.3**

Swift 资源

- Swift官网
  - https://docs.swift.org（https://swift.org）
  - https://github.com/apple/swift
  - https://developer.apple.com/swift/



- 众多资源
  - https://github.com/SwiftGGTeam/the-swift-programming-language-in-chinese
  - https://github.com/davecom/ClassicComputerScienceProblemsInSwift

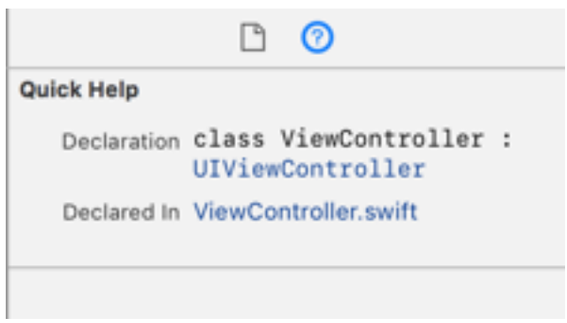


体验 Swift

- Mac：最佳选择 Xcode，做APP
  - playground（最快方式；iOS、Mac等Source）
  - Swift REPL （Read-Eval-Print-Loop；playground的命令行交互版本）
    - 启动：命令行下，输入命令xcrun swift
    - 测试：>提示符，输入swift代码
    - 退出：输入：，调用命令模式；输入q，退出；
- Windows
  - VS Code + Swift for Windows
- Ubuntu等
  - https://swift.org/downlaod
- iPad的APP：Swift Playgrounds （入门体验小游戏；iOS 10l ater）

Xcode中使用帮助

- 方法一： Option键
  - 光标在待查询对象上，按option，单击(概要)
- 方法二：Command键
  - 光标在待查询对象上，按option，单击(Jump to Definition)
- 方法二：Quick help
  - 光标在待查询对象上，使用Inspector



- 方法三：帮助文档
  - Help -> Developer Documentation

Swift语言基础

- Objective-C： 是C语言的一个严格的超集。在Swift推出之前，Mac和iOS的应用主要是用Objective-C 编写。

- Swift兼容Objective-C。在一定程度上，被视为“没有C的Objective-C”。（在C语言的基础上可快速学习）

- Swift的三大特性
  - 安全：提供多种安全措施，如值使用前初始化、自动内存管理等
  - 强大：高度优化的LLVM编译器生成，如低级的类似C语言的函数等
  - 现代：采纳了许多现代语言的特性，如闭包、范型、元组、函数式编程等；

注释

- Single-line comments
```
// This is a comment.
```

- Multiline comments
```
/* This is also a comment
but is written over multiple lines. */
```

- **Nested multiline comments**
```
/* This is the start of the first multiline comment.
 /* This is the second, nested multiline comment. */
This is the end of the first multiline comment. */
```

常量和变量

- Declaring Constants and Variables (**let 和 var**)

```
let maximumNumberOfLoginAttempts = 10

var currentLoginAttempt = 0

var x = 0.0, y = 0.0, z = 0.0 //multiple

var welcomeMessage: String   //Type Annotations
```

- **Naming Constants and Variables**

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = “dogcow"

tips: (1)don’t begin with a number; (2)can’t contain whitespace
characters, mathematical symbols, arrows, private-use Unicode scalar
values, or line- and box-drawing characters
```

Standard I/O

- print()

  print(“Hello World!”)

- readLine
  - Playground 中不能使用
  - Mac的 命令行工具 应用 中使用File->Project->macOS->Command Line Tool

```
let input = readLine()?.split(separator: " ")

if let inp = input{
    for item in inp{
        print(item)
    }
}
```

;

- Swift doesn't require you to write a semicolon (;) after each statement, although you can

- However, semicolons *are* required if you want to write multiple separate statements on a single line

```
let cat = "🐱"; print(cat)
```

基本数据类型

- 整数
  - 无小数部分的数字
  - Int，Int8，Int16，Int32，Int，UInt，……
  - Int的位数与操作系统一致。只要有可能，就尽量用Int
  - Int.min, Int.max
- 实数
  - 有小数部分的数字
  - Float（32位，精度6位），Double（64位，精度15位）
- Bool
  - Bool型文字：true，false

类型安全和类型推导

- **Swift is a *type-safe* language.**
  - A type safe language encourages you to be clear about the types of values your code can work with
  - it performs *type checks* when compiling your code and flags any mismatched types as errors

- **Type Inference**
  - *type check* doesn't mean that you have to specify the type of every constant and variable that you declare.
  - Swift uses *type inference* to work out the appropriate type and requires far fewer type declarations than languages such as C or Objective-C.

```
let meaningOfLife = 42
// meaningOfLife is inferred to be of type Int

 let pi = 3.14159
// pi is inferred to be of type Double
```

Type Conversion

- you must opt in to numeric type conversion on a case-by-case basis. This opt-in approach prevents hidden conversion errors and helps make type conversion intentions explicit in your code.

```
let three = 3
let pointOneFourOneFiveNine = 0.14159
let pi = Double(three) + pointOneFourOneFiveNine
// pi equals 3.14159, and is inferred to be of type Double
```

- Swift's type safety prevents non-Boolean values from being substituted for Bool.

Basic Operators

- Swift supports the operators you may already know from languages like C, and improves several capabilities to eliminate common coding errors.

- The assignment operator (=) doesn’t return a value,
- 余数运算符%，可用于浮点数
- tips：二元运算符强制两端空格

- 提供了一些新的运算符，如范围运算符等

String and Character

- String Literal：surrounded by double quotation marks (")

```
let someString = "Some string literal value”

let string1 = "hello"
let string2 = " there"
var welcome = string1 + string2
```

- Character Literal

```
let exclamationMark: Character = “!”

welcome.append(exclamationMark)
// welcome now equals "hello there!"
```

- String Interpolation

```
let multiplier = 3
let message = "\(multiplier) times 2.5 is \(Double(multiplier) * 2.5)"
// message is "3 times 2.5 is 7.5"
```

String and Unicode

- Swift's native String type is built from *Unicode scalar values*.
  - A Unicode scalar value is a unique 21-bit number for a character or modifier:`U+0061` for `LATIN SMALL LETTER A` (`"a"`), or `U+1F425` for `FRONT-FACING BABY CHICK` (“🐥”).
  - Every instance of Swift's Character type represents a single *extended grapheme cluster, which* is a sequence of one or more Unicode scalars that (when combined) produce a single human-readable character.

```
let eAcute: Character = "\u{E9}"                         // é
let combinedEAcute: Character = "\u{65}\u{301}"          // e followed by ́
// eAcute is é, combinedEAcute is é
```

- Counting Characters: .count

```
var word = "cafe"
print("the number of characters in \(word) is \(word.count)")
// Prints "the number of characters in cafe is 4"

word += "\u{301}"    // COMBINING ACUTE ACCENT, U+0301
print("the number of characters in \(word) is \(word.count)")
// Prints "the number of characters in café is 4"
```

String Index

- You can access any character (of type Character) in a String using [] notation. But the indexes inside the [] are not Int, they are a type called String.Index.

```
let s: String = “hello”
let firstIndex: String.Index = s.startIndex
let firstChar: Character = s[firstIndex]
let secondIndex: String.Index = s.index(after: firstIndex) //“h”
let secondChar: Character = s[secondIndex] //“e”
let fifthChar: Character = s[s.index(firstIndex, offsetBy: 4)]  //“o”
let substring = s[firstIndex…secondIndex]    // “he”
```

元组（Tuples）

- *Tuples* group multiple values into a single compound value. The values within a tuple can be of any type

```
let http404Error = (404, "Not Found")
// http404Error is of type (Int, String), and equals (404, "Not Found")
```

- *decompose* a tuple's contents into separate constants or variables

```
let (statusCode, statusMessage) = http404Error
print("The status code is \(statusCode)")
// Prints "The status code is 404”

let (justTheStatusCode, _) = http404Error
print("The status code is \(justTheStatusCode)")
// Prints "The status code is 404”

print("The status code is \(http404Error.0)")
// Prints "The status code is 404"
```

- name the individual elements in a tuple when the tuple is defined

```
let http200Status = (statusCode: 200, description: “OK")
print("The status code is \(http200Status.statusCode)")
// Prints "The status code is 200"
```

- Tuples are particularly useful as the return values of functions.

Control Flow

- 顺序执行
- 循环

  for in, while, while repeat

- 条件

  if, switch

- 控制转移

  break, continue, **fallthrough**, return, throw

- Early Exit

  guard

for in

- 用来更简单地遍历数组(array),字典(dictionary),区间(range),字符串(string)和其他序列类型

```
for index in 1...5 {
    print("\(index) times 5 is\ (index * 5)")
}
```

- 不需要区间序列内每项的值,用下划线(_)替代变量名

```
let base = 3 ,  power = 10
var answer = 1
for _ in 1...power {
    answer *= base
}
```

Range Operator

- ..<　　(exclusive of the upper bound)：半闭合
- ...　　(inclusive of both bounds)：闭合
- This is sort of a pseudo-representation of Range

```
struct Range<T> {
    var startIndex: T
    var endIndex: T
}
```

T is restricted (e.g. comparable), startIndex is less than endIndex

Countable Range

- There are other, more capable, Ranges. If the upper/lower bound is “strideable by Int”, ..< makes a CountableRange.

- CountableRange is enumeratable with for in.
  - for i in 0..<20 { }

- Floating point numbers don’t stride by Int, they stride by a floating point value. There’s a global function that will create a CountableRange from floating point values!
  - for i in stride(from: 0.5, through: 15.25, by: 0.3) { }

while

- 从计算一个条件开始。如果条件为 true,会重复运行一段语句,直到条件变为 false

```
while condition {
    statements
}
```

repeat-while

- 和 while 的区别是在判断循环条件之前,先执行一次循环的代码 块。然后重复循环直到条件为 false

```
repeat{
    statements
} while condition
```

if

```
var temperatureInFahrenheit = 30
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
}
// Prints "It's very cold. Consider wearing a scarf.”
```

```
temperatureInFahrenheit = 90
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear
sunscreen.")
} else {
    print("It's not that cold. Wear a t-shirt.")
}
// Prints "It's really warm. Don't forget to wear sunscreen."
```

switch

- swift的switch 语句比 C 语言中更加强大安全。
- 在 C 语言中,如果某个 case 不小心漏写了 break ,这个 case 就 会贯穿至下一个 case,Swift 无需写 break ,不会发生这种贯穿的情况。
- case 还可以匹配很多不同的模 式,包括间隔匹配(interval match),元组(tuple)和转换到特定类型。 switch 语句的 case 中匹配的值可 以绑定成临时常量或变量,在case体内使用,也可以用 where 来描述更复杂的匹配条件
- switch 语句必须是完备的。这就是说,每一个可能的值都必须至少有一个 case 分支与之对应。在某些不可能涵 盖所有值的情况下,你可以使用默认(default)分支来涵盖其它所有没有对应的值,这个默认分支必须在 swit ch语句的最后面。

- 完备

```
let someCharacter: Character = "z"
switch someCharacter {
case "a":
    print("The first letter of the alphabet")
case "z":
    print("The last letter of the alphabet")
default:
    print("Some other character")
}
```

- 区间匹配

```
let approximateCount = 62
let countedThings = "moons orbiting Saturn"
var naturalCount: String
switch approximateCount {
case 0:
    naturalCount = "no"
case 1..<5:
    naturalCount = "a few"
case 5..<12:
    naturalCount = "several"
case 12..<100:
    naturalCount = "dozens of"
case 100..<1000:
    naturalCount = "hundreds of"
default:
    naturalCount = "many"
}
```

- Tuple

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    print("(0, 0) is at the origin")
case (_, 0):
    print("(\(somePoint.0), 0) is on the x-axis")
case (0, _):
    print("(0, \(somePoint.1)) is on the y-axis")
case (-2...2, -2...2):
    print("(\(somePoint.0), \(somePoint.1)) is inside the box")
default:
    print("(\(somePoint.0), \(somePoint.1)) is outside of the box”)
}
```

- 值绑定(Value Bindings)

```
let anotherPoint = (2, 0)
switch anotherPoint {
case (let x, 0):
    print("on the x-axis with an x value of \(x)")
case (0, let y):
    print("on the y-axis with a y value of \(y)")
case let (x, y):
    print("somewhere else at (\(x), \(y))")
}
```

- where

```
let yetAnotherPoint = (1, -1)
switch yetAnotherPoint {
case let (x, y) where x == y:
    print("(\(x), \(y)) is on the line x == y")
case let (x, y) where x == -y:
    print("(\(x), \(y)) is on the line x == -y")
case let (x, y):
    print("(\(x), \(y)) is just some arbitrary point")
}
```

- 复合匹配(Compound Cases)

```
let someCharacter: Character = "e"
switch someCharacter {
case "a", "e", "i", "o", "u":
    print("\(someCharacter) is a vowel")
case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
     "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
    print("\(someCharacter) is a consonant")
default:
    print("\(someCharacter) is not a vowel or a consonant")
}
```

- 复合匹配同样可以包含值绑定。复合匹配里所有的匹 配模式,都必须包含相同的值绑定。

```
let stillAnotherPoint = (9, 0)
switch stillAnotherPoint {
case (let distance, 0), (0, let distance):
    print("On an axis, \(distance) from the origin")
default:
    print("Not on an axis")
}
```

fallthrough

- 每个需要贯穿特性的 case 分支使用 fallthrough

```
let integerToDescribe = 5
var description = "The number \(integerToDescribe) is"
switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " a prime number, and also"
    fallthrough
default:
    description += " an integer."
}
print(description)
// 输出 "The number 5 is a prime number, and also an integer."
```

continue

- continue 语句告诉一个循环体立刻停止本次循环,重新开始下次循环

```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
for character in puzzleInput.characters {
    switch character {
    case "a", "e", "i", "o", "u", " ":
        continue
    default:
        puzzleOutput.append(character)
    }
}
print(puzzleOutput) // 输出 "grtmndsthnklk"
```

break

- break语句会立刻结束整个控制流的执行。要更早的结束一个代码块或者一个循环体时,使用break语句。

```
let numberSymbol: Character = "三" // 简体中文里的数字 3
var possibleIntegerValue: Int?
switch numberSymbol {
case "1", "?", "一", "?":
  possibleIntegerValue = 1
case "2", "?", "二", "?":
  possibleIntegerValue = 2
case "3", "?", "三", "?":
  possibleIntegerValue = 3
case "4", "?", "四", "?":
   possibleIntegerValue = 4
default:
  break
}
```

带标签的语句

- 显式指明break语句想要终止的是哪个循环体或者条件语句,会很有用。
- 类似地,如果有许多嵌套的循环体,显式指明continue语句想要影响哪一个循环体也非常有用。

```
gameLoop: while square != finalSquare {
    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }
    switch square + diceRoll {
    case finalSquare:
        // diceRoll will move us to the final square, so the game is over
        break gameLoop
   case let newSquare where newSquare > finalSquare:
        // diceRoll will move us beyond the final square, so roll again
        continue gameLoop
    default:
        // this is a valid move, so find out its effect
        square += diceRoll
        square += board[square]
    }
}
print("Game over!”)
```

guard(early exit)

- You use a guard statement to require that a condition must be true in order for the code after the guard statement to be executed.
  (check illegal condition)

- like an if statement, executes statements depending on the Boolean value of an expression. Unlike an if statement, a guard statement always has an else clause—the code inside the else clause is executed if the condition is not true.

```
guard let score >= 0 else {
      return
}
```

可选类型（Optionals）

- 数据类型？　可能是某类型的值或没有值nil

```
let optionalInt: Int? = 9
```

```
var myString = "7"
var possibleInt = Int(myString)
print(possibleInt)
```

```
myString = "banana"
possibleInt = Int(myString)
print(possibleInt)
```

强制解包(unwrap)

- 强制解包操作，使用!

```
let x : String? = "hello"
let y  = x!
```

- 如果不合法，可能报错

```
let x : String?
let y  = x!
```

隐式解包可选类型

- 数据类型！

```
let x : String! = "Hello"
let y  = x
```

- 使用时不需要强制解包

?? Optional default

```
let s: String? = ...        // might be nil

display.text = s ?? " "
```

if let 可选绑定

- 使 可选绑定(optional binding，使用let)来判断可选类型是否包含 值,如果包含就把值赋给 个临时常 或者变 。

```
var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

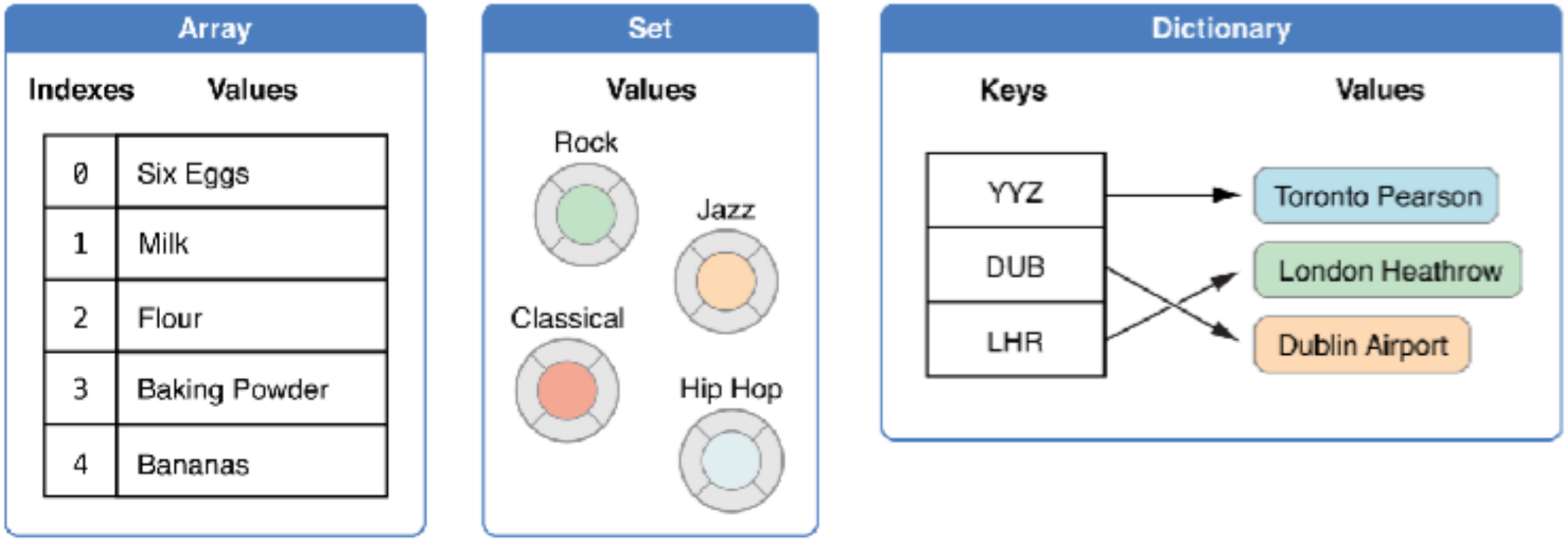
Chained

```
var display: UILabel?
if let temp1 = display {
      if let temp2 = temp1.text{
          let x = temp2.hashValue
      }
}
```

... Optional chaining using ? ...

```
if let x = display?.text?.hashValue { ... }    // x is an Int
let x = display?.text?.hashValue { ... }    // x is an Int?
```

集合类型


Array
Indexes
Values
0
Six Eggs
1
Milk
2
Flour
3
Baking Powder
4
Bananas
Set
Values
Rock
Jazz
Classical
Hip Hop
Dictionary
Keys
Values
YYZ
DUB
LHR
Toronto Pearson
London Heathrow
Dublin Airport

数组

- An *array* stores values of the same type in an ordered list
- 声明

```
var shoppingList  = ["Eggs", "Milk"]

// var shoppingList: [String] = ["Eggs", “Milk"]
//var shoppingList: Array<String> = ["Eggs", “Milk"]
 // shoppingList has been initialized with two initial items
```

```
 var someInts = [Int]()
print("someInts is of type [Int] with \(someInts.count) items.")
 // Prints "someInts is of type [Int] with 0 items.”
```

- 索引和下标：基于0（0-based)

```
shoppingList[0]
let array = [“a”,”b”,”c”,”d”]
let a = array[2…3]    // [”b”,”c”]
```

- 其它方法和属性
  - count, first, last
  - isEmpty, append, insert, +, =, ……

集合

- A *set* stores distinct values of the same type in a collection with no defined ordering
- 声明

```
var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"]
 // favoriteGenres has been initialized with three initial items
```

- 集合操作

```
let oddDigits: Set = [1, 3, 5, 7, 9]
let evenDigits: Set = [0, 2, 4, 6, 8]
let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]

oddDigits.union(evenDigits).sorted()  // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
oddDigits.intersection(evenDigits).sorted() // []
oddDigits.subtracting(singleDigitPrimeNumbers).sorted() // [1, 9]
oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()  // [1, 2, 9]

if favoriteGenres.contains("Funk") {
    print("I get up on the good foot.”)

}
let houseAnimals: Set = ["🐶", "🐱"]
let farmAnimals: Set = ["🐮", "🐔", "🐑", "🐶", "🐱"]
let cityAnimals: Set = ["🐦", "🐭"]

houseAnimals.isSubset(of: farmAnimals) // true
farmAnimals.isSuperset(of: houseAnimals)  // true
farmAnimals.isDisjoint(with: cityAnimals)  // true
```

字典

- A *dictionary* stores associations between keys of the same type and values of the same type in a collection with no defined ordering.（Hashable）

- 声明

```
 var airports: [String: String] = ["YYZ": "Toronto Pearson", "DUB":
"Dublin"]
```

- 通过key访问

```
airports["LHR"] = "London"
 for (airportCode, airportName) in airports {
     print("\(airportCode): \(airportName)")
 }
```

- 可替换switch（高级用法）

函数（Function）

```
func greet(to person: String) -> String {
    let greeting = "Hello " + person + "!"
    return greeting
}
greet(to:"Bob")
```

- 函数定义和调用
  - func
  - 函数名
  - 参数列表，每个参数有外部名(label，调用)和内部名(name)
  - 返回值： -> 类型

外部参数名称(Argument Label)

- 参数标签的使用能够让一个函数在调用时更有表达力(调用者更好理解)，更类似自然语言，并且仍保持了函数内部的可读性。

- 忽略外部参数

```
func greet(_ person: String) -> String {
    let greeting = "Hello " + person + "!"
    return greeting
}
greet("Bob")
```

- 如果参数只有一个名字，那么内外同名

```
func greet(person: String) -> String {
    let greeting = "Hello " + person + "!"
    return greeting
}
greet(person:"Bob")
```

输入输出参数

- 函数参数默认是常量。试图在函数体中更改参数值将编译错误
- 定义时，使用inout关键字；调用时，参数名前加 &

```
func swap(first a : inout Int, second b : inout Int)
{
    let temp = a
    a = b
    b = temp
}

var x = 10,y = 20
swap (first: &x, second: &y)
print(x,y)
```

可变参数

- 一个可变参数(variadic parameter)可以接受零个或多个值。
- 在变量类型名后面加入(...)的方式来定义可变参数。调用时,可以传入不确定数量的参数。可变参数的传入值在函数体中变为此类型的一个数组。

```
func arithmeticMean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
// returns 3.0, which is the arithmetic mean of these five numbers
arithmeticMean(3, 8.25, 18.75)
// returns 10.0, which is the arithmetic mean of these three
numbers
```

函数返回值

- 函数可以没有返回值。也可以使用元组(tuple)类型让多个值作为一个复合值从函数中返回

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        } }
    return (currentMin, currentMax)
}

let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \(bounds.max)")
```

函数类型(Funciton type)

- 每个函数都有种特定的函数类型,函数的类型由函数的参数类型和返回类型组成。没有返回类型使用Void
- 在 Swift 中,使用函数类型就像使用其他类型一样。可以定义函数的常量或变量，也可以作为函数的参数和返回值。

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

var mathFunction: (Int, Int) -> Int = addTwoInts
print("Result: \(mathFunction(2, 3))")

func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)

func chooseStepFunction(flag : Bool) -> (Int,Int) -> Int {
    return flag ? addTwoInts : multiplyTwoInts
}
```

闭包(Closure)

- *Closures* are self-contained blocks of functionality that can be passed around and used in your code.
- Global and nested functions, as introduced in Functions, are actually special cases of closures.

Closures take one of three forms.

- Global functions are closures that have a name and don’t capture any values.
- Nested functions are closures that have a name and can capture values from their enclosing function.
- Closure expressions are unnamed closures written in a lightweight syntax that can capture values from their surrounding context.

闭包表达式

- Syntax

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

- Example

```
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 } )
```

闭包表达式简化

- Inferring Type From Context

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

- Implicit Returns from Single-Expression Closures

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

- Shorthand Argument Names

```
reversedNames = names.sorted(by: { $0 > $1 } )
```

- Operator Methods

```
reversedNames = names.sorted(by: >)
```

尾闭包

- Syntax

```
func someFunctionThatTakesAClosure(closure: () -> Void) {
    // function body goes here
}

// Here's how you call this function without using a trailing closure:

someFunctionThatTakesAClosure(closure: {
    // closure's body goes here
})

// Here's how you call this function with a trailing closure instead:

someFunctionThatTakesAClosure() {
    // trailing closure's body goes here
}
```

- Example

```
reversedNames = names.sorted() { $0 > $1 }
reversedNames = names.sorted { $0 > $1 } //only argument
```

Data Structures in Swift

- 4 fundamental building blocks of data structures in Swift
  - Classes （class）
  - Structures （struct）
  - Enumerations （emum）
  - Protocols （protocol）

Similarities

- Declaration syntax

```
class    ViewController:    … {
}
struct    CalculatorBrain{

}
enum Op  {
}
```

Similarities

- Properties and Functions

```
fun    doit(argx    argi: Type) -> ReturnValue {
}
var    storedProperty   = <initial value>    //(not enum)

var computedProperty:  Type{
      get {}
      set {}
}
```

Similarities

- Initializers (again, not enum)

```
init(arg1x  arg1i: Type, arg2x arg2i: Type,  …)   {
}
```

Differences

- Inheritance (class only)

Differences

- Value type(struct and enum)

  Copied when passed as an argument to a function

  Copied when assigned to a different variable

  Immutable if assigned to a variable with let (function parameters are let)

  You must note any func that can mutate a struct/enum with the keyword mutating

- Reference type (class)

  Stored in the heap and reference counted (automatically)

  Constant pointers to a class (let) still can mutate by calling methods and changing properties

  When passed as an argument, does not make a copy (just passing a pointer to same instance)

How to Choose

- Class VS Struct
    - Inheritance
    - value type VS reference type
    - mutating
    - default initializer
- Use of enum is situational (any time you have a type of data with discrete values).

Method

- Parameters Names

All parameters to all functions have an internal name and an external name The internal name is the name of the local variable you use inside the method The external name is what callers use when they call the method

You can put _ if you don't want callers to use an external name at all for a given parameter This would almost never be done for anything but the first parameter.

If you only put one parameter name, it will be both the external and internal name.

```
fun foo(externalFirst first:    Int, externalSecond second:    Double)     {
      var sum      =    0.0
       for _     in   0..<first{     sum +=      second}
}
func     bar()    {
let  result   =    foo(externalFirst: 123,   externalSecond: 5.5) }

func     foo(_    first:     Int, externalSecond     second:     Double)     {
  var     sum      = 0.0
   for      _ in     0..<first{     sum      += second}
}
func     bar()    { letresult   =    foo(123,     externalSecond:     5.5) }

func     foo(first:     Int, second:     Double)     {
   var sum   = 0.0
   for      _ in     0..<first {    sum      += second }
}
func     bar()    { letresult   =    foo(first:     123,     second:     5.5)}
```

Method

- You can override methods/properties from your superclass
  - Precede your func or var with the keyword override
  - A method can be marked final which will prevent subclasses from being able to override
  - Entire classes can also be marked final

Method

- Both types and instances can have methods/properties
- Type methods and properties are denoted with the keyword static.

```
static func abs(d: Double) -> Double
{ if d < 0 { return -d } else { return d } }
static var pi : Double{return 3.1415926 }
let     d       =       Double.pi  // d    =       3.1415926
let     d       =       Double.abs(-324.44)  // d    =       324.4
let x: Double = 23.85    let e = x.pi // no! pi is not an instance var
let e = x.abs(-22.5)4 // no! abs is not an instance method
```

Properties

- Property Observers
  - You can observe changes to any property with willSet and didSet
  - Will also be invoked if you mutate a struct (e.g. add something to a Dictionary)
  - One very common thing to do in an observer in a Controller is to update the user-interface

```
var someStoredProperty: Int = 42 {
   willSet { newValue is the new value }
   didSet { oldValue is  the old value}
}
```

Properties

- Lazy Initialization
  - A lazy property does not get initialized until someone accesses it.
  - You can allocate an object, execute a closure, or call a method if you want
  - This still satisfies the “you must initialize all of your properties” rule
  - Things initialized this way can’t be constants (i.e., var ok, let not okay)
  - This can be used to get around some initialization dependency conundrums

```
lazy var brain = CalculatorBrain() // if CalculatorBrain used lots of resources
lazy var someProperty: Type = {
        // construct the value of someProperty here
        return <the constructed value>
}()
lazy var myProperty = self.initializeMyProperty()
```

Date & Data

- Date

  Value type used to find out the date and time right now or to store past or future dates See also Calendar, DateFormatter, DateComponents

  If you are displaying a date in your UI, there are localization ramifications, so check these out!

- Data

  A value type “bag o’ bits”. Used to save/restore/transmit raw data throughout the iOS SDK.

NSObject & NSNumber

- NSObject

  Base class for all Objective-C classes

  Some advanced features will require you to subclass from NSObject (and it can't hurt to do so)

- NSNumber

  Generic number-holding class (i.e., reference type)

  let n = NSNumber(35.5)

  or

  let n: NSNumber = 35.5

  let intified: Int = n.intValue // also oubleValue, boolValue, etc.

Casting

- as

  cast any type with as into any other type that makes sense.
  Mostly casting an object from one of its superclasses down to a subclass.
  But it could also be used to cast any type to a protocol it implements.

- always use as? it with if let …
- check if something can be converted with the is keyword (true/false)

  let vc: UIViewController = CalculatorViewController()
  The type of vc is UIViewController (because we explicitly typed it to be).
  And the assignment is legal because a CalculatorViewController is a UIViewController.
  But we can't say, for example, vc.displayValue, since vc is typed as a UIViewController.

  However, if we cast vc to be a CalculatorViewController, then we can use it …

```
if let calcVC = vc as? CalculatorViewController {
    calcVC.displayValue = 3.1415 // this is okay
}
```

Any & AnyObject

- Any can hold something of any type
- AnyObject holds classes only
- Any & AnyObject are special types

These types used to be commonly used for compatibility with old Objective-C APIs

But not so much anymore in iOS 10 since those old Objective-C APIs have been updated

Swift is a strongly typed language, though, so you can't invoke a method on an Any. You have to convert it into a concrete type first.

One of the beauties of Swift is its strong typing, so generally you want to avoid Any.

Any & AnyObject

- Where will you see it in iOS?

  Sometimes (rarely) it will be an argument to a function that can take different sorts of things.
  Here's a UIViewController method that includes a sender (which can be of any type).
  func prepare(for segue: UIStoryboardSegue, sender: Any?)
  The sender is the thing that caused this "segue" (i.e., a move to another MVC) to occur.
  The sender might be a UIButton or a UITableViewCell or some custom thing in your code.
  It's an Optional because it's okay for a segue to happen without a sender being specified.

  It could be used to contain a array of things with different types (e.g. [AnyObject]).
  But in Swift we'd almost certainly use an Array of an enum instead (like in CalculatorBrain).
  So we'd only do this to be backwards-compatible with some Objective-C API.

  You could also use it to return an object that you don't want the caller to know the type of.
  var cookie: Any

Any & AnyObject

- How do we use a variable of type Any?

  We can't usually use it directly (since we don't know what type it really is)

  Instead, we must convert it to another, known type

```
let unknown: Any = …
// we can't send unknown a message because it's "typeless"
if let foo = unknown as? MyType {
    // foo is of type MyType in here
    // so we can invoke MyType methods or access MyType vars in foo
    // if unknown was not of type MyType, then we'll never get here
}
```