Swift

Swift

A short guide to using Apple's new programming language, Swift.

Swift Cheat Sheet

This is a fork from Grant Timmerman's work...

Basics

```
println("Hello, world")
var myVariable = 42                               // variable (can't be nil)
let π = 3.1415926                                 // constant
let (x, y) = (10, 20)                             // x = 10, y = 20
let explicitDouble: Double = 1_000.000_1          // 1,000.0001
let label = "some text " + String(myVariable)     // Casting
let piText = "Pi = \(π)"                          // String interpolation
var optionalString: String? = "optional"          // Can be nil
optionalString = nil

/* Did you know /* you can nest multiline comments */ ? */
```

Arrays

```
// Array
var shoppingList = ["catfish", "water", "lemons"]
shoppingList[1] = "bottle of water"               // update
shoppingList.count                                // size of array (3)
shoppingList.append("eggs")
shoppingList += "Milk"

// Array slicing
var fibList = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 5]
fibList[4..6] // [3, 5]. Note: the end range value is exclusive
fibList[0..fibList.endIndex] // all except last item
// Subscripting returns the Slice type, instead of the Array type.
// You may need to cast it to Array in order to satisfy the type checker
Array(fibList[0..4])

// Variants of creating an array. All three are equivalent.
var emptyArray1 = String[]()
var emptyArray2: String[] = []
var emptyArray3: String[] = String[]()
```

Dictionaries

```
// Dictionary
var occupations = [
    "Malcolm": "Captain",
    "kaylee": "Mechanic"
]
occupations["Jayne"] = "Public Relations"
var emptyDictionary = Dictionary<String, Float>()
```

Control Flow

```
// for loop (array)
let myArray = [1, 1, 2, 3, 5]
for value in myArray {
    if value == 1 {
        println("One!")
    } else {
        println("Not one!")
    }
}

// for loop (dictionary)
var dict = [
    "name": "Steve Jobs",
    "title": "CEO",
    "company": "Apple"
]
for (key, value) in dict {
    println("\(key): \(value)")
}

// for loop (range)
for i in -1...1 { // [-1, 0, 1]
    println(i)
}
// use .. to exclude the last number

// for loop (ignoring the current value of the range on each iteration of the loop)
for _ in 1...3 {
    // Do something three times.
}

// while loop
var i = 1
while i < 1000 {
    i *= 2
}

// do-while loop
do {
    println("hello")
} while 1 == 2

// Switch
let vegetable = "red pepper"
switch vegetable {
case "celery":
    let vegetableComment = "Add some raisins and make ants on a log."
case "cucumber", "watercress":
    let vegetableComment = "That would make a good tea sandwich."
case let x where x.hasSuffix("pepper"):
    let vegetableComment = "Is it a spicy \(x)?"
default: // required (in order to cover all possible input)
    let vegetableComment = "Everything tastes good in soup."
}

// Switch to validate plist content
let city:Dictionary<String, AnyObject> = [
    "name" : "Qingdao",
    "population" : 2_721_000,
    "abbr" : "QD"
]
switch (city["name"], city["population"], city["abbr"]) {
    case (.Some(let cityName as NSString),
        .Some(let pop as NSNumber),
        .Some(let abbr as NSString))
    where abbr.length == 2:
        println("City Name: \(cityName) | Abbr.:\(abbr) Population: \(pop)")
    default:
        println("Not a valid city")
}
```

Functions

Functions are a first-class type, meaning they can be nested in functions and can be passed around

```
// Function that returns a String
func greet(name: String, day: String) -> String {
    return "Hello \(name), today is \(day)."
}
greet("Bob", "Tuesday") // call the greet function

// Function that returns multiple items in a tuple
func getGasPrices() -> (Double, Double, Double) {
    return (3.59, 3.69, 3.79)
}

// Function that takes variable number of arguments, collecting them into an array
```

```
func setup(numbers: Int...) {
    // do something
}
setup(5, 16, 38) // call the setup function with array of inputs

// Nested functions can organize code that is long or complex
func printWelcomeMessage() -> String {
    var y = "Hello,"
    func add() {
        y += " world"
    }
    add()
    return y
}
printWelcomeMessage() // Hello world

// Passing and returning functions
func makeIncrementer() -> (Int -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
var increment = makeIncrementer()
increment(7)
```

Closures

Functions are special case closures ({})

```
// Closure example.
// `->` separates the arguments and return type
// `in` separates the closure header from the closure body
var numbers = [1, 2, 3, 4, 5]
numbers.map({
    (number: Int) -> Int in
    let result = 3 * number
    return result
    })

// When the type is known, like above, we can do this
numbers = [1, 2, 6]
numbers = numbers.map({ number in 3 * number })
println(numbers) // [3, 6, 18]

// When a closure is the last argument, you can place it after the )
// When a closure is the only argument, you can omit the () entirely
// You can also refer to closure arguments by position ($0, $1, ...) rather than name
numbers = [2, 5, 1]
numbers.map { 3 * $0 } // [6, 15, 3]
```

Classes

All methods and properties of a class are public. If you just need to store data in a structured object, you should use a `struct`

```
// A parent class of Square
class Shape {
    init() {
    }

    func getArea() -> Int {
        return 0;
    }
}

// A simple class `Square` extends `Shape`
class Square: Shape {
    var sideLength: Int

    // Custom getter and setter property
    var perimeter: Int {
        get {
            return 4 * sideLength
        }
        set {
            sideLength = newValue / 4
        }
    }

    init(sideLength: Int) {
        self.sideLength = sideLength
        super.init()
    }

    func shrink() {
        if sideLength > 0 {
            --sideLength
        }
    }

    override func getArea() -> Int {
        return sideLength * sideLength
    }
}
var mySquare = Square(sideLength: 5)
print(mySquare.getArea()) // 25
mySquare.shrink()
print(mySquare.sideLength) // 4

// Access the Square class object,
// equivalent to [Square class] in Objective-C.
Square.self

//example for 'willSet' and 'didSet'
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            println("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue  {
                println("Added \(totalSteps - oldValue) steps to 'totalSteps'")
            }
        }
    }
}
var stepCounter = StepCounter()
stepCounter.totalSteps = 100 // About to set totalSteps to 100 \n Added 100 steps to 'totalS
stepCounter.totalSteps = 145 // About to set totalSteps to 145 \n Added 45 steps to 'totalSt

// If you don't need a custom getter and setter, but still want to run code
// before an after getting or setting a property, you can use `willSet` and `didSet`
```

Enums

Enums can optionally be of a specific type or on their own. They can contain methods like classes.

```
enum Suit {
    case Spades, Hearts, Diamonds, Clubs
    func getIcon() -> String {
        switch self {
        case .Spades: return "♤"
        case .Hearts: return "♡"
        case .Diamonds: return "♢"
        case .Clubs: return "♧"
        }
    }
}
```

Protocols

A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality.

```
protocol SomeProtocol {
    // protocol definition goes here
}
```

Extensions

Add extra functionality to an already created type

```
// adds the methods first and rest to the array type
extension Array {
    func first () -> Any? {
        return self[0]
    }
    func rest () -> Array {
        if self.count >= 1 {
            return Array(self[1..self.endIndex])
        } else {
            return []
        }
    }
}
```

Operator Overloading

You can overwrite existing operators or define new operators for existing or custom types.

```
// Overwrite existing types
@infix func + (a: Int, b: Int) -> Int {
    return a - b
}
var x = 5 + 4 // x is 1
```

You can't overwrite the = operator

Add operators for new types

```
struct Vector2D {
    var x = 0.0, y = 0.0
}
@infix func + (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y + right.y)
}
```

Operators can be `prefix`, `infix`, or `postfix`.

You have to add `@assignment` if you wish to define compound assignment operators like +=, ++ or -=

```
@assignment func += (inout left: Vector2D, right: Vector2D) {
    left = left + right
}
```

Operator overloading is limited to the following symbols: / = - + * % < > ! & | ^ . ~

Generics

Generic code enables you to write flexible, reusable functions and types that can work with any type.

```
// Generic function, which swaps two any values.
func swapTwoValues<T>(inout a: T, inout b: T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

```
// Generic collection type called `Stack`.
struct Stack<T> {
    var elements = T[]()

    mutating func push(element: T) {
        elements.append(element)
    }

    mutating func pop() -> T {
        return elements.removeLast()
    }
}
```

We can use certain type constraints on the types with generic functions and generic types. Use `where` after the type name to specify a list of requirements.

```
// Generic function, which checks that the sequence contains a specified value.
func containsValue<
    T where T: Sequence, T.GeneratorType.Element: Equatable>
    (sequence: T, valueToFind: T.GeneratorType.Element) -> Bool {

    for value in sequence {
        if value == valueToFind {
            return true
        }
    }

    return false
}
```

In the simple cases, you can omit `where` and simply write the protocol or class name after a colon. Writing `<T: Sequence>` is the same as writing `<T where T: Sequence>`.

Emoji/Unicode support

You can use any unicode character (including emoji) as variable names or in Strings.

```
  var 😄 = "Smiley"
  println(😄) // will print "Smiley"
  let □ = "□□😺🐭"
  var □: String[] = []
  for □ in □ {
      □.append(□+□)
  }
  println(□) // will print [□□, □□, 😺😺, 🐭🐭]
```

Which, in Xcode looks like

GoodBye

Links

- [Homepage]
- [Guide]
- [Book]

Contributing

Feel free to send a PR or mention an idea, improvement or issue!

And this GitBook is on MHM5000's GitHub page to contribute.

Table of Contents

Introduction 2

Basics 2

Arrays 2

Dictionaries 2

Control Flow 3

Functions 4

Closures 4

Classes 5

Enums 5

- Protocols 5
- Extensions 5
- Operator Overloading 5
- Generics 7
- Emoji/Unicode support 7

GoodBye 7