Swift Montréal Meetup

{ FP in }

Fatih Nayebi | 2016-04-06 18:00 | Tour CGI

Agenda

- Introduction
- First-class, Higher-order and Pure Functions
- Closures
- Generics and Associated Type Protocols
- Enumerations and Pattern Matching
- Optionals
- Functors, Applicative Functors and Monads

Introduction

- Why Swift?
  - Hybrid Language (FP, OOP and POP)
  - Type Safety and Type Inference
  - Immutability and Value Types
  - Playground and REPL
  - Automatic Reference Counting (ARC)
- Why Functional Programming matters?

Functional Programming

- A style of programming that models computations as the evaluation of expressions
- Avoiding mutable states
- Declarative vs. Imperative
- Lazy evaluation
- Pattern matching


SHUT UP AND
SHOW ME THE CODE

Declarative vs. Imperative

```swift
let numbers = [9, 29, 19, 79]

// Imperative example
var tripledNumbers:[Int] = []
for number in numbers {
    tripledNumbers.append(number * 3)
}
print(tripledNumbers)

// Declarative example
let tripledIntNumbers = numbers.map({
    number in 3 * number
})
print(tripledIntNumbers)
```

Shorter syntax

```
let tripledIntNumbers = numbers.map({
    number in 3 * number
})
```

```
let tripledIntNumbers = numbers.map { $0 * 3 }
```

Lazy Evaluation

```
let oneToFour = [1, 2, 3, 4]
let firstNumber = oneToFour.lazy.map({ $0 * 3}).first!

print(firstNumber) // 3
```

Functions

- First-class citizens
    - Functions are treated like any other values and can be passed to other functions or returned as a result of a function
- Higher-order functions
    - Functions that take other functions as their arguments
- Pure functions
    - Functions that do not depend on any data outside of themselves and they do not change any data outside of themselves
    - They provide the same result each time they are executed (**Referential transparency makes** it possible to conduct **equational reasoning**)

Nested Functions

```
func returnTwenty() -> Int {
    var y = 10
    func add() {
        y += 10
    }
    add()
    return y
}
returnTwenty()
```

Higher-order Functions

```
func calcualte(a: Int,
               b: Int,
               funcA: AddSubtractOperator,
               funcB: SquareTripleOperator) -> Int
{
    return funcA(funcB(a), funcB(b))
}
```

Returning Functions

```swift
func makeIncrementer() -> (Int -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}

var increment = makeIncrementer()
increment(7)
```

Function Types

```swift
let mathOperator: (Double, Double) -> Double

typealias operator = (Double, Double) -> Double

var operator: SimpleOperator

func addTwoNumbers(a: Double, b: Double) -> Double
{ return a + b }

mathOperator = addTwoNumbers
let result = mathOperator(3.5, 5.5)
```

First-class Citizens

```
let name: String = "John Doe"

func sayHello(name: String) {
    print("Hello \(name)")
}

// we pass a String type with its respective
value
sayHello("John Doe") // or
sayHello(name)

// store a function in a variable to be able to
pass it around
let sayHelloFunc = sayHello
```

Function Composition

```swift
let content = "10,20,40,30,60"

func extractElements(content: String) -> [String] {
    return content.characters.split(“,").map { String($0) }
}

let elements = extractElements(content)

func formatWithCurrency(content: [String]) -> [String] {
    return content.map {"\($0)$"}
}

let contentArray = ["10", "20", "40", "30", "60"]

let formattedElements = formatWithCurrency(contentArray)
```

Function Composition

```
let composedFunction = { data in
    formatWithCurrency(extractElements(data))
}

composedFunction(content)
```

Custom Operators

```
infix operator |> { associativity left }
func |> <T, V>(f: T -> V, g: V -> V ) -> T -> V {
    return { x in g(f(x)) }
}

let composedFn = extractElements |> formatWithCurrency

composedFn("10,20,40,30,80,60")
```

Closures

- Functions without the **func** keyword
- Closures are self-contained blocks of code that provide a specific functionality, can be stored, passed around and used in code.
- Closures are **reference types**

Closure Syntax

```
{ (parameters) -> ReturnType in
    // body of closure
}
```

Closures as function parameters/arguments:

```
func({(Int) -> (Int) in
//statements
})
```

Closure Syntax (Cntd.)

```
let anArray = [10, 20, 40, 30, 80, 60]

anArray.sort({ (param1: Int, param2: Int) -> Bool in
    return param1 < param2
})

anArray.sort({ (param1, param2) in
    return param1 < param2
})

anArray.sort { (param1, param2) in
    return param1 < param2
}

anArray.sort { return $0 < $1 }

anArray.sort { $0 < $1 }
```

Types

- Value vs. reference types
- Type inference and Casting
- Value type characteristics
    - **Behaviour** - Value types do not behave
    - **Isolation** - Value types have no implicit dependencies on the behaviour of any external system
    - **Interchangeability** - Because a value type is copied when it is assigned to a new variable, all of those copies are completely interchangeable.
    - **Testability** - There is no need for a mocking framework to write unit tests that deal with value types.

struct vs. class

```
struct ourStruct {
    var data: Int = 3
}
var valueA = ourStruct()
var valueB = valueA // valueA is copied to valueB
valueA.data = 5 // Changes valueA, not valueB

class ourClass {
    var data: Int = 3
}
var referenceA = ourClass()
var referenceB = referenceA // referenceA is copied
to referenceB
referenceA.data = 5 // changes the instance
referred to by referenceA and referenceB
```

Type Casting (is and as)

- A way to check type of an instance, and/or to treat that instance as if it is a different superclass or subclass from somewhere else in its own class hierarchy.

- Type check operator - `is` - to check wether an instance is of a certain subclass type

- Type cast operator - `as` and `as?` - A constant or variable of a certain class type may actually refer to an instance of a subclass behind the scenes. Where you believe this is the case, you can try to downcast to the subclass type with the as.

Enumerations

- Common type for related values to be used in a type-safe way
- Value provided for each enumeration member can be a string, character, integer or any floating-point type.
- Associated Values - Can define Swift enumerations to store Associated Values of any given type, and the value types can be different for each member of the enumeration if needed (discriminated unions, tagged unions, or variants).
- Raw Values - Enumeration members can come pre-populated with default values, which are all of the same type.
- Algebraic data types

Enum & Pattern Matching

```swift
enum MLSTeam {
    case Montreal
    case Toronto
    case NewYork
}
let theTeam = MLSTeam.Montreal

switch theTeam {
case .Montreal:
    print("Montreal Impact")
case .Toronto:
    print("Toronto FC")
case .NewYork:
    print("Newyork Redbulls")
}
```

Algebraic Data Types

```
enum NHLTeam { case Canadiens, Senators, Rangers,
Penguins, BlackHawks, Capitals}

enum Team {
    case Hockey(NHLTeam)
    case Soccer(MLSTeam)
}

struct HockeyAndSoccerTeams {
    var hockey: NHLTeam
    var soccer: MLSTeam
}
enum HockeyAndSoccerTeams {
    case Value(hockey: NHLTeam, soccer: MLSTeam)
}
```

Generics

- Generics enable us to write flexible and reusable functions and types that can work with any type, subject to requirements that we define.

```
func swapTwoIntegers(inout a: Int, inout b: Int) {
    let tempA = a
    a = b
    b = tempA
}

func swapTwoValues<T>(inout a: T, inout b: T) {
    let tempA = a
    a = b
    b = tempA
}
```

Functional Data Structures

```
enum Tree <T> {
    case Leaf(T)
    indirect case Node(Tree, Tree)
}

let ourGenericTree =
Tree.Node(Tree.Leaf("First"),
Tree.Node(Tree.Leaf("Second"),
Tree.Leaf("Third")))
```

Associated Type Protocols

```
protocol Container {
    associatedtype ItemType
    func append(item: ItemType)
}
```

Optionals!?

```
enum Optional<T> {
    case None
    case Some(T)
}

func mapOptionals<T, V>(transform: T -> V, input:
T?) -> V? {
    switch input {
    case .Some(let value): return transform(value)
    case .None: return .None
    }
}
```

Optionals!? (Cntd.)

```
class User {
    var name: String?
}

func extractUserName(name: String) -> String {
    return "\(name)"
}

var nonOptionalUserName: String {
    let user = User()
    user.name = "John Doe"
    let someUserName = mapOptionals(extractUserName, input:
user.name)
    return someUserName ?? ""
}
```

fmap

```
infix operator <^> { associativity left }

func <^><T, V>(transform: T -> V, input: T?) -> V? {
    switch input {
    case .Some(let value): return transform(value)
    case .None: return .None
    }
}

var nonOptionalUserName: String {
    let user = User()
    user.name = "John Doe"
    let someUserName = extractUserName <^> user.name
    return someUserName ?? ""
}
```

apply

```
infix operator <*> { associativity left }

func <*><T, V>(transform: (T -> V)?, input: T?) -> V? {
    switch transform {
    case .Some(let fx): return fx <^> input
    case .None: return .None
    }
}

func extractFullUserName(firstName: String)(lastName: String) -> String {
    return "\(firstName) \(lastName)"
}

var fullName: String {
    let user = User()
    user.firstName = "John"
    user.lastName = "Doe"
    let fullUserName = extractFullUserName <^> user.firstName <*> user.lastName
    return fullUserName ?? ""
}
```

Monad

- Optional type is a Monad so it implements map and flatMap

```
let optArr: [String?] = ["First", nil, "Third"]
let nonOptionalArray = optArr.flatMap { $0 }
```

Functor, Applicative Functor & Monad

- Category Theory
- **Functor**: Any type that implements map function
- **Applicative Functor**: Functor + apply()
- **Monad**: Functor + flatMap()

Immutability

- Swift makes it easy to define immutables
- Powerful value types (struct, enum and tuple)

References

- The Swift Programming Language by Apple (swift.org)
- Addison Wesley - The Swift Developer's Cookbook by Erica Sadun
- Packt Publishing - Swift 2 Functional Programming by Fatih Nayebi