Functional Programming in Swift

iOS Bootcamp

Callum McColl

Redeye

April 29, 2017

Overview

Functional vs Object-Oriented Programming

- Split problem into several pieces.
- Object-Oriented
  - Each piece is encapsulated as a class/protocol.
- Functional Programming
  - Each piece is encapsulated as a function.
  - Declarative vs Imperative thinking.
- Swift is a Multi-Paradigm Language
  - Object-Oriented
  - Functional Programming

Functional vs Object-Oriented Programming cont.

- Functional Programming concepts are orthogonal to Object-Oriented Programming.
  - This allows you to pick and choose how you want to solve a problem.
  - You can mix and combine both approaches.
- Some problems are easier to solve in certain paradigms.
  - GUI's are best done in Object-Oriented Languages.
  - Parsers are easier using Functional Programming Languages.
- Combining paradigms results in less error-prone programs.

Functions

- Functions are first class citizens in Swift.
  - Functions may be passed as arguments to other functions.
  - This is known as a higher order function:

```
DispatchQueue.global(qos: .userInitiated).async {
    // Execute code in the background
}
```

  - Functions can return new functions:

```
func add(_ lhs: Int) -> (Int) -> Int {
    return { (rhs: Int) -> Int
        return lhs + rhs
    }
}
add(2)(3) // 5
```

Functions cont.

- Remember the syntax for closures:

```swift
let double: (Int) -> Int = { (num: Int) -> Int in
    return num * 2
}

let double: (Int) -> Int = { num in
    return num * 2
}

let double: (Int) -> Int = { $0 * 2 }

print(double(3)) // 6
```

Mutability vs Immutability

```swift
var x = 2
runProgram() // do some work referencing x
x = 7
runProgram() // do the same work, with different
             // results.
```

- What have we really done here?
- We have made time a part of determining the behaviour of our program.
- The solution is to not use vars:

```swift
let x = 2
let myNewX = runProgram(x) // do some work
runProgram(myNewX) // do the work, with different
                   // input and different results.
```

Mutability vs Immutability cont.

- Swift uses immutability by default.
  - function parameters are immutable.

- The standard library exclusively uses value type semantics.

```
func double(_ point: Point2D) {
    var p = point
    p.x *= 2
    p.y *= 2
}
```

Classes vs Structs

- Using a class:

```
class Point2D {

    var x: Int = 2

    var y: Int = 2

}
var p = Point2D() // p.x = 2, p.y = 2
double(p)
// p.x = 4, p.y = 4
```

- Classes pass parameters by reference.

Classes vs Structs cont.

- Using a struct:

```
struct Point2D {

    var x: Int = 2

    var y: Int = 2

}
var p = Point2D() // p.x = 2, p.y = 2
double(p)
// p.x = 2, p.y = 2
```

- Structs pass parameters by copy.

Classes vs Structs cont.

- Structs enforce immutability:

```
struct Point2D { ... }

let p = Point2D()
p.x = 5 // Error: p must be a var.
```

- Classes do not:

```
class Point2D { ... }

let p = Point2D()
p.x = 5 // Ok
```

Referential Transparency

- An element of a program is referentially transparent if the reference can be replaced by the definition.
- The inputs of a function must map to the outputs of the function.
- There should be no side-effects.
- There should be no mutable state.

```
var i: Int = 0

func add(_ num: Int) -> Int {
    i += num
    return i
}

add(2) // 2
add(2) // 4 <- breaks referential transparency!
```

Referential Transparency cont.

- Object-Oriented Programming *isolates* side-effects:

```
class Adder {

    fileprivate var i: Int

    init(i: Int) { self.i = i }

    func add(_ num: Int) -> Int {
        self.i += num
        return self.i
    }

}
let adder = Adder(i: 0)
adder.add(2) // 2
adder.add(2) // 4
```

Referential Transparency cont.

- So how do we make our accumulative adder?

```
struct Adder {

    let i: Int

    init(i: Int) { self.i = i }

    func add(_ num: Int) -> Adder {
        return Adder(i: self.i + num)
    }

}
let adder = Adder(i: 0)
let adder2 = adder.add(2) // adder2.i is 2
let adder3 = adder2.add(2) // adder3.i is 4
```

Referential Transparency cont.

- Guarantees consistency.
  - Nothing besides the input of the function determines the output.

- Easier to Test.
  - To accurately test something you must consider all inputs.
  - If your function is influenced by side-effects, then they also must be considered.
  - How can we accurately test the following:

```swift
private var i: Int = 0 // hidden from tests.

func add(_ num: Int) -> Int {
    i += num
    return i
}
```

Higher Order Functions

- Swift has functional API's built into the standard library.

- Very Helpful with Collections!
  - Sorting an Array:

```
var unsortedArray = [10, 7, 5, 8, 9, 6, 4, 1, 3, 2]
unsortedArray.sort { $0 < $1 } // [1, 2, 3, 4 ...]
```

  - `sort` takes a function that compares two elements and returns whether the first is less than the second:
    - `mutating func sort(by: (Element, Element) -> Bool)`

  - But this breaks Referential Transparency!

Higher Order Functions cont.

- Collections in Swift generally contain functions that have referentially transparent variants.
  - Sorting an Array without breaking Referential Transparency:

```swift
let unsortedArray = [10, 7, 5, 8, 9, 6, 4, 1, 3, 2]
let sortedArray = unsortedArray.sorted { $0 < $1 }
```

  - `sorted` takes a function that compares two elements and returns whether the first is less than the second, and sorts the elements into a new array.
    - `func sorted(by: (Element, Element) -> Bool) -> Array<Element>`
  - Removes the need to use vars, thus enforcing Referential Transparency.

The Advantage of Higher Order Functions

- Remember `sorted` just takes a function:

```
func lessThan(lhs: Int, rhs: Int) -> Bool {
    return lhs < rhs
}
let sortedArray = unsortedArray.sorted(lessThan)
```

- In Swift, operators are just functions:

```
let sortedArray = unsortedArray.sorted(<)
```

- What if we wanted to sort in descending order?

```
let sortedDescArray = unsortedArray.sorted(>)
```

Manipulating Arrays

- Say we wanted to convert our Array of Ints into an Array of Strings.

- How would we normally do this?

```
var converted: [String] = []
for num in sortedArray {
    converted.append("\(num)")
}
print(converted) // ["1", "2", "3", "4" ...]
```

- Specifies the *how*, not the *what*.

- We can do this with a higher order function.

map

- `map` allows you to easily convert each element within the array.

```
let converted = sortedArray.map { "\($0)" }
    // ["1", "2", "3", "4", ...]
```

- `map` takes a function which takes an element and returns a generic type:

  - `func map<T>(_: (Element) -> T) -> [T]`

- Equivalent to a traditional for loop but:
  - Avoids telling the compiler *how* to do each step.
  - Leads to less code.

flatMap

- Sometimes we want to perform a map using a function that returns another collection:

```
let words = ["this", "is", "a", "string"]
let chars = words.map { $0.characters }
    // [["t", "h", "i", "s"], ["i", "s"], ...]
```

- flatMap can be used to *flatten* the collection:

```
let words = ["this", "is", "a", "string"]
let chars = words.flatMap { $0.characters }
    // ["t", "h", "i", "s", "i", "s", ...]
```

filter

- `filter` can be used to pull out certain elements of a collection:

```
let evenNums = sortedArray.filter { $0 % 2 == 0 }
    // [2, 4, 6, 8, 10]
```

- `filter` takes a function which takes an element of the array as a parameter and returns a Bool indicating whether that value should be included in the new array:
  - `func filter(_: (Element) -> Bool) -> [Element]`

- So we return `true` if we want to include the element, and we return `false` when we want to remove the element.

reduce

- `reduce` can be used to combine all the elements of the array into a single value:

```
let sum = sortedArray.reduce(0, +)
```

```
func reduce<Result>(
    _ initialResult: Result,
    _ nextPartialResult: (Result, Element) -> Result
) -> Result
```

`reduce` cont.

- How would we normally calculate the sum of all the elements in an array?

```swift
var sum = 0
for num in sortedArray {
    sum = sum + num
}
```

reduce cont.

```
func reduce<Result>(
    _ initialResult: Result,
    _ nextPartialResult: (Result, Element) -> Result
) -> Result
```

- So how does this work?

```
// This:
let sum = sortedArray.reduce(0, +)

// is Equivalent to:
let initialResult = 0
let nextPartialResult: (Int, Int) -> Int = +

var sum = initialResult
for num in sortedArray {
    sum = nextPartialResult(sum, num)
}
```

reduce cont.

- We can also use `reduce` to convert the array to a String:

```
let str = sortedArray.reduce("") { $0 + " \($1)" }
    // " 1 2 3 4 ..."

// Equivalent to:
let initialResult = ""
let nextPartialResult: (String, Int) -> String = {
    $0 + " \($1)"
}

var str = initialResult
for num in sortedArray {
    str = nextPartialResult(str, num)
}
```

`reduce` cont.

- And perform a map:

```
// Never do this!
let converted = sortedArray.reduce([]) {
    var arr = $0
    arr.append("\($1)")
    return arr
}
```

- So `reduce` is the higher order function equivalent of a generic for loop.
- Anything that you can do with a for loop, you should be able to accomplish with `reduce`.

Lazy Evaluation

- If the input to a function guarantees a certain output, do we need to always invoke the function immediately?

- Does the order in which functions are invoked really matter?

- Lazy Evaluation is the way in which operations are invoked *only* when their outputs are used.

Lazy Evaluation cont.

- Let's take a look at an example:
  - Let's say we want to create a new array which contains the even numbers within `sortedArray`, converted to Strings:

```
let evenNumbers = sortedArray.filter { $0 % 2 == 0 }
let evenStrings = evenNumbers.map { "\($0)" }
```

  - How many times does this go through each element in the array?
    - Twice
    - Goes through each element in the array for the filter, and again for the map.

  - We can make this faster.

Lazy Evaluation cont.

```
let lazyCollection = sortedArray.lazy
let evenNums = lazyCollection.filter { $0 % 2 == 0 }
let evenStrings = evenNums.map { "\($0)" }
```

- How many times do we go through each element in the array?
  - Zero
  - Remember the output value is only generated if it is being used.
  - We are not using any of the values in `evenStrings`, so there is no need to perform the filter or the map:

```
for str in evenStrings {
    print("The String is: \(str)")
}
```

Lazy Evaluation cont.

- Let's modify this example, and print some messages:

```
let lazyCollection = sortedArray.lazy
let evenNums = lazyCollection.filter {
    print("Filtering: \($0)")
    return $0 % 2 == 0
}

let evenStrings = evenNums.map { (num) -> String in
    print("Converting: \(num)")
    return "\(num)"
}

for str in evenStrings {
    print("The String is: \(str)")
}
```

Lazy Evaluation cont.

- What does this program print?

```
Filtering: 1
Filtering: 2
Converting: 2
The String is: 2
Filtering: 3
Filtering: 4
Converting: 4
The String is: 4
Filtering: 5
Filtering: 6
Converting: 6
The String is: 6
Filtering: 7
...
```

Summary

- Swift is a multi-paradigm language.
  - Supports Object-Oriented and Functional Programming.
- Functional Programming uses the idea of Referential Transparency.
  - Inputs must map to the outputs.
  - No side-effects.
  - No var references.
- Swift contains functional programming API's:
  - `sorted`
  - `map`
  - `flatMap`
  - `filter`
  - `reduce`
- Lazy Evaluation can be leveraged for a more efficient execution in certain situations.

Questions?

Challenges

- There are four challenges.
  1. Sorted List
  2. Grouped Sorted List
  3. Calculate Phone usage
  4. Custom Lazy Sequence

- Each challenge gets progressively harder.

- Each challenge adds more functionality to a data visualisation app.

- The data is an array of tuples where each tuples is a pair consisting of:
  - The name of a user, as a String.
  - The phone that that user owns.

- The `Phone` type is an enum.

Challenges cont.

```swift
enum Phone: String {

    case iPhone4 = "iPhone 4"
    case iPhone5 = "iPhone 5"
    case iPhone6 = "iPhone 6"
    case iPhone7 = "iPhone 7"
    case GalaxyS4 = "Galaxy S4"
    case GalaxyS5 = "Galaxy S5"
    case GalaxyS6 = "Galaxy S6"
    case GalaxyS7 = "Galaxy S7"
    case Other = "Other"
    case None = "None"

}
```

Challenges cont.

- If you ever want to convert a phone to a String, simply write: `phone.rawValue`
- Some rules to follow:
  - For challenges 1 – 3, you are not allowed to use any loops.
  - For challenges 1 – 3, you are not allowed to use vars, unless you are appending to an array.
  - For challenge 4, there are no rules.

Challenges cont.

- Download the challenges at: https://github.com/halfcharged/redeye

```swift
func sorted(by: (Element, Element) -> Bool) -> [Element]
```
- Sorts the array.

```swift
func map<T>(_: (Element) -> T) -> [T]
```
- Allows you to easily convert each element within the array.

```swift
func flatMap<T>(_: (Element) -> [T]) -> [T]
```
- A map, but *flattens* the sub-arrays.

```swift
func filter(_: (Element) -> Bool) -> [Element]
```
- Allows you to create a new array, but only include certain elements.

```swift
func reduce<Result>(
    _ initialResult: Result,
    _ nextPartialResult: (Result, Element) -> Result
) -> Result
```
- Used for generic transformations.