λ Functional Programming with Swift

Dario A Lencina-Talarico

October 12, 2016

Agenda

1) Definition.
2) Motivation.
3) Side effects.
4) aVoid Void.
5) Optionals.
6) Arrays functional extensions: map, filter, reduce etc.
7) Filtering a Dictionary.
8) Q&A.

λ Functional Programming

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

λ Functional Programming

Object Oriented

```
class CalculatorOO {
    var a : Int = 0
    var b : Int = 0

    func add() -> Int {
        return a + b
    }
}
```

Functional

```
class CalculatorFunctional {

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

}
```

λ Functional Programming

Object Oriented (Extreme)

```
class CalculatorOO {
  var a : Int = 0
  var b : Int = 0
  var result : Int = 0

  func add() -> Void {
     result = a + b
  }
}
```

Functional

```
class CalculatorFunctional {

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

}
```

Motivation

Have you seen this kind of error handling?

```
public func performHeartSurgery() -> Bool {
    return openTorax() && insertCoronaryStent() && closeTorax()
}
```

Motivation

Have you seen this kind of error handling?

```
public func performHeartSurgery() -> Bool {
  do {
     try openTorax()
     try insertCoronaryStent()
     try closeTorax()
     return true
  } catch {
     return false
  }
}
```

Motivation

Have you seen this kind of error handling?

```
public func performHeartSurgery() -> Bool {
  do {
     try openTorax()
     try insertCoronaryStent()
     try closeTorax()
     return true
  } catch {
     return false
  }
}
```

1) What does Bool mean?

2) What does true mean? Did the patient die?

3) What was the problem, is the patient alive?

4) If you are going to return Bool, you might
spare the do catch block

Motivation

Have you seen this kind of error handling?

```
public func performHeartSurgery2() throws -> Patient? {
   try openTorax()
   try insertCoronaryStent()
   try closeTorax()
   return self.patient
}
```

Motivation

Async version

```
public func performHeartSurgery(onResult : @escaping OnResult) -> Void {
    openTorax { (p1, surgeryError) in
        if let _ = surgeryError {
            onResult(p1,surgeryError)
        } else {
            self.insertCoronaryStent { (p2, coronaryStentError) in
                if let _ = coronaryStentError {
                    onResult(p2,coronaryStentError)
                } else {
                    self.closeTorax { (p3, closeToraxError) in
                        onResult(p3,closeToraxError)
                    }
                }
            }
        }
    }
}
```

Motivation

Async version

```
public func performHeartSurgery(onResult : @escaping OnResult) -> Void {
    openTorax { (p1, surgeryError) in
        if let _ = surgeryError {
            onResult(p1,surgeryError)
        } else {
            self.insertCoronaryStent { (p2, coronaryStentError) in
                if let _ = coronaryStentError {
                    onResult(p2,coronaryStentError)
                } else {
                    self.closeTorax { (p3, closeToraxError) in
                        onResult(p3,closeToraxError)
                    }
                }
            }
        }
    }
}
```

Motivation

Async version

```
public func performHeartSurgery(onResult : @escaping OnResult) -> Void {
  openTorax { (p1, surgeryError) in
     if let _ = surgeryError {
        onResult(p1,surgeryError)
     } else {
        self.insertCoronaryStent { (p2, coronaryStentError) in
           if let _ = coronaryStentError {
              onResult(p2,coronaryStentError)
           } else {
              self.closeTorax { (p3, closeToraxError) in
                 onResult(p3,closeToraxError)
              }
           }
        }
     }
  }
}
```

Crap

Motivation

Async version

```
public func performHeartSurgery(onResult : @escaping OnResult) -> Void {
  openTorax { (p1, surgeryError) in
    if let _ = surgeryError {
      onResult(p1,surgeryError)
    } else {
      self.insertCoronaryStent { (p2, coronaryStentError) in
        if let _ = coronaryStentError {
          onResult(p2,coronaryStentError)
        } else {
          self.closeTorax { (p3, closeToraxError) in
            onResult(p3,closeToraxError)
          }
        }
      }
    }
  }
}
```

Callback Hell

Motivation

Functional version:

```
func performHeartSurgery(p : Patient) -> Future<Patient,HeartSurgeryError> {
  return self.openTorax(p : p).flatMap {
          self.insertCoronaryStent(p: $0)}.flatMap {
          self.closeTorax(p: $0)}
}
```

Side effects

Wouldn’t be great if the type of a function would describe exactly what the function is doing?

```
func add(a:Int, b:Int) -> Int {
   launchPolarisMissiles() //side effect
   solveWorldHunger() // side effect
   return a + b
}
```

Side effects

Wouldn’t be great if the type of a function would describe exactly what the function is doing?

```
func add(a:Int, b:Int) -> Int {
   launchPolarisMissiles() //side effect
   solveWorldHunger() // side effect
   return a + b
}
```

Try to not use Void

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

Optionals

Designed to do “optional chaining” or “happy path”

“Optional chaining is a process for querying and calling properties, methods, and subscripts on an optional that might currently be nil. If the optional contains a value, the property, method, or subscript call succeeds; if the optional is nil, the property, method, or subscript call returns nil. Multiple queries can be chained together, and the entire chain fails gracefully if any link in the chain is nil.”

- (https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/OptionalChaining.html)

Optionals

```
func withdraw(account : Int, amount : Int) -> Account? {
    if let account = accounts[account] {
        if account.balance >= amount {
            let newAccount = Account(id: account.balance, balance: account.balance - amount)
            accounts[newAccount.id] = newAccount //side effect
            return newAccount
        } else {
            return nil
        }
    } else {
        return nil
    }
}
```

Optionals

```
func withdraw(account : Int, amount : Int) -> Account? {
  let newAccount: Account? = accounts[account]
      .filter {a in return a.balance >= amount}
       .flatMap {a in return Account(id:a.id, balance:a.balance - amount)}

  newAccount.map {a in accounts[a.id] = a} //side effect
  return newAccount
}
```

Optionals

```
func withdraw(account : Int, amount : Int) -> Account? {
  let newAccount: Account? = accounts[account]
      .filter { $0.balance >= amount}
      .flatMap { Account(id:$0.id, balance:$0.balance - amount)}

  newAccount.map {accounts[$0.id] = $0} //side effect
  return newAccount
}
```

Arrays

```
map<T>(_ transform: (Self.Iterator.Element) throws -> T) rethrows -> [T]
```

```
[1, 2, 3].map {n in n * 2}
```

```
[1, 2, 3].map {$0 * 2}
```

Procedural equivalent

```
var result : [Int] = []
for element in [1,2,3] {
   result.append(element * 2)
}
```

Arrays

```
filter(_ isIncluded: (Self.Iterator.Element) throws -> Bool) rethrows ->
[Self.Iterator.Element]

[1, 2, 3].filter {s in s >= 2}

[1, 2, 3].filter {$0 >= 2}

var result : [Int] = []
for element in [1,2,3] {
   if(element >= 2) {
       result.append(element)
   }
}
```

Arrays

```
flatMap<SegmentOfResult : Sequence>(_ transform: (Self.Iterator.Element) throws -> SegmentOfResult)
rethrows -> [SegmentOfResult.Iterator.Element]

let numbers = [1, 2, 3, 4]
let mapped = numbers.map { Array(count: $0, repeatedValue: $0) }
[[1], [2, 2], [3, 3, 3], [4, 4, 4, 4]]

let flatMapped = numbers.flatMap { Array(count: $0, repeatedValue: $0) }
[1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
```

Returns an array containing the concatenated results of calling the given transformation with each element of this sequence.

Use this method to receive a single-level collection when your transformation produces a sequence or collection for each element.

Filtering a Dictionary

```
func filterAccounts0(f : (Account) -> Bool) -> [Int : Account] {
    var newDict = [Int : Account]()
    for accountTuple in self.accounts {
        if f(accountTuple.value) {
            newDict[accountTuple.key] = accountTuple.value
        }
    }
    return newDict
}
```

Filtering a Dictionary

Wrong type!

```
public func filter(_ isIncluded: (Self.Iterator.Element) throws -> Bool) rethrows -> [Self.Iterator.Element]
```

I had to create

```
public func filterCopy(_ isIncluded: (Key, Value) throws -> Bool) rethrows -> [Key : Value]
```

Filtering a Dictionary

```
func filterAccounts1(f : (Account) -> Bool) -> [Int : Account] {
    return accounts.filterCopy { (key: Int, value: Account) -> Bool in
        f(value)
    }
}

func filterAccounts2(f : (Account) -> Bool) -> [Int : Account] {
    return accounts.filterCopy { f($1)}
}
```

Filtering a Dictionary

```
let customFilter = [123:"bla",124 : "bla2"].filterCopy {(key,value) in value == "bla"}
//[123 : "bla"]

let customFilter2 = [123:"bla",124 : "bla2"].filterCopy {$1 == "bla2"}
print(customFilter2)
//[124 : "bla2"]
```