FizzBuzz In Swift; A Talk With 3 Codas

Abizer Nasir | @abizern

The Setup

You've turned up for an interview, they would like you to generate FizzBuzz for the first 100 numbers.

FizzBuzz

- If the number is a multiple of 3, print "Fizz"
- If the number is a multiple of 5, print "Buzz"
- If the number is a multiple of 3 and 5, print "FizzBuzz"
- Otherwise, just print the number

Simple solution that works

```
func fizzBuzzSimple(number: Int) -> String {
    if number % 15 == 0 {
        return "FizzBuzz"
    } else if number % 5 == 0 {
        return "Buzz"
    } else if number % 3 == 0 {
        return "Fizz"
    } else {
        return "\(number)"
    }
}
```

Using Pattern Matching

```
func fizzBuzzWithPatternMatching(number: Int) -> String {
    switch (number % 5, number % 3) {
    case (0, 0):
        return "FizzBuzz"
    case (0, _):
        return "Buzz"
    case (_, 0):
        return "Fizz"
    default:
        return "\(number)"
    }
}
```

FizzBuzzBar

- The same as FizzBuzz except...
- If the number is a multiple of 7, add the string "Bar"
- e.g 105 -> "FizzBuzzBar"

```swift
func fizzBuzzBarWithPatternMatching(number: Int) -> String {
    switch (number % 7, number % 5, number % 3) {
    case (0, 0, 0):
        return "FizzBuzzBar"
    case (0, 0, _):
        return  "BuzzBar"
    case (_, 0, 0):
        return  "FizzBuzz"
    case (0, _, 0):
        return "FizzBar"
    case (0, _, _):
        return "Bar"
    case (_, 0, _):
        return "Buzz"
    case (_, _, 0):
        return "Fizz"
    default:
        return "\(number)"
    }
}
```

This is becoming hard to maintain

With three clauses, its simple enough to see on one screen and to reason it's correctness.

```
func fizzBuzzWithPatternMatching(number: Int) -> String {
    switch (number % 5, number % 3) {
    case (0, 0):
        return "FizzBuzz"
    case (0, _):
        return "Buzz"
    case (_, 0):
        return "Fizz"
    default:
        return "\(number)"
    }
}
```

With more, you start thinking about needing tests.

```
func fizzBuzzBarWithPatternMatching(number: Int) -> String {
    switch (number % 7, number % 5, number % 3) {
    case (0, 0, 0):
        return "FizzBuzzBar"
    case (0, 0, _):
        return  "BuzzBar"
    case (_, 0, 0):
        return  "FizzBuzz"
    case (0, _, 0):
        return "FizzBar"
    case (0, _, _):
        return "Bar"
    case (_, 0, _):
        return "Buzz"
    case (_, _, 0):
        return "Fizz"
    default:
        return "\(number)"
    }
}
```

The requirements are likely to change again.

Pass options into the function.

```
func fizzBuzzWithOptions(number: Int, options: [(Int, String)]) -> String {
    var output = ""

    for (divisor, description) in options {
        if number % divisor == 0 {
            output += description
        }
    }

    if output.isEmpty {
        output = "\(number)"
    }

    return output
}

var simpleOptions = [(3, "Fizz"), (5, "Buzz"), (7, "Bar")]
```

Simpler to call

```
var simpleOptions = [(3, "Fizz"), (5, "Buzz"), (7, "Bar")]

results = [String]()
for number in 1...105 {
    results.append(fizzBuzzWithOptions(number, simpleOptions))
}

results == kFizzBuzzBar
```

Make an extension to Int so that the number parameter doesn't need to be passed in.

```
extension Int {
    func fizzBuzzIntExtension(options: [(Int, String)]) -> String {
        var output = ""

        for (divisor, description) in options {
            if self % divisor == 0 {
                output += description
            }
        }

        if output.isEmpty {
            output = "\(self)"
        }

        return output
    }
}
```

Just as simple to call.

```
var simpleOptions = [(3, "Fizz"), (5, "Buzz"), (7, "Bar")]

results = [String]()
for number in 1...105 {
    results.append(number.fizzBuzzIntExtension(simpleOptions))
}

results == kFizzBuzzBar
```

FizzBuzzBarFooWhatever is just a sequence that follows a rule. Sequence. That sounds familiar.

SequenceType

```
protocol SequenceType : _Sequence_Type {
    typealias Generator : GeneratorType
    func generate() -> Generator
}
```

GeneratorType

```
protocol GeneratorType {
    mutating func next() -> Element?
}
```

Define a generator

```
struct FizzBuzzGenerator: GeneratorType {
    let start: Int
    let end: Int
    var number: Int

    init(start: Int, end: Int) {
        self.start = start
        self.end = end
        self.number = start
    }

    typealias Element = String
    mutating func next() -> Element? {
         while number <= end {
                return fizzBuzzBarWithPatternMatching(number++)
            }
            return nil
    }
}
```

define a sequence that uses this generator

```
struct FizzBuzzSequenceSimple: SequenceType {
    let start: Int
    let end: Int

    typealias Generator = FizzBuzzGenerator
    func generate() -> Generator {
        return FizzBuzzGenerator(start: start, end: end)
    }
}

results = [String]()
for result in FizzBuzzSequenceSimple(start: 1, end: 105) {
    results.append(result)
}
results

results == kFizzBuzzBar
```

Let's move the generator inside the sequence

```
struct FizzBuzzSequenceComposed: SequenceType {
    let generator: FizzBuzzGenerator

    init(start: Int, end: Int) {
        generator = FizzBuzzGenerator(start: start, end: end)
    }

    struct FizzBuzzGenerator: GeneratorType {
        let start: Int
        let end: Int
        var number: Int

        init(start: Int, end: Int) {
            self.start = start
            self.end = end
            self.number = start
        }

        typealias Element = String
        mutating func next() -> Element? {
            while number <= end {
                return fizzBuzzBarWithPatternMatching(number++)
            }
            return nil
        }
    }

    typealias Generator = FizzBuzzGenerator
    func generate() -> Generator {
        return generator
    }
}
```

This doesn't look right

- There is a lot of state being passed between the values.
- There is no way to change the generator without editing the function.

GeneratorOf<T>

```
struct GeneratorOf<T> : GeneratorType, SequenceType {
    init(_ next: () -> T?)
    init<G : GeneratorType where T == T>(_ self_: G)
    mutating func next() -> T?
    func generate() -> GeneratorOf<T>
}
```

Takes a closure, returns the function to use as the generator.

Pass the options directly to `GeneratorOf<T>`

```
struct FizzBuzzSequenceWithOptions: SequenceType {
    let start: Int
    let end: Int
    let options: [(Int, String)]

    func generate() -> GeneratorOf<String> {
        var number = self.start
        return GeneratorOf<String> {
            while number <= self.end {
                return fizzBuzzWithOptions(number++, self.options)
            }
            return nil
        }
    }
}
```

This is just as simple to call

```
results = [String]()
for result in FizzBuzzSequenceWithOptions(start: 1, end: 105, options: simpleOptions) {
    results.append(result)
}

results == kFizzBuzzBar
```

What if they get even sneakier? FizzBuzz doesn't have to be restricted to numbers.

Generalized FizzBuzzBarFooWhatever

- For numbers, behaves as already described.
- For Strings, base the output on the length of the string.
- For Arrays, base the output on the length of the array.
- For Shapes, base the output on the number of sides.
- etc.

Start by defining an Interface for the Type to conform to

```
protocol FizzBuzzable {
    func fizzBuzz(options: [(Int, String)]) -> String
}
```

Extend Types to conform to this Interface

Int

```
extension Int: FizzBuzzable {
    func fizzBuzz(options: [(Int, String)]) -> String {
        var output = ""

        for (divisor, description) in options {
            if self % divisor == 0 {
                output += description
            }
        }

        if output.isEmpty {
            output = "\(self)"
        }

        return output
    }
}
```

String

```
extension String: FizzBuzzable {
    func fizzBuzz(options: [(Int, String)]) -> String {
        let length = countElements(self)
        var output = ""


        for (divisor, description) in options {
            if length % divisor == 0 {
                output += description
            }
        }

        if output.isEmpty {
            output = "\(self)"
        }

        return output
    }
}
```

Define an Array to work on.

Arrays in Swift must be typed, but Interfaces are also types.

```
var typedArray: [FizzBuzzable] = [3, 5, 7, 11, "How", "Brown", "Penguin", "Marmoset"]

var expected = ["Fizz", "Buzz", "Bar", "11", "Fizz", "Buzz", "Bar", "Marmoset"]

results = [String]()
for element: FizzBuzzable in typedArray {
    results.append(element.fizzBuzz(simpleOptions))
}

results == expected
```

You can even have different FizzBuzz options for each type

```
var typedArray: [FizzBuzzable] = [3, 5, 7, 11, "How", "Brown", "Penguin", "Marmoset"]

results = [String]()
for element: FizzBuzzable in typedArray {
    var result: String
    if let number = element as? Int {
        result = element.fizzBuzz([(3, "Fizz")])

    } else if let string = element as? String {
        result = element.fizzBuzz([(3, "Hello"), (5, "Goodbye")])

    } else {
        result = "\(element)"
    }

    results.append(result)
}

expected = ["Fizz", "5", "7", "11", "Hello", "Goodbye", "Penguin", "Marmoset"]

results == expected
```

Coda 1. First Person

Objective-C without the C

Please, not Java!

It turned out to be not so bad

- I know a little Haskell.
- It made me a better Objective-C programmer.
- It's going to make me a better Swift Programmer.
- Swift isn't a purely functional language, but some of the concepts can be applied.
- For most people, it’s the Cocoa Frameworks that are the hurdle, not the language.

Coda 2. Second Person

What can you do about it?

Learn A functional Programming Language.

I like Haskell, but other products are available: Scala, Clojure, OCAML, F#...

Some handy tips you'll pick up and understand will be...

Think of Types and Interfaces rather than classes and operations. It's like OOP, but from a different perspective.

Aim for immutability in your objects. Structs rather than classes.

Write pure functions as much as possible, keep methods that have side effects to a minimum.

Don’t just re-write your Objective-C code with Swift Syntax.

Coda 3. Third Person

“Write the code, change the world”

“Find the bugs, file the radars”

Don’t give up on Objective-C just yet. You’ll be writing it for a while, and reading it for even longer.

Thank You!

http://downloads.abizern.org/FizzBuzzery.zip

http://learnyouahaskell.com

http://book.realworldhaskell.org