A Swift Start

Brandon Williams

Optional Types

Optional Types

You could have created optional types

Optional Types

You could have created optional types

You would have created optional types

Optional Types

You could have created optional types

Let's re-create optional types!

Overview of Optional Types

- Given a type "`A`", its optional type is "`A?`"
- A value of type `A?` holds something of type `A` or nothing, aka `nil`
- e.g.
  - `Int?`
  - `String?`
  - `[Int?]`
  - `[Int]?`

Why?

1. Swift is a strongly typed language

   All values need an explicit type

2. It can be useful to have the absence of a value

   But in a strongly typed language, how does one do that?

3. Objective-C was pretty close to this with `nil`

   But still possible to smudge types

How to use?

(live coding in playground)

How to use?

```
let colorsByName: [String: Int] = [
  "red": 0xff0000,
]

let redColor: Int? = colorsByName["red"]
```

```
if let redColor = redColor {
  redColor / 2
} else {
  redColor == nil
}
```

Real world example

Functions need a well-defined way to accept/return values that might possibly not exist.

```
func findInt(xs: [Int], x: Int) -> Int? {
  for (idx, element) in enumerate(xs) {
    if element == x { return idx }
  }
  return nil
}

find([1, 2, 3, 4], 3) // => 2
find([1, 2, 3, 4], 5) // => nil
```

Let's build Optional types

New data type that either has a value, or doesn't.

Enums are perfect for this

Enums are perfect for this

```
enum Maybe <A> {
   case Just(A)
   case Nothing
}
```

Enums are perfect for this

```
enum Maybe <A> {
   case Just(A)
   case Nothing
}
```

```
let x = Maybe<Int>.Just(4)
let y = Maybe<String>.Nothing
```

How to use?

(live coding in playground)

How to use?

```
enum Maybe <A> {
  case Just(A)
  case Nothing

  func description () -> String {
    switch self {
    case let .Just(value): return "{Just \(value)}"
    case .Nothing: return "{Nothing}"
    }
  }
}

let x = Maybe<Int>.Just(5)

switch x {
case let .Just(x):
  x * x
case .Nothing:
  "nothing to do"
}
```

Compare with Swift syntactic sugar

```
let x: Int? = 2
let y = Maybe<Int>.Just(2)

if let x = x {
  // x is now an honest Int
} else {
  // handle no value
}

switch y {
case let .Just(y):
  // y is now an honest Int
case .Nothing:
  // handle no value
}
```

Function composition

Function composition

```
func h: A -> B { /* body */ }
func g: B -> C { /* body */ }
func f: C -> D { /* body */ }

f(g(h(x)))
```

Imagine there are methods on arrays of `Int`'s:

```
square([1, 2, 3]) // => [1, 4, 9]
addOne([1, 2, 3]) // => [2, 3, 4]
sort([3, 1, 2])   // => [1, 2, 3]
```

Then we can do:

```
sort(addOne(square([3, 1, 5, -1])))
// => [2, 2, 10, 26]
```

Maybe values kinda mess up function composition

```
func squareRoot (x: Float) -> Maybe<Float> {
  if x >= 0 {
    return .Just(sqrtf(x))
  }
  return .Nothing
}

func invert (x: Float) -> Maybe<Float> {
  if x != 0.0 {
    return .Just(1.0 / x)
  }
  return .Nothing
}

invert(squareRoot(2.0)) // won't compile!
```

Something to fix function composition

Something to fix function composition

```
func >>= <A, B> (x: Maybe<A>, f: A -> Maybe<B>) -> Maybe<B>
```

Something to fix function composition

```
func >>= <A, B> (x: Maybe<A>, f: A -> Maybe<B>) -> Maybe<B>
```

- This operator is called `bind`.
- Think of it as trying to stuff the value on the left into the function on the right.

(live coding in playground)

Something to fix function composition

```
func >>= <A, B> (x: Maybe<A>, f: A -> Maybe<B>) -> Maybe<B> {
  switch x {
  case let .Just(x):
    return f(x)
  case .Nothing:
    return .Nothing
  }
}

let y = squareRoot(2.0)
  >>= { .Just($0 - 1.0) }
  >>= { invert($0) }
  >>= { squareRoot($0) }

# => {Just 1.553773999214 17}
```

A peak into Swift's `Optional`

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

Conclusion

- We've built our own version of `Optional`
- It can do everything the native `Optional` type can do.
- We don't have Swift's syntactic sugar...
- ...but we created functions to aid in composition
    - those functions could (and should) be defined for Swift's optionals.

For the objc peeps

- The fact that you can pass messages to `nil` is *nearly* `bind`

For the Ruby peeps

- Ruby's `try` is *nearly* `bind`.
  - `object.try(:method)` will call `method` on `object`, unless `object` is `nil`, in which case it returns `nil`.
- Important note: this `try` has nothing to do with `try/catch`.