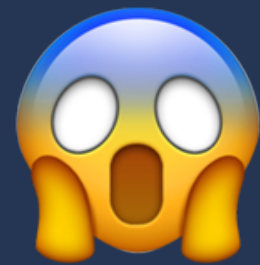


Functors and Monads



Don't Panic!

Context - a magic box/container which wraps a value

```
struct Array<Element> { }  
enum Optional<Wrapped> { }  
enum Result<T, ErrorType> { }  
struct MyStruct<B> { }
```

FUNCTORS

Declaration

```
Class Functor f where  
    fmap :: (a -> b) -> fa -> fb
```

Or

```
protocol Functor {  
    static func map<IntType, OutType>  
        (transform: Intype -> OutType, input: Self<IntType>) -> Self<Outtype>  
}
```

Make it *swifty*

```
protocol Functor {  
    static func map<InType, OutType>  
        (transform: InType -> OutType, input: Self<InType>) -> Self<OutType>  
}
```

become

```
protocol Mappable {  
    associatedtype Element  
    static func map<OutType>  
        (transform: Element -> OutType) -> Self<OutType>  
}
```

Optional

```
extension Optional: Mappable {  
    associatedtype Element = Wrapped  
    func map<OutType>(transform: (Element) -> OutType) -> OutType? {  
        guard let x = self else { return nil }  
        return transform(x)  
    }  
}
```

```
var date: Date? = //some date
```

```
// without map
```

```
if let date = date {  
    var formatted: String? = DateFormatter().string(from: date)  
} else {  
    //no date :(  
}
```

```
//with map
```

```
var formatted = date.map(DateFormatter().string)
```


Array

```
extension Array: Mappable {  
    associatedtype Element = Generator.Element  
    func map<OutType>(transform: Element -> OutType) -> [OutType] {  
        var result: [OutType] = []  
        for x in self {  
            result.append(transform(x))  
        }  
  
        return result  
    }  
}
```

```
func doubleValue(value: Double) -> Double
```

```
//without map
```

```
var newArray = [Double]  
for value in array {  
    newArray.append(doubleValue(value))  
}
```

```
//with map
```

```
let newArray = array.map(doubleValue)
```

Other Functor types?

`Array<T>: func map(transform: T -> U) -> Array<U>`

`Optional<T>: func map(transform: T -> U) -> Optional<U>`

`Promise<T>: func then(transform: T -> U) -> Promise<U>`

`Result<T>: func map(transform: T -> U) -> Result<U>`



MONADS

Declaration

```
protocol Monads: Mappable {  
    static func flatMap<InType, OutType>  
    (transform: InType -> Self<OutType>, input: InType) -> Self<OutType>  
}
```

or

```
protocol FlatMappable: Mappable {  
    associatedType: Element  
    static func flatMap<OutType>  
    (transform: Element -> Self<OutType>) -> Self<OutType>  
}
```

But why?

```
func stringToDate(string: String) -> Date?
```

```
let maybeDate = maybeString.map(stringToDate)  
maybeDate // Date??
```

```
func commentsById(id: String) -> [String]
```

```
let maybeComments = maybeIds.map(commentsById)  
maybeComments // [[String]]
```

Optional

```
extension Optional: FlatMappable {  
    associatedtype Element = Wrapped  
    func flatMap<OutType>  
    (transform: (Wrapped) -> OutType?) -> OutType? {  
        guard let x = self else { return nil }  
        return transform(x)  
    }  
}
```


Array

```
extension Array: FlatMappable {  
    associatedtype Element = Generator.Element  
    func flatMap<OutType>  
    (transform: Element -> [OutType]) -> [OutType] {  
        return self.map(transform).reduce([], +)  
    }  
}
```

```
func stringToDate(string: String) -> Date?
```

```
let maybeDate = maybeString.flatMap(stringToDate)  
maybeDate // Date?
```

```
func commentsById(id: String) -> [String]
```

```
let maybeComments = maybeId.flatMap(commentsById)  
maybeComments // [String]
```


Benefits

No more boilerplate code

Functional chaining

Readable and elegant code

Conclusion

Q&A

igor.atamanciuc@yopeso.com

