Swift Enums

Michał Januszewski @ 10Clouds

Not just C enums.

Seriously. Much. Much more…

Swift enums

- common type for a group of related values
- value is string, character or a number
- first-class types
- computer properties
- instance methods
- initializers
- extendable
- can conform to protocols

Basics

Defining enums

```
enum Side {
    case Left, Right, Top, Bottom
}
let mySide = Side.Left

switch mySide {
    case .Left: print("Left")
    default: print("Other")
}
```

Enum values

```
enum Constants: Double {
    case π = 3.14159
    case e = 2.71828
    case φ = 1.61803398874
    case λ = 1.30357
}
```

Enum values

```
enum Fruit: Int {
    case Apple = 1, Banana, Orange, Kiwi
}
enum Fruit: String {
    case Apple, Banana, Orange, Kiwi
}
if Fruit.Apple.rawValue == "Apple" { print("true") }
```

Init from existing value

```
let apple = Fruit(rawValue: 1)
```

Embedded enums

```
enum Vehicle {
    enum FourWheel {
        case Ferrari, Lamborghini, Lotus
    }
    enum TwoWheel {
        case Honda, Yamaha, Kawasaki
    }
}
let hondaBike = Vehicle.TwoWheel.Honda
```

Embedding enums

```
struct Car {
    enum Engine {
        case V6, V8, V10
    }
    enum WheelSize {
        case Size16, Size17, Size18
    }
    let name: String
    let engine: Engine
    let wheelSize: WheelSize
}

let myCar = Car(name: "My car", engine: .V6, wheelSize: .Size18)
```

Associated values

```
enum Trade {
    case Buy(stock: String, amount: Int)
    case Sell(stock: String, amount: Int)
}
let trade = Trade.Buy(stock: "APPL", amount: 500)

if case let Trade.Buy(stock, amount) = trade {
    print("Buy \(amount) of \(stock)")
}
```

Associated values

```
enum Trade {
    case Buy(String, Int)
    case Sell(String, Int)
}
```

Tuple arguments

```
let tp = (stock: "TSLA", amount: 100)
let trade = Trade.Sell(tp)
```

Tuple arguments

```
typealias CarConfig = (Power: Int, Turbo: Bool, AC: Bool)

func updatePower(config: CarConfig) -> CarConfig {
    return (Power: 240, Turbo: config.Turbo, AC: config.AC)
}
func updateTurbo(config: CarConfig) -> CarConfig {
    return (Power: config.Power, Turbo: true, AC: config.AC)
}
func updateAC(config: CarConfig) -> CarConfig {
    return (Power: config.Power, Turbo: config.Turbo, AC: true)
}

enum Setup {
    case Low(CarConfig)
    case Mid(CarConfig)
    case High(CarConfig)
}

let myCar = Setup.High(updateAC(updateTurbo(updatePower((0, true, true)
as CarConfig))))
```

Tuple arguments

```
infix operator <^> { associativity left }
func <^>(a: CarConfig, f: (CarConfig) -> CarConfig) -> CarConfig {
    return f(a)
}
let config = (0, true, true) <^> updatePower  <^> updateTurbo <^>
updateAC
let myCar = Setup.High(config)
```

Methods and properties

```
struct Car {
    enum Engine {
        case V6, V8, V10
    }
    enum WheelSize {
        case Size16, Size17, Size18
    }
    let name: String
    let engine: Engine
    let wheelSize: WheelSize

    func printEngine() { print("My car engine: \(engine)") }
}
```

Methods and properties

```
enum Engine {
    case V6, V8, V10
    var power: Int {
        switch self {
        case V6: return 240
        case V8: return 340
        case V10: return 480
        }
    }
}
```

Static methods

```
enum Engine {
    case V6, V8, V10
    var power: Int {
        switch self {
        case V6: return 240
        case V8: return 340
        case V10: return 480
        }
    }
    static func fromPower(power: Int) -> Engine? {
        if power == 240 { return .V6 }
        else { return nil }
    }
}
```

Mutating methods

```
enum Engine {
    case V6, V8, V10

    mutating func upgrade() {
        switch self {
        case V6: self = V8
        case V8: self = V10
        case V10: self = V10
        }
    }
}
```

Struct vs enum

- categorization
- hierarchy

Struct vs enum

```
struct Point {
    let x: Int
    let y: Int
}
struct Rect {
    let x: Int
    let y: Int
    let width: Int
    let height: Int
}

enum GeometricEntity {
    case Point(x: Int, y: Int)
    case Rect(x: Int, y: Int, width: Int, height: Int)
}
```

Protocols

Enums can protocols!*

* Enums can implement protocols ;-)

Enums and protocols

```
enum Engine: CustomStringConvertible {
    case V6, V8, V10
    var description: String {
        switch self {
        case .V10: return "I am V10!"
        case .V6: return "I am V6"
        case .V8: return "I am V8"
        }
    }
}

print(Engine.V8)
```

Extensions

Enum extensions

```
enum Engine {
    case V6, V8, V10
}

extension Engine {
    func start() { print("Starting: \(self)") }
    func stop() { print("Stopping: \(self)") }
}
```

Generics

Enums and generics

```
let nope = Optional<Int>.None
let ave = Optional<Int>.Some(666)
```

Enums and generics

```
enum Either<T1, T2> {
    case Left(T1)
    case Right(T2)
}

func getValue(isValid: Bool) -> Either<Int, Int> {
    if isValid { return .Left(0) } else { return .Right(1) }
}
```

Generics constraints

```
enum MyCollection<T: SequenceType where T.Generator.Element ==
Equatable> {
    case Empty
    case Full(elements: T)
}
```

Recursive types

```
enum FileNode {
    case File(name: String)
    indirect case Folder(name: String, files: [FileNode])
}

or

indirect enum FileNode {
    case File(name: String)
    case Folder(name: String, files: [FileNode])
}
```

Custom type as value

```
extension CGSize: StringLiteralConvertible {
    public init(stringLiteral value: String) {
        let size = CGSizeFromString(value)
        self.init(width: size.width, height: size.height)
    }

    public init(extendedGraphemeClusterLiteral value: String) {
        let size = CGSizeFromString(value)
        self.init(width: size.width, height: size.height)
    }

    public init(unicodeScalarLiteral value: String) {
        let size = CGSizeFromString(value)
        self.init(width: size.width, height: size.height)
    }
}

enum Screen: CGSize {
    case Small = "{100, 100}"
    case Large = "{300, 300}"
}
```

Enum equability

```
enum Trade {
    case Buy(stock: String, amount: Int)
    case Sell(stock: String, amount: Int)
}
func ==(lhs: Trade, rhs: Trade) -> Bool {
    switch (lhs, rhs) {
        case let (.Buy(stock1, amount1), .Buy(stock2, amount2))
            where stock1 == stock2 && amount1 == amount2:
            return true
        case let (.Sell(stock1, amount1), .Sell(stock2, amount2))
            where stock1 == stock2 && amount1 == amount2:
            return true
        default: return false
    }
}
```

Iterating through enums

```
enum ProductCategory: String {
    case Washers, Dryers, Toasters
    static let allValues = [Washers, Dryers, Toasters]
}

for category in ProductCategory.allValues {
    print(category)
}
```

Std lib enums

- Bit
- FloatingPointClassification
- Mirror.AncestorRepresentation
- Mirror.DisplayStyle
- Optional
- Process

Error handling!

```
enum VendingMachineError: ErrorType {
    case InvalidSelection
    case InsufficientFunds(coinsNeeded: Int)
    case OutOfStock
}
```

Real world use cases

- Status codes
- Api endpoints (we use it!)
- Observer pattern (Change: Insertion, Deletion…)
- JSON (JSONString, JSONNumber…)
- UIKit ids
- Units (mm, cm, m, km, …)
- LinkedLists? ;-)

Questions?

Jesteście piękni!