Code Smells Better with Swift Enumerations

@bwhiteley

Swift Enums

- Raw Values
- Methods
- Properties (computed only)
- Associated Values
- Initializers
- Extensible

Raw Values

- String
- Character
- Int
- Float

Raw Values

```swift
enum Direction:String {
    case North="North", South="South", East="East", West="West"
}

let south:Direction? = Direction(rawValue: "South")
let str = south?.rawValue
let up = Direction(rawValue: "up") // failable initializer returns nil
// Note: The book not up-to-date.
```

Associated Values

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

Return Values

```
enum ImageReturn {
    case Success (UIImage)
    case Error (String)
}
    func getImage(completion: (imageReturnType:ImageReturn) -> Void) {
       // retrieve image.
            if success {
                completion(ImageReturnType.Success(image)
            }
            else {
                completion(ImageReturnType.Error("404")
            }
    }
```

Generic Return Values

```
class Box<T> {
    let unbox: T
    init(_ value: T) {
        self.unbox = value
    }
}
// Box class is to work around a current Swift limitation
enum Result<T> {
    case Value(Box<T>)
    case Error(NSError)
}

func dataWithContentsOfFile(file: String, encoding: NSStringEncoding) -> Result<NSData> {
    var error: NSError?

    if let data = NSData(contentsOfFile: file, options: .allZeros, error: &error) {
        return .Value(Box(data))
    }
    else { return .Error(error!) }
}
```

Example

```
enum Gender {
    case Male
    case Female
    case Unknown

    var key:String {
        get {
            switch self {
            case .Male: return KeyMale
            case .Female: return KeyFemale
            case .Unknown: return KeyUnknownGender
            }
        }
    }

    var displayName:String {
        get {
            switch self {
            case .Male: return String.localizedStringForKey("gender_male", table: nil)
            case .Female: return String.localizedStringForKey("gender_female", table: nil)
            case .Unknown: return String.localizedStringForKey("gender_unknown", table: nil)
            }
        }
    }

    static func fromKey(str:String) -> Gender {
        switch str {
        case self.Male.key: return .Male
        case self.Female.key: return .Female
        default: return .Unknown
        }
    }
}
```

Example: Extend for Specific Purpose

```swift
// For use with a UISegmentedControl gender selector
private extension Gender {
    var segmentIndex:NSInteger {
        get {
            switch self {
            case .Male: return 0
            case .Female: return 1
            case .Unknown: return 2
            }
        }
    }

    init?(segmentIndex:NSInteger) { // failable initializer
        switch segmentIndex {
        case 0:
            self = .Male
        case 1:
            self = .Female
        case 2:
            self = .Unknown
        default:
            return nil
        }
    }
}
```

Extensions

- You can extend Objective C enums
- You can extend the `Optional` enum

Tables: The problem

- Show/hide rows depending on data/context
- Sometimes you want decoration rows
- No context in DataSource/Delegate methods
- Reconstruct effective index based on various state
- Common solutions have a bad smell

Tables: This Smells Better

```
enum CellType: String {
    case Switch = "switch"
    case Text = "textField"
    case Segment = "segment"
}

enum RowType {
    case FirstName
    case LastName
    case Location
    case Text
    case Gender
    case SafeMode

    var cellType:CellType {
        get {
            var ctype:CellType
            switch self {
            case .FirstName, .LastName, .Location, .Text: ctype = .Text
            case .Gender: ctype = .Segment
            case .SafeMode: ctype = .Switch
            }
            return ctype
        }
    }

    var reuseIdentifier:String {
        get { return cellType.rawValue }
    }
}
```

Tables: Declare The Rows

```
lazy private var schema:[RowType] = {
            if let item = self.dataItem {
                var rv:[RowType]
                switch item.type {
                case .Date, .Description:
                    rv = [.Text]
                case .Name:
                    rv = [.FirstName, .LastName]
                case .Place:
                    rv = [.Location]
                case .Gender:
                    rv = [.Gender]
                }
                if self.showSafeMode {
                    rv += [.SafeMode]
                }
                return rv
            }
            return []
        }()
```

Tables: DataSource and Delegate

```
func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return self.schema.count
}

func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    var cell:UITableViewCell;

    let rtype = schema[indexPath.row]

    cell = tableView.dequeueReusableCellWithIdentifier(rtype.reuseIdentifier,
                                forIndexPath: indexPath) as UITableViewCell

    switch rtype {
    case .Text:
    ...
    case .FirstName:
    ...
```

Tables: Associated Values

```swift
enum CellType  {
    case Info(DataItem, SectionInfo?)
    case SectionHeader(SectionInfo)
    func identifier() -> String {
        switch self {
        case .SectionHeader:
            return "SectionHeaderCell"
        case .Info:
            return "InfoCell"
        }
    }
}

func loadSchema() {
    var ra = self.data.map({ CellType.Info($0, nil) }) // add a section
    for sectionInfo in altSections {
        ra += [CellType.SectionHeader(sectionInfo)] // add separator
        ra += sectionInfo.items.map({CellType.Info($0, sectionInfo) } ) // add other sections
    }
    self.schema = ra
}
```

DataSource

```
override func tableView(tableView: UITableView?, numberOfRowsInSection section: Int) -> Int {
    return self.schema.count
}

override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cellType = self.schema[indexPath.row]

    switch cellType {
    case let .Info(data, _):
        let cell = tableView.dequeueReusableCellWithIdentifier(cellType.identifier(),
                                                forIndexPath: indexPath) as InfoCell
        self.configureInfoCell(cell, data: data)
        return cell

    case let .SectionHeader(SectionInfo):
        let cell = tableView.dequeueReusableCellWithIdentifier(cellType.identifier(),
                                                forIndexPath: indexPath) as UITableViewCell
    ...
```

??