Swift講習会 #3

Yuhel Tanaka

今回やること

- クロージャ
- エラー処理
- **Extension**

Yuhel Tanakaより

これで最後(3/3回)なので頑張りましょう！

クロージャ

関数自体をインスタンス化する機能

{(引数名:型) -> 戻り値の型(戻り値がない場合はVoid) in 処理}

```
let say = {(name:String) -> Void in
    print(name)
}
say("tanaka") // tanaka
```

何がうれしいの？？？

クロージャのいいところ

クロージャは変数に代入できる！

ということは・・・

クロージャのいいところ

ほかのメソッドにクロージャを渡せる！

≒ **処理を渡せる！**

クロージャの使用例 #1

```
func hello(name:String, completed:((String) -> Void)) {
    print("hello")
    // 処理完了
    completed(name) // クロージャで渡した処理を実行する
}

let say = {(name:String) -> Void in
    print(name)
}

hello(name:"yuhei", completed:say) // "hello" "yuhei"
```

クロージャの使用例 #2

AlertでOKボタンを押したときの処理がクロージャで渡されている(handler引数)

```
let okAction = UIAlertAction(title: "OK", style: .default,
    (action: UIAlertAction!) in
    let defaults = UserDefaults.standard
    defaults.set(userName, forKey: "userName")
    if defaults.synchronize() {
        self.loadState()
    }
})
```

エラー処理

- エラーを出してみる
- エラーを処理してみる

エラーを出してみる

- エラーを出すには**throw文**を使う
- エラーを出すメソッドには**throwsキーワード**を使う

```
struct NameError:Error {
}

func throwError(name:String) throws -> String {
    if name != "tanaka" {
        throw NameError()
    }
    return name
}
```

エラーを処理してみる

パターンは3種類

- エラーの種類によって場合分けしたい
- エラーが起きたらnilにすればいいや
- エラーは絶対に起きない！

エラーの種類によって場合分けしたい

do-catch文を使う

do-catch文

throwsキーワードが使われたメソッドを呼び出すにはtryキーワードが必要

```
func throwError(name:String) throws -> String {
    if name != "tanaka" {
        throw NameError()
    }
    return name
}
do {
    let name = try throwError("tanaka")
    print(name)
}
catch {
    print("error")
}
```

エラーによる場合分け

do-catch文ではエラーによって場合分けもできる

```
let name = ""

do {
  let output = try outputName(name: name)
  print(output)
}
catch NameError.invalid {
  print("invalid name")
}
catch let error {
  print(error)
}
```

エラーが起きたらnilにすればいいや

try?キーワードを使う

do-catch文なしで使える

try?キーワード

unwrapと一緒に使うパターン

```
struct NameError:Error {
}
func throwError(name:String) throws -> String {
    if name != "tanaka" {
        throw NameError()
    }
    return name
}
if let name = try? throwError("tanaka") {
    print("tanaka")
}
```

エラーは絶対に起きない！

try!キーワードを使う

エラーが起きたらクラッシュ！

try!キーワード

```
struct NameError:Error {
}
func throwError(name:String) throws -> String {
    if name != "tanaka" {
        throw NameError()
    }
    return name
}
let name1 = try! throwError("tanaka")
print(name1) // "tanaka"

let name2 = try! throwError("hoge") // クラッシュ
```

Extension

what??

Extensionとは

オブジェクトに対する機能追加です

- 追加できるもの
  - メソッド
  - Computed Property

クラス, 構造体, 列挙体に対する**Extension**

クラス, 構造体, 列挙体に対するExtension

```
class ClassMan {
    var height: Double
    var weight: Double
    func calcBMI() -> Double {
        return self.weight / self.height / self.height
    }
}
extension ClassMan { // ClassManクラスに対して機能拡張
    func sayHello() {
        print("hello")
    }
}
```

≡

よくみるExtensionの例

おみくじアプリでの例

```
class SecondViewController: UIViewController {
    // UIViewControllerを継承したSecondViewController
}

extension SecondViewController: UITableViewDataSource {
    // UITableViewDataSourceプロトコルを適合させる
}

extension SecondViewController: UITableViewDelegate {
    // UITableViewDelegateプロトコルを適合させる
}
```

プロトコルに対する Extension

プロトコルも実装を持てる！

```
protocol Car {
    var speed:Double {get set}
    var weight:Double {get set}
    func run();
}

extension Car { // Carプロトコルにデフォルト実装を定義
    func run() {
        print("running")
    }
}
```

プロトコルに対する **Extension**

```
struct VolksWagen:Car {
    var speed: Double
    var weight: Double
    init(speed:Double, weight:Double) {
        self.speed = speed
        self.weight = weight
    }
}

let wagen = VolksWagen(speed: 80, weight: 1000)
wagen.run() // "running" -> デフォルト実装
```

終わり

これで全部終わりです。