mocka
negócios tecnológicos

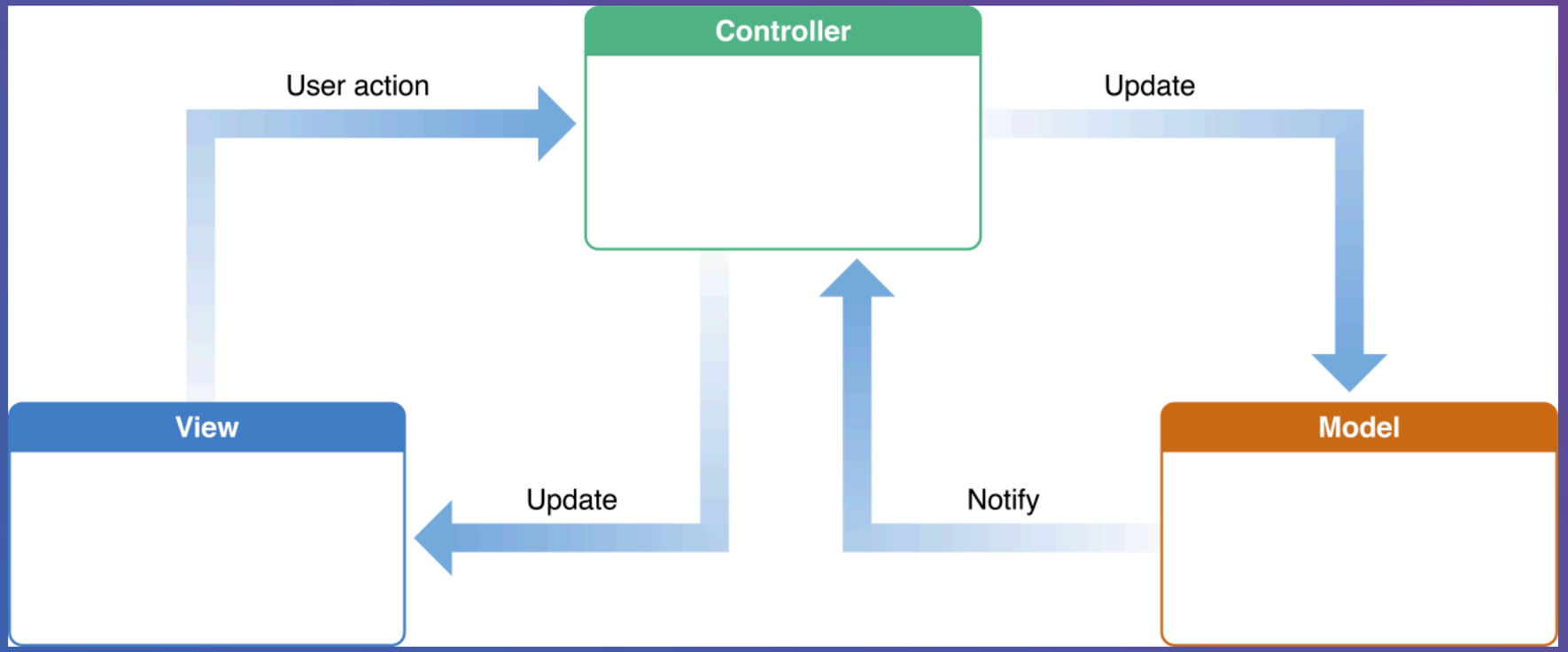
Aula 3

Matheus Frozzi Alberton

iOS

MVC, Extension, Error Handling, Generics

MVC



Model

The Model is where your data resides. Things like persistence, model objects, parsers and networking code normally live there.

View

The View layer is the face of your app. Its classes are typically reusable, since there aren’t any domain-specific logic in them. For example, a UILabel is a view that presents text on the screen, and it’s easily reusable.

Controller

The Controller mediates between the view and the model, typically via the delegation pattern. In the ideal scenario, the controller entity won’t know the concrete view it’s dealing with. Instead, it will communicate with an abstraction via a protocol. A classic example is the way a UITableView communicates with its data source via the UITableViewDataSource protocol.

Model Example

```
class User {
    var id: String = "0"
    var name: String = ""
    var email: String = ""
    var password: String = ""
```

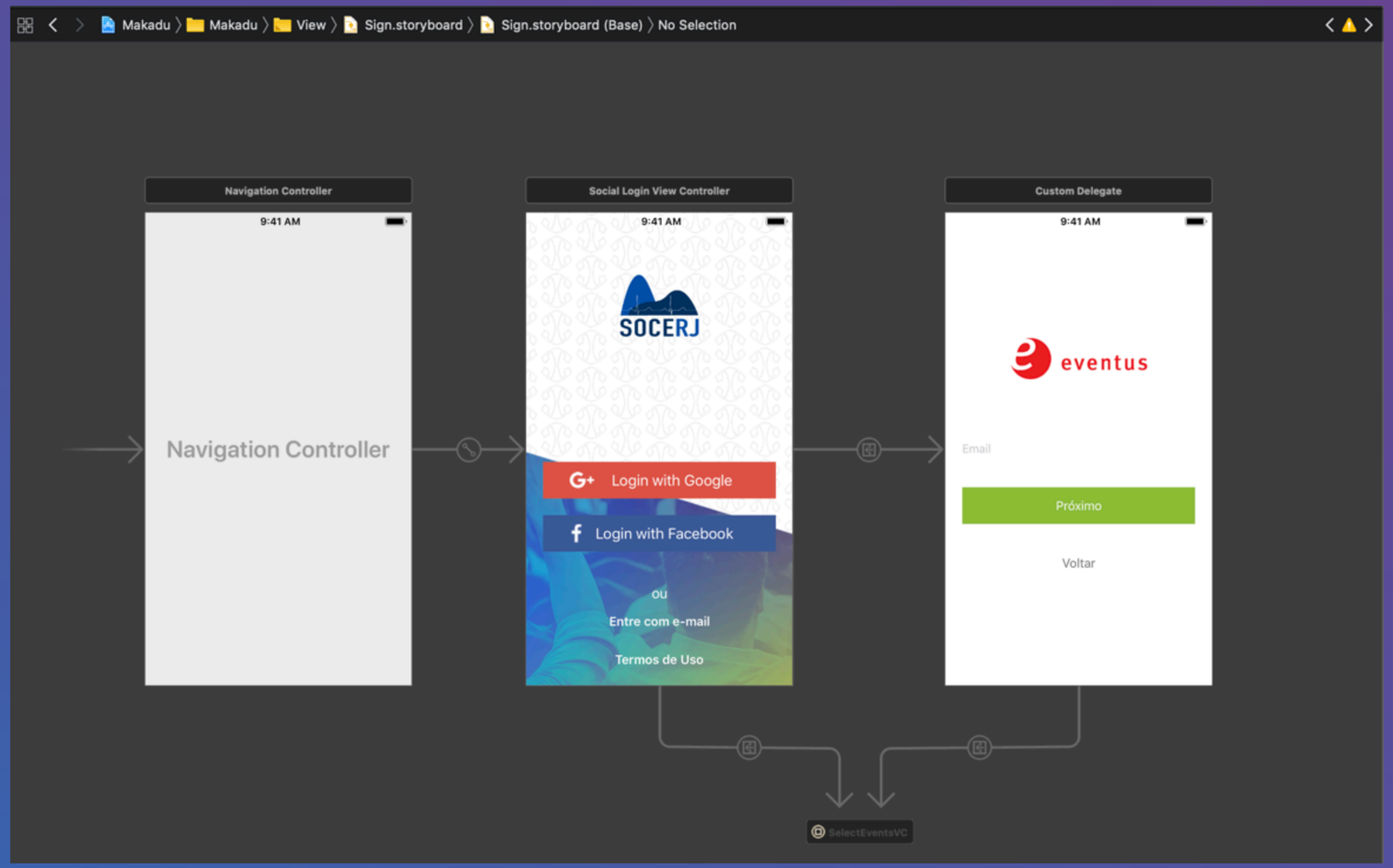
Model Example

```
    func login(callback: @escaping (_ user: User?, _ success: Bool)->()) {

        let params: [String: Any] = [
            "email": email,
            "password": password
        ]

        API.login.request(params: params) { (_Data, _Error) in

            guard let json = _Data as? JSON, _Error == nil else {
                callback(nil, false)
                return
            }

            let user = User(json: json)
            User.current = user

            callback(user, true)
        }
    }
```

Model Example

```
    func register(callback:@escaping (_ user: User?, _ success: Bool)->()) {

        let params: [String: Any] = [
            "name": name,
            "email": email,
            "password": password
        ]

        API.newUser.request(params: params) { _Data, _Error in

            guard let json = _Data as? JSON, _Error == nil else {
                callback(nil, false)
                return
            }

            let user = User(json: json)
            User.current = user

            callback(user, true)
        }
    }
```

View Example



Controller Example

```
class EmailLoginViewController: UIViewController {
    // MARK: - Outlets
    @IBOutlet weak var textFieldEmail: UITextField!
    @IBOutlet weak var textFieldPassword: UITextField!
    @IBOutlet weak var textFieldFullName: UITextField!
```

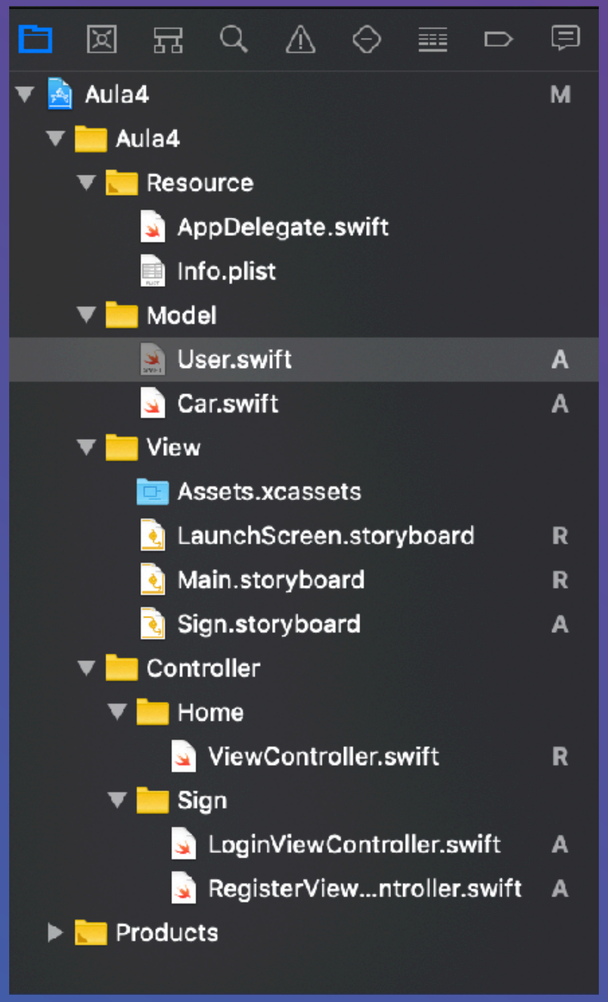
Controller Example

```
    @IBAction func submitButton(_ sender: UIButton) {
        guard let email = textFieldEmail.text?.removeWhitespace(), let name =
textFieldFullName.text, let password = textFieldPassword.text else {

            print("Fill all fields")
            return
        }

        guard email.isValidEmail() else {
            print(“This email isn’t valid")
            return
        }

        let user = User()
        user.email = email
        user.name = name
        user.password = password
```

Controller Example

```
user.register { user, success in

    guard user != nil && success else {

        print(“An error occurred")
        return
    }

    print(“Go to the next screen as a registered user")
}
```

MVC


Aula4
M
Aula4
Resource
AppDelegate.swift
Info.plist
Model
User.swift
A
Car.swift
A
View
Assets.xcassets
LaunchScreen.storyboard
R
Main.storyboard
R
Sign.storyboard
A
Controller
Home
ViewController.swift
R
Sign
LoginViewController.swift
A
RegisterView...ntroller.swift
A
Products

Extension

Extensions add new functionality to an existing class, structure, enumeration, or protocol type. This includes the ability to extend types for which you do not have access to the original source code (known as retroactive modeling). Extensions are similar to categories in Objective-C. (Unlike Objective-C categories, Swift extensions do not have names.)

Extensions

```
public protocol ClassMaster: class {
    var objectId: String? { get set }
    var createdAt: Date? { get set }
    var updatedAt: Date? { get set }

    init?(json: [String: Any])
    init()
}
```

Extensions

```
extension ClassMaster {
    func timeFromLastUpdate() -> String? {
        guard let updatedAt = updatedAt else {
            return nil
        }

        let components = 
Calendar.current.dateComponents([.day, .minute, .hour], from: Date(), to: 
updatedAt)

        return "\(components.day ?? 0) dias, \(components.hour ?? 0) horas e \
(components.minute ?? 0) minutos"
    }
}
```

Extensions

```
class User: ClassMaster {

    var objectId: String?
    var createdAt: Date?
    var updatedAt: Date?

    required init() { }
    required init?(json: [String : Any]) {

    }
}

let user = User()
user.updatedAt = Date()

print(user.timeFromLastUpdate())
```

Extensions

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
    var ft: Double { return self / 3.28084 }
}

let oneInch = 25.4.mm
print("One inch is \(oneInch) meters")
// Prints "One inch is 0.0254 meters"
let threeFeet = 3.ft
print("Three feet is \(threeFeet) meters")
// Prints "Three feet is 0.914399970739201 meters"
```

Extensions

```
extension Date {
    var millisecondsSince1970: Int {
        return Int((self.timeIntervalSince1970 * 1000.0).rounded())
    }

    func increaseDateWith(days: Int) -> Date {
        let calendar = Calendar(identifier: .gregorian)

        var dateComponent = DateComponents()
        dateComponent.day = days

        let newDate = calendar.date(byAdding: dateComponent, to: self)
        return newDate ?? Date()
    }
}
```

Extensions

```swift
extension Int {
    func repetitions(task: () -> Void) {
        for _ in 0..<self {
            task()
        }
    }

    mutating func square() {
        self = self * self
    }
}

3.repetitions {
    print("Hello!")
}

var someInt = 3
someInt.square()
print(someInt)
// Will print 9
```

Extensions

```
extension Int {
    enum Kind: String {
        case negative = "+", zero = "0", positive = "-"
    }

    var kind: Kind {
        switch self {
        case 0:
            return .zero
        case let x where x > 0:
            return .positive
        default:
            return .negative
        }
    }
}

print(3.kind.rawValue)
// Will print "+"
```

Error Handling

Error handling is the process of responding to and recovering from error conditions in your program. Swift provides first-class support for throwing, catching, propagating, and manipulating recoverable errors at runtime.

Error Handling

```
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

Error Handling

```
struct Item {
    var price: Int
    var count: Int
}

class VendingMachine {
    var inventory = [
        "Candy Bar": Item(price: 12, count: 7),
        "Chips": Item(price: 10, count: 4),
        "Pretzels": Item(price: 7, count: 11)
    ]
    var coinsDeposited = 0
```

Error Handling

```
func vend(itemNamed name: String) throws {
        guard let item = inventory[name] else {
            throw VendingMachineError.invalidSelection
        }

        guard item.count > 0 else {
            throw VendingMachineError.outOfStock
        }

        guard item.price <= coinsDeposited else {
            throw VendingMachineError.insufficientFunds(coinsNeeded: item.price -
coinsDeposited)
        }

        coinsDeposited -= item.price

        var newItem = item
        newItem.count -= 1
        inventory[name] = newItem

        print("Dispensing \(name)")
 }
```

Error Handling

```
let snack = "Candy Bar"

let vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 3
try? vendingMachine.vend(itemNamed: snack)

// This code will be ignored, because we put this as optional and the vending
machine dont got enough coins to buy this snack
```

Error Handling

```
vendingMachine.coinsDeposited = 15
try? vendingMachine.vend(itemNamed: snack)
print(vendingMachine.coinsDeposited) // Will print 3, because the "Candy Bar"
price is 12
```

Error Handling

```
struct PurchaseHistory {
    let name: String
    init(name: String, vendingMachine: VendingMachine) throws {
        try vendingMachine.vend(itemNamed: name)
        self.name = name
    }
}

let purchase = try? PurchaseHistory(name: snack, vendingMachine:
vendingMachine)
// purchase will be nil because the vending machine dont have enough coins to
buy this snack

vendingMachine.coinsDeposited = 10
let purchase2 = try! PurchaseHistory(name: snack, vendingMachine:
vendingMachine)
// purchase2 will have an instance of the class Purchase History
```

Error Handling

```
let snack = "Candy Bar"
let vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8

do {
    try vendingMachine.vend(itemNamed: snack)
    print("Success! Yum.")
} catch VendingMachineError.invalidSelection {
    print("Invalid Selection.")
} catch VendingMachineError.outOfStock {
    print("Out of Stock.")
} catch VendingMachineError.insufficientFunds(let coinsNeeded) {
    print("Insufficient funds. Please insert an additional \(coinsNeeded)
coins.")
} catch {
    print("Unexpected error: \(error).")
}
// Prints "Insufficient funds. Please insert an additional 2 coins."
```

Generics

Generic code enables you to write flexible, reusable functions and types that can work with any type, subject to requirements that you define. You can write code that avoids duplication and expresses its intent in a clear, abstracted manner.

Generics

```
func swapTwoStrings(_ a: inout String, _ b: inout String) {
    let temporaryA = a
    a = b
    b = temporaryA
}

func swapTwoDoubles(_ a: inout Double, _ b: inout Double) {
    let temporaryA = a
    a = b
    b = temporaryA
}

func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

Generics

```
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

Generics

```
var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)
// someInt is now 107, and anotherInt is now 3

var someString = "hello"
var anotherString = "world"
swapTwoValues(&someString, &anotherString)
// someString is now "world", and anotherString is now "hello"
```

Equatable

A type that can be compared for value equality.

Equatable

```
let students = ["Kofi", "Abena", "Efua", "Kweku", "Akosua"]

let nameToCheck = "Kofi"
if students.contains(nameToCheck) {
    print("\(nameToCheck) is signed up!")
} else {
    print("No record of \(nameToCheck).")
}
```

Generics / Equatable

```
func find<T: Equatable>(item:T, inArray:[T]) -> Int? {
    var index:Int = 0
    var found = false

    while (index < inArray.count && found == false) {
        if item == inArray[index] {
            found = true
        } else {
            index = index + 1
        }
    }

    if found {
        return index
    } else {
        return nil
    }
}
```

Generics

```
let myFriends:[String] = ["John", "Dave", "Jim", "Arthur", "Lancelot"]

let findIndexOfFriend = find(item: "John", inArray: myFriends)
// returns 0

let findIndexOfFriend1 = find(item: "Arthur", inArray: myFriends)
// returns 3

let findIndexOfFriend2 = find(item: "Guinevere", inArray: myFriends)
// returns nil
```

Equatable

```
class Person: Equatable {
    var name:String
    var weight:Int
    var sex:String

    init(weight:Int, name:String, sex:String) {
        self.name = name
        self.weight = weight
        self.sex = sex
    }

    static func == (lhs: Person, rhs: Person) -> Bool {
        if lhs.weight == rhs.weight &&
            lhs.name == rhs.name &&
            lhs.sex == rhs.sex {
            return true
        } else {
            return false
        }
    }
```

Generics / Equatable

```
let Joe = Person(weight: 180, name: "Joe Patterson", sex: "M")
let Pam = Person(weight: 120, name: "Pam Patterson", sex: "F")
let Sue = Person(weight: 115, name: "Sue Lewis", sex: "F")
let Jeb = Person(weight: 180, name: "Jeb Patterson", sex: "M")
let Bob = Person(weight: 200, name: "Bob Smith", sex: "M")

let myPeople = [Joe, Pam, Sue, Jeb]

let indexOfOneOfMyPeople = find(item: Jeb, inArray: myPeople)
// returns 3 from custom generic function

let indexOfBob = myPeople.index(of: Bob)
// returns nil because Bob isn't in the array
```

Hashable

A type that can be hashed into a Hasher to produce an integer hash value.

Hashable

```
class Person: Equatable, Hashable {
    var name:String
    var weight:Int
    var sex:String

    init(weight:Int, name:String, sex:String) {
        self.name = name
        self.weight = weight
        self.sex = sex
    }

    static func == (lhs: Person, rhs: Person) -> Bool {
        if lhs.weight == rhs.weight &&
            lhs.name == rhs.name &&
            lhs.sex == rhs.sex {
            return true
        } else {
            return false
        }
    }

    func hash(into hasher: inout Hasher) {
        hasher.combine(name)
        hasher.combine(weight)
        hasher.combine(sex)
    }
}
```

Hashable

```
let myPeople2 = [Joe, Pam, Sue, Jeb, Pam, Joe]
print(myPeople2)
// Will print 6 items

let mySetPeople = Set(myPeople2)
print(mySetPeople)
// Will print 4 items, removing the duplicates
```

Project

Criar um app de compra, onde exista Usuários, Maquina de Cartão (ou Checkout), Cartão de Credito e Histórico, onde o usuário comprará na Maquina de Cartão usando seu cartão de crédito e acrescentará no histórico de compras dele, que terá o status de compra (Concluido, Pagamento negado, Cancelado).

Project

- Criar novo projeto no Xcode;
- Criar 3 ou + classes/models com parâmetros e funções;
- Uma model ao menos esteja relacionada com a outra;
- Uma model com o init com Throw;
- Uma model com Hashable e Equatable;
- Criar um protocolo para sua model e uma extensão dele;
- Usar enum e closure em alguma model;
- Criar 3 controllers e aplicar as funções de cada model em cada uma controller;
- Organizar projeto com MVC;
- Rodar projeto no simulador;

Dúvidas?

matheus@mocka.email
51 9 9388.5121

http://bit.do/cursoiOS