에러처리

강사 주영민

예외처리

- 프로그램의 오류 조건에 응답하고 오류 조건에서 복구하는 프로세스입니다

- Swift는 런타임시 복구 가능한 오류를 던지고, 포착하고, 전파하고, 조작하는 기능을 제공합니다.

- 에러는 Error 프로토콜을 준수하는 유형의 값으로 나타냅니다. 실제로 Error프로토콜은 비어 있으나 오류를 처리할수 있는 타입임을 나타냅니다.

에러 발생 메소드 만들기 순서

1. Error Type 만들기

    - Error protocol을 채택한 Enum만들기

2. 에러전달 가능 메소드 만들기

    - Return type앞에 throws 키워드 작성

3. 에러 상황일때 에러 발생

    1. throw 키워드를 사용 에러 반환

에러 타입 만들기

- Error Protocol을 채택해서 Error Type 만들기

```
enum ErrorEnum:Error
{
    case noDataError
    case networkError
    case unknowError
}
```

에러전달

- 함수의 작성 중 에러가 발생할수 있는 함수에는 매개변수 뒤에 throws 키워드를 작성하여 에러가 전달될수 있는 함수를 선언 합니다.

```
//에러전달 가능성 함수
func canThrowErrors() throws -> String
```

```
//에러전달 가능성이 없는 함수
func cannotThrowErrors() -> String
```

에러발생

- throw 키워드를 통해 에러를 발생 시킬수 있다.

```
func canThrowErrors(inputNum:String?) throws -> Int
{
    guard let numStr = inputNum  else {
        throw ErrorEnum.noDataError
    }
    guard let num = Int(numStr) else {
        throw ErrorEnum.unknowError
    }
    if num < 0
    {
        throw ErrorEnum.minusError
    }
    return num
}
```

에러처리

- 함수가 에러를 throw하면 프로그램의 흐름이 변경되므로 에러가 발생할 수있는 코드의 위치를 신속하게 식별 할 수 있어야합니다.

- do-catch 을 통한 에러 처리

- Converting to Optional Value

do - catch

```
do {
    try expression
    statements
} catch pattern 1 {
    statements
} catch pattern 2 {
    statements
}
```

- pattern에서 모든 에러처리를 위한 패턴 처리가 필요하다.

예제

Type1

```
do {
    let num = try canThrowErrors(inputNum: "3")
} catch  {
    //모든 에러 처리
    print(“error")
}
```

Type2

```
do {
    let num = try canThrowErrors(inputNum: "-3")
} catch let error {
    //모든 에러에 대한 처리
    print(error)
}
```

예제

Type3

```
do {
    let num = try canThrowErrors(inputNum: "3")
} catch ErrorEnum.unknowError {
    //모르는 에러에 대한 처리
} catch ErrorEnum.noDataError {
    //없는 데이터에 대한 처리
} catch ErrorEnum.minusError {
    //음수에 대한 처리
} catch
{
    //나머지 에러에 대한 처리
}
```

Type4

```
do {
    let num = try canThrowErrors(inputNum: "3")
} catch let error as ErrorEnum{
    //ErrorEnum 타입의 에러 처리
} catch
{
    //나머지 에러에 대한 처리
}
```

Converting Errors to Optional Value

- 에러를 옵셔널 Value로 여겨서 처리할수 있다.

```
if let num = try? canThrowErrors(inputNum: “-3")
{
    //에러가 발생되지 않을때 실행
}
```

Specifying Cleanup Actions (후처리)

- 에러에 의해 함수의 문제가 생기더라도 꼭! 해야할 행동이 있다면!!
- defer 구문은 블럭이 어떻게 종료되던 꼭 실행된다는 것을 보장.

```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }

        while let line = try file.readline() {
            // Work with the file.
        }
        // close(file) is called here, at the end of the scope.
    }
}
```

Notification

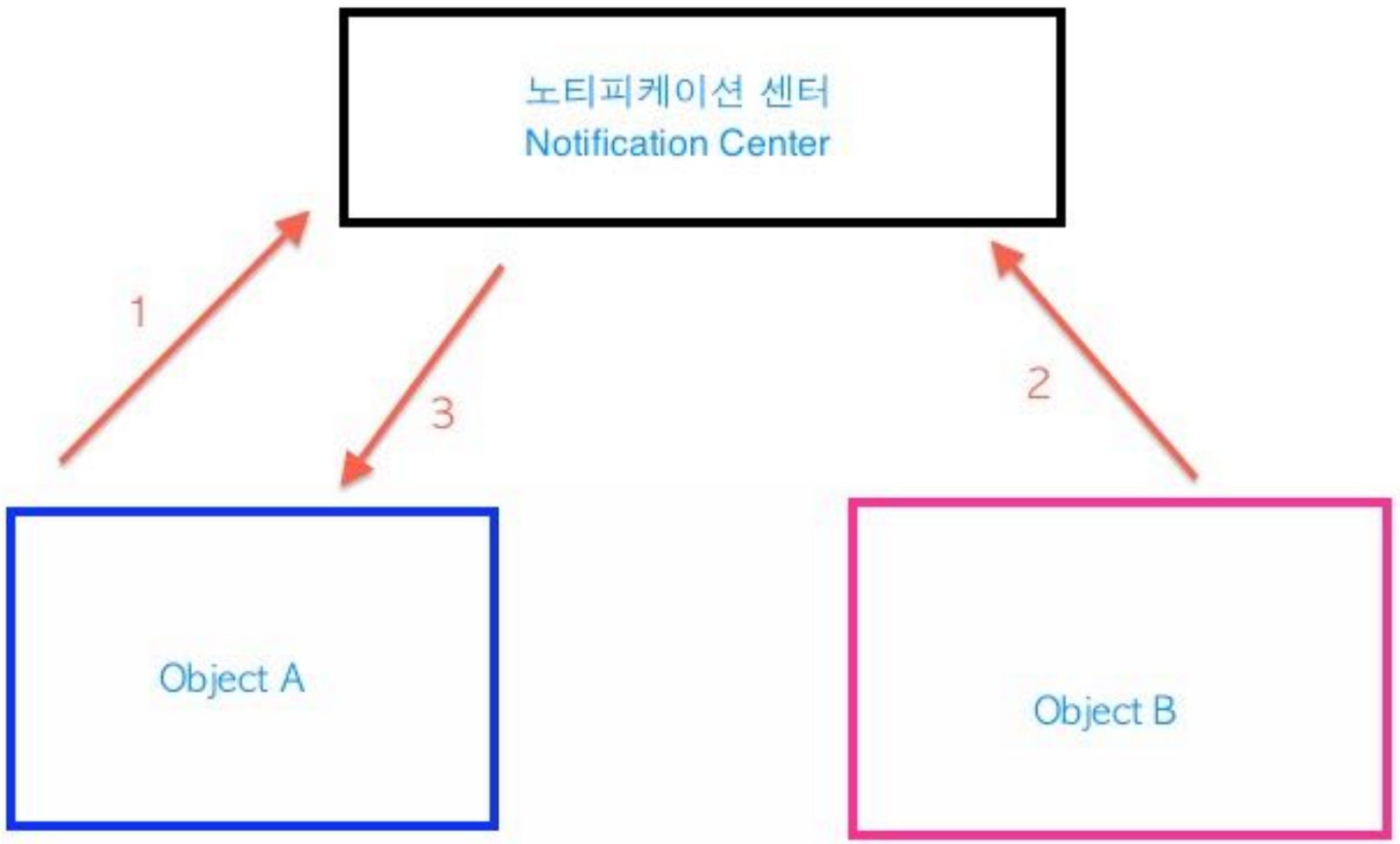
강사 주영민

Fast campus

NotificationCenter

- 특정 이벤트가 발생 하였음을 알리기 위해 불특정 다수의 객체에게 알리기 위해 사용하는 클래스
- 어떤 객체라도 특정 이벤트가 발생했다는 알림을 받을 것이라고 관찰자(Observer)로 등록을 해두면 노티피케이션 센터가 모든 관찰자 객체에게 알림을 준다

Notification 구조



1. 객체A가 노티피케이션 센터에 자신이 노티피게이션을 받을 것이라고 등록.(addObserver)

2. 객체B가 필요한 시점에 노티피케이션 송출 (postNotification)

3. 노티피케이션 센터에서 적절한 객체와 메소드를 찾아 호출

Notification 주요 Method

- Initializing

```
open class var `default`: NotificationCenter { get }
```

- Add Observer

```
open func addObserver(_ observer: Any,
                      selector aSelector: Selector,
                          name aName: NSNotification.Name?,
                        object anObject: Any?)
```

```
open func addObserver(forName name: NSNotification.Name?,
                                object obj: Any?,
                                     queue: OperationQueue?,
                                using block: @escaping (Notification) -> Swift.Void)
                                -> NSObjectProtocol
```

- Post Notification

```
open func post(name aName: NSNotification.Name,
                 object anObject: Any?,
               userInfo aUserInfo: [AnyHashable : Any]? = nil)
```

- Remove Observer

```
open func removeObserver(_ observer: Any)
```

예제

Observer

```
let notiCenter = NotificationCenter.default
notiCenter.addObserver(forName:Notification.Name(rawValue:"keyName"),
                       object: nil,
                        queue: nil)
{ (noti) in

    //노티가 왔을때 실행될 영역

}
```

.....

Poster

```
func postNoti() {
    NotificationCenter.default.post(name:
NSNotification.Name(rawValue: "key"), object: nil)
}
```

Fast campus

- 한번 만들어 볼까요?

System Notification

Observer

```
func observerNoti(noti:Notification){
        NotificationCenter.default.addObserver(self,
                    selector: #selector(self.trakingPost(noti:)),
                        name: Notification.Name.UIKeyboardWillShow,
                      object: nil)
}
```

```
func trakingPost(noti:Notification)
{
    //noti 내용
}
```

Poster

키보드가 올라올때 시스템에서 자동으로 Noti를 post해준다.

코드 확장

Subscript

Subscript

- 클래스, 구조체, 열거형의 collection, list, sequence의 멤버에 접근 가능한 단축문법인 Subscript를 정의 할수 있다.
- Subscript는 별도의 setter/getter없이 index를 통해서 데이터를 설정하거나 값을 가져오는 기능을 할 수 있다.
- Array[index] / Dictionary[“Key”] 등의 표현이 Subscript이다.

문법

```
subscript(index: Type) -> Type {
    get {
        // return an appropriate subscript value here
    }
    set(newValue) {
        // perform a suitable setting action here
    }
}
```

.....

```
subscript(index: Type) -> Type {
    // return an appropriate subscript value here
}
```

*연산 프로퍼티와 문법이 같음

예제 - Array

```
class Friends {
    private var friendNames:[String] = []

    subscript(index:Int) -> String
    {
        get {
            return friendNames[index]
        }
        set {
            friendNames[index] = newValue
        }

    }
}


let fList = Friends()
fList[0] = "joo"
```

예제 - struct

```
struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}
```

```
let threeTimesTable = TimesTable(multiplier: 3)
print("six times three is \(threeTimesTable[6])")
```

예제 - 다중 parameter

```
struct Matrix {
    let rows: Int, columns: Int
    var grid: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(repeating: 0.0, count: rows * columns)
    }

    subscript(row: Int, column: Int) -> Double {
        get {
            return grid[(row * columns) + column]
        }
        set {
            grid[(row * columns) + column] = newValue
        }
    }
}

var metrix = Matrix(rows: 2, columns: 2)
metrix[0,0] = 1
metrix[0,1] = 2.5
```

Extension

Extensions

- Extensions 기능은 기존 클래스, 구조, 열거 형 또는 프로토콜 유형에 새로운 기능을 추가합니다
- Extensions으로 할수 있는것은…

1. Add computed instance properties and computed type properties
2. Define instance methods and type methods
3. Provide new initializers
4. Define subscripts
5. Define and use new nested types
6. Make an existing type conform to a protocol

문법

```
extension SomeType {
    // new functionality to add to SomeType goes here
}
```

```
extension SomeType: SomeProtocol, AnotherProtocol {
    // implementation of protocol requirements goes here
}
```

유형 : Compute Properties

```
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"
```

유형 : init

```
extension Rect {
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size:
size)
    }
}
```

유형 : method

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


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

유형 : mutating method

```
extension Int {
    mutating func square() {
        self = self * self
    }
}

var someInt = 3
someInt.square()
```

유형 : Subscript

```
extension Int {
    subscript(digitIndex: Int) -> Int {
        var decimalBase = 1
        for _ in 0..<digitIndex {
            decimalBase *= 10
        }
        return (self / decimalBase) % 10
    }
}

746381295[0]
// returns 5
746381295[1]
// returns 9
746381295[2]
// returns 2
746381295[8]
// returns 7
```

Generic

Generic

- 어떤 타입에도 유연한 코드를 구현하기 위해 사용되는 기능
- 코드의 중복을 줄이고, 깔끔하고 추상적인 표현이 가능하다.

왜 Generic을 사용하는가?

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

두 Int를 받아 서로 바꿔주는 스왑함수가 있다.
우리는 Int 외에도 Double, String 등 다양한 타입의 데이터를 스왑하고 싶다면 어떻게 해야될까?

Generic을 사용한 swap함수

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

```
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"
```

Framework확인

- Array / Dictionary 파일 확인하기

Type Parameters

- 제넥릭에 사용된 “T”는 타입의 이름으로 사용되었다기 보다는 placeholder 역할로 사용되었다.

- 타입은 꺾쇠<> 로 감싸 표시한다.

- 타입의 이름은 보통 사용되는 속성에 맞게 네이밍 할수 있으나 아무런 연관이 없는 타입의 경우에는 T,U,V 같은 알파벳으로 사용된다.

Generic만들기 : Stack