Class & Struct (3)
Enum

G4ENG

15 JUL 19

Agenda

- init( )
- Optional chain
- Enumeration

init( )

- init 초기화 메소드 형식

- init(<매개변수>: <타입>, <매개변수>: <타입>, …) {
  1. 매개변수의 초기화
  2. 인스턴스 생성 시 기타 처리할 내용
  }

init( )

- 초기화 메소드의 특성

    1. 초기화 메소드의 이름은 init으로 통일

    2. 매개변수의 이름, 개수, 타입을 임의로 정의할 수 있다

    3. 매개변수의 이름과 개수, 타입이 서로 다른 여러 개의 초기화 메소드를 정의할 수 있다

    4. 정의된 초기화 메소드는 직접 호출되기도 하지만, 대부분 인스턴스 생성 시 간접적으로 호출된다

init( )

```
struct Resolution2 {
    var width = 0, height = 0

    init(width: Int) {
        self.width = width
    }
}

class VideoMode2 {
    var resoltuion = Resolution2(width: 2048)
    var interlaced = false
    var frameRate = 0.0
    var name: String?

    init(interlaced: Bool, frameRate: Double) {
        self.interlaced = interlaced
        self.frameRate = frameRate
    }
}
```

init( )

```
let resolution = Resolution2.init(width: 4096)
let videoMode = VideoMode2.init(interlaced: true, frameRate: 40.0)
// 구조체나 클래스의 인스턴스 생성할 때에는 init 메소드를 호출해주는 것이 원칙. 한편, 호출 시 init 메소드 생략 가능. 인스턴스 생성 구문은 init 메소드가 생략된 형태
let resolu = Resolution2(width: 2048)
let videMo = VideoMode2(interlaced: true, frameRate: 40.0)
```

init( )

- 초기화 구문 오버라이딩
  - 클래스에서는 초기화 구문도 일종의 메소드이므로, 자식 클래스에서 오버라이딩 가능
  - 마찬가지로 override 붙임
  - 기본 초기화 구문 init()은 부모 클래스에서 명시적으로 선언된 적이 없더라도 이를 상속받은 자식 클래스에서는 반드시 오버라이딩 형식으로 작성해야 한다

init( )

```
class Base {
    var baseValue: Double
    init(inputValue: Double) {
        self.baseValue = inputValue
    }
}

class ExBase: Base {
    override init(inputValue: Double) {
        super.init(inputValue: 10.5)
    }
}
```

init( )

- 초기화 구문 델리게이션
  - 상위 초기화 구문의 호출이 연속으로 이어지면 최상위에 있는 초기화 구문까지 호출이 이어지면서, 모든 초기화 구문이 누락되는 일이 없이 실행됨
  - 이처럼 연쇄적으로 오버라이딩된 자식 클래스의 초기화 구문에 대한 호출이 발생하는 것을 초기화 구문 델리게이션이라 함
  - 부모 클래스에 초기화 구문 외에 다른 형식의 초기화 구문이 추가됐으면, 자식 클래스에서 기본 초기화 구문을 오버라이딩할 때 명시적으로 부모 클래스의 기본 초기화 구문을 호출해야 함

Optionial chain

- 옵셔널 타입의 문제점
  - 옵셔널이란 nil이 할당될 수 있다는 것
  - nil은 초기에 값 할당이 일어나지 않았을 때 대입되지만, 값을 처리하는 과정에서 오류가 발생했을 때 대입되는 값이기도 함
  - 그래서 if 구문을 써서 바인딩을 하는게 코드가 길어짐
  - -> 그래서 간단하게 ? 연산자 이용하자

Optionial chain

```swift
struct Human {
    var name: String?
    var man: Bool = true
}

var boy: Human? = Human(name: "홍길동", man: true)
// 구조체를 인스턴스로 생성하여 변수에 할당하되 옵셔널 타입의 변수 boy에 할당하고 있음. 일단 옵셔널 타입으로 선언된 이상, 변수 boy를 사용하려면 옵셔널 타입에 대한 안전성 검사가 필요하다. boy 인스턴스부터 name 프로퍼티를 참조하려면 이
    역시 옵셔널 타입이므로 다시 안전성 검사가 필요하다.
if boy != nil {
    if boy!.name != nil {
        print("\(boy!.name!)")
    }
}

// 비강제 해제 구문

if let b = boy {
    if let name = b.name {
        print("\(name)")
    }
}
```

Optionial chain

```swift
// 어느 방식이든 안전성을 담보하려면 if 구문의 처리를 피할 수 없다. 만약 Human 구조체를 다른 구조체나 클래스가 프로퍼티로 사용하되, 이를 옵셔널 타입으로 설정한다면 더 복잡해진다.

struct Company {
    var ceo: Human?
    var companyName: String?
}

var startup: Company? = Company(ceo: Human(name: "나대표", man: false), companyName: "루비페이퍼")

if let company = startup {
// 가장 먼저 해야할 일은 startup의 옵셔널 타입 해제
    if let ceo = company.ceo {
    // Company라는 상수에는 옵셔널이 해제된 Company 타입의 인스턴스가 들어있게 된다. 다음으로 이 Company를 사용하여 ceo 프로퍼티의 옵셔널을 해제
        if let name = ceo.name {
            // name역시 옵셔널이므로 해제 과정
            print("\(name)")
        }
    }
}
```

Optionial chain

```swift
// 옵셔널 체인은 프로퍼티뿐만 아니라 메소드에서도 사용 가능. 메소드에서는 주로 반환값이 구조체나 클래스, 또는 열거형 등으로 구성되어 그 내부에 있는 프로퍼티나 메소드를 사용해야 할 때 옵셔널 체인을 적절히 사용하면 효율적
struct Company1 {
    var ceo: Human?
    var companyName: String?
    func getCEO() -> Human? {
        return self.ceo
    }
}

// 앞에서 사용했떤 Company 구조체에 getCEO 메소드 추가. 이 메소드는 Human 타입의 값을 반환함. 다만 내부적으로 self.ceo 프로퍼티를 반환하는 만큼 그에 맞는 옵셔널 타입으로 반환되도록 정의되어 있음. 이 메소드를 거쳐 ceo의 name 값을 참조해보자

var someCompany: Company1? = Company1(ceo: Human(name: "팀 쿡", man: true), companyName: "Apple")

let name = someCompany?.getCEO()?.name
if name != nil {
    print("\(name!)")
}
```

Enumeration

- 하나의 주제로 연관된 데이터들이 구성되어 있는 자료형 객체
- 열거형에서 데이터들은 열거형 객체를 정의하는 시점에 함께 정의됨
- 데이터를 함부로 삭제하거나 변경할 수 없음

Enumeration

- 집단 데이터 사용보다 열거형을 사용하는게 좋은 조건
  - 원치 않는 값이 잘못 입력되는 것을 막고 싶을 때
  - 입력받을 값을 미리 특정할 수 있을 때
  - 제한된 값 중에서만 선택할 수 있도록 강제하고 싶을 때

Enumeration

```
enum 열거형 이름 {
    // 열거형의 멤버 정의
    case 멤버값 1
    case 멤버값 2
    case ...
}

// 한줄로 정의 가능
enum 열거형 이름 {
    case 멤버값 1, 멤버값 2, ...
}
```

Enumeration

```
enum Direction {
    case east, west, south, north
}
```

Enumeration

```
enum Direction {
    case east, west, south, north
}
```

```
var directionToHead = Direction.west
```

Enumeration

```
enum Direction {
    case east, west, south, north
}
```

```
var directionToHead = Direction.west
```

```
directionToHead = .east
```

Enumeration

```
enum Direction {
    case east, west, south, north
}
```

```
var directionToHead = Direction.west
```

```
directionToHead = .east
```

```
var directionToBottom: Direction = .south
```

Enumeration

```
switch 비교 대상 {
    case 열거형.멤버1 :
        // 실행할 구문
    case 열거형.멤버2 :
        // 실행할 구문
}
```

Enumeration

```
switch directionToHead {
case .north:
    print("북쪽입니다")
case .south:
    print("남쪽입니다")
case .west:
    print("서쪽입니다")
case .east:
    print("동쪽입니다")
}
```

Enumeration

- 멤버와 값 분리
  - 필요한 데이터 집합을 열거형의 멤버로 구성할 때, 데이터만으로도 의미 전달이 어려울 때도 있음
  - 주로 한눈에 구분하기 힘든 숫자들일때가 이에 해당

Enumeration

```
enum HTTPCode1: Int{
    case OK = 200
    case NOT_MODIFY = 304
    case NOT_FOUND = 404
    case SERVER_ERROR = 500
}
```

Enumeration

```
enum Rank: Int {
    case one = 1
    case two, three, four, five
}

Rank.one.rawValue                   1
Rank.two.rawValue                   2
Rank.four.rawValue                  4
```

Enumeration

```
enum ImageFormat {
    case JPEG
    case PNG(Bool)
    case GIF(Int, Bool)
}

var newImage = ImageFormat.PNG(true)
newImage = .GIF(256, false)
```

Enumeration

```swift
enum HTTPCode: Int {
    case OK = 200
    case NOT_MODIFY = 304
    case NOT_FOUND = 404
    case SERVER_ERROR = 500

    var value: String {
        return "HTTPCode number is \(self.rawValue)"
    }
    // 연산 프로퍼티

    func getDescription() -> String {
        switch self {
        case .OK:
            return "Success. HTTP Code is \(self.rawValue)"
        case .NOT_MODIFY:
            return "Not modify. code is \(self.rawValue)"
        case .NOT_FOUND:
            return "Not found. code is \(self.rawValue)"
        case .SERVER_ERROR:
            return "Server Error. code is \(self.rawValue)"
        }
    }
    // 메소드

    static func getName() -> String {
        return "This Enumeration is HTTPCode"
    }
    // 타입 메소드
}

var response = HTTPCode.OK
response = .NOT_MODIFY

response.value
response.getDescription()
HTTPCode.getName()
```

Enumeration

Enumeration

UIImagePickerControllerSourceType

The source to use when picking an image or when determining available media types.

Declaration

```
typedef enum UIImagePickerControllerSourceType : NSInteger {
    ...
} UIImagePickerControllerSourceType;
```

Overview

A given source may not be available on a given device because the source is not physically present or because it cannot currently be accessed.

Enumeration

```
enum UIImagePickerControllerSourceType: NSInteger {
    case photoLibrary = 0
    case camera = 1
    case savedPhotosAlbum = 2
}
```