CHAPTER 8

옵셔널

옵셔널Optionals은 1장에서 언급한 스위프트의 특징 중 하나인 안전성Safe을 문법으로 담보하는 기능입니다. C 언어 또는 Objective-C에서는 찾아볼 수 없었던 콘셉트이기도 합니다.

옵셔널은 단어 뜻 그대로 '선택적인', 즉 값이 '있을 수도, 없을 수도 있음'을 나타내는 표현입니다. 이는 '변수나 상수 등에 꼭 값이 있다는 것을 보장할 수 없다. 즉, 변수 또는 상수의 값이 `nil`*일 수도 있다'는 것을 의미합니다. 라이브러리의 API 문서를 작성하거나 읽어본 사람은 문서에 `It can be NULL` 또는 `It can NOT be NULL` 등의 부연설명을 본 적이 있을 겁니다. 그리고 전달인자로 `NULL`이 전달되어도 되는지 문서를 보기 전에는 알 수가 없습니다. 그러나 스위프트에서는 옵셔널 하나만으로도 이 의미를 충분히 표현할 수 있기 때문에 (문서에 명시하지 않아도) 문법적 표현만으로 모든 의미를 전달할 수 있습니다. 게다가 옵셔널과 옵셔널이 아닌 값은 철저히 다른 타입으로 인식하기 때문에 컴파일할 때 바로 오류를 걸러낼 수 있습니다.

8.1 옵셔널 사용

데이터 타입(83쪽)에서 `nil`을 언급했지만, 아직 한 번도 `nil`을 사용한 적이 없습니다. 혹시 이미 `nil`을 할당해본 독자가 있나요? 그랬다면 아마 바로 컴파일 오류와 마주했을 겁니다. 그 이유는 옵셔널 변수 또는 상수가 아니면 `nil`을 할당할 수 없기 때문입니다. `Int` 타입의 변수

* NULL을 스위프트에서는 nil로 표기합니다.

에 0을 할당했다면 값이 없다는 의미인가요? 아닙니다. 0도 하나의 값입니다. ""로 **빈 문자열**을 만들었다면 이 또한 '빈 문자열'이라는 값이지, 값이 없는 것은 아닙니다. 변수 또는 상수에 정말 값이 없을 때만 nil로 표현합니다. 함수형 프로그래밍 패러다임에서 자주 등장하는 모나드Monad* 개념과 일맥상통합니다.

그래서 옵셔널의 사용은 많은 의미를 축약하여 표현하는 것과 같습니다. 옵셔널을 읽을 때 '해당 변수 또는 상수에는 값이 없을 수 있다. 즉, 변수 또는 상수가 nil일 수도 있으므로 사용에 주의하라'는 뜻으로 직관적으로 받아들일 수 있습니다. 값이 없는 옵셔널 변수 또는 상수에 (강제로) 접근하려면 런타임 오류가 발생합니다. 그렇게 되면 OS가 프로그램이 강제 종료시킬 확률이 매우 높습니다.

[코드 8-1]과 [코드 8-2]의 예를 통해 확인해보겠습니다.

코드 8-1 오류가 발생하는 nil 할당

```
var myName: String = "yagom"
myName = nil        // 오류!
```

nil은 옵셔널로 선언된 곳에서만 사용될 수 있습니다. 옵셔널 변수 또는 상수 등은 데이터 타입 뒤에 물음표(?)를 붙여 표현해줍니다.

코드 8-2 옵셔널 변수의 선언 및 nil 할당

```
var myName: String? = "yagom"
print(myName)   // yagom

myName = nil

print(myName)   // nil
```

사실 var myName: Optional<String>처럼 옵셔널을 조금 더 명확하게 써줄 수도 있습니다.** 그러나 물음표를 붙여주는 것이 조금 더 편하고 읽기도 쉽기 때문에 굳이 긴 표현을 사용하지는 않습니다.

---

* 모나드의 개념은 3부에서 다룹니다.

** 이는 제네릭을 사용한 문법으로 제네릭(392쪽)에서 확인할 수 있습니다.

옵셔널은 어떤 상황에 사용할까요? 왜 굳이 변수에 nil이 있음을 가정해야 할까요? 이 질문에 답할 수 있는 예로 우리가 만든 함수에 전달되는 전달인자의 값이 잘못된 값일 경우 제대로 처리하지 못했음을 nil을 반환하여 표현하는 것을 들 수 있습니다. 물론 기능상 심각한 오류라면 별도로 처리해야겠지만, 간단히 nil을 반환해서 오류가 있음을 알릴 수도 있습니다. 또는, 매개변수를 굳이 넘기지 않아도 된다는 뜻으로 매개변수의 타입을 옵셔널로 정의할 수도 있습니다. 스위프트 프로그래밍을 하면서 매개변수가 옵셔널일 때는 '아, 이 매개변수에는 값이 없어도 되는구나'라는 것을 API 문서를 보지 않고도 알아야 합니다. 이렇게 물음표 하나만으로 훌륭하고 암묵적인 커뮤니케이션을 완성했습니다.

[코드 4-19]는 100쪽에서 보았던 원시 값을 이용한 열거형 초기화 예제 코드입니다.

코드 4-19 원시 값을 통한 열거형 초기화

```
let primary = School(rawValue: "유치원")     // Primary
let graduate = School(rawValue: "석박사")    // nil

let one = Numbers(rawValue: 1)      // One
let three = Numbers(rawValue: 3)    // nil
```

[코드 4-19]에서 수상한 점을 혹시 발견했나요? 지금까지는 이 책 전반에 걸쳐 대부분 변수나 상수 뒤에 데이터 타입을 명시했습니다. 그러나 이 예제에서는 데이터 타입을 명시해주지 않고 타입 추론 기능을 사용했습니다. 왜 그랬을까요? nil을 할당하는 경우가 생기기 때문입니다. 컴파일러는 아마도 primary 및 graduate 상수의 데이터 타입을 School?이라고 추론했을 것입니다. 또, one과 three 상수의 데이터 타입은 Numbers?라고 추론했을 겁니다. 이때 원시 값이 열거형의 case에 해당하지 않으면 열거형 인스턴스 생성에 실패하여 nil을 반환하는 경우가 생깁니다. 앞에서 설명한 함수의 처리 실패 유형에 해당하는 것이죠.

옵셔널의 더 놀라운 점은 열거형으로 구현되어 있다는 점입니다. 옵셔널의 정의를 한 번 찾아봤습니다.

코드 8-3 옵셔널 열거형의 정의

```
public enum Optional<Wrapped> : ExpressibleByNilLiteral {
    case none
    case some(Wrapped)
```

```
    public init(_ some: Wrapped)
    /// 중략…
}
```

[코드 8-3]에서 옵셔널은 제네릭이 적용된 열거형입니다. ExpressibleByNilLiteral 프로토콜을 따른다는 것도 확인할 수 있습니다(제네릭이니 프로토콜이니 하는 용어는 뒤에서 배울 예정이니 걱정마세요). 여기서 알아야 할 것은 옵셔널이 값을 갖는 케이스와 그렇지 못한 케이스 두 가지로 정의되어 있다는 것입니다. 즉, nil일 때는 none 케이스가 될 것이고, 값이 있는 경우는 some 케이스가 되는데, 연관 값으로 Wrapped가 있습니다. 따라서 옵셔널에 값이 있으면 some의 연관 값인 Wrapped에 값이 할당됩니다. 즉, 값이 옵셔널이라는 열거형의 방패막에 보호되어 래핑되어 있는 모습이라는 겁니다.

옵셔널 자체가 열거형이기 때문에 옵셔널 변수는 switch 구문을 통해 값이 있고 없음을 확인할 수 있습니다.

코드 8-4 switch를 통한 옵셔널 값의 확인

```
func checkOptionalValue(value optionalValue: Any?) {
    switch optionalValue {
    case .none:
        print("This Optional variable is nil")
    case .some(let value):
        print("Value is \(value)")
    }
}

var myName: String? = "yagom"
checkOptionalValue(value: myName)  // Value is yagom

myName = nil
checkOptionalValue(value: myName)  // This Optional variable is nil
```

[코드 8-5]처럼 여러 케이스의 조건을 통해 검사하고자 한다면 더욱 유용하게 쓰일 수도 있습니다. 그럴 땐 세련되게 where 절과 병합해서 쓰면 더욱 좋습니다. where에 대해서는 where 절(447쪽)에서 조금 더 자세히 다루겠습니다.

코드 8-5 switch를 통한 옵셔널 값의 확인

```
let numbers: [Int?] = [2, nil, -4, nil, 100]

for number in numbers {
    switch number {
    case .some(let value) where value < 0:
        print("Negative value!! \(value)")
    case .some(let value) where value > 10:
        print("Large value!! \(value)")

    case .some(let value):
        print("Value \(value)")

    case .none:
        print("nil")
    }
}

// Value 2
// nil
// Negative value!! -4
// nil
// Large Value!! 100
```

그러나, 단 하나의 옵셔널을 switch 구문을 통해 매번 값이 있는지 확인하는 것은 매우 불편할 것입니다. 그래서 옵셔널 타입에서 값을 조금 더 안전하고 편리하게 추출하는 방법에 대해 알아보겠습니다.

8.2 옵셔널 추출

열거형의 some 케이스로 꼭꼭 숨어있는 **옵셔널의 값을 옵셔널이 아닌 값으로 추출**하는 옵셔널 추출 Optional Unwrapping 방법에 대해 알아보겠습니다.

8.2.1 강제 추출

옵셔널 강제 추출Forced Unwrapping 방식은 옵셔널의 값을 추출하는 가장 간단하지만 **가장 위험한 방법**입니다. 런타임 오류가 일어날 가능성이 가장 높기 때문입니다. 또, 옵셔널을 만든 의미가 무색해지는 방법이기도 합니다. 옵셔널의 값을 강제 추출하려면 옵셔널 값의 뒤에 느낌표(!)를 붙여주면 값을 강제로 추출하여 반환해줍니다. 만약 강제 추출 시 옵셔널에 값이 없다면, 즉 `nil`이라면 런타임 오류가 발생합니다.

코드 8-6 옵셔널 값의 강제 추출

```
var myName: String? = "yagom"

// 옵셔널이 아닌 변수에는 옵셔널 값이 들어갈 수 없습니다. 추출해서 할당해주어야 합니다.
var yagom: String = myName!

myName = nil
yagom = myName!     // 런타임 오류!

// if 구문 등 조건문을 이용해서 조금 더 안전하게 처리해볼 수 있습니다.
if myName != nil {
    print("My name is \(myName!)")
} else {
    print("myName == nil")
}
// myName == nil
```

런타임 오류의 가능성을 항상 내포하기 때문에 옵셔널 강제 추출 방식은 사용하는 것을 지양해야 합니다.

8.2.2 옵셔널 바인딩

[코드 8-6]에서 사용한 `if` 구문을 통해 `myName`이 `nil`인지 먼저 확인하고 옵셔널 값을 강제 추출하는 방법은 다른 프로그래밍 언어에서 `NULL` 값을 체크하는 방식과 비슷합니다. 앞서 설명한 것처럼 옵셔널을 사용하는 의미도 사라집니다. 그래서 스위프트는 조금 더 안전하고 세련된 방법으로 옵셔널 바인딩Optional Binding을 제공합니다.

옵셔널 바인딩은 옵셔널에 값이 있는지 확인할 때 사용합니다. 만약 옵셔널에 값이 있다면 옵셔널에서 추출한 값을 일정 블록 안에서 사용할 수 있는 상수나 변수로 할당해서 옵셔널이 아닌 형태로 사용할 수 있도록 해줍니다. 옵셔널 바인딩은 if 또는 while 구문 등과 결합하여 사용할 수 있습니다.

코드 8-7 옵셔널 바인딩을 사용한 옵셔널 값의 추출

```
var myName: String? = "yagom"

// 옵셔널 바인딩을 통한 임시 상수 할당
if let name = myName {
    print("My name is \(name)")
} else {
    print("myName == nil")
}
// My name is yagom

// 옵셔널 바인딩을 통한 임시 변수 할당
if var name = myName {
    name = "wizplan"   // 변수이므로 내부에서 변경이 가능합니다.
    print("My name is \(name)")
} else {
    print("myName == nil")
}
// My name is wizplan
```

[코드 8-7]의 예제에서는 if 구문을 실행하는 블록 안쪽에서만 name이라는 임시 상수를 사용할 수 있습니다. 즉, if 블록 밖에서는 사용할 수 없고 else 블록에서도 사용할 수 없습니다. 그렇기 때문에 위와 아래에서 모두 별도로 name을 사용했지만 충돌이 일어나지 않았습니다. 또, 상수로 사용하지 않고 변수로 사용하고 싶다면 if var를 통해 임시 변수로 할당할 수도 있습니다. [코드 8-7]에서는 if와 else 블록만을 사용했지만, else if 블록도 추가할 수 있습니다.

옵셔널 바인딩을 통해 한 번에 여러 옵셔널의 값을 추출할 수도 있습니다. 쉼표(,)를 사용해 바인딩 할 옵셔널을 나열하면 됩니다. 단, 바인딩하려는 옵셔널 중 하나라도 값이 없다면 해당 블록 내부의 명령문은 실행되지 않습니다.

코드 8-8 옵셔널 바인딩을 사용한 여러 개의 옵셔널 값의 추출

```
var myName: String? = "yagom"
var yourName: String? = nil

// friend에 바인딩이 되지 않으므로 실행되지 않습니다.
if let name = myName, let friend = yourName {
    print("We are friend!")
}

yourName = "eric"

if let name = myName, let friend = yourName {
    print("We are friend! \(name) & \(friend)")
}
// We are friend! yagom & eric
```

옵셔널 바인딩은 옵셔널 체이닝과 환상의 결합을 이룹니다. 옵셔널 체이닝은 옵셔널 체이닝과 빠른종료(261쪽)에서 다루겠습니다.

8.2.3 암시적 추출 옵셔널

때때로 nil을 할당하고 싶지만, 옵셔널 바인딩으로 매번 값을 추출하기 귀찮거나 로직상 nil 때문에 런타임 오류가 발생하지 않을 것 같다는 확신이 들 때* nil을 할당해줄 수 있는 옵셔널이 아닌 변수나 상수가 있으면 좋을 겁니다. 이때 사용하는 것이 바로 암시적 추출 옵셔널 Implicitly Unwrapped Optionals입니다. 옵셔널을 표시하고자 타입 뒤에 물음표(?)를 사용했지만, 암시적 추출 옵셔널을 사용하려면 타입 뒤에 느낌표(!)를 사용해주면 됩니다.

암시적 추출 옵셔널로 지정된 타입은 일반 값처럼 사용할 수 있으나, 여전히 옵셔널이기 때문에 nil도 할당해줄 수 있습니다. 그러나 nil이 할당되어 있을 때 접근을 시도하면 런타임 오류가 발생합니다. [코드 8-9]는 암시적 추출 옵셔널 변수에 nil이 할당되어 있을 때 사용하면 발생하는 오류 상황을 연출한 코드입니다.

* 예로 들었지만 매우 위험한 생각입니다. 실제 프로젝트용 프로그래밍 중에 오류가 생기지 않겠다라는 확신이 드는 순간은 드물겠죠.

코드 8-9 암시적 추출 옵셔널의 사용

```
var myName: String! = "yagom"
print(myName)   // yagom
myName = nil

// 암시적 추출 옵셔널도 옵셔널이므로 당연히 바인딩을 사용할 수 있습니다.
if var name = myName {
    print("My name is \(name)")
} else {
    print("myName == nil")
}
// myName == nil

myName.isEmpty   // 오류!!
```

옵셔널을 사용할 때는 강제 추출 또는 암시적 추출 옵셔널을 사용하기보다는 옵셔널 바인딩, `nil` 병합 연산자를 비롯해 뒤에서 배울 옵셔널 체이닝 등의 방법을 사용하는 편이 훨씬 안전합니다. 또한 이렇게 하는 편이 스위프트의 지향점에 부합합니다.