공식문서 읽기 : Enumerations

Enumerations - The Swift Programming Language (Swift 5.5)

An enumeration defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code. If you are familiar with C, you will know that C enumerations assign related names to a set of integer values.

https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html

Enumerations

열거형은 연관성이 있는 값들을 모아놓은 것을 말한다. Swift가 제공하는 열거형은 C언어보다 융통성이 있어 열거의 각 경우에 값을 꼭 제공할 필요는 없다. raw value로 알려진 각 케이스의 값은 String, Character, Int, Float, Double과 같은 값일 수 있다.

Swift의 열거형은 일급 객체이다. 또한 클래스와 비슷하게 프로퍼티를 계산하고 추가적인 정보를 처리하고 관련 기능을 제공하는 인스턴스 메서드를 지원한다. 생성자를 정의할 수 있고 원래 구현된 기능 이상의 것을 추가할 수 있도록 확장 기능도 제공한다. 프로토콜을 채택하는 것도 가능하다.

열거형은 값타입(Value Type)이다.

Enumeration Syntax

열거형을 선언할 때는 enum 키워드를 사용한다.

```
enum CompassPoint {
  case north
  case south
  case east
  case west
}
```

열거형의 이름은 CompassPoint, 열거 케이스는 north, south, east, west 이다. Swift에서는 C언어와 Objective-C와 다르게 기본적으로 정숫값이 설정되어 있지 않다.\

```
enum Planet {
  case mercury, venus, earth, mars, jupiter, saturn, uranos, neptune
}
```

위의 코드와 같이 한 줄에 나열하는 것도 가능하다.

열거형을 만들면 하나의 새로운 타입처럼 사용할 수 있다. 따라서 Swift의 명명 규칙에 따라 enum의 이름은 대문자로 시작해야한다.

```
var directionToHead = CompassPoint.west
```

위의 코드와 같이 열거형의 하나의 케이스에 접근하는 것이 가능하다. 또한 이제 directionToHead는 CompassPoint 타입이므로 다음과 같이 수정이 가능하다.

```
directionToHead = .east
```

Matching Enumeration Values with a Switch Statement

열거형은 switch 구문과 함께 사용하면 다양하게 활용할 수 있다.

```
directionToHead = .south
switch directionToHead {
  case .north:
    print("Lost of planets have a north")
  case .south:
    print("Watch out for penguins")
  case .east:
    print("Where the sun rises")
  case .west:
    print("Where the skies are blue")
}

// Prints "Watch out for penguins"
```

위 코드를 보면 directionToHead는 .south 케이스입니다. 해당 구문은 열거형의 모든 케이스를 가지고 있기 때문에 default는 사용하지 않았습니다. 만약 하나라도 빠진다면 default를 반드시 사용해야합니다.

Iterating over Enumeration Cases

열거형을 사용할 때 모든 케이스에 바로 접근할 수 있으면 편리할 것이다. 그래서 Swift는 이 기능을 제공한다. 하지만 열거형에서 기본적으로 제공하지는 않고, CaseIterable 프로토콜을 사용해야 이 기능을 사용할 수 있다.

```
enum Beverage: CaseIteravble {
  case coffee, tea, juice
}

let numberOfChoices = Beverage.allCases.count
print(numberOfChoices)
// Prints 3
```

위 코드와 같이 열거형에서 CaseIterable 프로토콜을 채택하고 allCases 프로퍼티를 사용하면 케이스들이 Array로 반환된다.

```
for beverage in Beverage.allCases {
  print(beverage)
}
// coffee
// tea
// juice
```

물론 배열처럼 for in loop에서도 사용할 수 있다.

Associate Values

지금까지의 예는 모두 열거형의 케이스가 명시적으로 정의되는 것이었다. 가끔은 다른 타입의 값과 열거형의 케이스를 함께 저장하는 것이 유용할 수 있다. 이렇게 열거형의 케이스와 저장되는 다른 값을 Associate Values라고 한다.

어떤 타입의 값도 열거형과 함께 사용될 수 있고 각 열거 케이스마다 다른 타입을 사용할 수 있다.

(discriminated unions, tagged unions, variants in other programming language)

예를들어 재고를 관리하는 시스템이 서로 다른 두 타입의 바코드(일반 바코드, QR코드)로 재고를 관리한다고 가정했을 때 일부 제품에는 UPC(일반 바코드)형태의 1D 바코드가 있고 0~9 까지의 숫자를 사용해 제조업체와 제품에 대한 코드숫자를 나타낸다고 가정했다.

또 어떠한 제품은 QR코드 타입의 2D 바코드로 표시되며 이는 IOS-8859-1 문자를 사용할 수 있고 최대 2953 길이의 문자열을 인코딩 가능하다고 가정했다.

즉 이처럼 다른 두 가지 타입의 케이스를 열거형에서 한 번에 다루려면 다음과 같이 나타낼 수 있다.

```
enum Barcode {
  case upc(Int, Int, Int, Int)
  case qrCode(String)
}
```

upc 바코드 케이스는 Int 타입을 네개 사용하는 튜플의 형태고, QR코드 케이스는 String 타입으로 열거 케이스를 나눌 수 있다. 각각의 타입마다 새로운 값을 생성할 수 있다.

```
var productBarcode = Barcode.upc(8, 85090, 51226, 3)
productBarcode = .qrCode("ABCDEFGHIJKLMNOP")
```

이렇게 되면 productBarcode 변수는 한 시점에 하나의 값만 저장할 수 있게 된다. 즉 upc 바코드 타입이나 QR코드 타입 중 하나만 저장할 수 있다는 의미이다.

```
switch productBarcode {
  case .upc(let numberSystem, let manufacture, let product, let check):
    print("UPC: \(numberSystem), \(manufacture), \(product), \(check).")
  case .qrCode(let productCode):
    print("QR code: \(productCode).")
}
```

즉 위와 같이 switch와 함께 사용해서 해당 값에 있는 값을 추출할 수 있다. 또한 여러값이 들어있는 associated value를 위해 각 값마다 변수 혹은 상수를 정해 이름을 붙여 사용할 수 있다.

Raw Value

위에서 본 Associated Value가 열거형 타입이 여러 타입으로 이루어진 값을 저장하는 방법이라면, raw Value는 기본값을 미리 선언할 수 있는 타입이다.

```
enum ASCIIControlCharacter: Character {
  case tab = "\t"
  case lineFeed = "\n"
  case carriageReturn = "\r"
}
```

위 코드에서 ASCIIControlCharacter 열거형은 Character 타입으로 정의된다. Raw Value는 위와 같이 Character일 수도 있고, Int, Float, Double, String 타입이 될 수도 있다. 하지만 각 케이스마다 raw Value는 고유한 값이어야 한다.

Raw Value는 Associated Value와는 다르다. Raw Value는 열거형을 처음 정의할 때 미리 채워진 값으로 설정하는 것이고 Associated Value는 타입만 정해두고 나중에 채우는 것이다. 즉 Raw Value는 새로운 인스턴스가 모두 같은 값을 가지지만 Associated Value는 변할 수 있다.

Implicitly Assigned Raw Value

Int, String으로 열거형의 Raw Value를 선언하면 모든 케이스에 대해 Raw Value를 선언하지 않아도 된다. 물론 선언하면 그 값이 해당 케이스의 Raw Value가 되겠지만 선언하지 않으면 Swift가 값을 알아서 할당한다.

```
enum Planet {
  case mercury = 1, venus, earth, mars, jupiter, saturn, uranos, neptune
}
let earthsOrder = Planet.earth.rawValue
// earthsOrder: 3
```

예를 들어 위의 코드를 보면 Raw Value를 첫번째 케이스에만 1이라고 할당했다. 그 뒤 케이스에는 Raw Value를 선언하지 않았지만 swift에서 2, 3, 4, ... 이렇게 알아서 순차적으로 할당한다.

```
enum CompassPoint: String {
  case north, south, east, west
}
```

```
let sunsetDirection = CompassPoint.west.rawValue
// sumsetDirection: "west"
```

만약 위 코드처럼 아무것도 선언하지 않으면 케이스의 이름이 String 타입으로 Raw Value가 된다.

Initializing from a Raw Value

열거형의 새로운 인스턴스를 생성할 때 변수나 상수에 rawValue라는 매개변수로 초기화할 수 있는데, 이렇게 되면 Raw Value가 존재할 경우 해당 케이스의 값이 반환되고, 존재하지 않는 Raw Value라면 nil을 반환한다. 즉 Optional Type으로 반환된다는 말이다.

```
let possiblePlanet = Planet(rawValue: 7)
// type: Planet?
// return: Planet.uranus
```

위와 같이 값이 있다면 다행이지만 rawValue가 없는 경우 nil이 반환되기 때문에 이는 실패할 수 있는 초기화 방법이다. 따라서 if let 구문을 통한 옵셔널 바인딩을 사용해 안전하게 추출해야한다.

```
let positionToFind = 11
if let somePlanet = Planet(rawValue: positionToFind) {
  switch somePlanet {
    case .earth:
      print("Mostly harmless")
    default:
      print("Not a safe place for humans")
  }
} else {
  print("There isn't a planet at position \(positionToFind)")
}
```

위 코드처럼 11이라는 rawValue는 없기 때문에 else 블록을 수행한다.

Recursive Enumerations

Recursive Enumerations는 재귀 열거형으로 어떤 열거형의 케이스에 Assosiated value 타입으로 자신의 열거형 타입이 존재하는 경우를 말한다. 이렇게 자신의 열거형 타입을 사용하면 해당 케이스 앞에 indirect라는 키워드를 사용해야 한다.

```
enum ArithmeticExpression {
  case number(Int)
  indirect case addition(ArithmeticExpression, ArithmeticExpression)
  indirect case multiplication(ArithmeticExpression, ArithmeticExpression)
}
```

여러번 반복해서 inderect 키워드를 사용하고 싶지 않다면 아래와 같이 enum 앞에 작성해도 된다.

```
indirect enum ArithmeticExpression {
  case number(Int)
  case addition(ArithmeticExpression, ArithmeticExpression)
  case multiplication(ArithmeticExpression, ArithmeticExpression)
}
```

위 예제는 addition, multiplication 케이스에서 Recursive Enumerations이 사용됐고 해당 케이스에서는 associated value 타입으로 자신의 열거형 타입을 가지고 있다.

```
let five = ArithmeticExpression.number(5)
let four = ArithmeticExpression.number(4)

let sum = ArithmeticExpression.addition(five, four)
let product = ArithmeticExpression.multiplication(sum, ArithmeticExpression.number(2))
```

위와 같이 사용할 수 있다.

이를 실제 재귀 함수에서 사용할 수 있는데 예시는 아래와 같다.

```
func evaluate(_ expression: ArithmeticExpression) -> Int {
  switch expression {
    case let .number(value):
      return value
    case let .addition(left, right):
      return evaluate(left) + evaluate(right)
    case let .multiplication(left, right):
      return evaluate(left) * evaluate(right)
```

```
  }
}

print(evaluate(product))
// Prints "18"
```