




공식문서 읽기 : Structures and Classes

Structures and Classes - The Swift Programming Language (Swift 5.5)

Structures and classes are general-purpose, flexible constructs that become the building blocks of your program's code. You define properties and methods to add functionality to your structures and classes using the same syntax you use to define constants, variables, and functions.

 <https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html>

구조체와 클래스는 보통 프로그램 코드 블록을 유연성있게 구축하기 위해 사용한다. 상수, 변수, 함수를 정의하는 것과 같은 문법을 사용해 구조체와 클래스에 property, method를 정의한다.

다른 언어와 달리 Swift는 구조체와 클래스로부터 인터페이스와 구현을 분리하지 않아도 된다. 구조체 또는 클래스를 하나의 파일에 정의하면 Swift가 자동으로 해당 클래스와 구조체를 사용할 수 있는 인터페이스를 만들어준다.



클래스의 인터페이스는 객체로 알려져 있다. 하지만 Swift 구조체와 클래스는 다른 언어보다 더 기능성(functionality)에 가깝다.

Comparing Structures and Classes

구조체와 클래스의 공통점

- 값을 저장하기 위한 프로퍼티를 정의한다.
- 기능성을 제공하기 위한 메소드를 정의한다.
- subscript 문법을 사용하는 값에 접근하기 위한 subscript 문법을 정의한다.
- 초기화 상태를 설정하기 위한 Initializer를 정의한다.
- 기본적인 구현을 넘어서 기능을 확장한다.
- 특정한 종류의 표준 기능성을 제공하기 위한 프로토콜을 정한다.

클래스의 추가적인 기능(구조체에는 없는)

- 상속(inheritance): 한 클래스가 다른 클래스를 상속할 수 있다.
- 타입캐스팅(type casting): 런타임 시 클래스의 인스턴스의 타입을 확인하고 이해하기 위한 타입 캐스팅이 가능하다.
- 소멸자(deinitializer): 할당된 자원을 해제시킬 수 있다.
- 참조카운팅(reference counting): 클래스 인스턴스에 하나 이상의 참조를 가능케 한다.

클래스가 지원하는 추가적인 기능은 복잡성의 증가로 인한 비용 상승을 초래한다. 가이드라인에 따르면 사용하기 구조체의 사용이 더 쉽기 때문에 구조체의 사용을 더 선호한다. 클래스는 필요시에 사용한다. 대부분의 커스텀 데이터 타입은 구조체 혹은 열거형으로 정의된다는 뜻이다.

Definition Syntax

구조체와 클래스는 비슷하게 정의한다. struct 또는 class 키워드로 구조체나 클래스를 선언한다.

```
struct SomeStructure {  
  
}  
class SomeClass {  
  
}
```

구조체나 클래스를 정의할 때마다 새로운 Swift 타입이 정의된다. UpperCamelCase로 작성하는 것을 원칙으로 한다. 프로퍼티나 메소드는 lowerCamelCase로 선언한다.

```
struct Resolution {  
    var width = 0  
    var height = 0  
}  
  
class VideoMode {  
    var resolution = Resolution()  
    var interlaced = false  
    var frameRate = 0.0  
    var name: String?  
}
```

위 예제는 새로운 구조체인 화면의 해상도를 묘사하는 Resolution을 정의한다. 이 구조체는 width와 height라는 두 프로퍼티를 저장한다. 프로퍼티는 구조체 또는 클래스의 일부분으로 저장되고 묶이는 상수 또는 변수이다.

비디오 화면의 특정 모드를 묘사하는 VideoMode 클래스도 정의한다. 이 클래스는 네 개의 변수를 가지고 있다. resolution은 새로운 Resolution 구조체 인스턴스로 초기화된다.

Structure and Classes instance

Resolution 구조체 정의와 VideoMode 클래스 정의는 두 구조체/클래스가 어떻게 생겼는지만 나타낸다. 그들 자신은 특정한 해상도와 비디오 모드를 묘사하지 않는다. 이를 위해서는 새로운 인스턴스를 만들어야 한다.

인스턴스 생성 문법은 구조체와 클래스가 서로 비슷하다

```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

구조체와 클래스는 모두 새로운 인스턴스를 위한 initializer 문법을 사용한다. 간단한 형태는 구조체, 클래스의 타입 이름을 빈 소괄호와 함께 사용하는 것이다. 이는 새로운 인스턴스를 생성하며, 그 인스턴스의 모든 프로퍼티는 기본 값으로 초기화된다.

Accessing Properties

점문법을 사용해 인스턴스의 프로퍼티에 접근할 수 있다. 인스턴스 이름 뒤에 점과 프로퍼티 이름을 표기한다.

```
print("The width of someResolution is \(someResolution.width)")
// prints "The width of someResolution is 0"
```

위 예시에서 `someResolution.width`는 `someResolution`의 `width` 프로퍼티로 추론되며, 기본값인 0을 반환한다.

`VideoMode`의 `resolution` 프로퍼티 안 `width` 프로퍼티와 같이 subproperty에 깊게 들어갈 수 있다.

```
print("The width of someVideoMode is \(someVideoMode.resolution.width)")
// prints "The width of someVideoMode is 0"
```

변수 프로퍼티에 새로운 값을 할당하기 위해 점문법을 사용할 수 있다.

```
someVideoMode.resolution.width = 1280
print("The width of someVideoMode is \(someVideoMode.resolution.width)")
// prints "The width of someVideoMode is 1280"
```

Memberwise Initializers for Structure Type

모든 구조체는 자동으로 memberwise initializer를 생성한다. 이는 새로운 구조체 인스턴스의 멤버 프로퍼티를 초기화하는데 사용할 수 있다. 새로운 인스턴스의 프로퍼티에 대한 초기값은 이름에 의해 memberwise initializer에 들어간다.

```
let vga = Resolution(width: 640, height: 480)
```

구조체와 달리 클래스 인스턴스는 기본 멤버 생성자를 받지 않는다.

Structures and Enumerations Are Value Types

값 타입은 상수/변수에 할당되거나 함수에 들어갈 때 그 값이 복사되어 전달된다는 의미를 지닌다. Swift에서 모든 기본타입(정수, 실수, Boolean, 문자열, 배열, Dictionary)은 값 타입이고, 구조체로써 구현된다.

모든 구조체와 열거형 역시 값 타입이다. 이는 생성하는 모든 구조체와 열거형은 항상 복사되어 코드에 전달된다는 뜻이다.



Array, Dictionary, String과 같은 기본 라이브러리에 의해 정의된 컬렉션은 복사를 줄이기 위한 최적화를 실시한다. 즉시 복사본을 생성하는 것이 아니라 원본과 복사본이 메모리 공간을 공유하게 만든다. 만약 컬렉션의 복사본 중 하나가 수정될 경우, 수정 전에 그 원소는 복사된다.

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

Resolution은 구조체이기 때문에, 기존 인스턴스의 복사본이 만들어진다. 그 복사본이 cinema 변수에 할당된다. hd와 cinema가 같은 너비와 높이를 갖고 있을지라도 둘은 완벽히 다른 인스턴스이다.

cinema의 width 프로퍼티를 2048로 변경한 결과는 다음과 같다.

```
cinema.width = 2048
print("cinema is now \$(cinema.width) pixels wide")
// prints "cinema is now 2048 pixels wide"
print("hd is still \$(hd.width) pixels wide")
// prints "cinema is now 1920 pixels wide"
```

hd에 저장된 값은 새로운 cinema 인스턴스에 복사된다. 같은 숫자값을 가지는 두 개의 분리된 인스턴스가 만들어지는 것이다. 분리된 인스턴스이기 때문에 cinema의 width를 수정해도 hd의 width는 영향을 받지 않는다.

열거형에도 같은 원리가 적용된다.

```
enum CompassPoint {
    case north, south, east, west
    mutating func turnNorth() {
        self = .north
    }
}

var currentDirection = CompassPoint.west
let rememberedDirection = currentDirection
currentDirection.turnNorth()

print("The current direction is \$(currentDirection)")
// prints "The current direction is north"
print("The remembered direction is \$(rememberedDirection)")
// Prints "The current direction is west"
```

Classes Are Reference Types

값 타입과 달리, 참조 타입은 변수나 상수에 할당되거나 함수에 전달될 때 복사되지 않는다. 참조는 기존의 동일한 인스턴스를 활용한다.

```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0

let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

클래스는 참조타입이기 때문에, `tenEighty`와 `alsoTenEighty`는 같은 `VideoMode` 인스턴스를 참조한다. 하나의 같은 인스턴스를 위한 두 가지의 이름이 있는 것이다.

`tenEighty`의 `frameRate` 프로퍼티가 30.0으로 변한 것을 확인할 수 있다.

```
print("The frameRate property of tenEighty is now \"tenEighty.frameRate\"")
// prints "The frameRate property of tenEighty is now 30.0"
```

위 예시는 참조 타입이 추론하기 어렵다는 것을 보여준다. `tenEighty`와 `alsoTenEighty`가 지금 작성하는 코드에서 멀리 떨어져있다면 바뀐 `VideoMode`를 모두 확인하는 것은 어려운 일이 될 것이다. `tenEighty`를 사용중이더라도 `alsoTenEighty`를 사용하는 코드를 고려해야 한다. 반대의 경우도 마찬가지이다.

`tenEighty`와 `alsoTenEighty`를 상수로 선언했음에도 `tenEighty.frameRate`와 `alsoTenEighty.frameRate`를 변경할 수 있었다. `tenEighty`와 `alsoTenEighty` 상수 그 자체의 값은 사실상 바뀌지 않기 때문이다. `tenEighty`와 `alsoTenEighty`는 `VideoMode` 인스턴스를 저장하지 않는 `VideoMode` 인스턴스를 참조한다. `VideoMode`의 상수 레퍼런스 값이 아니라 `frameRate` 프로퍼티가 변하는 것이다.

Identity Operators

클래스는 참조 타입이어서 하나의 동일한 클래스 인스턴스를 여러 상수/변수가 참조하는 것이 가능하다.

구조체나 열거형은 항상 복사본을 저장하기 때문에 이와는 다르다.

두 상수/변수가 정확히 같은 클래스의 인스턴스를 참조하고 있는지를 확인하기 위해 두 식별 연산자를 사용한다.

- `===` : 동일할 경우(identical) 참
- `!==` : 동일하지 않을 경우(not identical) 참

```
if tenEighty === alsoTenEighty {
    print("tenEighty and alsoTenEighty refer to the same Video instance.")
}
// prints "tenEighty and alsoTenEighty refer to the same Video instance."
```

`===`, `==`가 같은 의미가 아니라는 것이 중요하다. `===`는 클래스 타입의 두 상수/변수가 정확히 같은 클래스 인스턴스를 참조하는지를 뜻한다. `==`는 두 인스턴스가 상등하거나 동등한지를 고려한다.

Pointers

C, C++, Objective-C는 메모리의 주소를 참조하기 위한 포인터를 사용한다. 몇몇 참조 타입의 인스턴스를 참조하는 Swift의 상수와 변수는 C의 포인터와 비슷하다. 하지만 메모리 주소에 대한 직접적인 포인터는 아니다. 새로운 레퍼런스를 생성한다는 것을 표시하기 위해 별표(*)를 작성할 필요도 없다. 대신 이러한 레퍼런스는 Swift의 다른 상수/변수와 같이 정의된다. 표준 라이브러리는 직접적으로 포인터와 상호작용하고자 할 경우 포인터와 버퍼 타입을 제공한다.

클래스와 구조체의 선택

클래스와 구조체 모두 프로그램의 코드를 조직화하고 특정 타입을 선언하는 데 사용된다. 그리고 앞서 설명했던 것처럼 클래스 인스턴스가 인자로 사용될 때는 참조가 넘어가고 구조체는 값이 넘어간다.

일반적으로 다음 중 1개 이상의 조건을 만족하면 구조체를 사용하는 것을 고려할 수 있다.

- 구조체의 주 목적이 관계된 간단한 값을 캡슐화(encapsulate)하기 위한 것인 경우
- 구조체의 인스턴스가 참조되기보다 복사되기를 기대하는 경우
- 구조체에 의해 저장된 어떠한 프로퍼티가 참조되기보다 복사되기를 기대하는 경우
- 구조체가 프로퍼티나 메소드 등을 상속할 필요가 없는 경우

예를 들면 다음과 같다. double형을 갖는 width와 height를 캡슐화해 특정 지형의 크기로 사용하는 경우, Int형을 갖는 start와 length를 캡슐화해 특정 값의 범위를 나타내는 경우, Double형으로 구성된 x,y,z를 캡슐화 해 3D 좌표 시스템의 point로 사용하는 경우