06. Swift 2.0으로 iOS 코딩하기 - Swift Tour (6)

Enumeration

Enumeration. 열거형 이라고 번역하고 많은 언어에서 `enum` 이라는 키워드를 쓰는 이것은 Swift 에서도 존재하지만, Swift 에서는 정말 많은 변화(추가?) 가 있습니다. 이것이 다른 언어의 `enum` 과 Swift 의 `enum` 을 완전히 다른 것으로 만들어 놓기도 하는데요, 이 추가사항은 뒤에 얘기하기로 하고 일단 기본문법부터 봅니다.

<a>

```
5  enum Rank: Int {
6      case Ace = 1
7      case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
8      case Jack, Queen, King
9      func simpleDesc() -> String {
10         switch (self) {
11         case .Ace: return "ace"                          "ace"
12         case .Jack: return "jack"
13         case .Queen: return "queen"
14         case .King: return "king"
15         default: return String(self.rawValue)            "5"
16         }
17     }
18 }
19
20 let ace = Rank.Ace                                      Ace
21 let rv = ace.rawValue                                   1
22 let five = Rank.Five                                    Five
23 ace.simpleDesc()                                       "ace"
24 five.simpleDesc()                                      "5"
25
```

기본문법부터 보자고 해 놓고 `enum` 에 함수를 추가했네요. 네, `class` 뿐 아니라 Swift 에서는 `enum` 도 함수를 가질 수 있습니다. `enum` 키워드 뒤에 일단 눈에 띄는 것은 `Int` 를 상속받는 것처럼 적은 것인데요, 이것은 `enum` 의 RawValue 의 타입을 지정하는 의미입니다. Raw Value 란 말그대로 `enum` 의 각 `case` 를 구별하기 위해 사용하는 내부 타입입니다. 다른 언어에서는 `enum` 에 `int` 만 사용하지만, Swift 에서는 `Int`, `Float`, `Double`, `String` 등을 사용할 수 있습니다. 생략하면 rawValue 를 가질 수 없습니다. `Int` 를 사용하는 경우 위의 Ace 처럼 값을 지정하면 이후의 값은 자동으로 증가하여 매겨집니다.

`case` 는 한 줄에 comma (`,`) 로 구별해서 써도 되고, 여러 줄에 걸쳐서 적어도 됩니다. 위의 예제에서는 함수가 추가된 것 말고 특이한 것은 별로 없어보이네요. 그런데 라인 11 이후의 `switch`/`case` 부분을 보면 `Rank.Ace` 로 쓰지 않고 그냥 `.Ace` 로 쓴 것을 볼 수 있습니다. Swift 에서는 `enum` 을 쓸 때 컴파일러가 해당 `enum` type 이 쓰여야 하는 것을 알 수 있으면 type 이름을 생략하고 dot (`.`) 부터 시작해도 됩니다. 다음 코드를 보죠.

<b>

```
26  var six = Rank.Six                          Six
27  six.simpleDesc()                            "6"
28  six = .Five                                 Five
29  six.simpleDesc()                            "5"
30
```

코드가 조금 말이 되지는 않지만 라인 28 를 보면 Rank 를 생략한 것을 볼 수 있습니다. 라인 27 에서는 six 의 type 을 알지 못하기 때문에 우변에 Rank.Six 로 써야 하지만 라인 29 에서는 컴파일러는 이미 six 의 type 이 Rank 인 것을 알고 있기 때문에 .Five 만 써도 됩니다. 위의 코드 <a> 에서도 라인 10 의 switch (self) 를 보고 컴파일러는 case 들에 나올 값이 Rank 타입인 것을 알아차리게 됩니다. 따라서 case 에서 .Ace, .Jack 처럼 쓸 수 있게 됩니다. 이것은 var 로 선언된 변수에 값을 대입하거나 function argument 를 전달할 때도 마찬가지가 됩니다.

자 이제 Swift enum 의 강력한 기능에 대한 소개가 나갑니다. 바로 value association 입니다.

<c>

```
5   enum ServerResponse {
6       case Result(String, String)
7       case Error(String)
8   }
9
10  let success = ServerResponse.Result("Kim", "Seoul")    Result("Kim", "Seoul")
11  let failure = ServerResponse.Error("Invalid name")      Error("Invalid name")
12
13  func desc(sr: ServerResponse) -> String {
14      switch (sr) {
15      case let .Result(name, city):
16          return "Name: \(name) City: \(city)"            "Name: Kim City: Seoul"
17      case let .Error(cause):
18          return "Error: \(cause)"                        "Error: Invalid name"
19      }
20  }
21
22  desc(success)                                           "Name: Kim City: Seoul"
23  desc(failure)                                           "Error: Invalid name"
24
```

Swift enum 의 각 case 는 어떤 값(들) 을 연동시킬 (associate) 수 있습니다. 그리고 그 값들은 case 별로 다른 종류와 개수일 수 있다는 점에서 매우 강력합니다. 라인 5 부터 정의하는 ServerResponse 라는 enum type 은 다른 언어에서의 enum 처럼 결과 혹은 에러 값을 가집니다. 그런데 단순히 [결과가 있음] 과 [에러] 의 상태만 가지는 것이 아니라 각 상태에 따른 값도 포함할 수 있다는 것이죠. 즉 .Result 일 때는 결과값 String 두 개가 매달리고, .Error 일 때는 원인을 나타내는 String 한 개를 매달아 둘 수 있습니다.

코드 <c> 는, Value-associated enum 의 값을 switch/case 에서 사용하는 예제가 되겠습니다. 라인 42 에서 보면 .Result 일 경우 매달려 있는 두 개의 String 을 각각 name 과 city 로 이름을 주어 접근할 수 있도록 하고 있습니다. 조금만 생각을 해 보시면, 연동된 값을 접근할 수 있는 곳은 switch/case 에서뿐이라는 것을 알 수 있습니다. 연동된 값에 접근이 가능하려면 해당 case 여야만 하기 때문입니다.

Struct

이번엔 Struct 를 살펴보겠습니다. 다른 언어에서의 `struct` 와 비슷합니다만, Swift 에서의 `struct` 는 `class` 와 거의 같고 한 가지 특성이 다릅니다. 바로 `class` 는 Reference Type 이고 `struct` 는 Value Type 이라는 점입니다. 그 외에는 상수/변수/함수/생성자 등등 거의 모든 속성이 동일합니다.

그렇다면 Value Type 과 Reference Type 이 어떻게 다른지를 이해하면 `struct` 에 대한 이해를 하는 것이라고 볼 수 있겠네요. 다음 코드를 보겠습니다.

<d-1>

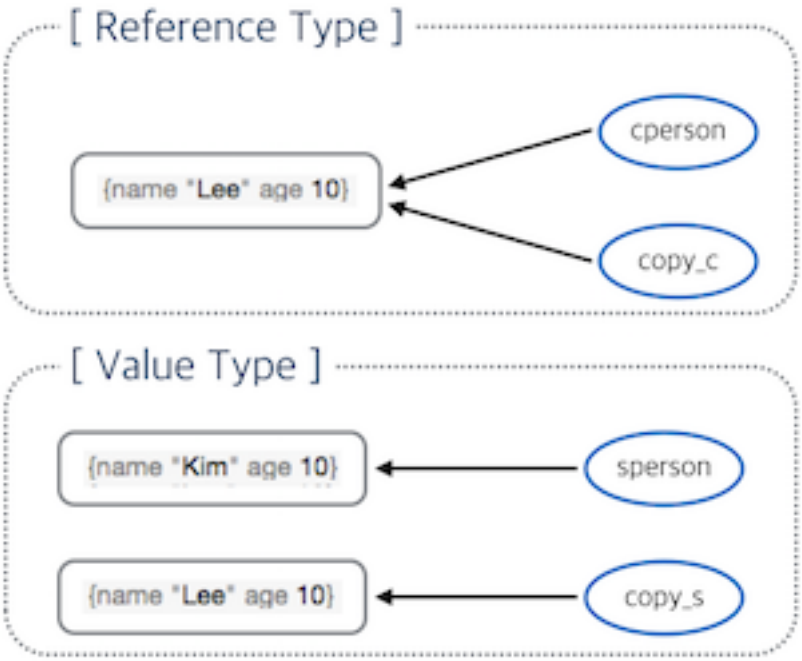
```
53  class ClassPerson {
54      var name: String
55      var age: Int
56      init(name: String, age: Int) {
57          self.name = name
58          self.age = age
59      }
60  }
61
62  let cperson = ClassPerson(name: "Kim", age: 10)      {name "Kim" age 10}
63  var copy_c = cperson                                 {name "Kim" age 10}
64  copy_c.name = "Lee"                                  {name "Lee" age 10}
65
66  cperson                                              {name "Lee" age 10}
67  copy_c                                               {name "Lee" age 10}
```

<d-2>

```
69  struct StructPerson {
70      var name: String
71      var age: Int
72      init(name: String, age: Int) {
73          self.name = name
74          self.age = age
75      }
76  }
77
78  let sperson = StructPerson(name: "Kim", age: 10)     {name "Kim", age 10}
79  var copy_s = sperson                                 {name "Kim", age 10}
80  copy_s.name = "Lee"                                  {name "Lee", age 10}
81
82  sperson                                              {name "Kim", age 10}
83  copy_s                                               {name "Lee", age 10}
```

코드 <d-1> 과 <d-2> 는 `class` 와 `struct` 만을 달리한 코드입니다. 두 코드 모두 객체를 하나 만들고, 다른 변수를 만들어 대입한 뒤 새로운 변수를 통해 속성 값을 변경하고 있습니다. 그런데 결과가 다르죠.

<d-3>



위 그림 <d-3> 과 같이 표현해 볼 수 있겠습니다. `class` 로 만든 `cperson` 은 `copy_c` 라는 변수를 생성하여 대입할 때 동일한 객체를 가리키는 반면, `struct` 로 만든 `sperson` 은 `copy_s` 라는 변수를 생성하여 대입할 때 새로운 객체가 만들어 대입됩니다. 따라서 `copy_c.name` 의 값을 변경시키면 `cperson` 에도 영향을 미치지만, `copy_s.name` 의 값을 변경한 것으로는 `sperson` 에 영향을 미치지 않습니다.

그렇다면 언제 `class` 를 쓰고 언제 `struct` 를 쓰는가? 하는 것이 관건이 되겠네요. Swift 에서는 Value Type 을 활용하는 것을 권장하고 있습니다. 이것은 Mutable 과 Immutable 을 구별해서 쓰는 Swift 의 철학과 닿아 있습니다. 그리고 기본 타입들인 `String`, `Array`, `Dictionary` 등이 모두 Value Type 입니다. 다른 언어에서라면 거의 Reference Type 으로 유지하는 개념들이죠. Mutable/Immutable 을 구별할 때, "잘 모르면 `let` 을 쓴다. 변경할 일이 생기면 `var` 로 쓴다." 라는 원칙을 언급한 적이 있었죠. `struct` 와 `class` 도 약간 비슷한 느낌으로 구별할 수 있는데, "잘 모르면 `struct` 를 쓴다. 두 개의 Reference 가 동일한 객체를 가리켜야 하는 모델의 경우엔 `class` 를 쓴다" 정도로 생각하면 될 듯합니다.

위 예제 코드 <d-1> 과 <d-2> 에서는 `var` 를 사용했는데, `class` 의 경우에는 `let` 을 사용해도 되지만 `struct` 는 `let` 을 사용하면 에러가 납니다. 이것은 `struct` 의 `mutating` 관련 정책 때문인데, 이 부분은 차후에 다루기로 하죠.

2015 년 WWDC 동영상 중에 `struct` 를 활용하여 value type 중심으로 설계하는 것에 대한 것이 있습니다. 한번 보시는 것도 좋지만 그놈의 영어가... ^^ 그래도 영어자막이라도 나오니 중간중간 끊어가면서 볼 수 있습니다.

Protocol

다음은 Protocol 에 대해 알아봅니다. Swift 는 protocol-oriented 언어다 라고 주장을 하고 있군요. 역시 2015년 WWDC 영상 중에 Protocol 중심으로 설계하는 것에 대한 것도 있습니다.

Swift 의 Protocol 은 Java 의 interface 와 목적과 활용이 매우 비슷합니다. 함수 집합의 규격을 선언해 두고 그 함수들을 구현하는 객체들끼리 묶어서 처리할 수 있도록 하는 개념입니다. 하지만 Swift 에서는 Protocol 을 `class` 뿐 아니라 `struct` 와 `enum` 도 구현할 수 있게 열어두고 있어서 더욱 폭넓은 활용을 할 수 있습니다. 또한 함수만 넣을 수 있는 것이 아니고 Computed Property 도 포함될 수 있습니다.

<e>

```
86  protocol Annotation {
87      var location: CGPoint { get }
88      var title: String { get }
89      func show()
90  }
91
92  struct City : Annotation {
93      var location = CGPoint(x: 0, y: 0)
94      var title = ""
95      func show() {
96          print("Showing me: \(title)")
97      }
98  }
99
100 enum EnumCity : Annotation {
101     case Seoul, Busan
102     var location: CGPoint {
103         return CGPoint(x: 0, y: 0)
104     }
105     var title: String {
106         return self == .Seoul ? "Seoul" : "Busan"                    "Seoul"
107     }
108     func show() {
109         print("EnumCity: \(self)")                                   "EnumCity: Seoul\n"
110     }
111 }
112
113 let e_city = EnumCity.Seoul                                          Seoul
114 e_city.title                                                         "Seoul"
115 e_city.show()                                                        Seoul
116
117 var city = City()                                                    City
118
119 let annotations:[Annotation] = [ e_city, city ]                      [Seoul, {location {x 0 y 0}, title ""}]
```

Annotation 이라는 이름의 Protocol 을 선언해 두고 `struct` City 와 `enum` EnumCity 가 그 Protocol 을 구현하도록 하고 있습니다. 이 두 객체는 라인 119 에서처럼 동일한 type 인 것으로 처리될 수도 있습니다. Protocol 을 구현할 때는 라인 92 나 라인 100 처럼 colon (`:`) 을 쓰고 뒤에 protocol 이름을 쓰는데, 마치 `class` 에서 상속받는 부모 클래스를 적어주는 것처럼 하죠. 상속과 달리 Protocol 은 두 개 이상을 구현하는 것이 가능하기 때문에 이때는 comma (`,`) 로 구별하여 열거해 주기도 합니다.

`struct` 가 value type 이기 때문에 다음과 같이 나오는 것도 주목해 주세요

<e-1>

```
119 let annotations:[Annotation] = [ e_city, city ]                   [(Enum Value), {location {x 0 y 0}, title ""}]
120
121 city.location = CGPoint(x: 10, y: 20)                             {location {x 10 y 20}, title ""}
122 city.title = "Incheon"                                            {location {x 10 y 20}, title "Incheon"}
123 city.show()                                                       {location {x 10 y 20}, title "Incheon"}
124 annotations[1]                                                    {location {x 0 y 0}, title ""}
```

city 와 annotations[1] 은 동일한 객체에서 출발했지만 복사된 후에 값이 바뀌었기 때문에 라인 124 의 객체는 “Incheon” 이 아닙니다.

Extension

Extension 은 Objective-C 를 통해 알게 된 기능인데, 너무나 환상적인 특징이라고 생각하고 있습니다. 기존의 다른 OOP 언어들은 기존 `class` 에 어떤 기능을 추가하기 위해서는 그 `class` 를 상속하여 새로은 `class` 를 정의해야 했습니다. 그런데 이 방법은 내가 새로운 객체를 생성할 때에만 해당되고, 라이브러리가 리턴한 객체에 대해서는 wrapper class 를 이용하는 등 불편함이 있었습니다. Objective-C 에서는 기존에 존재하는 `class` 에게 함수를 추가할 수 있도록 하는 `extension` 이라는 기법을 소개했는데, 이는 Swift 에서도 그대로 받아들이고 있습니다.

예를 들어 String 이라는 `class` 에 함수를 추가하고 싶다고 해 보죠. 다음과 같이 합니다.

<f>

```
140 extension String {
141     func addPrefix(prefix: String) -> String {
142         return prefix + self
143     }
144 }
145
146 let s = "Nice"
147 let a = s.addPrefix("Very ")
148 "cookies".addPrefix("a lot of ")
```

(2 times)

"Nice"
"Very Nice"
"a lot of cookies"

`String` 클래스에 함수를 추가했습니다. 이제 `String` 객체이기만 하면 `addPrefix()` 를 사용할 수 있게 됩니다.

존재하는 `class` 나 `struct` 등의 type 에게 protocol 을 구현하도록 할 수도 있습니다. 위의 코드 <e> 와 연결해서 생각해 보겠습니다.

<g>

```
150 extension String: Annotation {
151     var title: String {
152         return self
153     }
154     var location: CGPoint {
155         return CGPoint(x: 0, y: 0)
156     }
157     func show() {
158         println("show: \(self)")
159     }
160 }
161
162 "String".show()
```

"show: String"

`show()` 는 원래 `String` 에는 없는 함수이지만 `String` 이 Annotation 이라는 protocol 을 구현하는 것으로 확장하였으므로 Annotation 의 함수인 `show()` 를 `String` 에 대해서도 호출할 수 있게 되었습니다.

<h>

```
164 protocol Incresing {
165     func increasedValue(amount: Int) -> Int
166     func increase(amount: Int)
167 }
168
169 extension Int: Incresing {
170     func increasedValue(amount: Int) -> Int {
171         return self + amount                    9
172     }
173     func increase(amount: Int) {
174         self += amount
175     }       Binary operator '+=' cannot be applied to two Int operands
176 }
177
178 var value = 3                                   3
179 value.increasedValue(6)                         9
180 value.increase(4)
```

위 코드 <h> 에서는 `Int` 에 `Increasing` 이라는 protocol 을 구현하게 하고 있습니다. `increasedValue()` 는 정상적으로 잘 작동하느 것처럼 보이는데, `increase()` 에서는 에러가 납니다. 이것은 protocol 을 구현하는 동안 `self` 에 대한 변경을 할 수 없도록 하는 규칙 때문에 그렇습니다. `self` 를 변경할 수도 있는 함수는 반드시 `mutating` 으로 선언되어야 합니다.

<h-1>

```
164 protocol Incresing {
165     func increasedValue(amount: Int) -> Int
166     mutating func increase(amount: Int)
167 }
168
169 extension Int: Incresing {
170     func increasedValue(amount: Int) -> Int {
171         return self + amount                    9
172     }
173     mutating func increase(amount: Int) {
174         self += amount                          7
175     }
176 }
177
178 var value = 3                                   3
179 value.increasedValue(6)                         9
180 value.increase(4)                               7
```

`self` 를 변경하는 함수는 `mutating` 으로 선언하여 에러를 없앤 것을 볼 수 있습니다. 그렇다면 이 `mutating` 을 왜 구별해서 써야 하느냐? 다음 코드 <h-2> 를 보면 알 수 있습니다.

<h-2>

```
178  let value = 3                                          3
179  value.increasedValue(6)                                9
180  value.increase(4)                                      7
181      Immutable value of type 'Int' only has mutating members named 'increase'
182
```

이번에는 value 가 `let` 으로 선언되었습니다. `var` 일 때는 상관 없지만, `let` 으로 선언된 값은 `mutating` 함수를 호출할 수 없습니다. 드디어 첫시간에 배운 `let` 과 `var` 구별의 구현부분에 대한 언급이네요. Swift 에서는 Immutable 객체를 쓰도록 유도하고 있다는 사실을 다시 한 번 깨닫는 순간입니다.

Generics

<i>

```
5   func makeArray<T>(item: T, times: Int) -> [T] {
6       var items = [T]()                                  (2 times)
7       for i in 0..<times {
8           items.append(item)                             (7 times)
9       }
10      return items                                       (2 times)
11  }
12
13  makeArray("K", 4)                                      ["K", "K", "K", "K"]
14  makeArray(10, 3)                                       [10, 10, 10]
```

Generics 란 type 에 관계 없이 어떤 것을 정의하고자 할 때 사용하는 기법으로, 최신 언어들은 거의 차용하고 있습니다. 논외의 얘기를 조금 하면, Objective-C 는 Generics 의 개념이 없습니다. 왜냐하면 `id` 라고 하는 무식한? 타입을 써서 무슨 타입이든지 상관하지 않고 막 전달하기 때문이죠. Swift 가 설계되면서 Objective-C 의 이러한 `id` 정책은 그 위험성 때문에 철퇴를 맞습니다. Swift 에도 `Any` 라는 type 을 쓸 수 있기는 하지만 type-strict 한 특성상 별로 권장되지 않죠. 하지만 XCode7 부터는 Objective-C 에도 Light-weight Generics 라는 개념이 추가되어 활용되고 있습니다.

다시 Swift 얘기로 돌아오죠. 위의 코드 <i> 를 보면 임의의 type T 에 대해 `makeArray` 라는 함수를 정의하는 것을 볼 수 있습니다. 사실 이 함수는 Any 를 써서 구현하는 것도 가능합니다.

<i-1>

```
16 func makeAnyArray(item: Any, times: Int) -> [Any] {
17     var items = [Any]()                                  (2 times)
18     for i in 0..<times {
19         items.append(item)                               (7 times)
20     }
21     return items                                         (2 times)
22 }
23
24 makeAnyArray("K", 4)                                    ["K", "K", "K", "K"]
25 makeAnyArray(10, 3)                                     [10, 10, 10]
```

하지만, 이 경우 `makeAnyArray()` 가 리턴한 값은 라인 24와 라인 25 모두 `[Any]` 입니다. 리턴 받은 쪽 입장에서는 혹시 다른 타입이 섞여 들어오지 않았는지도 걱정해야 하고 type casting 도 해야 합니다. 하지만 generics 버전인 코드 <i> 에서는 라인 13 과 라인 14 의 리턴타입이 각각 `[String]` 과 `[Int]` 입니다. 타입캐스팅이 필요 없죠.

불편한 것 뿐 아니라 아예 불가능한 것도 있습니다.

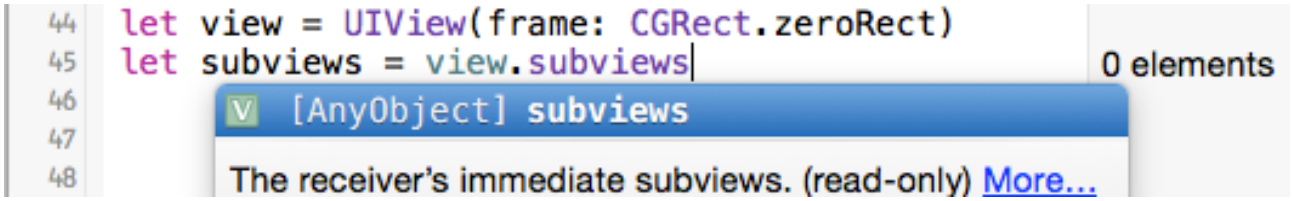
<j>

```
27 func swapGenerics<T>(inout a: T, inout b: T) {
28     let t = a                                            10
29     a = b                                                20
30     b = t                                                10
31 }
32
33 func swapAny(inout a: Any, inout b: Any) {
34     let t = a
35     a = b
36     b = t
37 }
38
39 var a1 = 10, a2 = 20                                    (2 times)
40 swapGenerics(&a1, &a2)
41 a1                                                      20
42 a2                                                      10
43
44 swapAny(&a1, &a2)
45  Cannot invoke 'swapAny' with an argument list of type '(inout Int, in...
```

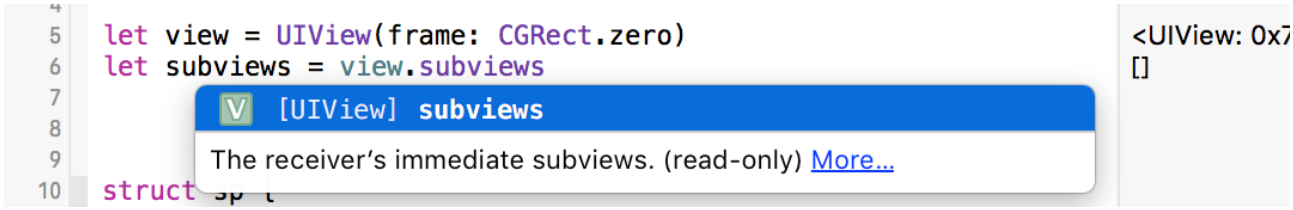
이처럼, 동일한 것이 확실할 때에는 Any 같은 상위 type 을 쓰지 않고 Generics 를 사용하는 것이 훨씬 도움이 됩니다.

XCode 7.0 과 Objective-C 의 Light-weight Generics 덕에 `NSArray` 가 `id` 만을 가지지 않고 특정 타입을 지칭할 수 있게 되어서 이제 더이상 `UIView.subviews` 로 loop 를 돌 때 type casting 을 하지 않아도 되게 되었습니다.

<k-1>



<k-2>



Swift 1.2 를 사용했었던 XCode 6.4 에서는 위의 코드 <k-1> 처럼 `subviews` 의 type 이 `[AnyObject]` 입니다. 하지만 XCode 7.0 부터는 <k-2>처럼 `[UIView]` 로 되어 상당히 편리해 졌습니다. Swift 의 발전이 Objective-C 에 영향을 준 경우입니다.

이렇든 Generics 는 워낙 프로그래밍을 편리하게 해 주는 도구라서, 제 생각에는 Swift 2.0 과 XCode 7.0 을 만들면서 Objective-C 로부터 넘어오는 부분에 Generics 를 적용하기 어려워지니까 Objective-C 에 강제로 Generics 개념을 추가해준 것 같기도 합니다.

마침

다음 시간부터는 iOS 앱을 만드는 것에 집중해 보기로 하죠.

감사합니다.