04. Swift 2.0으로 iOS 코딩하기 - Swift Tour (4)

네번째 시간입니다. Function 과 Closure 에 대해 알아보겠습니다.

Function

먼저 함수에 대해 알아봅니다. 함수는 `func` 라는 키워드와 함께 함수 이름, 인자, 리턴타입을 명시하는 다음과 같이 명시하는 문법을 씁니다.

<a>

```
5  func greet(name:String, day:String) -> String {
6      return "Hello \(name), today is \(day)"
7  }
8
9  let gr1 = greet("Dave", day: "Monday")
10 let gr2 = greet("Kelly", day: "Sunday")
11
```

(2 times)

"Hello Dave, today is Monday"
"Hello Kelly, today is Sunday"

함수는 동일한 기능을 여러 번 사용하고 싶을 때 사용하기도 하지만, 복잡한 일을 단위 작업으로 나누기 위해서 사용하기도 합니다. 몇 가지 언어에서처럼 Swift 도 가변 인자를 지원하는데, type 에 ... 을 붙여 사용합니다.

<b>

```
5  func average(numbers: Int...) -> Double {
6      var sum = 0
7      for num in numbers {
8          sum += num
9      }
10     return Double(sum) / Double(numbers.count)
11 }
12
13 let avg1 = average(10, 20, 23)
14 let avg2 = average(1,2,3,4,5,6,10)
15
```

(2 times)

(10 times)

(2 times)

[Int] numbers

17.66666666666667
4.428571428571429

위와 같이 Int... 으로 지정한 numbers 의 타입은 [Int] 로 인식되는 것을 알 수 있습니다.

Swift 에서는 두 개 이상의 값을 묶어서 하나의 변수/상수로 가리킬 수 있도록 하는 Tuple 을 쓸 수 있습니다. C 의 `struct` 와 비슷한 개념일 수 있지만, Swift 에도 `struct` 가 있습니다. Tuple 은 type 을 지정하지 않은 `struct` 정도로 생각해도 좋겠네요. 괄호로 여러 개의 값을 묶어서 사용합니다.
`struct` 로 사용하려면 미리 정의해 두고 사용해야 하지만 Tuple 은 그때그때 즉흥적으로 만들어서 사용할 수 있는 장점이 있습니다.

<c-1>

```
5  let pair = (12, 13.0)
6  pair
7  pair.0
8  pair.1
9  pair
```

(Int, Double) pair

```
(.0 12, .1 13)
(.0 12, .1 13)
12
13
(.0 12, .1 13)
```

코드 <c-1> 에서 pair 의 type 은 `(Int, Double)` 입니다. 각 멤버는 `.0` 혹은 `.1` 의 인덱스로 접근합니다. 인덱스뿐 아니라 이름을 붙여 줄 수도 있는데, 아래 코드 <c-2> 처럼 합니다.

<c-2>

```
12 let point = (x:10, y:15)
13 point.0
14 point.1
15 point.x
16 point.y
17 point
```

(x: Int, y: Int) point

```
(.0 10, .1 15)
10
15
10
15
```

이러한 Tuple 은 함수가 여러 개의 값을 리턴하고자 할 때 꽤 유용하게 사용할 수 있습니다. 물론 미리 `struct` 등을 정의해 놓고 사용해도 되기는 합니다만, Tuple 이 편하게 쓰기 좋죠. 아래 코드 <d> 에서는 주어진 정수들의 합과 평균을 동시에 리턴하는 함수인 `sum_average()` 를 정의합니다.

<d>

```
5  func sum_average(numbers: Int...) -> (sum:Int, avg:Double) {
6      var sum = 0
7      for num in numbers {
8          sum += num
9      }
10     let avg = Double(sum) / Double(numbers.count)
11     return (sum:sum, avg:avg)
12 }
13
14 let sa1 = sum_average(10, 20, 23)
15 let sa2 = sum_average(1,2,3,4,5,6,10)
16
17 sa1
```

(sum: Int, avg: Double) sa1
(sum: Int, avg: Double) sa2

```
(2 times)
(10 times)
(2 times)
(2 times)
(.0 53, .1 17.66666666666667)
(.0 31, .1 4.428571428571429)
(.0 53, .1 17.66666666666667)
```

함수는 nest 될 수 있는 특징을 가집니다. nest 된다는 말은 어떤 것 안에 또다시 그걸 할 수 있는 것을 말합니다. 즉, 함수 안에 함수를 정의할 수 있습니다. 이렇게 **함수(1)** 안에 정의된 **함수(2)**는 scope 가 **함수(1)** 이므로 **함수(1)** 밖에서는 사용이 불가능합니다.

아래 코드 <e> 는 특이한 함수인 `calculate()` 를 정의하는데, <d> 에서처럼 합과 평균을 구하는데 특이한 점은 각 숫자가 2나 5의 배수이면 합에 각각 1씩 추가해 주는 녀석입니다. 라인 7 의 `increaseSumIfNumber(_, devidedBy:)` 함수는 `calculate()` 내에서 정의되어 `calculate()` 내에서만 사용하는 함수입니다.

<e>

```
5  func calculate(numbers: Int...) -> (sum:Int, avg:Double) {
6      var sum = 0                                                          (2 times)
7      func increaseSumIfNumber(num:Int, devidedBy devider:Int) {
8          if (num % devider) == 0 {
9              sum++                                                        (10 times)
10         }
11     }
12     for num in numbers {
13         sum += num                                                       (10 times)
14         increaseSumIfNumber(num, devidedBy: 2);
15         increaseSumIfNumber(num, devidedBy: 5);
16     }
17     let avg = Double(sum) / Double(numbers.count)                        (2 times)
18     return (sum:sum, avg:avg)                                            (2 times)
19 }
20
21 let sa1 = calculate(10, 20, 23)                                         (.0 57, .1 19)
22 let sa2 = calculate(1,2,3,4,5,6,10)                                     (.0 37, .1 5.285
23
24 sa1.sum                                                                 57
25
```

코드 <e> 의 7번째 라인에서는 함수의 새로운 기능이 소개되는데, 바로 함수 인자 (argument) 의 이름을 붙이는 것입니다. 두번째 인자인 devider 의 이름을 쓰지 않았다면 호출하는 14 라인에서는 increaseSumIfNumber(num, 2) 와 같이 썼을테지만, 이렇게 쓰는 경우 호출하는 쪽 (caller) 코드에서는 두번째 인자가 무엇을 의미하는지 알기 어려운 때가 많습니다. 이럴 때 위와 같이 devidedBy: 를 붙여서 쓰면 의미가 명확해지곤 합니다. 첫번째 인자는 함수 이름을 통해 힌트를 얻게 되기 때문에 보통은 두번째 이후의 인자들에게 이름을 부여합니다.

위와 같이 함수 인자에 이름을 부여하는 경우 함수 인자의 이름을 포함한 것이 전체 함수의 이름이 됩니다. 함수의 이름이 된다는 것은 나중에 함수를 구별하고 검색할 때 이름을 쓰게 되기 때문에 중요해 집니다. 위의 경우 함수의 이름은 `increaseSumIfNumber()` 가 아니고 `increaseSumIfNumber(_, devidedBy:)` 가 됩니다.

Swift 는 함수형 언어 (Functional Language) 의 특징을 여럿 가지는데, 그 중 하나가 함수도 하나의 type 이 된다는 것입니다. 예를 들어 **"Int 를 인자로 받아 Bool 을 리턴하는 함수"** 는 `Int -> Bool` 로 표현합니다. `Int` 와 `Double` 을 인자로 받아 `String` 을 리턴하는 함수는 `(Int, Double) -> String` 으로 표현합니다. 인자 목록과 리턴 타입이 동일한 함수는 같은 타입의 객체로 취급됩니다.

<f>

```
5  func points(num:Int, bonus:Int->Bool) -> Int {
6      var pt = num * 10                                    (4 times)
7      if bonus(num) {
8          pt++                                             (2 times)
9      }
10     return pt                                            (4 times)
11 }
12
13 func odd(num:Int) -> Bool {
14     return (num % 2) != 0                                (2 times)
15 }
16
17 func negative(num:Int) -> Bool {
18     return num < 0                                       (2 times)
19 }
20
21 points(5, bonus: odd)                                   51
22 points(6, bonus: odd)                                   60
23 points(7, bonus: negative)                              70
24 points(-1, bonus: negative)                             -9
```

위 코드 <f> 에서, `points()` 는 숫자 하나와 함수 하나를 입력 인자로 받습니다. 그래서 입력받은 숫자에 10 을 곱한 값을 리턴하는데, 전달받은 함수를 호출하여 참이면 보너스로 1점을 추가해 줍니다. 라인 21~24 에서 여러 값으로 호출하는데, 라인 22 와 라인 23 의 경우 전달받은 함수를 적용했을 때 참이 아니므로 그냥 10배 값이 리턴되지만, 라인 21 과 라인 24 에서는 전달된 함수에 전달된 값을 인자로 호출했을 때 참을 리턴하므로 1 을 더해서 리턴하게 됩니다.

Swift 가 **type-strict 한 언어**라고 여러 번 강조했는데, 동일한 type 인지를 판단하는 기준이 Java 와는 좀 다릅니다. Java 의 경우 멤버 구성이 완전히 동일하더라도 두 개의 type (혹은 class) 은 영원히 호환되지 않지만, Swift 는 내부 구성 요소가 같기만 하면 서로 호환이 됩니다. Tuple 의 경우도 그렇고, 위의 함수 type 만 하더라도 `points()`, `odd()`, `negative()` 세 함수에서 `Int->Bool` 함수를 언급할 뿐 그것을 하나의 type 으로 정의하지조차 않습니다. 이런 점은 다른 strict 한 언어들에 비해 좀 더 자유로움을 추구했다고 할 수도 있겠습니다.

함수를 함수의 인자로 전달할 수 있을 뿐 아니라 당연히 리턴값으로도 사용할 수 있습니다.

<g>

```
5  func incrementor(num:Int) -> (Void -> Int) {
6      var value = 0                                  (2 times)
7      func increase() -> Int {
8          value += num                               (5 times)
9          return value                               (5 times)
10     }
11     return increase;                               (2 times)
12 }
13
14 let inc3 = incrementor(3)                          (Function)
15 inc3()                                             3
16 inc3()                                             6
17
18 let inc7 = incrementor(7)                          (Function)
19 inc7()                                             7
20 inc7()                                             14
21
22 inc3()                                             9
23
```

코드 <g> 에서 함수 `incrementor()` 는 `Int` 하나를 입력받아 함수를 리턴하는 함수입니다. `Void` 라는 type 이 처음 나왔는데, 함수의 인자나 리턴타입이 없을 때 사용하는 type 입니다.

라인 14 와 18 에서 각각 3 증가시키는 함수와 7 증가시키는 함수를 만들어서 여러 번 부르고 있습니다. 특이할만한 점은 바로 라인 6 의 value 라는 변수인데, 이 변수는 `incrementor()` 호출마다 하나씩 생성되어 유지되는 값입니다. 나중에 class 를 만들고 member 변수를 선언해서 사용할 일이 있겠지만, 그것의 간편 버전으로 생각해도 좋습니다.

Closure

Closure 는 함수와 동일하지만 `func` 키워드도, 함수의 이름도 없습니다. 정의로는 반대로 함수가 closure 의 특수한 형태라고 생각하는 것이 맞습니다. Closure 야말로 Functional Paradigm 의 꽃이라고 할 수 있습니다. 코드 덩어리를 객체로 취급하는 방식인데, C 에는 이런 개념이 없고 Objective-C 의 block 과 같습니다. Java 에는 Anonymous Interface Implementation 즉 class 이름 없이 interface 를 구현하기만 하는 것과 비슷하고, Javascript 같은 스크립팅 언어에서는 매우 빈번히 쓰이는 기법입니다.

Block/closure 을 사용하기 전에는 delegation 이나 Objective-C 의 selector 개념을 많이 사용했습니다. Closure 를 쓰게 되어 좋아진 점을 예로 들어보면, 화면 중앙에 간단한 메시지와 버튼 몇 개를 보여주는 `UIAlertView` 라는 클래스가 있습니다. Alert 객체를 만들어 title, message, button 등의 속성을 정하고 띄우고 나면 사용자가 어떤 응답을 했는지 미리 지정한 함수를 통해 (delegation) 알려줍니다. 여기까지는 문제가 없는데, 문제는 이러한 Alert 가 두 개 이상인 경우에 복잡해진다는 것입니다. 사용자가 Yes 를 답했는지 No 를 답했는지에 따라 다른 코드를 실행해야 하는데, 이 결과가 두 개의 전혀 다른 Alert 에 대해 동일한 함수로 전달된다는 것입니다. 이때문에 Button Index 를 전달받고도 이 응답이 어떤 질문에 대한 응답인지를 판단하는 코드를 다시 써서 분기를 해 주어야 합니다. 코드를 읽기가 매우 나빠지죠.

저는 Alert 를 사용할 때 위와 같은 문제 때문에 block 을 쓰는 기법을 도입했습니다. 일을 시키는 코드와 그 일의 결과에 대한 처리를 하는 코드가 인접할 수 있게 되어 읽기가 훨씬 쉬워집니다. 그런데 iOS8 부터는 `UIAlertView` 대신 `UIAlertController` 라는 클래스가 추가되어 block 기반의 처리를 할 수있도록 변경되었죠. iOS 측에서 프로그래머의 불편한 부분을 긁어 준 셈입니다.

잡설이 길었는데, 위 코드 <f> 를 closure 를 사용하는 코드로 변경해 보겠습니다.

<h>

```
5  func points(num:Int, bonus:Int->Bool) -> Int {
6      var pt = num * 10                                    (4 times)
7      if bonus(num) {
8          pt++                                             (2 times)
9      }
10     return pt                                            (4 times)
11 }
12
13 points(5, bonus: { (num:Int) -> Bool in
14     return (num % 2) != 0                                (2 times)
15 })
16
17 let negative = { (num:Int) -> Bool in
18     return num < 0                                       (2 times)
19 }
20 points(-1, bonus: negative)
21
22 let positive = { $0 > 0 }
23
24 let mod3 = points(9, bonus: { ($0 % 3) == 0 })
25 mod3
26 let mod7 = points(9) { ($0 % 7) == 0 }
27 mod7
```

함수 `points()` 는 코드 <f> 의 것과 동일합니다. 그런데 홀수인지 판단하는 코드와 음수인지 판단하는 코드를 함수로 만들지 않고 모두 closure 로 작성하였습니다. 라인 13 ~ 15 를 보면 알 수 있듯이, Closure 는 { 로 시작해서 } 로 끝나는 하나의 코드 덩어리 입니다. 함수처럼 인자와 리턴타입이 있고 { 와 `in` 사이에 그 타입을 적습니다. <f> 에서는 `negative` 를 함수로 선언했지만, <h> 에서는 closure 로 선언했습니다. 하지만 이후에 `negative` 를 사용할 때는 완전히 동일한 것으로 취급됩니다. <h> 의 라인 20 과 <f> 의 라인 24 는 완전히 동일한 코드죠.

경우에 따라 함수 인자와 리턴 타입, return statement 를 생략하게 되는 수가 있습니다. 라인 22 를 보면, positive 라는 closure 는 인자에 이름을 주지 않고 $0 을 사용했습니다. 이것은 첫번째 함수 인자 라는 의미입니다. 단일 expression 으로만 되어 있어서 `return` 까지 생략했습니다.

라인 24의 mod3 은 9 가 3의 배수이므로 10배 후 1이 추가되는 것을, 라인 26의 mod7 은 9 가 7의 배수가 아니므로 10배만 하는 것을 보여줍니다. 그런데 중괄호 (curly braces) 의 위치가 특이하죠. Swift 에서는 **함수의 마지막 인자가 closure 인 경우 괄고를 미리 닫고 closure 를 쓸 수 있도록 허용**하는 문법을 제공합니다. **유일한 인자가 closure 이면 괄호조차 생략**합니다.

다음 코드 <i> 는 `[Int]` 를 정렬하는 함수 `sortInPlace()` 를 호출하는 예제입니다. 임의의 정수 배열을 선언한 다음 오름차순과 내림차순 정렬을 반복하면서 closure 사용 용법을 보여 줍니다.

<i>

```
5  var numbers = [ 1, 12, 32, 2, 4, 43, 3]                    [1, 12, 32, 2, 4, 43, 3]
6  numbers.sortInPlace { (n1:Int, n2:Int) -> Bool in           [1, 2, 3, 4, 12, 32, 43]
7      return n1 < n2                                          (14 times)
8  }
9  numbers.sortInPlace({n1, n2 in n1 > n2})                    (22 times)
10 numbers                                                     [43, 32, 12, 4, 3, 2, 1]
11
12 numbers.sortInPlace { $0 < $1 }                             (22 times)
13 numbers                                                     [1, 2, 3, 4, 12, 32, 43]
14
15 numbers.sortInPlace(>)                                      [43, 32, 12, 4, 3, 2, 1]
16 numbers                                                     [43, 32, 12, 4, 3, 2, 1]
17
18 numbers.sortInPlace()                                       [1, 2, 3, 4, 12, 32, 43]
```

라인 6~8 이 정상적으로 모두 생략하지 않은 코드입니다. `Int` 두 개를 전달받아서 앞의 것이 먼저 와야 하면 `true` 를 리턴하는 closure 를 전달하고 있죠. 라인 10을 보면 꽤 많은 것들이 생략되어 있습니다. 인자의 이름만 남기고 모두 생략했죠. 라인 13은 좀 더 생략합니다. 인자의 이름 대신 `$0`, `$1` 를 사용하고 있고 `sortInPlace` 함수 호출시의 `()` 도 생략했습니다. 기존 언어에 익숙한 사람들에게 가장 황당한 것은 라인 16 인데, 바로 operator function 이라는 것입니다. Swift 에서는 모든 operator 가 closure 로 사용됩니다. > 라는 operator 도 인자 두 개를 받아서 `Bool` 을 리턴하는 역할을 하기 때문에, `sortInPlace` 의 인자로 아주 적절하죠.

마지막 18 라인에서는 sortInPlace 에 아예 아무런 closure 도 전달하지 않았습니다. 이것은 2.0 에서 sort 관련하여 변경된 것 중 하나인데, 언어가 바뀌었다기 보다는 API 가 바뀐 경우입니다. 2.0 부터는 Number 들과 String 은 기본적으로 Comparable 을 구현하고 있어서 비교 closure 를 제공하지 않아도 됩니다.

마침

네번째 시간은 여기까지입니다. 다음 시간에 Class/Object 와 함께 만나요.