

05. Swift 2.0으로 iOS 코딩하기 - Swift Tour (5)

다섯번째 시간입니다. 클래스와 객체에 대해 알아봅시다.

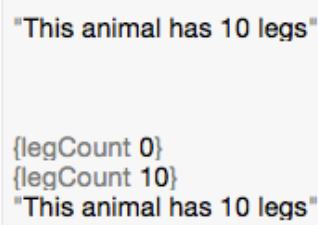
Class

붕어빵을 만들어서 팔려고 합니다. 그런데 반죽을 손으로 빚어서 붕어빵 하나를 만드는 데는 시간이 너무 오래 걸립니다. 붕어 비늘 하나 하나 만들어 주려면 꽤 어려운 작업이 됩니다. 문제는 붕어빵을 하나 더 만들려면 그 작업을 다시 또 해야 한다는 점입니다. 그래서 붕어빵 여러개를 손쉽게 만들기 위해서 철로 붕어빵틀을 제작했습니다. 이제는 붕어빵틀에다가 반죽을 붓기만 하면 붕어빵을 만들 수 있게 되었습니다.

class 와 객체 (object instance) 는 붕어빵틀과 붕어빵 으로 이해할 수 있습니다. 붕어빵틀을 원하는 모양대로 잘 제작하는 것은 class 디자인이며, 붕어빵 10개를 찍어 내었다면 객체 10개를 생성한 것에 비유할 수 있겠습니다.

다음 코드 <a> 는 Animal 이라는 이름의 붕어빵틀을 만들고 animal 이라는 이름의 붕어빵을 찍어낸 것입니다.

```
<a>
5  class Animal {
6      var legCount = 0
7      func simpleDescription() -> String {
8          return "This animal has \(legCount) legs"
9      }
10 }
11
12 let monster = Animal()
13 monster.legCount = 10
14 monster.simpleDescription()
```



이제 OOP 용어로 돌아 오면, Animal 이라는 class 는 legCount 라는 이름의 “멤버 변수” 가 있으며, simpleDescription() 이라는 이름의 함수를 가집니다. Swift 에서는 class 이름에 괄호를 써서 해당 class 의 객체를 생성합니다. 위 코드 <a> 의 라인 12 에서 하는 일입니다. 그리고는 legCount 라는 property 의 값을 변경시킨 뒤, simpleDescription() 함수를 호출하였습니다.

위 코드 <a> Animal 에는 중요한 요소가 하나 빠졌는데, 바로 생성자 입니다. 생성자는 init 라는 이름을 가집니다.


```
5 class Animal {
6     var legCount = 0
7     var name:String
8     init(name:String) {
9         self.name = name
10    }
11    func simpleDescription() -> String {
12        return "This \(name) has \(legCount) legs"
13    }
14 }
15
16 let lark = Animal(name: "Lark")
17 lark.legCount = 2
18 lark.simpleDescription()
```

"This Lark has 2 legs"

{legCount 0 name "Lark"}
{legCount 2 name "Lark"}
"This Lark has 2 legs"

코드 에서 새로 정의한 Animal 은 이제 생성자를 가집니다. 그런데 유일한 생성자가 인자를 가지기 때문에, 라인 16 처럼 이제 Animal 객체를 생성할 때는 name: 인자를 반드시 넘겨주어야 합니다.

생성자에 인자를 주는 것과 주지 않는 것의 차이는, 해당 특성을 가지지 않는 객체가 존재하는 것이 가능하게 할 것인지의 문제입니다. 위 예제는 조금 잘못된 예일 수 있는데, 하나의 Animal 이 이름 없이 존재할 수 있을까요? 혹은 다리 갯수가 0 인 상태로 존재할 수 있을까요? 이 질문에 대한 답이 yes 라면 생성자 인자로 전달하는 것이 맞고, no 라면 생성 후 property 설정으로 하는 것이 맞을 수 있겠습니다.

Inheritance

클래스를 상속받는 경우를 살펴보겠습니다.

<c>

```
20 class Mammal:Animal {
21     override init(name: String) {
22         super.init(name: name)
23         legCount = 4
24     }
25     override func simpleDescription() -> String {
26         return "This mammal named \(name) has \(legCount) legs"
27     }
28     func giveBirth() -> Mammal {
29         println("Brought forth a baby")
30         return baby()
31     }
32     func baby() -> Mammal {
33         return Mammal(name: "A baby \(name)")
34     }
35 }
36
37 let el = Mammal(name: "Elephant")
38 el.legCount
39 el.simpleDescription()
40 let baby = el.giveBirth()
41
```

"This mammal named Elephant has 4 legs"

"Brought forth a baby"
{legCount 4 name "A baby Elephant"}

{legCount 4 name "A baby Elephant"}

{legCount 4 name "Elephant"}
4
"This mammal named Elephant has 4 legs"
{legCount 4 name "A baby Elephant"}

코드 <c> 의 20 라인을 보면, Mammal 이라는 class 는 Animal 을 상속합니다. 상속한다는 것은 그 속성을 모두 이어받는다는 뜻입니다. 따라서 legCount, init() 및 simpleDescription() 을 그대로 사용할 수도 있었습니다. 그런데 init 와 simpleDescription 은 override 했네요. override 는 parent class 에 정의된 것을 무시하고 새롭게 정의할 때 사용하는 용어입니다.

Animal 의 경우는 생성했을 때 다리의 갯수 (legCount) 가 0 인 상태였기 때문에 생성 후 다리 갯수를 지정해 주어야 했습니다. 그런데, 새로 상속받은 Mammal 의 경우는 생성자에서 legCount 를 4 로 정

해주었네요. 라인 38 에서 이미 4가 되어 있는 것을 알 수 있습니다. `init()` 와 `simpleDescription()` 은 둘 다 `override` 되었지만, `init` 의 경우는 parent class 에 정의된 `init` 를 호출하고 있는데, 이것은 `super` 를 통해서 가능합니다.

`Mammal` 은 상속받고 `override` 만 하는 것이 아니라 `giveBirth()` 라는 새로운 함수도 제공합니다. 새로운 객체를 생성해서 리턴하고 있죠.

<d>

<pre>37 let el = Mammal(name: "Elephant") 38 el.legCount 39 el.simpleDescription() 40 let baby = el.giveBirth() 41 let grand = baby.giveBirth() 42</pre>	<pre>{{legCount 4 name "Elephant"}} 4 "This mammal named Elephant has 4 legs" {{legCount 4 name "A baby Elephant"}} {{legCount 4 name "A baby A baby Elephant"}}</pre>
--	--

이때 생성된 `baby` 는 다시 `Mammal` 객체이기 때문에 역시 `giveBirth()` 함수를 호출할 수 있게 됩니다.

이번에는 `Mammal` 을 상속하는 `Dog` 를 만들어 볼까요

<e>

<pre>43 class Dog:Mammal { 44 override func baby() -> Mammal { 45 return Dog(name: "\(name)'s Puppy") 46 } 47 func bark() -> String { 48 return "Bark!" 49 } 50 } 51 52 let jd = Dog(name: "Jindol") 53 let puppy = jd.giveBirth() 54 puppy.name 55 puppy.dynamicType 56 (puppy as! Dog).bark() 57 puppy.bark()</pre>	<pre>{{{...}}} "Bark!" {{{...}}} {{{...}}} "Jindol's Puppy" __lldb_expr_112.Dog "Bark!"</pre>
---	---

`Dog` 라는 class 는 새로운 기능은 없고 `baby()` 만 `override` 했습니다. 라인 49 에서 `Dog` 를 만든 다음 라인 50 에서 `giveBirth()` 를 호출하면 `Dog` 에서 `giveBirth()` 를 `override` 하지는 않았지만 `baby()` 를 `override` 했기 때문에 `Mammal` 객체가 생성되는 것이 아닌 `Dog` 객체가 생성됩니다. 주의할 점은 50라인에서 새로 만들어진 `puppy` 객체의 실제 type 은 `Dog` 이지만, 코드 상에서의 `puppy` 의 type 은 `Mammal` 이라는 점입니다. 따라서 상수 `puppy` 로는 `Dog` 만의 기능을 호출하지는 못합니다. `Dog` 만의 기능을 호출하기 위해서는 `as!` 키워드를 이용하여 type cast 를 해 주어야 합니다.

이번에는 오리너구리 (`Platypus`) 의 경우를 살펴보겠습니다.

<f>

```
59 class Platypus:Mammal {
60     override func giveBirth() -> Mammal {
61         println("Laying an egg")
62         return baby()
63     }
64     override func baby() -> Mammal {
65         return Platypus(name: "Platypus from an egg");
66     }
67 }
68
69 let pl = Platypus(name: "Platypus")
70 pl.name
71 let pb = pl.giveBirth()
72 pb.name
73
```

```
"Laying an egg"
{{{...}}}
```

```
{{{...}}}
```

```
{{{...}}}
"Platypus"
{{{...}}}
"Platypus from an egg"
```

Platypus 는 Mammal 의 기능을 모두 상속받는데, baby() 뿐 아니라 giveBirth() 를 override 합니다. 다른 Mammal 들은 새끼를 낳지만 Platypus 는 알을 낳기 때문입니다. 라인 71 에서 giveBirth() 가 호출되면 라인 61 이 실행되면서 “Laying an egg” 가 출력됩니다.

지난 1학기 기말고사 문제 중 하나가 바로 “오리너구리와 Polymorphism 의 관계에 대해 설명하라” 였습니다. 포유류들은 모두 새끼를 낳는 방식으로 자손번식을 하지만, 오리너구리는 포유류임에도 불구하고 자손번식 방법을 override 하여 알을 낳았다는 점에서 OOP 의 Polymorphism 의 좋은 예가 됩니다.

Polymorphism 을 활용하는 좋은 예는 실제 type 이 무엇인지 모를 때 극대화됩니다.

<g>

```
74 let mammals = [
75     Mammal(name: "Normal mammal"),
76     Dog(name: "Jindol"),
77     Platypus(name: "Platypus")
78 ]
79
80 for m in mammals {
81     let child = m.giveBirth()
82     child.simpleDescription()

```

```
This mammal named A baby Normal mammal has 4 legs
This mammal named Jindol's Puppy has 4 legs
This mammal named Platypus from an egg has 4 legs
```

```
83 }
or.
```

위 코드 <g> 의 81 번째 라인이 수행될 당시에는 m 의 타입이 무엇인지 상관하고 있지 않습니다. 다만 giveBirth() 함수를 부를 뿐이죠. 그러면 해당 객체에 바인딩되어 있는 giveBirth() 함수가 호출되면서 새끼를 낳는 버전이 두 번, 알을 낳는 버전이 한 번 호출됩니다. 그리고 다시 override 된 함수 baby() 를 통해 실제 생성되는 객체도 다를 수 있도록 구현되어 있습니다.

위와 같이 Polymorphism 을 활용하지 않는, 아주 나쁜 예를 보여드리겠습니다.

<pre><h> 80 for m in mammals { 81 if (m is Dog) { 82 println("Dog: " + m.simpleDescription()) 83 } else if (m is Platypus) { 84 println("Platypus: " + m.simpleDescription()) 85 } else { 86 println("Just a mammal: " + m.simpleDescription()) 87 } 88 } 89</pre>	<pre>"Dog: This mammal named Jindol has 4 legs" "Platypus: This mammal named Platypus has 4 legs" "Just a mammal: This mammal named Normal mammal has 4 legs"</pre>
--	---

위의 코드 <h> 는 코드 <h> 의 라인 80 에 있는 for loop 부분을 다시 쓴 것입니다. Loop 내에서 배열 내의 각 element 의 type 을 알아낸 다음 Dog 이면 dog 에 해당하는 코드를, Platypus 이면 Platypus 에 해당하는 코드를 호출하는 것입니다. 이것은 매우 잘못 디자인된 경우에 해당합니다. <g> 와 같이 Mammal 은 Mammal 로서 다루도록 하고, 각 클래스마다 다른 동작은 override 한 함수에서 이루어지도록 하는 것이 바람직합니다.

Computed Property

한참 위 코드 의 Animal 을 다시 떠올려보죠. 여기에는 legCount 라는 property 가 있었습니다. 이것은 property 중 stored property 라는 것으로, 그 값을 위한 저장소가 마련되어 있습니다. 그런데 stored property 말고 computed property 라는 것도 있습니다. 값이 객체내에 저장되는 것이 아니고 해당 property 에 접근할 때마다 코드가 실행되는 것입니다.

<i>

95	class Monster:Mammal {	
96	var kneeCount : Int {	
97	get {	
98	return legCount	(2 times)
99	}	
100	}	
101	var toeCount : Int {	
102	get {	
103	return legCount * 4	16
104	}	
105	set(count) {	
106	legCount = count / 4	{{{...}} eyeCount 0}
107	}	
108	}	
109	var eyeCount = 0 {	
110	didSet {	
111	legCount = 2 * eyeCount	{{{...}} eyeCount 123}
112	}	
113	}	
114	}	
115		
116	let m = Monster(name: "Goo")	{{{...}} eyeCount 0}
117	m.kneeCount	4
118	m.toeCount	16
119	m.toeCount = 20	{{{...}} eyeCount 0}
120	m.kneeCount	5
121		
122	m.eyeCount = 123	{{{...}} eyeCount 123}
123	m.legCount	246
124		

이번에는 포유류를 상속하는 괴물을 만들어 보았습니다. 이 class 는 computed property 를 두 개 추가하였습니다. 라인 117 에서 kneeCount 를 부르자 라인 98 의 코드가 실행됩니다. 라인 118 의 toeCount 을 얻어오려고 하니 라인 103 의 코드가 실행됩니다. kneeCount 는 getter 만 제공하지만 toeCount 는 setter 도 제공하고 있어서 라인 119 와 같은 방법으로 setter 를 호출합니다.

Read-only computed property 의 경우에는 위 코드 <i> 의 라인 97 처럼 get 을 명시해도 되지만, 아래 <i-1> 처럼 생략해도 됩니다.

<i-1>

```
95 class Monster:Mammal {
96     var kneeCount : Int {
97         return legCount
98     }
99     var toeCount : Int {
```

Stored property 에는 willSet 과 didSet 을 붙일 수 있습니다. 이것을 Property Observer 라 부릅니다. 이렇게 하면 해당 값이 설정되었을 때 특정 코드가 실행되게 하는 것이 가능해집니다. 이 괴물은 eyeCount 를 설정하면 그 갯수에 따라 다리의 갯수가 변하는군요. eyeCount 는 stored property 이기 때문에 123 값을 유지하게 되고 이 값이 설정되면 legCount 를 eyeCount 의 두 배로 유지하게 됩니다.

Optional Chain

Swift 에서 객체의 Optional 을 이용할 때 편리한 기능 중 하나가 Optional chain 이라는 것입니다. 원래는 Optional 을 unwrap 한 후에 사용해야 하는데, unwrap 하지 않고도 쓸 수 있도록 해 주는 기능입니다.

```
<j>
125 let animals = [
126     "jindol": Dog(name: "Jindol"),
127     "platypus": Platypus(name: "Platypus")
128 ]
129
130 let cat = animals["cat"]
131 let catdesc = cat?.simpleDescription()
132 catdesc.dynamicType
133
```

```
["platypus": {...}, "jindol": {...}]

nil
nil
Swift.Optional<Swift.String>
```

위 코드 <j> 의 라인 130 을 보면 cat 의 타입은 Mammal? 임을 알 수 있습니다. 실제 값은 nil 이구요. 이 cat 이라는 상수가 nil 인지 체크해서 nil 이 아닐 경우에만 simpleDescription() 등의 함수 호출을 할 수 있습니다. 그런데 Optional 의 값이 nil 일때도 함수호출을 할 수 있는 방법을 제공하는데, unwrap 을 하는 ! 대신 ? 를 쓰는 것입니다. 라인 131 과 같이 합니다. 이렇게 쓰면 cat 이 nil 이 아닐 때는 simpleDescription() 을 호출해 주고, 그렇지 않으면 (nil 이면) simpleDescription() 호출을 생략한 채 그 결과가 nil 이 됩니다. simpleDescription() 함수의 return type 은 원래 String 이지만, Optional chain 으로 호출하게 되면 그 결과가 nil 이 될 수 있기 때문에 전체 수식 결과의 type 은 String? 이 됩니다. 즉 라인 131 의 catdesc 의 type 은 String? 입니다. 값은 nil 이구요.

마침

다섯번째 시간은 여기서 마치도록 하겠습니다. 다음 시간에는 Enumeration, Struct, Protocol, Extension, Generics 등을 다루고 Swift Tour 를 마감하겠습니다.