03. Swift 2.0으로 iOS 코딩하기 - Swift Tour (3)

세번째 시간입니다. 오늘은 기본적인 Control Flow 와 Optional 에 대해 알아봅니다. Optional 은 Swift 에서 매우 중요한 개념이므로 잘 익히도록 해야겠습니다.

Control Flow: for-in, if, if-let

흐름제어를 위해 사용되는 키워드는 C 문법으로부터 넘어온 `if`, `switch-case`, `for`, `while` 등이 있고 Objective-C 에서 넘어 온 `for-in` 이 있습니다. 기본적으로는 비슷한 용법으로 사용되지만 꽤 다른 목적으로 사용되는 것들도 있습니다.

C, C++, Java, Javascript 등과 가장 먼저 만나는 문법상의 차이는 `if`, `for`, `switch` 등의 keyword 뒤에 괄호를 쓰지 않아도 된다는 것입니다. 다만, Curly brace ({}) 는 반드시 써야 합니다. C 나 Objective-C 는 `boolean` 과 `int` 를 구별하지 않지만 Java 는 구별하죠. Swift 도 형(type) 을 명확히 하는 언어답게 `boolean` 과 `int` 를 호환시키지 않습니다. 즉, `int` 값을 `if` 등의 판단조건으로 사용할 수 없다는 뜻입니다. Object reference 역시 `if (obj)` 와 같이 쓸 수 없고 `nil` 이 아닌지 판단하기 위한 표현식을 명시해 주어야만 합니다.

<a>

```
5  let ages = [13, 18, 34, 29]                    [13, 18, 34, 29]
6  var dollar = 0                                 0
7  for age in ages {
8      if age > 19 {
9          dollar += 20                           (2 times)
10     } else {
11         dollar += 15                           (2 times)
12     }
13 }
14 print("Total cost: \(dollar)")                  "Total cost: 70\n"
```

네 가족이 맛있는 식사를 한 모양입니다. 성인은 20불 청소년은 15불이네요. 모두 70불이 되었음을 계산하는 코드입니다. ages 라는 배열의 원소들 하나씩 loop 를 돌기 위해 `for-in` statement 를 사용하였습니다. 위의 코드는 ages 의 각 원소 하나씩 age 에 대입되어 이하의 코드를 실행시키게 됩니다. if 에 조건식을 적으면 이 조건식이 참일 경우 이하 코드가 실행되며 거짓인 경우에는 else 이하의 코드가 실행됩니다. 물론 else 는 생략이 가능합니다.

`if` 와 `if-let` 을 설명하기 위해 Optional 을 간단히 설명해야겠습니다. Optional 이란 Swift 전반적으로 매우 중요하게 사용되는 개념인데, C, C++, Java, Javascript 등 다른 언어에서는 찾아볼 수 없어서 매우 생소할 것입니다. 간단하게 말하면 "어떤 값이 존재하거나 아니면 아무런 값도 아니거나" 라고 할 수 있습니다. 다른 언어에서 가장 가까운 개념을 찾는다면 C 의 포인터 혹은 Java 의 object reference 라고 할 수 있을텐데, 어떤 값을 가리켜서 그 값이 의미가 있거나, 아니면 NULL 값을 가져서 아무런 의미도 없

는 상태를 말합니다. 그런데 이런 reference 및 NULL 로 할 수도 있는 개념을 왜 어렵게 optional 이란 것을 도입했는가? 하는 의문이 생길 수 있을텐데, 이에 대한 저의 생각을 적어 보겠습니다.

어떤 함수가 이름을 찾아 String 으로 리턴해야 한다고 합시다. 찾았을 경우는 문제가 없지만, 찾지 못했을 때 이 함수가 어떻게 해야 할까요? 이럴 때 전통적으로 해 오던 방식이 NULL 을 리턴하는 것입니다. 그러면 호출하는 쪽에서는 리턴값을 보고 NULL 인지 확인하면 찾았는지의 여부를 확인할 수 있습니다. 그럼 이번엔 문자열에서 특정 문자의 위치를 찾는 함수를 생각해 볼까요? 위치를 찾았다면 0 이상의 index 를 리턴하면 됩니다. 하지만 찾지 못했다면? 많은 언어들에서 이런 기능을 하는 API 는 -1 을 리턴합니다. 호출하는 쪽에서는 리턴된 index 가 음수라는 것은 말이 안되니 못찾은 것이구나 하고 실패로 처리하면 되죠. 이 경우는 다행히 정수 범위 내에 의미 없는 값이 있을 수 있기 때문에 의미 없는 값을 에러로 간주할 수 있었습니다.

그런데 모든 정수가 의미 있을 수 있는 경우는 어떨까요? 특정 사건이 얼마나 남았는지를 millisecond 단위로 리턴하는 어떤 함수는 해당 사건이 이미 지나버렸으면 지나버린 만큼 음수를 리턴합니다. 그런데, 그 사건이 아예 존재하지 않을 수도 있다고 해 보죠. 그러면 이 함수가 사건을 찾지 못했을 때의 리턴값은 무엇으로 해야 할까요? 이런 함수는 각 언어적인 특성을 많이 타게 됩니다. 어떤 언어는 두 개 이상의 리턴값을 전달할 수 있는 기능을 이용하기도 하고, 어떤 언어는 성공/실패 여부를 리턴하고 실제 찾은 값은 reference type 으로 전달한 함수 인자로 되돌려 받기도 합니다. 또 어떤 언어는 exception 을 발생시키는 방법을 쓰기도 합니다. 정 안되면 INT_MAX 같은 값을 에러값으로 간주하기도 하죠. 하지만 이런 상황이 리턴값에만 존재하는 것이 아니라 함수 파라미터에 사용될 수도 있고 객체의 멤버 변수에 적용되어야 할 수도 있는데, 그때마다 pair 로 쓰게 되는 것도 직관적이지 않죠.

Swift 에서는 이러한 경우를 직관적 (이라고 썼지만 처음 이해하는 것이 쉽지 않네요) 으로 표현하기 위해 Optional 의 개념을 도입하였습니다. 그로 인해 reference type 이 아닌 것들도 "값이 존재하지 않을 수 있음" 의 상태를 가질 수 있게 되었습니다. 익숙해 지고 나면 매우 편리하게 사용할 수 있습니다.

두번째 시간에서 Dictionary 의 값을 가져오면 Optional 이 된다고 했습니다. `[String:Int]` 타입의 dictionary 인 `ages` 에서 `ages["Kim"]` 을 하면 그 결과의 type 은 `Int?` (Optional Int 라고 읽습니다) 입니다. 해당 키로 값을 조회했더니 값이 존재할 수도 있고 그 키를 찾지 못해 값이 전혀 없을 수도 있죠. 존재하지 않을 때 `0` 을 리턴하면 곤란한 경우이고, `nil` 이 리턴되며, 이는 `0` 과 다릅니다. `nil` 은 존재하지 않는 것이며 `0` 은 존재하는 값이니까요.

<b-1>

```
5  let ages = [ "Kelly":13, "Mac":18, "John":34, "Kim":29 ]
6  var dollar = 0
7  var names = [ "Kelly", "Grace", "Kim" ]
8  for name in names {
9      let age = ages[name]
10     if age != nil {
11         if age! > 19 {
12             dollar += 20
13         } else {
14             dollar += 15
15         }
16     } else {
17         dollar += 20
18     }
19 }
20 print("Total cost: \(dollar)")
```

```
["Kelly": 13, "John": 34, "Kim": 29, "Mac": 18]
0
["Kelly", "Grace", "Kim"]

(3 times)


55

15


35


"Total cost: 55\n"
```

이 식당은 정책이 엄격하군요. Kelly, Mac, John, Kim 은 나이를 알 수 있어서 20불/15불을 받는데, Grace 의 나이는 확인할 방법이 없어서 성인 요금을 받습니다. `ages[name]` 을 통해 세 개의 이름이 전달되는데, Kelly 와 Kim 의 경우 나이를 확인할 수 있어서 값이 각각 13 과 29 로 들어 있지만 Grace 의

경우는 해당 키가 존재하지 않으므로 값은 `nil` 이 됩니다. 일단 optional 의 값이 `nil` 이 아닌지 확인한 10 라인 이하의 코드에서는 이 optional 의 값을 꺼내어 사용해야 합니다. 이것을 Unwrap 한다고 합니다. unwrap 할 때 사용하는 operator 가 `!`(느낌표) 입니다. Swift 의 optional 은 반드시 `nil` 이 아닐 때에만 unwrap 을 해야 하며, `nil` 인 상태의 optional 값을 unwrap 하려고 시도하면 런타임 에러를 발생시킵니다. 이는 다른 언어의 Null Pointer Exception 과 유사한 상황이 됩니다.

(첨언) 위 11 라인에서 age 에 `!` 를 붙이지 않아도 컴파일하는데에는 문제가 없는데, 이는 `>` 라는 operator 가 `Int` 뿐 아니라 `Int?` 에 대해서도 동작하도록 준비되어 있기 때문입니다. 하지만 지금은 Optional 에 대해 처음 배우고 있고, Optional 의 값을 꺼내어 사용할 때에는 unwrap 이라는 과정을 통하는 것이 일반적이므로 `!` 를 붙여야 하는 것으로 이해하고 넘어가는 편이 좋습니다.

<b-2>

```
8   for name in names {
9       if let age = ages[name] {
10          if (age > 19) {
11              dollar += 20
12          } else {
13              dollar += 15
14          }
15      } else {
16          dollar += 20
17      }
18  }
```

위 코드 <b-1> 중 8 라인 이하의 `for` loop 는 위 <b-2> 처럼 쓸 수도 있습니다. 바로 optional 이 될 수 있는 값에 `if-let` 을 사용하여 바로 unwrap 을 하는 것입니다. `if-let` 으로 지정된 상수는 그 값이 nil 이 아닐 때에만 이하 코드가 실행되고, `nil` 인 경우 `else` 의 코드가 실행됩니다. `nil` 이 아닐 때에만 `if-let` 으로 대입되므로 `if-let` 내의 코드에서는 unwrap 된 값으로 간주하여 사용할 수 있습니다. 즉 ages[name] 은 `Int?` 이지만, 코드 <b-2> 에서 `if-let` 으로 지정된 age 는 `Int?` 가 아니라 `Int` 가 됩니다. `if-let` 에 의해 이미 unwrap 이 되었으므로 `if` 내부에서 `!` 를 쓰지 않아야 합니다.

Swift 2.0 에서는 `guard` 라는 키워드가 추가되었습니다. 이는 `if-let` 의 불편함을 극복합니다. 보통 `if-let` 을 쓸 때는 optional 의 결과가 `nil` 이 아닐 경우 그 다음 step 을 진행하기 위해서인데, 그렇게 여러 가지의 조건을 쓰게 되면 들여쓰기의 단계가 많아지게 마련이죠. 게다가 `if-let` 구문을 쓰면 해당 `if` 의 scope 내에서만 optional 을 unwrap 하게 되어 그 밖에서는 쓸 수 없다는 단점도 가지고 있습니다.

`guard` 는 이와 반대로 하는 개념입니다. 즉 `if-let` 이 어떤 조건이 만족할 때 처리하는 것이라면, `guard` 는 어떤 조건을 정해주고 그렇지 않을때는 예외처리하겠다 라는 방식으로 접근합니다.

먼저 다음 코드를 보겠습니다. type 이나 함수는 아직 배우지 않은 것들이니 `if-let` 만 보도록 하죠.

<b-3>

```
let queue = NSOperationQueue()
queue.addOperationWithBlock {
    let imageUrl = NSURL(string: urlString)
    if let imageData = NSData(contentsOfURL: imageUrl!) {
        if let image = UIImage(data: imageData) {
            let mainQueue = NSOperationQueue.mainQueue()
            mainQueue.addOperationWithBlock {
                if let cell = tableView.cellForRowAtIndexPath(indexPath) {
                    cell.imageView?.image = image
                }
            }
        }
    }
}
```

어느 언어에서나 만나게 되는 Pyramid of doom 이라고 불리는 코드입니다. 특정 조건들이 여러 개가 만족되어야 하기 때문에 `if-let` 을 계속 타고 들어가도록 코딩을 했습니다. 이것을 early-return 이라고 불리는 코드로 만들어 보겠습니다.

<b-4>

```
let queue = NSOperationQueue()
queue.addOperationWithBlock {
    let imageUrl = NSURL(string: urlString)
    if imageUrl == nil { return }
    let imageData = NSData(contentsOfURL: imageUrl!)
    if (imageData == nil) { return }
    let image = UIImage(data: imageData!)
    if (image == nil) { return }
    let mainQueue = NSOperationQueue.mainQueue()
    mainQueue.addOperationWithBlock {
        let cell = tableView.cellForRowAtIndexPath(indexPath)
        cell?.imageView?.image = image
    }
}
```

`let` 으로 값을 지정한 다음 그 결과가 `nil` 이면 `return` 을 하는 방식으로 구현되었습니다. 들여쓰기가 여러 번되지 않은 장점은 있지만, 이후 코드에서 unwrap 이 되지 않아 사용이 불편한 부분이 있습니다. `guard` 를 사용하면 이 문제를 해결할 수 있습니다.

<b-5>

```
let queue = NSOperationQueue()
queue.addOperationWithBlock {
    guard let imageUrl = NSURL(string: urlString) else {
        return
    }
    guard let imageData = NSData(contentsOfURL: imageUrl) else {
        return
    }
    guard let image = UIImage(data: imageData) else {
        return
    }
    let mainQueue = NSOperationQueue.mainQueue()
    mainQueue.addOperationWithBlock {
        guard let cell = tableView.cellForRowAtIndexPath(indexPath) else {
            return
        }
        cell.imageView?.image = image;
    }
}
```

guard 를 사용하면 <b-5> 와 같이 nil 일 경우 else 절을 실행하고 nil 이 아닌 경우는 let 으로 unwrap 까지 되는 편리함을 누릴 수 있습니다.

Control Flow: switch

다음은 switch 에 대해 알아보겠습니다. C 언어가 switch-case 문법을 도입한 이래 많은 언어에서 이 문법을 차용했는데, Javascript 외에는 거의 case 에 primitive type 의 상수만 사용할 수 있을 뿐, 문자열조차 사용할 수 없습니다. 그런데 Swift 에서는 case 가 매우 강력한 패턴매칭을 사용합니다. 이 패턴매칭은 switch-case 뿐 아니라 if 에서도 사용이 가능하니 익숙해지면 매우 편리합니다.

상수 외에 변수나 수식을 사용할 수 있다는 것 외에도 Swift 의 case 는 몇 가지 특징을 가지고 있습니다.

- case 를 종료하기 위해 break 키워드를 사용하지 않습니다. 하나의 case 가 선택되어 코드가 실행되면 다음 case 가 나오는 곳까지 실행되고 switch 를 빠져나갑니다. C 등의 언어와 반대로, 다음 case 에 적힌 것도 실행하려면 fallthrough 라는 키워드를 별도로 사용합니다. 즉, 빠져나갈 때 키워드를 쓰지 않고 흘려내려갈 때 키워드를 씁니다. 흘려내려가는 경우가 빠져나가는 경우보다 드문 일이라는 것을 언어가 반영하고 있죠. 단, 특정 케이스에서 실행할 코드가 전혀 없을 때는 break 키워드를 씁니다.

<c>

```
let integerToDescribe = 5
var description = "The number \(integerToDescribe) is"
switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " a prime number, and also"
    fallthrough
default:
    description += " an integer."
}
print(description)
// prints "The number 5 is a prime number, and also an
           integer."
```

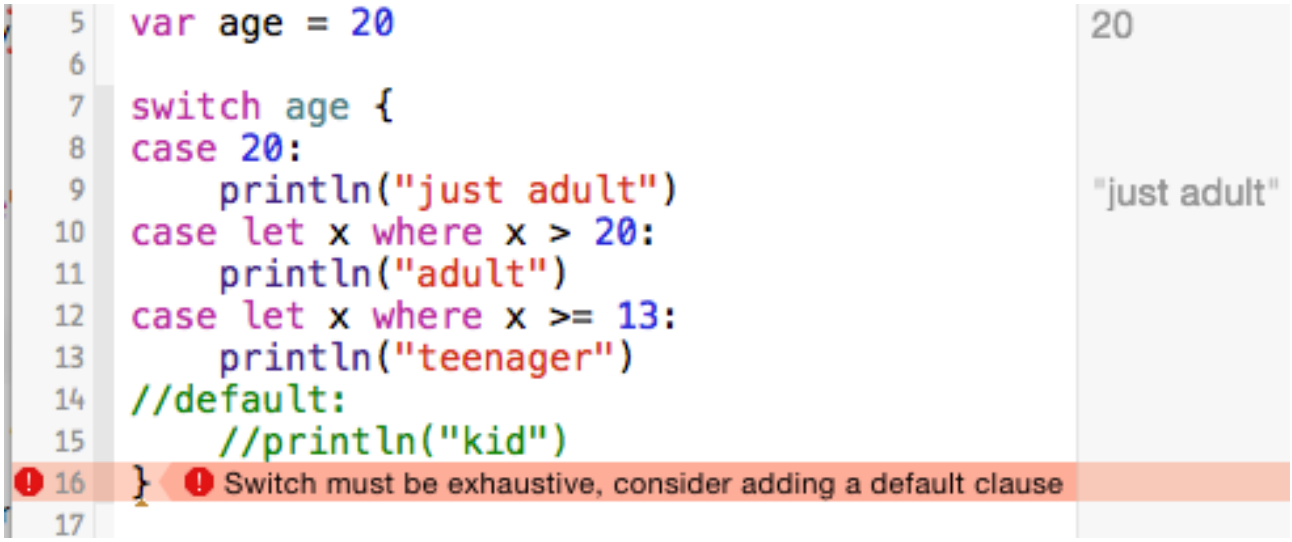
- 여러 개의 `case` 로 같은 코드를 실행할 때는 , (comma) 로 구분합니다. 다른 언어에서 여러 개의 `case` 를 열거하는 것과 다릅니다.

<d>

```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
for character in puzzleInput.characters {
    switch character {
    case "a", "e", "i", "o", "u", " ":
        continue
    default:
        puzzleOutput.append(character)
    }
}
print(puzzleOutput)
// prints "grtmndsthnklk"
```

- 첫번째 `case` 가 만족되지 않으면 다음 `case` 가 만족되는지 검사하여 마지막 `case` 까지 모든 경우가 커버되어야 합니다. 컴파일러가 모든 경우가 커버되지 않는다고 판단하면 에러를 냅니다. 따라서 이런 경우 마지막에 `default:` 를 넣어 주어야 합니다. 아래 <e> 코드에서 `default:` 가 없으면 아래와 같은 에러가 납니다.

<e>



Swift 에서 `case` 는 차례대로 적용되기 때문에 순서가 중요할 수 있습니다. 코드 <e> 에서 10 라인의 `case` 와 12 라인의 `case` 의 순서가 바뀌면 13 이상인지 비교해서 먼저 처리해버리므로 20 보다 큰 비교는 의미가 없어집니다.

위 <e> 에서는 `case-let` 을 이용하여 `switch` 에 제공된 값을 x 에 받은 다음 `where` 뒤에 수식을 쓰는 테스트를 할 수 있습니다. 또한 아래 <f> 와 같이 Range Operator 를 사용하여 범위를 지정해 줄 수도 있습니다. Range operator 는 양쪽이 모두 inclusive (포함) 인 `...` 과 왼쪽은 inclusive 이고 오른쪽은 exclusive (제외) 인 `..<` 의 두 가지가 있습니다. 즉, `10...99` 와 `10..<100` 은 동일한 결과입니다. 또한, 아래 <f> 의 18 라인처럼 긴 숫자의 경우 중간에 `_` (underscore) 를 넣어도 컴파일러는 무시하므로 읽기 쉬운 코드를 쓸 수 있는 장점이 있습니다.

<f>

```
let count = 1000                                    1000
var prefix:String
switch count {
case 0:
    prefix = "no"
case 1...3:
    prefix = "a few"
case 4...9:
    prefix = "several"
case 10...99:
    prefix = "tens of"
case 100..<1000:
    prefix = "hundreds of"
case 1000..<1_000_000:
    prefix = "thousands of"                         "thousands of"
default:
    prefix = "so many"
}

print ("\(prefix) stars")                           "thousands of stars\n"
```

Control Flow: for-in

Array 에 대해 Loop 를 돌았던 `for-in` 에 대해 조금 더 알아보겠습니다.

Dictionary 에 `for-in` 을 사용하는 경우, (key, value) Tuple 을 써서 각 element 에 대한 정보를 알아올 수 있습니다. Tuple 이라는 개념은 처음 나오는데, 나중에 다시 설명하기로 하고, 일단 괄호로 묶인 두 개 이상의 값이라는 정도로만 이해하기 바랍니다.

<g>

```
let ages = [ "Kelly":13, "Mac":18, "John":34, "Kim":29 ]    ["Kelly": 13, "John": 34, "Kim": 29, "Mac": 18]
for (name, age) in ages {
    let msg = "\(name) is \(age)"                           (4 times)
```

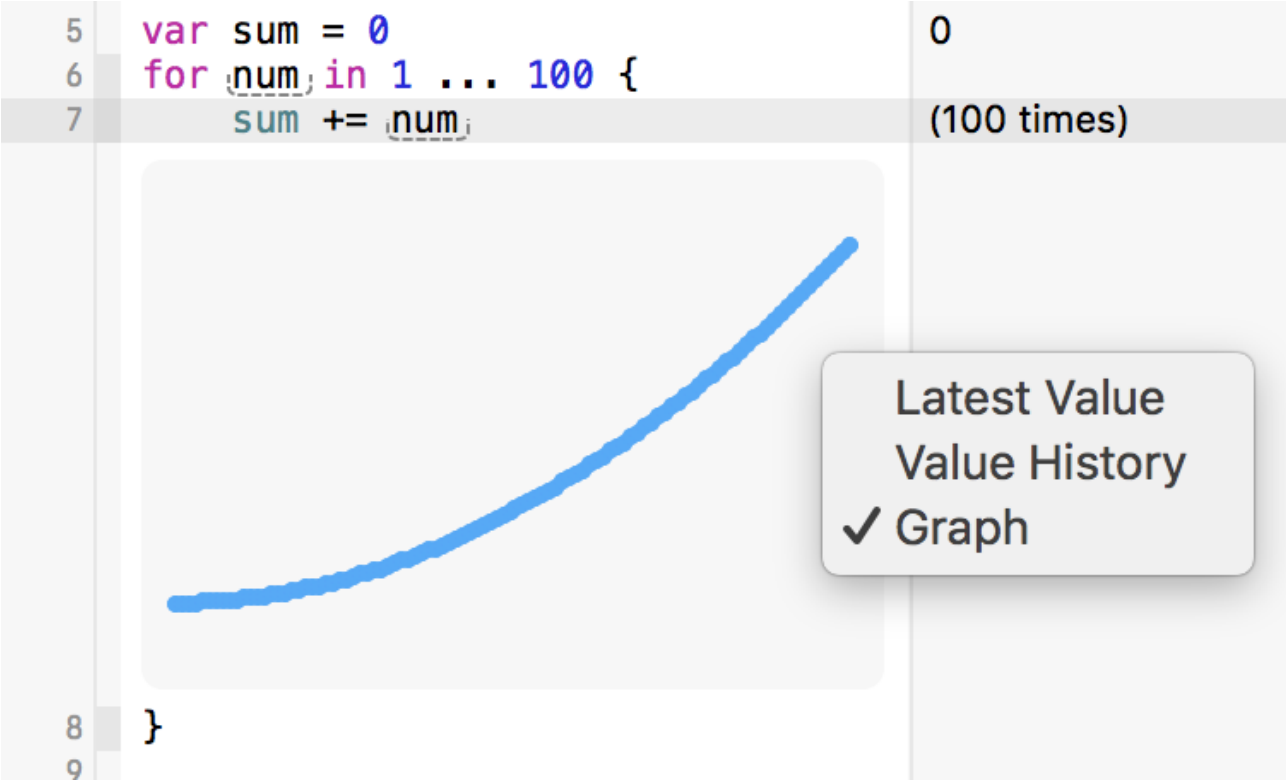
```
Kelly is 13
John is 34
Kim is 29
Mac is 18
```

```
}
```

위 코드 <g> 의 6 라인과 같은 표현을 통해 loop 내에서 각 element 의 key 와 value 를 접근할 수 있습니다. 다시 한 번 말하지만 Dictionary 는 저장된 순서를 희생하고 찾는 속도를 높인 자료구조이므로 순서은 어떻게 나올 지 알 수 없습니다.

XCode Playground 의 새로운 기능을 소개하게 되었는데, 원래 각 코드에서 대입 혹은 출력하는 내용이 오른쪽에 나오게 되는데, loop 내의 코드의 경우 여러 번 실행되므로 한 줄에 표현이 불가능해집니다. 이럴 때는 4 times 와 같은 형태로 출력되는데, 이때 오른쪽에 마우스 커서를 올려 동그라미를 누르게 되면 각 내용을 확인할 수가 있습니다. 문자열이 아니고 숫자라면 그 값의 변화를 아래 <h> 처럼 그래프로 확인시켜 주기도 합니다.

<h>



그래프 영역에 마우스 오른쪽 버튼을 누르면 나오는 세 개의 메뉴는 각각 최종값, 모든 값, 그래프 입니다.

Control Flow: while, repeat-while, for, do

C 기반의 다른 언어처럼 `while` loop 가 있습니다. 그런데, 이런 언어들과 달리 `do-while` 은 채용하고 있지 않습니다. 2.0 이전에는 `do-while` 이 있었지만, 2.0 으로 넘어오면서 `do-while` 이 없어지고 `repeat-while` 로 변경되었습니다. Swift 에서의 `do` 는 다른 의미로 사용됩니다.

C Style 의 `for` 도 가능합니다. `for var i = 0; i < 10; i++ {}` 와 같은 형태로 사용합니다.

`do` 는 다른 언어의 try 블럭과 비슷한 용도로 사용됩니다.

<i>

```
do {
    try canThrowAnError()
    // no error was thrown
} catch {
    // an error was thrown
}
```

<i> 에서 보는 것과 같이, try 를 블럭에 쓰지 않고 하나의 statement 앞에 씁니다. Error Handling 은 나중에 따로 다루는 편이 좋을 것 같습니다.

마침

다음 시간에는 Swift 에서 Optional 못지 않게 중요한 Closure 를 Function 과 함께 다루겠습니다. 다음 시간에 만나요.