공식문서 읽기 : Closure

Closures - The Swift Programming Language (Swift 5.5)

Closures are self-contained blocks of functionality that can be passed around and used in your code. Closures in Swift are similar to blocks in C and Objective-C and to lambdas in other programming languages. Closures can capture and store references to any constants and variables

https://docs.swift.org/swift-book/LanguageGuide/Closures.html#//apple_ref/doc/uid/TP40014097-CH11-ID94

Closure

- 클로저는 코드에서 전달, 사용할 수 있는 두가지의 자체 기능을 가진 블록입니다.
- 정의된 컨텍스트에서 모든 상수 및 변수에 대한 참조를 캡처하고 저장할 수 있습니다. (캡처: 블록 외부에 있는 값을 참조하는 행위)

Closure의 세 가지 형태

- 전역함수(global function)는 이름이 있고 값을 캡쳐하지 않는 클로저입니다.
- 중첩함수(nested function)는 이름이 있고 둘러싸는 함수에서 값을 캡처할 수 있는 클로저입니다.

- 클로저 표현(closure expression)은 경량화된 문법으로 쓰여지고, 관련 컨텍스트로부터 값을 캡처할 수 있는 이름이 없는 클로저입니다.

클로저의 표현식은 간단하고 깔끔한 구문을 지향합니다.

클로저의 최적화에포함되는 내용

- 컨텍스트에서 매개변수 및 반환 타입 추론
- 단일 표현식 클로저의 암시적 반환
- 축약된 인자 이름
- 후행 폐쇄 구문(후위 클로저 문법)

Closure Expressions

중첩함수보다 클로저를 사용할 때 유용한 경우가 있습니다. 함수가 인자로 하나 이상의 다른 함수를 받을 때 특히 더 그렇습니다. 클로저 표현은 인라인 클로저를 간단하게 작성할 수 있는 방법입니다.

The Sorted Method

Swift의 표준 라이브러리는 배열의 값을 정렬하는 sorted(by:) method를 제공합니다. 정렬 후 새로운 배열을 반환하는 method로, 원본에는 영향을 미치지 않습니다.

```
func backword(_ s1: String, s2: String) -> Bool {
  return s1 > s2
}

var reversedNames = names.sorted(by: backword)
```

Closure Expression Syntax

클로저 표현 문법의 기본적인 형태는 아래와 같습니다.

```
{ (parameters) -> return type in
  // statements
}
```

클로저의 매개변수는 in-out parameters가 될 수 있지만, 기본값을 가질수는 없습니다. 매개변수 집합을 사용할 수도 있고, 매개변수 타입과 반환 타입에 튜플을 사용할 수도 있습니다.

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
  return s1 > s2
})
```

backward(_:_:)로 선언하는 것과 위와 같이 작성하는 것은 동일한 방법입니다. 양쪽 모두 파라미터와 리턴 타입이 작성됩니다. 하지만 인라인 클로저는 매개변수와 반환타입을 바깥이 아닌 중괄호 안에 작성합니다.

클로저의 바디는 in 키워드로 구분됩니다.

in 전 : 파라미터와 리턴 타입

in 후 : 클로저 바디, 실행 구문

Inferring Type From Context

(컨텍스트에서의 타입 추론)

정렬 클로저가 메소드의 인자로 들어가기 때문에 이를 통해 매개변수와 반환타입을 추론할 수 있습니다.

sorted(by:) 메소드는 문자열 배열에서 호출되며, 인자는 (String, String) → Bool 이어야 합니다. 따라서 이를 직접 작성할 필요가 없습니다.

```
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 })
```

인라인 클로저 표현으로서 함수나 메소드에 클로저를 넣을 때, 항상 매개변수와 반환 타입을 추론할 수 있습니다. 클로저의 전체 형태를 작성하지 않아도 된다는 뜻이며 이는 생략이 가능합니다.

반드시 생략해야 하는 것은 아니기 때문에 코드를 명확하게 표현하기 위해 명시적으로 타입을 표시하는것 또한 가능합니다.

Implicit Returns from Single-Expression Closures

(단일 표현 클로저에서의 암시적 반환)

단일 표현 클로저는 return 키워드를 생략함으로서 결과 값을 암시적으로 반환할 수 있습니다.

```
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 })
```

sorted(by:) 메소드에 인자로 들어간 함수의 타입은 클로저에 의해 Bool 타입을 반환할 것이 명확합니다. 클로저의 바디가 포함하고 있는 단일표현 (s1>s2)가 Bool 값을 반환하기 때문입니다. 이렇게 명확할 때에는 return 키워드를 생략할 수 있습니다.

Shortand Argument Names

(인자 이름의 축약)

Swift는 자동으로 인라인 클로저의 축약된 인자 이름을 제공합니다. 이런 이름은 $0, $1, $2와 같이 표현됩니다.

만약 클로저 표현에서 이런 이름을 사용한다면 클로저의 정의에서 인자 목록을 생략할 수 있으며 in 키워드도 생략 가능합니다.

```
reversedNames = names.sorted(by: { $0 > $1 })
```

Operator Method

(연산자 메소드)

Swift의 String 타입 연산자에는 String끼리 비교할 수 있는 연산자가 구현되어 있습니다. 그렇기 때문에 그냥 이 연산자를 사용하는 것도 가능합니다.

```
reversedNames = names.sorted(by:>)
```

Trailing Closures(후위 클로저)

함수의 마지막 파라미터로 클로저를 넣어야할 때 그 표현이 길다면, 후위 클로저를 이용하는 것이 좋습니다. 후위 클로저는 호출하는 함수의 소괄호 이후 작성합니다.

```
// 파라미터로 클로저를 받는 함수
func someFunctionThatTakesAClosure(closure: () -> Void) {
    // function body goes here
}

// 위 함수 호출 방법
someFunctionThatTakesAClosure(closure: {
    // closure's body goes here
})

// 클로저가 길다면 아래와 같이 작성
someFunctionThatTakesAClosure() {
    // trailing closure's body goes here
}
```

위의 정렬 클로저는 후위 클로저를 사용해 다음과 같이 표현합니다.

```
reversedNames = names.sorted() { $0 > $1 }
```

만약 함수의 파라미터가 후위 클로저 하나뿐이라면 소괄호를 생략하는 것도 가능합니다.

```
reversedNames = names.sorted { $0 > $1 }
```

후위 클로저는 한 줄에 작성할 수 없을 정도로 클로저가 충분히 클 경우에 유용합니다. 대표적인 예로 map(_:) 메서드가 있습니다.

```
let digitNames = [
    0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]
let numbers = [16, 58, 510]

let strings = numbers.map { (number) -> String in
  var number = number
  var output = ""
  repeat {
    output = digitNames[number % 3]! + output
    number /= 10
```

```
  } while number > 0
  return output
}
```

Capturing Values (값 캡처)

클로저가 정의된 주변의 컨텍스트로부터 상수나 변수를 캡처할 수 있습니다. 클로저는 바디에서 이 상수/변수의 값을 보내고 수정할 수 있습니다. 원본 스코프가 존재하지 않더라도 캡처하고 있다면 가능합니다.

Swift에서 값을 캡처할 수 있는 가장 간단한 방법은 nested function(중첩함수)입니다. 중첩 함수는 바깥 함수의 인자나 그 안에서 선언된 상수/변수를 캡처해 사용할 수 있습니다.

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
  var runningTotal = 0
  func incrementer() -> Int {
    runningTotal += amount
    return runningTotal
  }
  return incrementer
}
```

위 함수의 반환값 타입은 `() → Int` 입니다. 이 함수는 매개변수가 없으며 Int를 반환합니다.

incrementer() 함수만 따로 놓으면 일반적인 함수처럼 보이지 않습니다.

```
func incrementer() -> Int {
  runningTotal += amount
  return runningTotal
}
```

incrementer() 함수는 매개변수가 없고, 블록 안에 runningTotal과 amount도 없습니다. 이는 상위 함수로부터 runningTotal과 amount의 참조를 캡처해 와서 incrementer()의 바디 안에서 사용하는 방식으로 작동합니다. makeIncrementer()가 종료되어도 참조의 캡처는 사라지지 않습니다.

위 함수를 사용한 예시입니다.

```
let incrementByTen = makeIncrementer(forIncrement: 10)

incrementByTen()
// returns a value of 10
incrementByTen()
// returns a value of 20
incrementByTen()
// returns a value of 30

let incrementBySeven = makeIncrementer(forIncrement: 7)
incrementBySeven()
// returns a value of 7

incrementByTen()
// returns a value of 40
```

처음 incrementByTen()를 호출할 때에는 runningTotal에 10씩 더하게 됩니다.

두번째로 호출한다면 새로운 runningTotal의 참조를 가지게 됩니다.

만약 클로저를 어떤 클래스의 인스턴스 프로퍼티로 할당하고, 그 클로저가 그 인스턴스를 캡처링하면 강한 순환참조에 빠지게 됩니다. 즉, 인스턴스의 사용이 끝나도 메모리 해제를 하지 못하는 것입니다.

이 문제를 해결하기 위해 Swift에서는 capture list를 사용합니다.

클로저는 참조 타입(Reference Type)입니다.

위 예제에서 incrementBySeven, incrementByTen은 상수입니다. 그럼에도 runningTotal 변수를 계속해서 증가시킬 수 있었던 이유는, 함수와 클로저는 참조 타입이기 때문입니다.

함수와 클로저를 상수나 변수에 할당할 때 실제로는 상수와 변수에 해당 함수나 클로저의 참조가 할당됩니다. 따라서 한 클로저를 두 상수나 변수에 할당하면 그 두 상수나 변수는 같은 클로저를 참조합니다.

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()
// returns a value of 50

incrementByTen()
// returns a value of 60
```

이스케이핑 클로저 (Escaping Closure)

클로저가 함수의 인자로 들어갔지만, 함수가 종료된 후 호출될 때 클로저는 함수를 벗어났다(escaping)고 합니다. 클로저를 매개변수 중 하나로 선언할 때, 이 클로저는 탈출을 허용한다는 뜻으로 @escaping 어노테이션을 매개변수 타입 앞에 작성합니다.

클로저가 탈출할 수 있는 하나의 방법은 함수 바깥에서 정의된 변수에 저장되는 것입니다. 예를 들어, 많은 함수가 클로저를 completion handler로 취하기 위해 비동기로 작동됩니다. 이 함수의 클로저는 조작이 끝날 때까지 호출되지 않음에도 조작을 시작한 이후 반환됩니다. 클로저는 이후에 호출되기 위해 탈출해야합니다.

```
var completionHandlers: [() -> Void] = []
func someFunctionWithEscapingClosure(completionHandler: @escaping () -> Void) {
    completionHandlers.append(completionHandler)
}
```

위 함수에서 인자로 전달된 completionHandler는 someFunctionWithEscapingClosure 함수가 종료된 후 나중에 처리됩니다. 만약 함수가 끝나고 실행되는 클로저에 @escaping 키워드를 붙이지 않으면 컴파일 에러가 발생합니다.

@escaping 을 사용하는 클로저에서는 self를 명시적으로 언급해야합니다.

```
func someFunctionWithNonescapingClosure(closure: () -> Void) {
    closure()
}

class SomeClass {
    var x = 10
    func doSomething() {
        someFunctionWithEscapingClosure { self.x = 100 }
        someFunctionWithNonescapingClosure { x = 200 }
    }
}

let instance = SomeClass()
instance.doSomething()
print(instance.x)
// Prints "200"

completionHandlers.first?()
print(instance.x)
// Prints "100"
```

자동 클로저 (Autoclosures)

자동 클로저는 인자 값이 없으며, 특정 표현을 감싸 다른 함수에 전달인자로 사용하도록 하는 클로저입니다. 자동클로저는 클로저를 실행하기 전까지 실제로 실행되지 않습니다. 그래서 계산이 복잡한 연산을 하는 데 유용합니다. 실제로 계산이 필요하기 전에는 사용되지 않기 때문입니다.

```
var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
print(customersInLine.count)
// Prints "5"

let customerProvider = { customersInLine.remove(at: 0) }
print(customersInLine.count)
// Prints "5"

print("Now serving \(customerProvider())!")
// Prints "Now serving Chris!"
print(customersInLine.count)
// Prints "4"
```

customersInLine 배열의 첫 번째 원소가 클로저 안의 코드에 의해 지워졌음에도 클로저가 호출될 때까지 배열의 원소는 실제로 삭제되지 않습니다. 만약 클로저가 호출되지 않는다면 클로저 안의 표현 역시 사용되지 않습니다.

함수의 인자로 클로저를 넣음으로써 같은 효과를 얻을 수 있습니다.

```
// customersInLine is ["Alex", "Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: () -> String) {
    print("Now serving \(customerProvider())!")
}
serve(customer: { customersInLine.remove(at: 0) } )
// Prints "Now serving Alex!"
```

위 방법에서 serve 함수는 인자로 ()→String 즉, 인자가 없고 String을 반환하는 클로저를 파라미터로 받습니다.

그리고 이 함수를 실행할 때는 클로저 { customersInLine.remove(at: 0) }를 명시적으로 직접 넣을 수 있습니다.

위와 같이 함수의 인자로 클로저를 넣을 때 명시적으로 넣는 대신 , @autoclosure 키워드를 이용할 수 있습니다. 이제 클로저 대신 String 인자를 받는 함수를 호출할 수 있습니다. customerProvider 매개변수 타입이 @autoclosure 속성으로 마크됐기 때문에, 인자는 자동적으로 클로저로 변환됩니다.

```
// customersInLine is ["Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: @autoclosure () -> String) {
    print("Now serving \(customerProvider())!")
}
serve(customer: customersInLine.remove(at: 0))
// Prints "Now serving Ewa!"
```

자동클로저가 탈출하는 것을 허용하고 싶을 경우, @autoclosure, @escaping 어노테이션을 모두 사용할 수 있습니다.