Swift4 Closure 정리.

Chap. 13 클로저

클로저

- 일정 기능을 하는 코드를 하나의 블록으로 모아놓은 것. 함수는 클로저의 한 형태라고 할 수있음.
- 일회용 함수를 작성할 수 있는 구문. 일회용 함수는 한 번만 사용하면 되므로 굳이 이름을 작성할 필요 없으므로 생략할 수 있음.
- Obj-C의 블록, 자바나 파이썬의 람다 함수와 유사한 역할.

클로저는 변수나 상수가 선언된 위치에서 참조를 획득하고 저장할 수 있음. 이것을 변수나 상수의 클로징(잠금)이라고 하며, 클로저라는 이름은 여기서 착안.
클로저는 다음과 같은 세 가지 경우 중 하나에 해당한다.

1. **전역 함수** 이름이 있으며, 주변 환경에서 캡처할(획득할) 어떤 값도 없는 클로저
2. **중첩 함수** 이름이 있으며 자신을 둘러싼 함수로부터 값을 캡처할 수 있는 클로저
3. **클로저 표현식** 이름이 없으며 주변 환경으로부터 값을 캡처할 수 있는 경량 문법으로 작성된 클로저

클로저 표현 방법 : 클로저가 함수가 아닌 하나의 블록의 모습으로 표현될 수 있는 방법.
위치에 따라 **'기본 클로저'**와 **'후행 클로저'**로 나눌 수 있음.

13.1 기본 클로저

- Sorted(by:) 메소드를 통해 동일한 기능을 하는 코드를 표현하는 방법

Sorted(by:) 는 배열의 값을 정렬시켜주는 기능을 하는 메소드.

```
public func sorted(by areInIncreasingOrder: (Element, Element) -> Bool) ->
[Element]
```

Sorted(by:) 메소드는 첫 번째 전달인자 값이 두 번째 전달인자 값보다 먼저 배치되어야 하는지에 대한 결과를 Bool값으로 리턴함.
다음 예제는 매개변수로 String 타입을 두 개 가지고, Bool값을 반환하는 함수를 구현하고, 구현된 함수를 sorted(by:)메소드의 전달인자로 전달하여 reversed라는 이름의 배열로 반환받는 예제.

예제 13.1

```
let names: [String] = ["wizplan", "eric", "yagom", "jenny" ]
func backwards(first: String, second: String) -> Bool {
    print("\(first) \(second) 비교 중")
    return first > second
}
let reversed: [String] = names.sorted(by: backwards)
```

```
print(reversed)    // [“yagom”, “wizplan”, “Jenny”, “eric”]
```

위 예제에서 문자열의 크고 작다는, 첫 글자의 알파벳이 뒷 순서일수록 큰 것.
위 backwards 함수는 함수 이름, 매개 변수 등 부가적 표현이 많으므로 클로저 표현으로 다음과 같이 간결히 표현.

```
{ (매개변수들) -> 반환 타입 in
    실행 코드
}
```

```
let names: [String] = ["wizplan", "eric", "yagom", "jenny" ]
//func backwards(first: String, second: String) -> Bool {
//    print("\(first) \(second) 비교 중")
//    return first > second
//}
let reversed: [String] = names.sorted(by: { (first: String, second: String)
-> Bool in return first > second})
print(reversed)    // [“yagom”, “wizplan”, “Jenny”, “eric”]
```

13.2 후행 클로저

위 예제를 후행 클로저 표현 방식으로 더 읽기 쉽게 바꿀 수 있음.
함수나 메소드의 마지막 전달인자로 위치하는 클로저는 **함수나 메소드의 소괄호를 닫은 후 작성해도 됨.**
클로저가 좀 길어지거나 가독성이 떨어진다고 하면 다음과 같이 후행 클로저를 사용하면 좋음. (Xcode의 자동완성 기능이 후행 클로저 사용을 유도함)
단, 후행 클로저는 맨 마지막 전달인자로 전달되는 클로저에만 해당되므로 전달인자로 클로저를 여러 개 전달할 땐 맨 마지막 클로저만 후행 클로저로 사용할 수 있음.
또한 sorted(by:) 처럼 단 하나의 클로저만 전달하는 경우 소괄호를 생략해도 무방.

```
let names: [String] = ["wizplan", "eric", "yagom", "jenny" ]

// 후행 클로저의 사용 sroted( Int, String, (String, String) -> Bool)
let reversed: [String] = names.sorted() { (first: String, second: String) -
> Bool in return first > second}

// sorted(by: ) 메소드의 소괄호 생략
let reversed2: [String] = names.sorted { (first: String, second: String) ->
Bool in return first > second}
```

13.3 클로저 표현 간소화

13.3.1 문맥을 이용한 타입 유추

메소드의 전달인자로 전달하는 클로저는 메소드에서 요구하는 형태로 전달해야함.
-> 매개변수의 타입, 개수, 반환 타입등이 같아야 전달인자로서 전달 가능.
-> 전달인자로 전달할 클로저는 이미 적합한 타입을 준수한다고 유추 가능.
-> 전달인자로 전달하는 클로저를 구현할 때는 매개변수의 타입, 반환값등을 생략할 수 있음.

```
let names: [String] = ["wizplan", "eric", "yagom", "jenny" ]
```

```
// 클로저의 매개변수 타입과 반환 타입을 생략하여 표현할 수 있음.
let reversed: [String] = names.sorted { (first, second) in return first >
second }
print(reversed)
```

13.3.2 단축 인자 이름

매개 변수의 이름 first와 second를 단축 인자 이름으로 간결히 표현 가능
단축 인자 이름은 첫 번째부터 순서대로 $0, $1, $2, … 로 표현하며, 단축 인자 표현을 사용하면 매개변수 및 반환타입과 실행 코드를 구분하기 위해 사용했던 'in'을 사용할 필요가 없음.

```
let names: [String] = ["wizplan", "eric", "yagom", "jenny" ]

let reversed: [String] = names.sorted { return $0 > $1 }
print(reversed)
```

13.3.3 암시적 반환 표현

클로저가 반환값을 갖는 클로저이고, 클로저 내부의 실행문이 단 한 줄이라면, 암시적으로 'return'을 생략할 수 있음.

```
let names: [String] = ["wizplan", "eric", "yagom", "jenny" ]

let reversed: [String] = names.sorted { $0>$1 }
print(reversed)
```

13.3.4 연산자 함수

클로저는 매개변수의 타입과 반환 타입이 연산자를 구현한 함수의 모양과 동일하다면 연산자만 표기하더라도 알아서 연산하고 반환함.

```
let names: [String] = ["wizplan", "eric", "yagom", "jenny" ]

let reversed: [String] = names.sorted(by: >)
print(reversed)
```

13.4 값 획득

클로저는 자신이 정의된 위치의 주변 문맥을 통해 **상수나 변수를 획득(캡쳐)** 할 수 있음.
값 획득을 통하여 주변에 정의한 상수나 변수가 더 이상 존재하지 않더라도 해당 상수나 변수의 값을 자신 내부에서 참조하거나 수정할 수 있음. (클로저는 비동기 작업에 많이 사용.)
예를 들어 중첩 함수의 경우, 자신을 포함하는 함수의 지역변수나 지역상수를 획득할 수 있음.

다음 예제는 incrementer라는 함수를 중첩 함수로 포함하는 makeIncrementer 함수로 값 획득에 대해 공부.

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

```
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}
```

makeIncrementer의 반환 타입은 " () -> Int " 형식의 클로저, 함수 객체를 반환한다는 의미.
반환하는 함수는 매개변수를 받지 않고 반환 타입은 Int인 함수. -> 호출할 때마다 Int값 반환.

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

incrementer를 밖으로 빼내어 따로 두면, 작동할 수 없는 형태가 됨.
함수 incrementer는 매개변수를 받지도 않으며, 구문 안에서 사용되는 runningTotal과 amount가 정의되어 있지 않음.
makeIncrementer의 안에서, 위 언급된 두 변수의 참조를 획득하며, 이 둘은 함수의 실행이 끝나도 사라지지 않음.

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

let incrementByTwo: (() -> Int) = makeIncrementer(forIncrement: 2)
let incrementByTwo2: (() -> Int) = makeIncrementer(forIncrement: 2)
let incrementByTen: (() -> Int) = makeIncrementer(forIncrement: 10)

let first: Int = incrementByTwo()   // 호출할 때마다 2씩 증가
let second: Int = incrementByTwo2()

let first2: Int = incrementByTen()  // 호출할 때마다 10씩 증가
let second2: Int = incrementByTen()
```

increment가 할당된 각 상수들은 호출될 때마다 자신만의 runningTotal 을 가지며, 다른 함수의 영향을 주지도 받지도 않음.
각각 자신만의 참조를 미리 획득했기 때문이다.

13.5 클로저는 참조 타입

함수나 클로저를 상수나 변수에 할당하는 것은 "상수나 변수에 함수나 클로저의 참조를 설정하는 것".

```
let incrementByTwo: (() -> Int) = makeIncrementer(forIncrement: 2)
```

```
let sameWithIncrementByTwo: (() -> Int) = incrementByTwo

let first: Int = incrementByTwo()
let second: Int = sameWithIncrementByTwo()
```

13.6 탈출 클로저

클로저를 매개변수로 갖는 함수를 선언할 때 매개변수 이름의 콜론 뒤에 **@escaping** 키워드를 사용하여 클로저가 탈출하는 것을 허용한다고 명시해줄 수 있음.

```
func callback(fn: () -> Void) {
    fn()
}
callback {
    print("closure가 실행되었습니다.")
}
```

정의된 함수 “callback” 은 매개변수를 통해 전달된 클로저를 함수 내부에서 실행하는 역할.

```
func callback(fn: () -> Void) {
    let f = fn
    fn()
}   // Non-escaping parameter 'fn' may only be called
```

위 코드에 오류가 발생하는 이유는 **인자값으로 전달되는 클로저는 기본적으로 탈출 불가(non-escaping)** 이기 때문.
-> 이것은 **1. 함수 내에서 / 2. 직접 실행을 위해서만 사용** 해야 하는 것을 의미.
-> 따라서 위 코드처럼 상수 or 변수에 대입할 수 없음.
만약 변수나 상수에 대입하는 것을 허용한다면, 내부 함수를 통한 캡처(획득) 기능을 통해 클로저가 함수 밖으로 탈출(함수 내부를 벗어나 실행되는 것) 할 수 있다.

```
func callback(fn: () -> Void) {
    func innerCallback(){
        fn()
    }
}
```

같은 이유로 인자값으로 전달된 클로저는 함수 내부에서 중첩된, 내부 함수에서도 사용할 수 없음. (내부 함수에서 사용할 수 있게되면, 컨텍스트(Context)의 캡처를 통해 탈출될 수 있음.)

코드를 작성하다보면 인자값으로 전달된 클로저를 변수나 상수에 대입하거나, 중첩 함수 내부에서 사용해야 하는 경우가 있는데 이 때 사용하는 것이 **@escaping** 속성이다.

```
func callback(fn: @escaping () -> Void) {
    let f = fn // 인자값으로 전달된 클로저를 상수 f에 대입
    f() // 대입된 클로저를 실행
}
```

```
callback{
    print("Closure가 실행되었습니다.")
}
```

클로저가 기본적으로 **탈출 불가**의 속성으로 설정된 이유는, 컴파일러가 코드를 최적화하는 과정에서의 성능 향상을 위함이다.
해당 클로저가 함수 밖으로 탈출할 수 없다는 것은 컴파일러가 메모리를 관리하는데 효율적임.

또한 **탈출 불가 클로저** 내에서는 self 키워드를 사용할 수 있지만 생략 가능.
하지만 @escaping 키워드를 사용하여 탈출 가능하게 된 클로저에서 해당 타입의 프로퍼티나 메소드, 서브스크립트 등에 접근하려면 self키워드를 명시적으로 사용해야함.

```
typealias VoidVoidClosure = () -> Void

func functionWithNoescapeClosure(closure: VoidVoidClosure) {
    closure()
}

func functionWithEscpaingClosure(completionHandler: @escaping
VoidVoidClosure) -> VoidVoidClosure {
    return completionHandler
}


class SomeClass {
    var x = 10

    func runNoescapeClosure() {
        //탈출 불가 클로저에서 self 키워드 사용은 선택 사항.
        functionWithNoescapeClosure {  x = 200  }
    }

    func runEscapingClosure() -> VoidVoidClosure {
        // 탈출 클로저에서는 명시적으로 self를 사용해야함.
        return functionWithEscpaingClosure { self.x = 100 }
    }
}

let instance: SomeClass = SomeClass()
instance.runNoescapeClosure()
print(instance.x)
let returnedClosure: VoidVoidClosure = instance.runEscapingClosure()
returnedClosure()
print(instance.x)
```

13.7 자동 클로저

자동 클로저 **(@autoclosure)** : @autoclosure 속성은 인자값으로 전달된 일반 구문이나 함수 등을 클로저로 래핑(Wrapping)하는 역할.

-> 일반 구문을 인자값으로 넣어도 컴파일러가 알아서 클로저로 인식.

이 속성을 적용하면 클로저 인자값을 '{ }' 형태가 아닌 '( )' 형태로 사용할 수 있다는 장점을 갖는다. (하지만 '{ }' 로 작성할 수 없음)

아래는 일반적인 함수의 선언과 실행 방법이다.

```
func condition(stmt: ()-> Bool) {
    if stmt() == true {
        print("결과가 참입니다.")
    } else {
        print("결과가 거짓입니다.")
    }
}

// 실행 방법 1 : 일반 구문
condition(stmt: {4>2})

// 실행 방법 2 : 클로저 구문
condition{4>2}


//STEP 1 : 경량화 되지 않은 클로저 전체 구문
condition {() -> Bool in
    return (4>2)
}

//STEP 2 : 클로저 타입 선언 생략
condition{
    return (4>2)
}

//STEP 3 : 클로저 반환구문 생략
condition{
    4>2
}
```

여기서 선언한 'condition' 함수에 전달하는 인자를 @autoclosure 키워드를 사용하면 다음과 같이 실행 가능하다.

```
func condition(stmt: @autoclosure ()-> Bool) {
    if stmt() == true {
        print("결과가 참입니다.")
    } else {
        print("결과가 거짓입니다.")
    }
}

// 실행 방법
```

```
condition(stmt: (4>2) )
```

지연된 실행 : 자동클로저는 클로저가 호출되기 전까지 클로저 내부의 코드가 동작하지 않음.
-> 연산을 지연시켜 코드의 실행을 제어하기 좋음.

```
var arrs = [String]()

func addVars(fn: @autoclosure () -> Void) {
    arrs = Array(repeating: "", count: 3)
    fn()
}

//arrs.insert("KR", at: 1)  <- 이 줄은 'arrs' 배열의 인덱스가 아직 0이기 때문에 오류가 발생함.

print(arrs)                              // 빈 배열 출력
addVars(fn: arrs.insert("KR", at:1))  // addVars 함수를 호출하면서 함수 안의 배열의 인덱스를 확장하는 명령이 실행되어 오류가 발생하지 않음.
print(arrs)                             // [ "", "KR" , "", "" ]
```