Closure

- 코드 블럭. 이름 없는 일회용 함수.
- 함수가 다른 곳의 매개변수로 전달되야 할 때 유용.
  다시 쓰지 않을 일회용 함수를 굳이 따로 정의하면 비효율적이고 불필요함. '클로저' 라는 방식으로 일회용 함수를 정의해서 사용하는 게 매우 편함.
- 클로저 표현의 최적화
  1. 인자 타입(parameter type)과 반환 타입(return type)의 추론
  2. 단일 표현 클로저에서의 암시적 반환
  3. 축약된 인자 이름
  4. 후위 클로저 문법

클로저 표현 (Closure Expressions)

```
func backward(_ s1: String, _ s2: String) -> Bool {
return s1 > s2
}
var reversedNames = names.sorted(by: backward)
// reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

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

- 인라인 클로저 : 함수가 따로 정의된 형태가 아닌 파라미터로 들어가 있는 형태의 클로저.
- 클로저의 몸통(body)은 in 키워드 다음에 시작.

축약

1. 문맥에서 타입 추론 (Inferring Type From Context)
   : **인자 타입, 리턴 타입** 생략.

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

2. 단일 표현 클로저에서의 암시적 반환 (Implicit Returns from Single-Express Closures)
   : 단일 표현 클로저에선 반환 값이 있는 경우, **return 키워드** 생략.

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

3. 인자 이름 축약 (Shorthand Arguments Names)
   : 전달인자의 이름이 굳이 필요없고, 컴파일러가 타입을 유추할 수 있는 경우 **축약된 전달인자 이름($0, $1, $2...)을 사용.**

축약 인자 이름을 사용하면 인자 값과 그 인자로 처리할 때 사용하는 인자가 같다는 것을 알기 때문에 인자를 입력 받는 부분과 **in 키워드 부분을 생략**할 수 있음.

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

+) 연산자 메소드
: Swift의 String 타입 연산자에는 String끼리 비교할 수 있는 비교 연산자(>) 를 구현해 둠.

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

후행 클로저(Trailing Closures)

- 함수 마지막 인자로 전달한 클로저가 길거나 가독성이 떨어질 때 사용.
- **마지막 전달인자로 전달되는 클로저에만 해당.**
- 마지막 매개변수 이름을 생략 → **함수 호출 괄호() 밖에서 클로저를 구현.**

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

```
result = calculate(a: 10, b: 10) {(left: Int, right: Int) -> Int) in
  return left + right
}
```

- 함수의 인자가 후위 클로저 하나 뿐이라면, 괄호()를 생략할 수 있음.
  But, 인자 값이 여러 개인 경우엔 소괄호 생략 x.

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

값 캡쳐 (Capturing Values)

- 일반적인 함수는 매개변수로 받은 값들만 사용 가능 하지만, 클로저는 자신이 선언된 곳 영역의 변수에도 접근 가능함.
- 클로저의 값 캡쳐 방식
  : 클로저는 값을 캡쳐할 때 Value/Reference Type과 관계 없이, Reference Capture를 함.
  그래서 클로저 내부에서 사용하는 변수들이 변경되면, 참조한 클로저 외부에 있는 변수 값도 변경됨.
- Swift에서 값을 캡쳐 하는 가장 단순한 형태는 중첩 함수(nested function).
  중첩 함수는 자신을 둘러싼 함수의 매개변수 & 정의된 상수 & 변수도 캡쳐할 수 있다.

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

```
}

=> incrementer 함수에서 runningTotal 도 사용하고 amount 도 사용.
중첩 함수 상위의 함수에서 인자로 받은 amount 는 값을 복사해서 가져옴.
하지만 중첩함수 선언 영역에 있는 runningTotal 는 참조하기 때문에 해당 값을 변경 할 수 있음.
```

- +) 만약 클로저를 어떤 클래스 인스턴스의 프로퍼티로 할당하고 그 클로저가 그 인스턴스를 캡쳐링하면 강한 순환참조에 빠지게 된다. 즉, 인스턴스의 사용이 끝나도 메모리를 해제하지 못하는 것. 그래서 Swift는 이 문제를 다루기 위해 캡쳐 리스트(capture list)를 사용한다. 더 많은 정보는 클로저의 강한 참조 순환을 참조.
- **ARC(Automatic Reference Counting) :** Swift에서 메모리는 ARC로 관리한다. 인스턴스를 참조하는 곳이 몇군데인지 체크하여 자동으로 레퍼런스 카운팅을 해주는 객체. 인스턴스가 변수나 함수의 인자값으로 할당되면 카운트가 1증가, 종료되면 1 감소. 레퍼런스 카운트가 0이되면 자동으로 메모리를 해제해준다.
- **강한 순환 참조 :** 두 객체가 서로 참조하는 경우 레퍼런스 카운팅이 절대로 0이 될 수 없는 상황이 발생할 수 도 있다. 이런 상황에서 메모리 누수가 발생한다. 강한 순환 참조가 이루어지지 않도록 주의해야 한다.
  ex) Parent 객체는 child라는 프로퍼티를 갖고 있고, Child 객체는 parent라는 프로퍼티를 갖고 있는 경우 서로 참조하는 상황이 되기 때문에 두 객체 모두 레퍼런스 카운트가 1로 유지된다. ⇒ 메모리 누수 발생!
- https://velog.io/@kimdo2297/클로져-캡쳐에-대해서-about-closure-capture

클로저는 참조 타입

- 함수와 클로저는 참조 타입이라, 함수와 클로저를 상수나 변수에 할당할 때 실제로는 상수와 변수에 해당 함수나 클로저의 **참조(reference)**가 할당 됨. 그래서 만약 한 클로저를 두 상수나 변수에 할당하면 그 두 상수나 변수는 같은 클로저를 참조하고 있음.

탈출 클로저(Escaping Closure)

- 해당 함수의 인자로 클로저가 전달되지만, 함수가 반환된 후 실행 되는 것을 의미.
- 함수의 인자 값으로 전달된 클로저를 저장해 두고, 나중에 다른 곳에서도 실행할 수 있게 허용해주는 속성.
- 탈출 클로저를 왜 사용할까?
  : 함수의 인자 값으로 전달된 클로저는 기본적으로 비탈출의 성격을 가진다. **비탈출 클로저**는 함수 내에서 **직접 실행을 위해서만 사용이 가능**하다는 것을 의미한다. 그래서 **함수 내부에서도 변수/상수에 대입할 수 없다.** 변수/상수에 대입을 허용하면 내부 함수를 통한 캡처 기능을 이용한 클로저가 함수 바깥으로 탈출(함수 내부 범위를 벗어난 실행) 할 수 있기 때문이다.
  +) 클로저를 변수/상수에 대입하거나 중첩 함수 내부에서 사용해야 하는 경우도 있는데,
  이럴 때에 @escaping을 쓰면 된다. 이 속성을 가지게 되면 탈출이 가능하게 되고 앞서 봤던 제약이 사라짐.

  1. 함수 내부의 클로저 인자를 외부에서 사용할 수 있음.
     : 기본적으로 함수의 인자로 들어오는 클로저는 **함수 안에서만** 사용할 수 있음. 그러나 클로저 타입 앞에 `@escaping` 를 붙이면 탈출 클로저로 바꿀 수 있고, 탈출 클로저는 해당 함수가 반환된 후 실행되므로, 일반적인 클로저와 달리 함수 외부에 저장하거나 다른 스레드에서 실행시킬 수 있음.
     탈출 클로저를 이용해 함수를 만들면 함수의 실행이 끝난 후에도 메모리에 해당 부분이 남아있어 외부에서 활용하기 편함.
     만약 탈출 클로저를 사용할 수 없다면 함수 내 클로저 인자를 함수 외부로 가져와서 가공할 수 없을 것임. 따라서 해당 값을 가공하려고 할 때마다 새로운 함수를 생성하여 클로저 인자를 만들고 새로 가공해야 하니 코드가 길고 복잡해질 것.
     +) 탈출 클로저를 이용하면 깔끔한 아키텍쳐 설계에도 도움.

2. 함수 간 실행 순서를 고려할 수 있다.
   : 탈출 클로저는 해당함수가 반환된 후 실행되는 클로저이므로, 함수 간 실행순서를 고려할 때 탈출클로저를 사용하여 코드를 짤 수 있음.
   비동기로 여러 개의 작업이 처리될 때 함수의 동작 순서를 지정할 때 사용할 수 있음.

- 탈출 클로저 사용하기

1. 함수 밖에서 정의한 변수에 저장하기
   : 함수 내 클로저 인자를 저장하기 위해서는 함수 밖에서 선언한 변수에 저장하면 됨. 아래 예시는 `completionHandlers`라는 변수에 `someFunctionWithEscapingClosure`라는 함수의 클로저 인자인 `completionHandler`를 append하여 저장.
   +) 파라미터 앞에 @escaping 을 써 주면 되고, 안 써 주면 함수 밖에 변수에 저장할 수 없음.

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

2. 비동기 실행
   : GCD를 활용해 dispatch queue에서 비동기식으로 클로저를 실행할 때 큐는 클로저를 메모리에 보관하고 나중에 사용할 수 있음. 이 경우 클로저가 언제 실행되는지 알 수 없음.
   아래 예시는 json이 담긴 url을 처리하여 `DispatchQueue.global(qos: .background).async`에서 처리하도록 하는 예시.

```
func getJson(path: String, params: [String: Any], completed: @escaping (Result<String, Error>) -> Void) {
    DispatchQueue.global(qos: .background).async {
        do {
            let url = URL(string: path)
            let json = try String(contentsOf: url!, encoding: .utf8)
            DispatchQueue.main.async {
                completed(Result.success(json))
            }

        } catch {
            DispatchQueue.main.async {
                completed(Result.failure(error))
            }
        }
    }
}
```

- @escaping 키워드로 탈출 클로저임을 명시한 경우, 클로저 내부의 프로퍼티, 메서드, 서브스크립트 등에 접근하려면 self 키워드를 필수로 사용해야 한다.
  @escaping이 없는 nonescaping 클로저는 self 를 안적어 줘도 됨.

```
// 함수 외부에 클로저를 저장하는 예시
// 클로저를 저장하는 배열
var completionHandlers: [() -> Void] = []

func withEscaping(completion: @escaping () -> Void) {
    // 함수 밖에 있는 completionHandlers 배열에 해당 클로저를 저장
    completionHandlers.append(completion)
}

func withoutEscaping(completion: () -> Void) {
    completion()
}

class MyClass {
    var x = 10
```

```
    func callFunc() {
        // escaping closure 는 명시적으로 self 를 기입해야 한다.
        withEscaping { self.x = 100 }
        // nonescaping closure 는 암시적으로 self 를 참조 할 수 있다
        withoutEscaping { x = 200 }
    }
}
let mc = MyClass()
mc.callFunc()
print(mc.x)
completionHandlers.first?()
print(mc.x)

// 결과
// 200
// 100
```

+) Completion Handler : 클로저를 많이 Completion Handler에 많이 응용함. 동작을 하기 위한 코드의 덩어리 (하나의 함수, 코드의 묶음).

+) 비동기 : 동시에, 순서대로 일어나지 않음.

+) https://velog.io/@dmsgk/Swift-탈출클로저Escaping-closure-이해하기
https://blog.naver.com/horajjan/222243694985
https://hcn1519.github.io/articles/2017-09/swift_escaping_closure

자동 클로저(Auto Closure)

- @autoclosure는 인자 값으로 전달된 일반 구문이나 함수 등을 클로저로 래핑 하는 역할을 한다.
  일반 구문을 인자 값으로 넣어도 컴파일러가 알아서 클로저로 만들어서 사용한다.
- 이때 사용되는 클로저는 인자가 없고 리턴값만 존재해야 한다.
- 자동 클로저는 적혀진 라인 순서대로 바로 실행되지 않고, 클로저를 호출할 때 지연 호출됨.
  부작용이 있거나 연산이 오래 걸리는 코드에서 유용. 왜냐하면 코드의 사용 시점을 제어할 수 있기 때문.

```
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"
```

- 클로저를 인자로 받는 함수에 @autoclosure를 사용하지 않은 경우

```
func normalPrint(_ closure: () -> Void) {
    closure()
}

normalPrint({ print("I'm Normal Expression") })
```

이 함수를 호출할 때 클로저 부분은 다음과 같이 대괄호 {...}로 묶어서 인자를 넣어야 합니다.

- @autoclosure를 사용한 경우

```
func autoClosurePrint(_ closure: @autoclosure () -> Void) {
    closure()
}

autoClosurePrint(print("I'm AutoClosure Expression"))
```

@autoclosure를 이용하면 **함수에 클로저를 인자로 사용할 때 일반 표현**을 사용할 수 있습니다.