Week 1-2

- Swifty Swift Code(스위프트 다운 스위프트 코드)
  - 좋은 코드의 8가지 특징
  - API가 깔끔하면
  - 스위프트에서 더 스위프트다운 구문을 구현하는 몇 가지 방법
  - 익스텐션 Extension
  - 연산 프로퍼티 Computed Property
  - 사용자 정의 연산자 Custom Operator
  - 다양한 클로저 표현
  - 옵셔널 Optional
  - 빠른 종료 guard
- Functional Programming
  - 왜 함수형 프로그래밍을 배워야 하는가?
    - 동시성 문제
    - 데이터 관리에 따른 부담
    - 함수형 프로그래밍은 모듈화되어 있습니다.
    - 더 빠르게 작업해야 합니다.
    - 함수형 프로그래밍은 단순함으로의 복귀입니다.
- 명령형 프로그래밍 vs 선언형 프로그래밍
  - 명령형 프로그래밍
  - 선언형 프로그래밍
- 함수형 프로그래밍의 특징
  - 변경 불가능한(Immutable) 값을 활용합니다
  - 일급 시민으로서의 함수
  - 클로저
  - 함수형 프로그래밍을 연습하는 방법
  - 함수란 무엇인가
- 클로저
- 고차함수
  - map
  - filter
  - reduce

map, filter, reduce 연습해보기
연습 1
연습 2
생각해보기
스위프트 프로그래머가 FP를 연습하는 방법
스위프트 프로그래머가 FP를 대하는 자세
이력서
1. 삶의 이력 정리하기
2. 나를 나타낼 수 있는 키워드 뽑아보기
3. 수치화 합시다

Swifty Swift Code(스위프트 다운 스위프트 코드)

좋은 코드의 8가지 특징

- 잘 작동합니다.
- 읽기 쉽습니다.
- 테스트 가능합니다.
- 관리가 쉽습니다.
- 외관이 보기 좋습니다.
- 변경이 쉽습니다.
- 간결합니다.
- 효율적입니다.

API가 깔끔하면

- 읽기 쉽습니다.
- 외관이 보기 좋습니다.
- 간결합니다.

스위프트에서 더 스위프트다운 구문을 구현하는 몇 가지 방법

- 익스텐션
- 연산 프로퍼티
- 사용자 정의 연산자
- 다양한 클로저 표현
- 옵셔널
- 빠른 종료

익스텐션 Extension

```
StringUtils.lastCharacter("yagom") // m static func lastCharacter(_
of: String) -> Character { return of.last ?? Character("") }
```

```
"yagom".lastCharacter() // m extension String { func lastCharacter
() -> Character { return self.last ?? Character("") } }
```

연산 프로퍼티 Computed Property

필요한 경우에 매개변수 없는 메서드를 연산 프로퍼티로 구현할 수 있습니다.

```
"yagom".lastCharacter() // m extension String { func lastCharacter
() -> Character { return self.last ?? Character("") } }
```

```
"yagom".lastCharacter // m extension String { var lastCharacter: Ch
aracter { return self.last ?? Character("") } }
```

사용자 정의 연산자 Custom Operator

```
postfix operator ~~ extension String { static postfix func ~~ (stri
ng: Self) -> Character { return string.last ?? Character("") } } "y
agom"~~ // m
```

다양한 클로저 표현


```
func calculate(a: Int, b: Int, method: (Int, Int) -> Int) -> Int {
return method(a, b) } var result: Int
```

```
// 후행 클로저 result = calculate(a: 10, b: 10) { (lhs: Int, rhs: Int)
-> Int in return lhs + rhs }
```

```
// 후행 클로저 + 반환타입 생략 result = calculate(a: 10, b: 10) { (lhs: I
nt, rhs: Int) in return lhs + rhs }
```

```
// 후행 클로저 + 반환타입 생략 + 단축인자 이름 result = calculate(a: 10, b: 10) { return $0 + $1 }
```

```
// 후행 클로저 + 반환타입 생략 + 단축인자 이름 + 암시적 반환 표현 result = calculate(a: 10, b: 10) { $0 + $1 }
```

```
// 후행 클로저 + 반환타입 생략 + 단축인자 이름 + 암시적 반환 표현 result = calculate(a: 10, b: 10) { $0 + $1 }
```

```
// 스위프트의 연산자는 메서드(or 함수)입니다 // 연산자 메서드 이름은 연산자 기호입니다
result = calculate(a: 10, b: 10, method: +)
```

옵셔널 Optional

옵셔널 사용에 있어 느낌표(!) 사용을 절대지양합니다. 확신할 수 있는 코드, 확신할 수 있는 상황은 매우 드뭅니다.
연습에서는 내 습관에 느낌표를 없애도록합니다. (IBOutlet 등 특수한 경우만 예외합니다)

빠른 종료 guard

`guard` 구문은 `if` 구문에 비해 추가로 가지는 의미가 있습니다.
`guard` 구문의 조건을 충족하지 못하면 해당 흐름은 중단됩니다. 꼭 필요한 확인을 위해서는 `if` 보다는 `guard`를 사용합니다.

```
func split(string: String, with token: Character) -> [String] { if token.isEmpty { return [] } // ... }
```

```
func split(string: String, with token: Character) -> [String] { gua
rd token.isEmpty == false else { return [] } // ... }
```

Functional Programming

왜 함수형 프로그래밍을 배워야 하는가?

동시성 문제

이제는 스마트폰마저 다중 프로세서를 가지고 있습니다. 여러 개의 프로세서를 가지고 있다는 의미는 동시성 문제에서 자유롭지 못하다는 뜻입니다. 동시성 문제를 해결하면서 안정적인 소프트웨어를 개발하는 것에 대한 중요성이 높아졌습니다. 데이터의 상태를 변경하는 객체 지향 프로그래밍 방식으로는 동시성 문제를 해결하는 데는 문제도 많고 자원의 소모비용도 큽니다.

데이터 관리에 따른 부담

대용량 데이터를 다루는 작업이 점점 더 많아지고 있습니다. 대용량 데이터를 처리할 때 데이터를 객체로 변환하는데 따른 부담이 큽니다. 대용량 데이터를 처리할 수 있는 효율적인 데이터 구조와 데이터 연산이 필요합니다.

함수형 프로그래밍은 모듈화되어 있습니다.

타입 단위의 모듈화는 가장 작은 단위의 모듈화가 아닙니다. 함수형 프로그래밍의 함수를 모듈화할 경우 수 많은 곳에서 재사용할 수 있습니다. 함수형 프로그래밍은 더 유용하고, 재사용이 편리하고, 구성이 용이하고, 테스트하기 더 간편한 추상화를 제공합니다.

더 빠르게 작업해야 합니다.

소프트웨어 개발 흐름은 점점 더 빠른 결과물을 확인할 수 있기를 기대하는 방향으로 변화되고 있습니다. 타입에 대한 모델링에 많은 시간을 투자하기보다 사용자 요구 사항에 대해서 최소한으로 충분한 수준을 유지하면서 동시에 변화에 대해서도 유연하게 대응하는데 도움을 줍니다.

함수형 프로그래밍은 단순함으로의 복귀입니다.

요구 사항 자체는 본질적으로는 복잡하지 않습니다. 다만 그 요구 사항을 구현하기 위해 선택된 방식(OOP 등)에서의 설계가 복잡해질 수 있는 것이죠. 함수형 프로그래밍은 때에 따라 이 복잡성을 단순화할 수 있습니다.

복잡한 문제를 작은 단위로 분리해 해결하는 능력이 프로그래머에게 특히 중요하죠. 함수형 프로그래밍을 학습하면 문제에 접근하는 방법, 문제를 작은 단위로 쪼개는 방법, 설계하는 과정, 프로그래밍하는 순서에서 새로운 시각을 배울 수 있습니다. 즉, 함수형 프로그래밍 방식을 학습하면 현재 프로그래밍 스타일을 개선해서 더 깔끔한 코드를 구현하는데 도움을 받을 수 있습니다.

명령형 프로그래밍 vs 선언형 프로그래밍

명령형 프로그래밍

프로그래밍의 상태와 상태를 변경시키는 구문의 관점으로 접근하는 프로그래밍 방식. 명령형 프로그래밍은 컴퓨터가 실행할 멍령을 실행 순서대로 구현해야 합니다.

대부분의 객체 지향 프로그래밍 언어가 명령형 프로그래밍 언어입니다. 알고리즘 처리 작업에 적합한 언어입니다.

선언형 프로그래밍

선언으로만 프로그램을 동작시키는 것을 의미합니다. 프로그램을 실행하기 위해 구체적인 작동 순서를 나열하지 않아도 됩니다.

완전하지 않지만 함수형 프로그래밍을 활용해 일정 수준의 선언형 프로그래밍을 할 수 있습니다. 함수형 프로그래밍은 선언형 프로그래밍의 한 종류로 볼 수 있습니다. 요즘 애플에서 이 선언형을 많이 강조하고 있는데요, SwiftUI가 선언형 프로그래밍을 사용하기 때문입니다. 뷰에게 명령하는 것이 아니라 뷰를 선언하기 때문이죠.

명령형 프로그래밍

```
나: "소피아, 장난감 정리하자. 땅에 어떤 장난감이 있지?" 소피아: "어. 공이 있네" 나: "그래. 공을 박스에 넣자. 땅에 또 뭐가 있지?" 소피아: "어. 인형이 있지" 나: "그래. 인형을 박스에 넣자. 땅에 또 뭐가 있지?" 소피아: "어. 책이 있네" 나: "그래. 책을 박스에 넣자. 땅에 또 뭐가 있지?" 소피아: "아니. 아무것도 없어" 나: "잘했네. 이제 다 했다"
```

선언형 프로그래밍

```
"소피아. 땅에 있는 모든 장난감을 박스에 넣으렴"
```

명령형 프로그래밍 스타일

```
func point(of target: Customer) -> Int { for customer in customers
{ if customer == target { return customer.point } } return 0 }
```

선언형 프로그래밍 스타일

```
func point(of target: Customer) -> Int { if let customer = customer
s.filter({ $0 == target }) { return customer.point } return 0 }
```

함수형 프로그래밍의 특징

- 작업을 어떻게 수행할 것인지, **How**에 집중합니다
- 구체적인 작업 방식은 라이브러리가 결정, **무엇(What)**을 수행할 것인지에 집중합니다
- side-effect가 발생하지 않습니다

변경 불가능한(Immutable) 값을 활용합니다

값이 변경되는 것을 허용하면 멀티 스레드 프로그래밍이 힘듭니다.
여러 스레드에서 경쟁적으로 값을 변경하려고하면 컨텍스트 스위칭도 많이 일어나고, 스레드 충돌 방어 기법도 많이 고려해야하기 때문입니다.
값을 변경할 수 없는 경우 프로그램의 정확성을 높여 버그의 발생 가능성을 줄일수 있습니다.

일급 시민으로서의 함수

함수형 프로그래밍에서는 함수가 일급 시민입니다. 함수를 일급 시민으로 활용이 가능할 경우 함수를 함수의 인자로 받거나 함수의 반환 값으로 활용하는 것이 가능합니다.

클로저

클로저는 전달 가능하고 실행가능한 코드의 묶음입니다. 함수는 이름이 있는 클로저입니다.
흔히 클로저를 익명함수라고 표현하지만, 스위프트에선 틀린 표현입니다. 함수가 이름이 있는 클로저이지, 클로저가 이름이 없는 함수는 아닙니다.

함수형 프로그래밍을 연습하는 방법

프로그래밍의 기본 틀은 객체 지향 프로그래밍, 함수 내부 구현은 함수형 프로그래밍을 지향, 인스턴스의 상태 관리는 불변을 지향합니다.

함수란 무엇인가

클로저

클로저는 다른 함수 또는 다른 코드로 전달할 수 있는 코드의 묶음입니다. 클로저를 활용하면 손쉽게 공통 코드 구조를 함수로 만들어낼 수 있습니다. 스위프트 표준 라이브러리와 코코아 터치 프레임워크에서도 클로저를 아주 많이 사용하고 있습니다.

버튼의 액션을 selector를 활용하여 추가하는 기존의 방식

```
let button = UIButton(type: .custom) button.addTarget(self, action: #selector(touchUpButton:), for: .touchUpInside) @objc func touchUpButton(_ sender: UIButton) { // action code... }
```

클로저를 활용해서 실행할 코드를 전달하는 간결한 방식

```
let actionHandler: (UIAction) -> Void = { action in // action code... } let action = UIAction(handler: actionHandler) let button = UIButton(primaryAction: action)
```

이 코드는 앞에서 살펴본 코드와 같은 일을 수행하지만 재사용성도 뛰어나며 코드의 분산도 적어져서 훨씬 더 간결하고 읽기 쉽습니다.

고차함수

스위프트 언어에서 함수는 일급 시민이기 때문에 고차함수를 적극 활용할 수 있습니다. 특히 스위프트의 각종 컬렉션에서 고차함수를 잘 활용하면 임시 변수 등을 생성하지 않고 상수를 적극 활용할 수 있습니다. 컬렉션에서 고차함수를 사용하면 새로운 컬렉션 혹은 값을 생성해줍니다.

```
people.map{ $0.name }.filter { $0.hasPrefix("김") }
```

이 연쇄 호출은 기존의 배열(`people`)로부터 map 호출 후 새로운 배열을 생성하고, 그 배열로부터 다시 filter 호출 후 새로운 배열을 생성합니다. `people` 에 값이 수백개 수만개라면 새로운 컬렉션으로 복사하는게 걱정된다고요? 걱정마세요. 그 정도는 스위프트 컴파일러가 스마트하게 해낼테니까요 :)

- *고차함수(Higher-order function)**는 '다른 함수를 전달인자로 받거나 함수실행의 결과를 함수로 반환하는 함수'를 뜻합니다.
  스위프트의 함수(클로저)는 일급시민이기 때문에 함수의 전달인자로 전달할 수 있으며, 함수의 결과값으로 반환할 수 있습니다.
  스위프트 표준라이브러리에서 제공하는 유용한 고차함수 몇가지에 대해 알아봅니다.
- `map`
- `filter`
- `reduce`

`map` , `filter` , `reduce` 함수는 스위프트 표준 라이브러리의 컨테이너 타입(`Array` , `Set` , `Dictionary` 등)에 구현되어 있습니다.

map

`map` 함수는 컨테이너 내부의 **기존 데이터를 변형(transform)하여 새로운 컨테이너를 생성합니다.**

> 변형하고자 하는 numbers와 변형 결과를 받을 doubledNumbers, strings

```
let numbers: [Int] = [0, 1, 2, 3, 4] var doubledNumbers: [Int] var strings: [String]
```

> 기존의 for 구문 사용

```
doubledNumbers = [Int]() strings = [String]() for number in numbers { doubledNumbers.append(number * 2) strings.append("\\(number)") } print(doubledNumbers) // [0, 2, 4, 6, 8] print(strings) // ["0", "1", "2", "3", "4"]
```

> map 메서드 사용

```
// numbers의 각 요소를 2배하여 새로운 배열 반환 doubledNumbers = numbers.map({ (number: Int) -> Int in return number * 2 }) // numbers의 각 요소를 문자열로 변환하여 새로운 배열 반환 strings = numbers.map({ (number: Int) -> String in return "\\(number)" }) print(doubledNumbers) // [0, 2, 4, 6, 8] print(strings) // ["0", "1", "2", "3", "4"] // 매개변수, 반환 타입, 반환 키워드(return) 생략, 후행 클로저 doubledNumbers = numbers.map { $0 * 2 } print(doubledNumbers) // [0, 2, 4, 6, 8]
```

filter

`filter` 함수는 컨테이너 내부의 **값을 걸러서 새로운 컨테이너로 추출**합니다.

> 기존의 for 구문 사용

```
// 변수 사용에 주목하세요 var filtered: [Int] = [Int]() for number in numbers { if number % 2 == 0 { filtered.append(number) } } print(filtered) // [0, 2, 4]
```

> filter 메서드 사용

```
// numbers의 요소 중 짝수를 걸러내어 새로운 배열로 반환 let evenNumbers: [Int] = numbers.filter { (number: Int) -> Bool in return number % 2 == 0 } print(evenNumbers) // [0, 2, 4] // 매개변수, 반환 타입, 반환 키워드(return) 생략, 후행 클로저 let oddNumbers: [Int] = numbers.filter { $0 % 2 != 0 } print(oddNumbers) // [1, 3]
```

reduce

`reduce` 함수는 컨테이너 내부의 **콘텐츠를 하나로 통합**합니다.

> 통합하고자 하는 someNumbers

```
let someNumbers: [Int] = [2, 8, 15] let someStrings: [String] = ["H", "E", "L", "L", "O"]
```

> 기존의 for 구문 사용

```
// 변수 사용에 주목하세요 var result: Int = 0 // someNumbers의 모든 요소를 더합니다 for number in someNumbers { result += number } print(result) // 25
```

> reduce 메서드 사용

```
// 초깃값이 0이고 someNumbers 내부의 모든 값을 더합니다. let sum: Int = someNumbers.reduce(0, { (first: Int, second: Int) -> Int in //print("\\(first) + \\(second)") //어떻게 동작하는지 확인해보세요 return first + second }) print(sum) // 25 // 초깃값이 0이고 someNumbers 내부의 모든 값을 뺍니다. var subtract: Int = someNumbers.reduce(0, { (first: Int, second: Int) -> Int in //print("\\(first) - \\(second)") //어떻게 동작하는지 확인해보세요 return first - second }) print(subtract) // -25 // 초깃값이 3이고 someNumbers 내부의 모든 값을 더합니다. let sumFromThree = someNumbers.reduce(3) { $0 + $1 } print(sumFromThree) // 28 someStrings.reduce("", { $0.appending($1) }) // "HELLO"
```

`reduce` 메서드에 전달하는 클로저의 매개변수 이름을 `first`, `second` 보다는 `result`, `currentItem`과 같은 이름으로 정정하는 것이 좋겠습니다. 첫 번째 매개변수는 초깃값으로부터 출발하여 마지막 요소까지 순회하는 내내의 결괏값입니다. `currentItem`은 현재 순회하고 있는 요소의 값을 뜻합니다. 결국 `return result + currentItem`이라고 표현한다면 이제까지 더해진 결괏값에 이번 요소의 값을 더한다는 뜻이 되겠습니다.

---

map, filter, reduce 연습해보기

연습 1

Array에 담긴 모든 숫자 중 3보다 큰 숫자를 2배 한 후 모든 값의 합을 구합니다. 지금까지 학습한 map, reduce, filter를 활용해 구현합니다.

연습 2

요소가 수백개인 Array에 여러 단어가 포함되어 있습니다.

- 단어의 길이가 12자를 초과하는 단어를 추출합니다.
- 12자가 넘는 단어 중 길이가 긴 순서로 100개의 단어를 추출합니다.
- 추출한 100개의 단어를 출력합니다. 모든 단어는 소문자로 출력해야합니다.

생각해보기

- 반복문과 조건문을 대체할 수 있는 고차함수에 대해 고민해보세요
- 반복문과 조건문을 대체할 수 있는 순환함수(Recursive Function) 사용에 대해 고민해보세요

스위프트 프로그래머가 FP를 연습하는 방법

OOP 사고를 가지는 프로그래머가 OOP와 FP를 모두 지원하는 언어로 FP를 연습하는 것이 쉽지 않습니다.
FP를 제대로 연습하고 싶다면 극단적으로 FP를 지향하는 언어로 연습하는 것도 좋은 방법입니다.
예를 들어 Lisp 계열인 scheme, JVM 언어인 clojure와 같은 언어가 좋은 연습도구가 될수도 있죠.

스위프트 프로그래머가 FP를 대하는 자세

순수 OOP와 순수 FP 기반으로 프로그래밍하는 것은 각각의 한계점이 있습니다.
FP의 장점을 스위프트에 도입하는 것이 OOP를 더 잘할 수 있는 길중에 하나일지도 모릅니다.
프로그래밍의 기본 틀은 OOP 기반,
메소드 내부 구현은 FP를 지향,
인스턴스의 상태 관리는 immutable instance를 지향해봅니다.

객체지향 언어에 흥미가 떨어지거나
새로운 프로그래밍 접근 방식을 도전하고 싶거나
현재의 역량을 한 단계 더 성장시키고 싶다면...
함수형 프로그래밍에 도전해보세요.

이력서

1. 삶의 이력 정리하기

- 사진은 굳이 넣지 않도록 합니다. 요구하면 따로 첨부합니다
- 학력을 잘 넣지 않는 추세이나, 달리 넣을 게 없다면 최종학력은 넣습니다
- 다른 업종에 종사한 이력도 있다면 넣는 게 좋습니다

- 이력마다 얼마간 무엇을 배웠는지 기술합니다

> 💬
> ——
> 2016.03. ~ 2019.02. 한국고등학교
> - 수학에 두각을 드러내고, 몰두하여 시험마다 n점 이상(혹은 n등급 이상)을 취득 (혹은 무슨 대회 입상)
> - 고전물리를 공부하면서 벡터들의 상호작용을 고민하며 논리력과 사고력을 익힘.
> - 또 이를 다른 학문에서도 적용하여 문제가 어떤 방향성을 갖는지 파악하고 해결할 수 있게 됨(음악, 수학 등)
> - 평소 소극적인 성격을 극복하고, 원만한 교우관계를 갖기 위해 노력함. 이를 위해 평소 못하던 구기종목을 배우고 참여함(근데 여전히 못함)
> - 남의 이야기를 경청하는 스킬을 익힘. 또 이를 발전시켜 다양한 문제에 공감하고, 해결할 수 있도록 도울 수 있게 됨. 이를 통해 현재까지도 친구들의 고민상담을 해주는 편.
> - 시창작 동아리에 참여하였으며 주마다 발표 및 낭송하는 활동을 함. 특히 동아리 내에서 리액션을 잘해, 좋은 분위기가 되도록 크게 이바지 함
> ——
> 2020.05. ~ 2020.10. 야곰아카데미
> - 개발 관련 무언가에 대해서 어떤 성취를 얼만큼 이룸
> - 특히 무슨 방법으로 공부하여 이러이러한 문제를 해결하기 위해 노력하고 기록을 남김
> - 커뮤니케이션에 대해서 고민하여 어떤 성취를 이룸
> ——

- 예시라서 항목마다 일관성이 없는데 일관성이 있으면 더 좋습니다

2. 나를 나타낼 수 있는 키워드 뽑아보기

- 이력서를 관통할 수 있는, 스스로를 나타낼 수 있는 키워드를 하나 뽑아보도록 합니다
  - 예) 접근성, UI/UX, Unit Test, TDD, 코드 리뷰 등
  - 내가 관심 많은 부분을 하나의 키워드로 잡고 이력서/포트폴리오 전역에 녹여서 관련 질문이 많이 들어오도록 합니다
- 분명하고 명료한 키워드일수록 좋습니다

- 그 외에도 자신을 어필할 수 있는 강점을 3가지 정도 뽑아봅니다
  - 미친 것처럼 보여도 강렬한 인상을 주면 좋습니다. **욕하란 건 아닙니다(진지함)**

> 💬
> - 밥값보다는 더 하는 개발자입니다
> - 시키지 않아도 알아서 잘하는 개발자입니다
> - 축하드립니다. 다시는 없을, 제 이력서를 받을 기회입니다.
> - apple/swift 컨트리뷰터입니다
> - ReactiveX/RxSwift 컨트리뷰터입니다