High-order Function

참고 링크

Swift Standard Library - Collection.swift
https://github.com/apple/swift/blob/master/stdlib/public/core/Collection.swift

Simple Higher Order Functions in Swift
https://goo.gl/1w8qxR

High-order Function

고차함수란?

- 하나 이상의 함수를 인자로 취하는 함수
- 함수를 결과로 반환하는 함수

※ High-order Function 이 되기 위해서는 함수가 First-class Citizen 이어야 한다.

First-class citizen

1급 객체 (First-class citizen)

- 변수나 데이터에 할당할 수 있어야 한다.
- 객체의 인자로 넘길 수 있어야 한다.
- 객체의 리턴값으로 리턴할 수 있어야 한다.

```swift
func firstCitizen() {
  print("function call")
}

func function(_ parameter: @escaping ()->()) -> (()->()) {
  return parameter
}

let returnValue = function(firstCitizen)
returnValue()
```

Example

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

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

let incrementer = makeIncrementer(forIncrement: 7)
print(incrementer())  // 7
print(incrementer())  // 14
```

High-order Functions in Swift

- **map**

  - 컬렉션의 각 요소(Element)에 동일 연산을 적용하여, 변형된 새 컬렉션 반환

- **filter**

  - 컬렉션의 각 요소를 평가하여 조건을 만족하는 요소만을 새로운 컬렉션으로 반환

- **reduce**

  - 컬렉션의 각 요소들을 결합하여 단 하나의 타입으로 반환. ex) Int, String

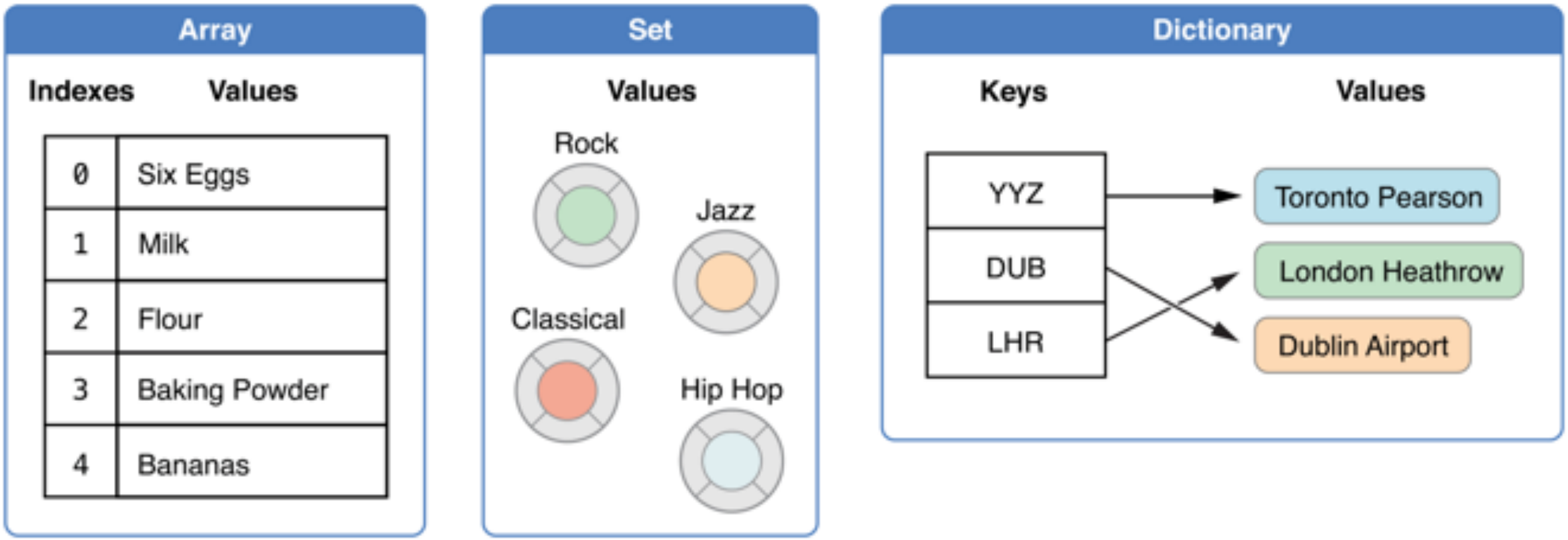
  - 입력되는 요소와 반환되는 요소가 서로 다른 타입일 수 있음.

- **flatMap**

  - 중첩된 컬렉션을 하나의 컬렉션으로 병합.

  - 요소 중 옵셔널이 있을 경우 제거

Collection Type



Practice

```
struct Pet {
  enum PetType {
    case dog, cat, snake, pig, bird
  }
  var type: PetType
  var age: Int
}
```

```
let myPet = [
  Pet(type: .dog, age: 13),
  Pet(type: .dog, age: 2),
  Pet(type: .dog, age: 7),
  Pet(type: .cat, age: 9),
  Pet(type: .snake, age: 4),
  Pet(type: .pig, age: 5),
]
```

Practice

1. 강아지들의 나이를 합산한 결과를 반환하는 sum 함수 구현

2. 모든 펫의 나이를 1씩 더한 배열을 반환하는 newAge 함수 구현

Case 1

```
func sum() -> Int {
  var ageSum = 0
  for pet in myPet {
    guard pet.type == .dog else { continue }
    ageSum += pet.age
  }
  return ageSum
}
```

```
func sum() -> Int {
  return myPet
    .filter { $0.type == .dog }
    .reduce(0) { $0 + $1.age }
}
```

Case 2

```
func newAge() -> [Pet] {
  var newAge = [Pet]()
  for pet in myPet {
    newAge.append(Pet(type: pet.type, age: pet.age + 1))
  }
  return newAge
}
```

```
func newAge() -> [Pet] {
  return myPet.map {
    Pet(type: $0.type, age: $0.age + 1)
  }
}
```

Practice

```
let immutableArray = Array(1…40)
```

1. 배열의 각 요소 * index 값을 반환하는 함수

2. 배열 요소 중 홀수는 제외하고 짝수만 반환하는 함수

3. 배열의 모든 값을 더하여 반환하는 구현

4. immutableArray 에 대해서 1~3 번 함수를 차례대로 적용한 최종 값을 반환

Function as argument

```
func multiplyByIndex<T: Numeric>(index: T, number: T) -> T {
  return index * number
}

func isEven(number: Int) -> Bool {
  return number & 1 == 0
}
```

```
func addingAllNumbers<T: BinaryInteger> (sum: T, number: T) -> T {
  return sum + number
}

immutableArray.enumerated()
  .map(multiplyByIndex(index:number:))
  .filter(isEven(number:))
  .reduce(0, addingAllNumbers(sum:number:))
```

Chaining

```
immutableArray.enumerated()
  .map { (offset, element) -> Int in
    return offset * element
  }.filter { (element) -> Bool in
    return element & 1 == 0
  }.reduce(0) { (sum, nextElement) -> Int in
    return sum + nextElement
}
```

Shorthand Argument Names

```
immutableArray.enumerated()
  .map { $0 * $1 }
  .filter { $0 & 1 == 0 }
  .reduce(0) { $0 + $1 }

immutableArray.enumerated()
.map( * )
.filter({ $0 & 1 == 0 })
.reduce(0, +)
```

swift/stdlib/public/core/Sequence.swift

```swift
@_inlineable
public func map<T>(
  _ transform: (Element) throws -> T
) rethrows -> [T] {
  let initialCapacity = underestimatedCount
  var result = ContiguousArray<T>()
  result.reserveCapacity(initialCapacity)

  var iterator = self.makeIterator()

  // Add elements up to the initial capacity without checking for regrowth.
  for _ in 0..<initialCapacity {
    result.append(try transform(iterator.next()!))
  }
  // Add remaining elements, if any.
  while let element = iterator.next() {
    result.append(try transform(element))
  }
  return Array(result)
}
```

swift/stdlib/public/core/Sequence.swift

```swift
@_inlineable
public func filter(
  _ isIncluded: (Element) throws -> Bool
) rethrows -> [Element] {
  return try _filter(isIncluded)
}

@_transparent
public func _filter(
  _ isIncluded: (Element) throws -> Bool
) rethrows -> [Element] {

  var result = ContiguousArray<Element>()

  var iterator = self.makeIterator()

  while let element = iterator.next() {
    if try isIncluded(element) {
      result.append(element)
    }
  }

  return Array(result)
}
```

map vs flatMap

```
let array = ["1j", "2d", "22", “33"]

let m1 = array.map({ Int($0) })
let f1 = array.flatMap({ Int($0) })

let m2 = array.map({ Int($0) }).first
let f2 = array.flatMap({ Int($0) }).first
```

swift/stdlib/public/core/FlatMap.swift

```
@_inlineable // FIXME(sil-serialize-all)
public func flatMap<SegmentOfResult>(
  _ transform: @escaping (Elements.Element) -> SegmentOfResult
) -> LazySequence<
  FlattenSequence<LazyMapSequence<Elements, SegmentOfResult>>> {
  return self.map(transform).joined()
}
```

```
@available(swift, deprecated: 4.1, renamed: "compactMap(_:)",
  message: "Please use compactMap(_:) for the case where closure returns an optional value")
@_inlineable // FIXME(sil-serialize-all)
public func flatMap<ElementOfResult>(
  _ transform: @escaping (Elements.Element) -> ElementOfResult?
) -> LazyMapCollection<
  LazyFilterCollection<
    LazyMapCollection<Elements, ElementOfResult?>>,
  ElementOfResult
> {
  return self.map(transform).filter { $0 != nil }.map { $0! }
}
```

Practice

```
let array: [[Int?]] = [[1, 2, 3], [nil, 5], [6, nil], [nil, nil]]
```

Q. map 과 flatMap 을 이용하여 다음 결과를 출력해보세요.

1. [[Optional(1), Optional(2), Optional(3)], [nil, Optional(5)], [Optional(6), nil], [nil, nil]]

2. [[1, 2, 3], [5], [6], []]

3. [Optional(1), Optional(2), Optional(3), nil, Optional(5), Optional(6), nil, nil, nil]

4. [1, 2, 3, 5, 6]

flatten and filter nil

```swift
let array: [[Int?]] = [[1, 2, 3], [nil, 5], [6, nil], [nil, nil]]

let m1 = array.map({ $0 })
let m2 = array.map({ $0.map({ $0 }) })
let m3 = array.map({ $0.flatMap({ $0 }) })

let f1 = array.flatMap({ $0 })
let f2 = array.flatMap({ $0.map({ $0 }) })
let f3 = array.flatMap({ $0.flatMap({ $0 }) })

// map
[[Optional(1), Optional(2), Optional(3)], [nil, Optional(5)], [Optional(6), nil], [nil, nil]]
[[Optional(1), Optional(2), Optional(3)], [nil, Optional(5)], [Optional(6), nil], [nil, nil]]
[[1, 2, 3], [5], [6], []]

// flatMap
[Optional(1), Optional(2), Optional(3), nil, Optional(5), Optional(6), nil, nil, nil]
[Optional(1), Optional(2), Optional(3), nil, Optional(5), Optional(6), nil, nil, nil]
[1, 2, 3, 5, 6]
```

Example

e.g. 프로젝트 적용 예

```
let aLabel = view.subviews.flatMap({ $0 as? UILabel }).first
//for object in objects {
//  if let object = object as? T {
//    return object
//  }
//}
```