Higher-order Function

Giftbot

Higher-order Function

고차함수란?

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

※ Higher-order Function이 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.

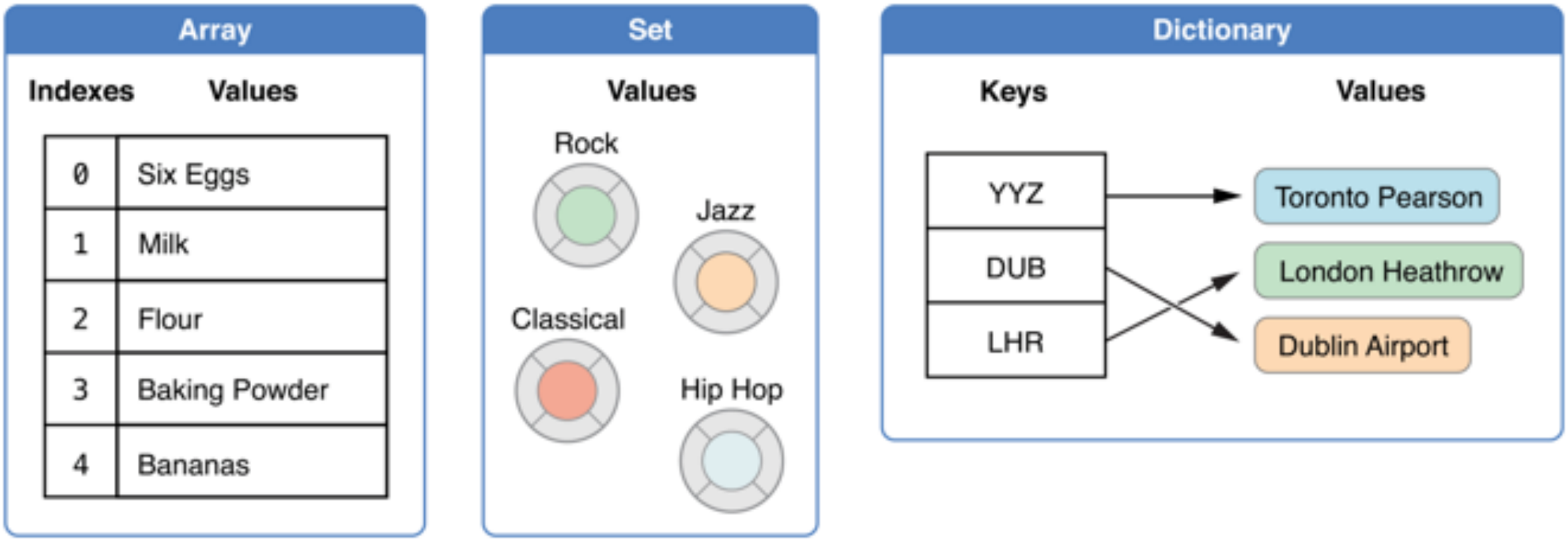
```
func firstClassCitizen() {
  print("function call")
}

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

let returnValue = function(firstClassCitizen)
returnValue()
```

Collection Type

고차함수는 일반적으로 컬렉션 타입에 사용한다(어레이, 셋, 딕셔너리)



Higher-order Functions in Swift

- **forEach**
  - 컬렉션의 각 요소(Element)에 동일 연산을 적용하며, 반환값이 없는 형태

- **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 타입을 지닌 값으로 반환. e.g. Int, String 타입

- **flatMap**
  - 중첩된 컬렉션을 하나의 컬렉션으로 병합

- **compactMap**
  - 컬렉션의 요소 중 옵셔널이 있을 경우 제거
  - (flatMap으로 사용하다가 Swift 4.1 에서 compactMap 으로 변경됨)

Playground

Practice 1

```
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

Input : myPet 배열 이용

[1번 문제]

Pet 타입의 배열을 파라미터로 받아 그 배열에 포함된 Pet 중
강아지의 나이만을 합산한 결과를 반환하는 sumDogAge 함수 구현

```
func sumDogAge(pets: [Pet]) -> Int
```

[2번 문제]

Pet 타입의 배열을 파라미터로 받아 모든 Pet이 나이를 1살씩 더 먹었을 때의
상태를 지닌 새로운 배열을 반환하는 oneYearOlder 함수 구현

```
func oneYearOlder(of pets: [Pet]) -> [Pet]
```

Case 1

```
func sumDogAge(pets: [Pet]) -> Int {
  var ageSum = 0
  for pet in pets {
    guard pet.type == .dog else { continue }
    ageSum += pet.age
  }
  return ageSum
}
```

Case 1

```
func sumDogAge(pets: [Pet]) -> Int {
  return pets
    .filter { $0.type == .dog }
    .reduce(0) { $0 + $1.age }
}
```

Case 2

```
func oneYearOlder(of pets: [Pet]) -> [Pet] {
  var oneYearOlderPets = [Pet]()
  for pet in pets {
    let temp = Pet(type: pet.type, age: pet.age + 1)
    oneYearOlderPets.append(temp)
  }
  return oneYearOlderPets
}
```

Case 2

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

Practice 2

```swift
let immutableArray = Array(1...40)
```

[문제]

immutableArray 배열의 각 인덱스와 해당 인덱스의 요소를 곱한 값 중
홀수는 제외하고 짝수에 대해서만 모든 값을 더하여 결과 출력

단, 아래 1 ~ 3번에 해당하는 함수를 각각 정의하고

이것들을 함께 조합하여 위 문제의 결과를 도출할 것

1. 배열의 각 요소 * index 값을 반환하는 함수

2. 짝수 여부를 판별하는 함수

3. 두 개의 숫자를 더하여 반환하는 함수

Define Function

```
func multiplyByIndex(index: Int, number: Int) -> Int {
  return index * number
}

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

func addTwoNumbers(lhs: Int, rhs: Int) -> Int {
  return lhs + rhs
}
```

Call Function

```
var sum = 0
for (index, num) in immutableArray.enumerated() {
  let multipliedNum = multiplyByIndex(index: index, number: num)

  if isEven(number: multipliedNum) {
    sum = addTwoNumbers(lhs: sum, rhs: multipliedNum)
  }
}
```

Function as argument

```
immutableArray.enumerated()
  .map(multiplyByIndex(index:number:))
  .filter(isEven(number:))
  .reduce(0, addTwoNumbers(lhs:rhs:))
```

Closures

```
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 }
```

Shorthand Argument Names

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

map vs compactMap

```
let array = ["1j", "2d", "3", "4"]

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

let m2 = array.map({ Int($0) })[0]
let f2 = array.compactMap({ Int($0) })[0]
```

map - 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)
}
```

filter - 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)
}
```

swift/stdlib/public/core/FlatMap.swift

```swift
@inlinable // lazy-performance
public func flatMap<SegmentOfResult>(
  _ transform: @escaping (Elements.Element) -> SegmentOfResult
) -> LazySequence<
  FlattenSequence<LazyMapSequence<Elements, SegmentOfResult>>> {
  return self.map(transform).joined()
}
```

```swift
@inlinable // lazy-performance
public func compactMap<ElementOfResult>(
  _ transform: @escaping (Elements.Element) -> ElementOfResult?
) -> LazyMapSequence<
  LazyFilterSequence<
    LazyMapSequence<Elements, ElementOfResult?>>,
  ElementOfResult
> {
  return self.map(transform).filter { $0 != nil }.map { $0! }
}
```