01.1주차1-3Groovy집합자료형

그루비의 집합 자료형

Table of Contents

범위 사용하기

그루비는 '범위'라는 개념을 제공한다. 범위에는 왼쪽과 오른쪽 경계가 있으며, 범위에 속한 각 요소에 대해 돌아가면서 '무언가'를 수행할 수 있다. 또 어떤 요소가 범위에 속하는지 검사도 할 수 있다.

범위 지정하기

범위는 점 두 개로 이루어진 연산자 .. 로 지정한다. 도트들의 왼쪽과 오른쪽에는 각각 경계를 표시한다. 이 연산자는 우선순위가 낮기 때문에 범위를 만들 때는 괄호로 감싸야하는 경우가 많다. 물론 생성자를 이용해서 만들 수도 있다.
또 ..< 는 반만 제외하는 범위를 만든다. 즉, 오른쪽 경계는 범위에 포함되지 않는다.
대개는 왼쪽 경계가 오른쪽보다 낮다. 반대로지정하는 경우는 "역 범위(reverserrange)"라고 부른다.

```groovy
//범위 지정하기 ( 작성중)

// 양쪽 경계를 포함하는 범위
assert(0..10).contains(0)
assert(0..10).contains(5)
assert(0..10).contains(10)
assert(0..10).contains(-1) == false
assert(0..10).contains(11) == false
// 한쪽 경계만 포함하는 범위
assert(0..<10).contains(9)
assert(0..<10).contains(10) == false

// 범위의 참조 변수
def a = 0..10
assert a instanceof Range
assert a.contains(5)
// 명시적 생성
a = new IntRange(0,10)
assert a.contains(5)
assert(0.0..1.0).containsWithinBounds(0.5)
// 날짜 범위
def today = new Date()
def yesterday = today -1
assert(yesterday..today).size() == 2
// 문자열 범위
assert('a'..'c').contains('b')
// 범위가 사용된 for루프
def log = ''
for (element in 5..9){
    log += element
}
assert log == '56789'
// 역 범위를 이용한 for 루프
log = ''
for (element in 9..5){
    log += element
}
assert log == '98765'
// 한쪽 경계만 포함한 역 범위를 클로저로 되풀이
log = ''
(9..<5).each{
    element -> log += element
}
assert log == '9876'
```

범위는 객체다.

모든 범위는 객체이므로 인자로 전달할 수도 있고, 메서드를 호출할 수도 있다. 범위에서 가장 눈에 띄는 메서드는 요소마다 지정된 클로저를 호출하는 each이고, 두번째로 contains 메서드다. contains 메서드는 주어진 값이 범위에 속하는지 검사한다.
범위도 기본 객체ㅣㅇ고 isCase와 같은 의미인 contains 메서드를 정의했기때문에 연산자 재정의에 참여할수 있다(grep 필터 와 switch case문에 범위를 사용할 수 있다.)

```
//범위는 객체다 ( 작성중)

// 범위를 이용한 되풀이
result = ''
(5..9).each{
    element -> result += element
}
assert result == '56789'

assert(0..10).isCase(5)
// 1. 분류에 범위 사용하기
age = 36
switch(age) {
    case 16..20 : insuranceRate = 0.05 ; break
    case 21..50 : insuranceRate = 0.06 ; break
    case 51..65 : insuranceRate = 0.07 ; break
    default : throw new IllegalArgumentException()
}
assert insuranceRate == 0.06
// 2.범위를 이용한 필터링
ages = [20,36,42,56]
midage = 21..50
assert ages.grep(midage) == [36,42]
```

범위 사용하기

어떤 자료형이든 다음 두가지 사항만 만족하면 범위에 쓸 수 있다.

- 자료형에 next와 previous를 구현한다. 즉, 자료형의 연산자 ++과 --를 재정의한다.
- 자료형이 java.lang.Comparable 인터페이스를 구현한다. 즉, compareTo 메서드를 정의함으로써 ' 비행접시' 연산자 ( )를 재정의한다.

일주일을 나타내는 Weekday 클래스를 만들고, 사용자 정의 범위를 표현하는 예제를 만들어보자

```
//사용자 정의 범위 - Weekday ( 작성중)

class Weekday implements Comparable{
    static final DAYS = [ 'Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
    private int index = 0
    Weekday(String day){
        index = DAYS.indexOf(day) // 생성자는 요일 이름을 인자로 받아서 인덱스로 만들어 저장한다.
    }
    Weekday next(){
        return new Weekday(DAYS[(index+1) % DAYS.size()])
    }
    Weekday previous(){
        return new Weekday(DAYS[index -1])
    }
    int compareTo(Object other) {
        return this.index <=> other.index;
    }
    String toString(){
        return DAYS[index]
    }
}

def mon = new Weekday('Mon')
def fri = new Weekday('Fri')

def worklog = ''
// 되풀이에 범위 사용
for(day in mon..fri){
    worklog += day.toString() + ' '
}
assert worklog == 'Mon Tue Wed Thu Fri '
```

리스트 사용하기

리스트 정의하기

기본적으로 쉼표로 구분된 요소들을 배열 기호로 감싸서 만들수 있다.

[item, item, item]
기호 사이를 비워두면 빈 리스트를 만들수 있다. 리스트는 java.util.ArrayList 객체이고, 생성자를 호출해서 명시적으로 생성할 수도 있다.
이렇게 생성한 리스트에 배열 기호 연산자로 접근할 수 있다. 또 범위에 toList를 호출하면 리스트를 만들면서 초기화할 수 있다.

```
//리스트 정의하기 ( 작성중)
myList = [1,2,3]

assert myList.size() == 3
assert myList[0] == 1
assert myList instanceof ArrayList

emptyList = []
assert emptyList.size() == 0

longList = (0..1000).toList()
assert longList[555] == 555

// myList 의 내용으로 내부를 채운다.
explicitList = new ArrayList()
explicitList.addAll(myList)
assert explicitList.size() == 3
explicitList[0] = 10
assert explicitList[0] == 10

explicitList = new LinkedList(myList)
assert explicitList.size() == 3
explicitList[0] = 10
assert explicitList[0] == 10
```

리스트의 연산자들

배열기호 연산자

```
//재정의된 배열 기호 연산자로 리스트의 일부에만 접근하기 ( 작성중)
myList = ['a', 'b', 'c', 'd', 'e', 'f']
assert myList[0..2] == ['a', 'b', 'c']
assert myList[0,2,4] == ['a', 'c', 'e']

myList[0..2] = ['x', 'y', 'z']
assert myList == ['x', 'y', 'z', 'd', 'e', 'f']

myList[3..5] = []
assert myList == ['x', 'y', 'z']

myList[1..1] = ['y', '1', '2']
assert myList == ['x', 'y', '1', '2', 'z']
```

음수도 인텍스 값에 쓸수 있다. 음수를 사용하면 리스트 끝부터 거꾸로 참조하게 된다. 리스트[0,1,2,3,4]는 리스트[-5,-4,-3,-2,-1]과 같다.
그리고 범위를 뒤집어서 사용하면 겨로가 리스트도 뒤집힌다. 따라서 list[4..0]은 [4,3,2,1,0]이 된다.

요수 추가하고 삭제하기

```
//요수의 추가와 삭제에 사용되는 리스트 연산자 ( 작성중)
myList = []

// plus(object)
myList += 'a'
assert myList == ['a']
// plus(Collection)
myList += ['b', 'c']
assert myList == ['a', 'b', 'c']

// 시프트 연산자는 append로 동작한다.
myList = []
myList << 'a' << 'b'
assert myList == ['a', 'b']
// minus(Collection)
assert myList - ['b'] == ['a']
// 곱하기
assert myList * 2 == ['a', 'b', 'a', 'b']
```

제어구조

```
//제어 구조에 사용되는 리스트 ( 작성중)
myList = ['a', 'b', 'c']

assert myList.isCase('a')
candidate = 'a'
switch(candidate){
    case myList : assert true; break // 1.포함 여부에 따른 분류
    default     : assert false
}
// 2. 교집합 필터
assert ['x', 'a', 'z'].grep(myList) == ['a']

// 3. 빈 리스트는 거짓(false)
myList = []
if(myList) assert false

// 4. for 루프에도 리스트를 사용할 수 있다.
log = ''
```

```
for( i in [1, 'x', 5]){
    log += i
}
assert log == '1x5'
```

- 1,2 패턴과 범위에서도 본것이다. isCase 메서드를 구현하면 grep과 switch에 쓸 수 있다.
- 3. 참/거짓(Boolean)검사에서 빈 리스트는 거짓이 된다.
- 4. 반복문에 리스트 같은 컬렉션을 사용하는 법과 리스트가 여러가지 자료형의 변수를 함께 담을 수 있음을 보여준다.

리스트 메서드 사용하기

리스트의 내용 다루기
리스트에 요소를 추가하거나 삭제하고, 다양한 방식으로 다른 리스트와 합친다. 또한 정렬하고, 뒤집고, 중첩된 리스트를 단일 리스트로 만들고, 기존 리스트에서 새로운 리스트를 만들어내기도 한다.

```
//리스트 내용을 수정하는 메서드 들( 작성중)
assert [1,[2,3]].flatten() == [1,2,3]

assert [1,2,3].intersect([4,3,1]) == [3,1]
assert [1,2,3].disjoint([4,5,6])

list = [1,2,3]
popped = list.pop() // 1. 리스트를 스택처럼
assert popped == 3
assert list == [1,2]

assert [1,2].reverse() == [2,1]
assert [3,1,2].sort() == [1,2,3]

def list = [ [1,0], [0,1,2] ]
list = list.sort { a,b -> a[0] <=> b[0] } // 2. 첫 번째 요소로 리스트 비교
assert list == [ [0,1,2], [1,0] ]

list = list.sort { item -> item.size() } // 3. 크기를 기준으로 리스트 비교
assert list == [ [1,0], [0,1,2] ]

list = ['a', 'b', 'c']
list.remove(2) // 4. 인덱스를 기준으로 요소 삭제
assert list == ['a', 'b']
list.remove('b') // 5. 값을 기준으로 요소 삭제
assert list == ['a']
// 6. 리스트 변환
def doubled = [1,2,3].collect {
    item -> item *2
}
assert doubled == [2,4,6]
// 7. 클로저를 만족하는 모든 요소 찾기
def odd = [1,2,3].findAll{
    item -> item % 2 == 1
}
assert odd == [1,3]
```

리스트 요소에는 무엇이든 넣을 수 있다. 다른 리스트를 중첩해서 넣을 수도 있다. 자바에서 다차원 배열을 만드는 것처럼, 그루비에서는 리스트의 리스트를 만들 수 있다. 중첩된 리스트를 평면적으로 만들기 위해서는 flatten 메서드를 사용한다.
리스트의 교집합에는 양쪽 리스트에 모두 존재하는 요소가 들어간다. disjoint 메서드를 호출해서 두 컬렉션의 교집합이 공집합인지 알수 있다.

- 1. push나 pop을 이용해 리스트를 스택처럼 사용하기도 한다. push()는 내부적으로 시프트 연산자(<<)를 호출 한다.
- 2,3 리스트 요소들이 Comparable 인터페이스를 구현하면, 추가 코딩없이 정렬할 수 있다. 정렬 방법으로 클로저를 지정할 수 있고, 이렇게 하면 정렬 대상이 되는 요소를 클로저에 전달하고, 그 결과 값을 내부에서 비교하게 된다.
- 4. 인덱스 번호로 요소를 삭제한다
- 5. 삭제할 요소의 값으로 삭제한다.
- 6. collect 메서드는 리스트의 각 요소에 되풀이해서 클로저를 호출하고, 그 결과를 모아 새로운 리스트를 만든다.
- 7. findAll을 이용하면, 클로저에서 참으로 평가한 요소들만 모인 리스트를 얻을 수 있다.

리스트 수정과 관련된 고려 사항이 두가지 있는데, 바로 중복 요소 제거와 널(null)값의 제거다. 중복 요소를 제거하는 한가지 방법은 중복없는 Set 같은 자료형으로 변환하는것이다. 리스트를 인자로 Set의 생성자를 부르면 된다.

```
//Set 생성자( 작성중)
def x = [1,1,1]
assert[1] == new HashSet(x).toList()
assert[1] == x.unique()
```

새로운 객체를 만들고 싶지 않다면 unique 메서드를 사용한다. unique 메서드는 요소의 순서를 바꾸지는 않는다.
리스트에서 널(null)값을 지우려면 널이 아닌 요소들을 지우지 않으면 된다. 예를 들어 앞에서 본 findAll 메서드를 이용한다.

```
//findAll 메서드를 이용한 예제 ( 작성중)
def x = [1,null,1]
assert [1,1] == x.findAll{ it != null }
assert [1,1] == x.grep{ it }
```

리스트의 내용에 접근하기

요소를 조사하는 메서드로는 요소의 개수를 얻는 메서드(count)나 최대값과 최솟값을 구하는 메서드(max/min), 주어진 클로저를 만족하는 첫번째 요소를 얻는 메서드(find) 그리고 주어진 클로저를 만족하는 모든, 또는 어떤 요소를 찾는 메서드(every/any) 등이 있다.
리스트에 있는 모든 요소에 대해 어떤 작업을 되풀이할 때는 each를 쓰면 앞으로 진행하고, eachReverse를 쓰면 반대로 진행한다.
값을 누적해주는 메서드도 있는데, join 메서드는 간단한 형태로, 모든 요소를 주어진 문자열과 합친 문자열로 만들어 준다.
inject 메서드는 클로저를 이용해서 새로운 기능을 삽입(inject)하게 해준다. 이기능은 중간 값과 현재 요소의 값에 대해 실행하게 된다.

```
//리스트의 조회, 되풀이, 누적 ( 작성중)
def list = [1,2,3]
// 조사
assert list.count(2) == 1
assert list.max() == 3
assert list.min() == 1

def even = list.find {
    item -> item % 2 == 0
}
assert even == 2

assert list.every { item -> item < 5 }
assert list.any { item -> item < 2 }

// 되풀이
def store = ''
list.each {
    item -> store += item
}
assert store == '123'

store = ''
list.reverseEach{
    item -> store += item
}
assert store == '321'

// 누적
assert list.join('-') == '1-2-3'
result = list.inject(0){
    clinks, guests -> clinks += guests
}
assert result == 0 + 1+2+3
assert list.sum() == 6

factorial = list.inject(1){
    fac, item -> fac *= item
}
assert factorial == 1 * 1*2*3
```

리스트 활용하기

토니호이 의 퀵소트 알고리즘을 구현하는 예제를 작성해보겠다.
문제를 자료형을 제한하지 않는 일반 버전으로 구현할 것이며, 여기에 오리형(duck type)에만 의존할 것이다.

퀵소트의 목적은 띄엄띄엄 비교하는 것이다. 이 전략이 성공하려면 서브리스트 두 개로 나눌 적절한 '중심축'을 찾아내야 한다. 두 서브리스트 중 한 쪽에는 중심축보다 작은 값들만 있고, 다른 쪽에는 큰 값들만 있어야 한다. 그 후에는 같은 전략을 각 서브리스트에 반복 적용하면 된다. 이 이론의 핵심은 둘로 쪼개진 리스트의 요소들은 서로 비교할 필요가 없다는데 있다. 매번 완벽한 중심축을 찾아내서 리스트를 정확히 둘로 쪼갤수 있다면, 알고리즘은 n*log(n)의 복잡도로 수행된다.
최악의 경우 매번 경계선에 있는 값이 중심축이 된다면 n2 의 복잡도로 수행된다.

```
//리스트를 이용한 퀵소트 ( 작성중)
def quickSort(list) {
    if (list.size() < 2) return list
    def pivot = list[list.size().intdiv(2)]
    def left = list.findAll { item -> item < pivot }
    // 1. 중심축을 기준으로 분류하기
    def middle = list.findAll { item -> item == pivot }
    def right = list.findAll { item -> item > pivot }
    return ( quickSort(left) + middle + quickSort(right)) // 재귀 호출
}

assert quickSort([]) == []
assert quickSort([1]) == [1]
assert quickSort([1,2]) == [1,2]
assert quickSort([2,1]) == [1,2]
assert quickSort([3,1,2]) == [1,2,3]
assert quickSort([3,1,2,2]) == [1,2,2,3]
assert quickSort([1.0f,'a',10,null]) == [null, 1.0f, 10, 'a'] // 2. 오리형 요소
assert quickSort('Karin and Dierk') == '  DKaadeiiknnrr'.toList() // 3 오리형 구조
```

- 1. 리스트를 세 개 사용했다. 중복되는 요소들을 버리지 않으려면 이런 시으로 구현해야 했다.
- 2. 오리형이라는 관점으로 접근했기 때문에 자료형이 서로 달라도 정렬할 수 있다.
- 3. 문자열도 정렬할 수 있다.(오리형 이라 가능)

맵 사용법

맵 정의 방법

맵을 정의하는 방법은 리스트와 비슷하다 다만, 배열기호 안에 어떤 자료형이든지 넣을 수있다는 점이 리스트와 다르다.
맵은 쉼표로 분리하며 요소들이 콜론으로 구분된 키-값의 쌍이라는것이다.
[키:값, 키:값, 키:값]
빈맵을 만들때에는 [:] 으로 만든다.

```
//맵 정의하기 ( 작성중)
def myMap = [a:1, b:2, c:3]

assert myMap instanceof HashMap
assert myMap.size() == 3
assert myMap['a'] == 1

def emptyMap = [:]
assert emptyMap.size() == 0

def explicitMap = new TreeMap()
explicitMap.putAll(myMap)
assert explicitMap['a'] == 1
```

보통 키(key)에는 문자열을 사용하기 때문에, 작은따옴표나 큰따옴표는 생략해도 된다. 단, 키에 특수문자가 없어야하고, 그루비의 키워드가 아니어야 한다.

맵의 연산자

객체를 얻는 첫 번째 방법은 배열 기호를 이용하는것이다. 이기능은 맵의 getAt 메서드로 구현했다.
두번째 방법은 "프로퍼티"처럼 도트 연산자와 키를 이용하는 방법이 있다. 프로퍼티에 대해서는 추후에 더 배울것이다.
세번째 방법은 get 메서드를 이용하는 방법이다. 이때 디폴트 값을 추가로 전달할 수 있는데, 맵에 주어진 키가 없으면 디폴트 값을 돌려준다. 디폴트 값을 전달하지 않으면 null이 사용된다.

```
//맵의 사용(GDK의 맵 매서드들) ( 작성중)
```

할당할때는 배열 기호를 이용하거나 '도트-키' 문법을 쓸수있다. '도트-키'문법을 쓸 때 키에 특수 문자가 있다면 다음처럼 키를 문자열로 만든다.

```
myMap = ['a.b':1]
assert myMap.'a.b' == 1
```

그냥 myMap.a.b 라고만하면 동작하지 않을텐데 myMap.getA().getB()라는 뜻이기 때문이다.

```
//맵의 요소 조회하기 ( 작성중)
def myMap = [a:1, b:2, c:3]
// 맵에서 요소 얻기
assert myMap['a'] == 1
assert myMap.a == 1
assert myMap.get('a') == 1
assert myMap.get('a',0) == 1
// 없는 요소 얻어보기
assert myMap['d'] == null
assert myMap.d == null
assert myMap.get('d') == null
// 디폴트 값을 지정하면서 얻기
assert myMap.get('d',0) == 0
assert myMap.d == 0
// 맵에 간단하게 할당하기
myMap['d'] = 1
assert myMap.d == 1
myMap.d = 2
assert myMap.d == 2
```

맵에서 정보를 얻는 방법을 예제를 통해서 보자.

```
// 맵의 요소 조회하기
def myMap = [a:1, b:2, c:3]
def other = [b:2, c:3, a:1]

assert myMap == other // 메서드 호출
// 일반적인 JDK 메서드
assert myMap.isEmpty() == false
assert myMap.size() == 3
assert myMap.containsKey('a')
assert myMap.containsValue(1)
assert myMap.keySet() == toSet(['a', 'b', 'c'])
assert toSet(myMap.values()) == toSet([1,2,3])
assert myMap.entrySet() instanceof Collection
// GDK에서 추가한 메서드
assert myMap.any() { entry -> entry.value > 2 }
assert myMap.every { entry -> entry.key < 'd' }
// 검증에서 쓰기위한 메서드
def toSet(list){
    new java.util.HashSet(list)
}
```

- 1. GDK에서 맵 조회용 메서드 두 개를 추가했다. any와 every이다.
- 주어진 클로저를 만족하는 요소가 하나라도 있는지(any)
- 모든 요소가 클로저를 만족하는지(every) 통해 참/거짓을 리턴한다.

맵이 주는 정보를 가지고 다양한 방법으로 요소들에 (혹은 키나 값들에) 되풀이 메서드를 호출할 수 있다.
keySet 나 entrySet 로 얻은 집합은 컬렉션이기 때문에 컬렉션을 지원하는 for루프를 쓸 수 있다.

```
//맵과 되풀이 메서드사용하기 (GDK) ( 작성중)
def myMap = [a:1, b:2, c:3]
// 요소들에 되풀이
def store = ''
myMap.each {
    entry ->
    store += entry.key
    store += entry.value
}
assert store.contains('a1')
assert store.contains('b2')
assert store.contains('c3')
// 키/값에 되풀이
store = ''
myMap.each {
    key, value ->
    store += key
    store += value
}
assert store.contains('a1')
assert store.contains('b2')
assert store.contains('c3')
// 키에만 되풀이
store = ''
for ( key in myMap.keySet()){
    store += key
}
assert store.contains('a')
assert store.contains('b')
assert store.contains('c')
// 값에만 되풀이
store = ''
for ( value in myMap.values()){
    store += value
}
assert store.contains('1')
assert store.contains('2')
assert store.contains('3')
```

다양한 방식으로 맵의 내용을 수정할수 있다.

- 주어진 컬렉션의 요소들을 키로 하는 모든 요소를 subMap 메서드로 얻기
- 주어진 클로저를 만족하는 요소들을 findAll 메서드로 얻기
- 주어진 클로저를 만족하는 요소 하나를 find 메서드로 얻기(맵에는 순서가 없기때문에 리스트와 달리 first 메서드가 없다)
- 클로저가 돌려주는 값들을 요소로 하는 새로운 리스트를 collect 메서드로 만들기(주어진 컬렉션에 이 리스트를 추가 할수도 있다.)

```
//맵의 내용 수정하기와 요소들로부터 새로 생성하기 ( 작성중)
def myMap = [a:1, b:2, c:3]
myMap.clear()
assert myMap.isEmpty()

myMap = [a:1, b:2, c:3]
myMap.remove('a')
assert myMap.size() == 2

myMap = [a:1, b:2, c:3]
def abMap = myMap.subMap(['a','b']) // 1. 원래의 맵에대한 새로운 뷰를 생성한다.
assert abMap.size() == 2

abMap = myMap.findAll {
    entry -> entry.value < 3
}
assert abMap.size() == 2
assert abMap.a == 1

def found = myMap.find {
    entry -> entry.value < 2
}
assert found.key == 'a'
assert found.value == 1

def doubled = myMap.collect {
    entry -> entry.value *= 2
}
assert doubled instanceof List
assert doubled.every {
    item -> item % 2 == 0
}

def addTo = []
myMap.collect(addTo){
    entry -> entry.value *= 2
}
assert doubled instanceof List
assert addTo.every{
    item -> item %2 == 0
}
```

- 1. subMap은 subList와 비슷하지만, 인자로 키들이 들어있는 Collection을 사용한다는 점에서 다른다.

3.3.3 맵 활용하기텍스트에서 단어 빈도를 세는 문제를 예제를 통해 다뤄보자.

```
//맵을 이용한 단어 출현 비도 분석 ( 작성중)
def textCorpus =
"""
Look for the bare necessities
The simple bare necessities
Forget about your worries and your strife
I mean the bare necessities
Old Mother Nature's recipes
That bring the bare necessities of life
"""

def words = textCorpus.tokenize()
def wordFrequency = [:]
words.each {
    word ->
    wordFrequency[word] = wordFrequency.get(word, 0) +1 // 1.
}
def wordList = wordFrequency.keySet().toList()
wordList.sort { wordFrequency[it] } // 2.

def statistic = "n"
wordList[-1..-6].each {
    word ->
    statistic += word.padLeft(12) + ': '
    statistic += wordFrequency[word] + "n"
}
assert statistic ==
"""
 necessities: 4
        bare: 4
         the: 3
        your: 2
        life: 1
          of: 1
"""
```

- 1. 소스에서는 그루비의 자료형들에 관한 지식을 잘 조합했다. 빈도수를 세는 부분은 실질적으로 한 줄이면 충분하다. 이절을 시작할 때 본 가상코드보다도 짧다.
- 2. wordList의 sort 메서드가 클로저를 받는 것이 굉장히 유용함을 알 수 있다. wordList의 메서드를 호출하면서 wordList와는 전적으로 다른 객체인 wordFrequency를 이용해 비교 로직을 구현하게 해주기 때문이다.