Chapter06. 컬렉션과 시퀀스

6장에서 다르는 내용

- 함수형 스타일로 컬렉션 다루기
- 시퀀스: 컬렉션 연산을 지연시켜 수행하기

컬렉션에 대한 함수형 API

- 컬렉션을 함수형 방식을 사용하면 일반적인 연산을 일관성 있게 표현할 수 있고 어휘를 공유할 수 있다.

원소 제거와 변환: filter와 map

```
fun main() {
    val list = listOf(1, 2, 3, 4)
    println(list.filter { it % 2 == 0 })
}


fun main() {
    val list = listOf(1, 2, 3, 4)
    println(list.map { it * it })
}
```

- filter, map 함수는 컬렉션을 다루는 토대
- 어떤 술어(predicate)를 바탕으로 컬렉션의 원소를 걸러내거나, 다른 형태로 변환시키고 싶을 때 사용
- map의 경우 키와 값을 처리하는 함수가 따로 존재한다.
    - `filterKeys, mapKeys`
    - `filterValues, mapValues`

컬렉션 값 누적: reduce와 fold

- 컬렉션의 정보를 종합하는 데 사용한다.
- 즉, 원소로 이뤄진 컬렉션을 받아서 한 값으로 반환한다.
- 값은 누적기를 통해 점진적으로 만들어진다.

reduce

```
fun main() {
    val list = listOf(1, 2, 3, 4)
    val summed = list.reduce { acc, element ->
        acc + element
    }
    println(summed)
    // 10
    val multiplied = list.reduce { acc, element ->
```

```
        acc * element
    }
    println(multiplied)
    // 24
}
```

- 컬렉션의 첫 번째 값을 누적기에 넣는다.
    - 따라서 빈 컬렉션에 이 함수를 호출해서는 안된다.
- 그 후 람다가 호출되면서 누적 값과 2번째 원소가 인자로 전달된다.

fold

```
data class Person(val name: String, val age: Int)

fun main() {
    val people = listOf(
        Person("Aleksei", 29),
        Person("Natalia", 28)
    )
    val folded = people.fold("") { acc, person →
        acc + person.name
    }
    println(folded)
}
// AlekseiNatalia
```

- 임의의 시작 값을 선택할 수 있다.

```
fun main() {
    val list = listOf(1, 2, 3, 4)
    val summed = list.runningReduce { acc, element →
        acc + element
    }
    println(summed)
    // [1, 3, 6, 10]  <1>
```

```
    val multiplied = list.runningReduce { acc, element ->
        acc * element
    }
    println(multiplied)
    // [1, 2, 6, 24] <1>
    val people = listOf(
        Person("Aleksei", 29),
        Person("Natalia", 28)
    )
    println(people.runningFold("") { acc, person ->
        acc + person.name
    })
    // [, Aleksei, AlekseiNatalia] <1>
}
```

- 중간 단계의 모든 누적 값을 뽑아내고 싶다면 `runningReduce, runningFold`
- 리스트를 반환한다.

컬렉션에 술어 적용: all, any, none, count, find

```
fun main() {
    val list = listOf(1, 2, 3)
    println(!list.all { it == 3 }) // true
    println(list.any { it != 3 }) // true
}
```

- 컬렉션의 모든 원소가 어떤 조건을 만족하는지 판단하는 연산
- `!all(조건) == any(!조건)`
    - 드모르간의 법칙
    - 가독성상 any 사용
- `!any() == none()`
- 술어에 대해 빈컬렉션 동작을 잘 생각하자.
    - `any()` : false

    - `none()` : true
    - `all()` : true
- count() vs filter & size
    - 전자는 원소의 개수만 추적한다.
    - 후자는 중간 컬렉션이 생긴다
- find
    - firstOrNull과 같다.

리스트를 분할해 리스트의 쌍으로 만들기 : partition

```
val canBeInClub27 = { p: Person → p.age <= 27 }

fun main() {
    val people = listOf(
        Person("Alice", 26),
        Person("Bob", 29),
        Person("Carol", 31)
    )
    val comeIn = people.filter(canBeInClub27)
    val stayOut = people.filterNot(canBeInClub27)
    println(comeIn)
    println(stayOut)
}


val (comIn, stayOut) = people.partition(canBeInClub27)
```

- 컬렉션을 어떤 술어를 만족하는 그룹과 그렇지 않은 그룹으로 나눠야할 때 사용

리슽를 여러 그룹으로 이뤄진 맵으로 바꾸기: groupBy

```
fun main() {
```

```
    val people = listOf(
        Person("Alice", 31),
        Person("Bob", 29),
        Person("Carol", 31)
    )
    println(people.groupBy { it.age })
}

// {31=[Person(name=Alice, age=31), Person(name=Carol, age=31)], 29=[Pers
```

- 컬렉션의 원소들을 두 그룹만으로 분리할 수 없는 경우
- 컬렉션의 원소를 어떤 특성에 따라 여러 그룹으로 나누고 싶을 때 사용

```
fun main() {
    val list = listOf("apple", "apricot", "banana", "cantaloupe")
    println(list.groupBy(String::first))
}
```

- 멤버참조도 사용가능

컬렉션을 맵으로 변환: associate, associateWith, associateBy

```
fun main() {
    val people = listOf(Person("Joe", 22), Person("Mary", 31))
    val nameToAge = people.associate { it.name to it.age }
    println(nameToAge) // {Joe=22, Mary=31}
    println(nameToAge["Joe"]) // 22
}
```

- 원소를 그룹화하지 않으면서 컬렉션으로부터 맵을 만들고 싶을 때
  - associate
- 원소로부터 키/값 쌍을 만들어내는 람다를 제공

```
fun main() {
    val people = listOf(
        Person("Joe", 22),
        Person("Mary", 31),
        Person("Jamie", 22)
    )
    val personToAge = people.associateWith { it.age }
    println(personToAge)
    // {Person(name=Joe, age=22)=22, Person(name=Mary, age=31)=31, Perso

    val ageToPerson = people.associateBy { it.age }
    println(ageToPerson)
    // {22=Person(name=Jamie, age=22), 31=Person(name=Mary, age=31)}
}
```

- 컬렉션의 원소와 다른 어떤 값 사이의 연관을 만들어내고 싶을 때
  - associateWith: 컬렉션의 원래 원소를 키로 사용, 람다가 만들어내는 값을 맵의 값으로
  - associateBy: 컬렉션의 원래 원소를 맵의 값으로 사용, 람다가 만들어내는 값을 맵의 키로
- 변환 함수가 키가 같은 값을 여러 번 추가하게 되면 마지막 결과가 덮어쓰게 된다.

가변 컬렉션의 원소 변경: replaceAll, fill

```
fun main() {
    val names = mutableListOf("Martin", "Samuel")
    println(names)
    names.replaceAll { it.uppercase() }
    println(names)
    names.fill("(redacted)")
    println(names)
}

// [Martin, Samuel]
```

```
// [MARTIN, SAMUEL]
// [(redacted), (redacted)]
```

- 가변 컬렉션으로 작업하면 더 편리한 경우가 있다.

컬렉션의 특별한 경우 처리: ifEmpty

```
fun main() {
    val empty = emptyList<String>()
    val full = listOf("apple", "orange", "banana")
    println(empty.ifEmpty { listOf("no", "values", "here") })
    println(full.ifEmpty { listOf("no", "values", "here") })
}
```

- 컬렉션이 비어있지 않은 경우에만 처리를 계속하는 것이 타당한 경우 사용한다.

```
fun main() {
    val blankName = " "
    val name = "J. Doe"
    println(blankName.ifEmpty { "(unnamed)" })
    println(blankName.ifBlank { "(unnamed)" })
    println(name.ifBlank { "(unnamed)" })
}
```

- `ifBlank` : 문자열에서 ifEmpty의 형제

컬렉션 나누기: chuncked와 windowed

```
val temperatures = listOf(27.7, 29.8, 22.0, 35.5, 19.1)


fun main() {
    println(temperatures.windowed(3))
```

```
    // [[27.7, 29.8, 22.0], [29.8, 22.0, 35.5], [22.0, 35.5, 19.1]]

    println(temperatures.windowed(3) { it.sum() / it.size })
    // [26.5, 29.0999999998, 25.5333333333]
}
```

- 컬렉션의 데이터를 연속적인 시간의 값들로 처리하고 싶을 때
  - 슬라이딩 윈도우
- 출력을 변환할 람다를 전달할 수도 있다.

```
val temperatures = listOf(27.7, 29.8, 22.0, 35.5, 19.1)


fun main() {
    println(temperatures.chunked(2))
    // [[27.7, 29.8], [22.0, 35.5], [19.1]]

    println(temperatures.chunked(2)) { it.sum() })
    // [57.5, 57.5, 19.1]
}
```

- 컬렉션을 어떤 주어진 크기의 서로 겹치지 않는 부분으로 나누고 싶을 때
- 람다를 전달하면, 출력을 변환시킨다.

컬렉션 합치기: zip

```
fun main() {
    val names = listOf("Joe", "Mary", "Jamie")
    val ages = listOf(22, 31, 22, 44, 0)
    println(names.zip(ages))
    // [(Joe, 22), (Mary, 31), (Jamie, 22)]

    println(names.zip(ages) { name, age -> Person(name, age) })
```

```
    // [Person(name=Joe, age=22), Person(name=Mary, age=31), Person(name
}
```

- 연관이 있는 데이터가 들어있는 별도의 두 리스트를 한꺼번에 종합해야 할 때
- 람다를 전달하면 출력을 변환할 수 있다.
- 결과 컬렉션의 길이는 더 짧은 컬렉션의 길이와 같다.
    - 즉 공통 길이의 인덱스에 해당하는 원소들만 처리한다.

```
fun main() {
    val names = listOf("Joe", "Mary", "Jamie")
    val ages = listOf(22, 31, 22, 44, 0)
    println(names zip ages)
}
```

- 중위 표기법을 쓸 때는 람다를 전달할 수 없다.

```
fun main() {
    val names = listOf("Joe", "Mary", "Jamie")
    val ages = listOf(22, 31, 22, 44, 0)
    val countries = listOf("DE", "NL", "US")
    println(names zip ages zip countries)
    // [((Joe, 22), DE), ((Mary, 31), NL), ((Jamie, 22), US)]

}
```

- 항상 2개의 리스트에 대해 작동하기 때문에 내포된 쌍의 리스트가 될 뿐이다.

내포된 컬렉션의 원소 처리: flatMap과 flatten

```
class Book(val title: String, val authors: List<String>)

val libaray = listOf(
    Book("1", listOf("A, B, C, D")),
```

```
    Book("2", listOf("C, E, F"))
)
```

```
fun main() {
    val authors = library.flatMap { it.authors }
    println(authors)
    // [A, B, C, D, C, E, F]
}
```

- `flatMap`
  - 컬렉션의 각 원소를 파라미터로 주어진 함수를 사용해 변환 또는 매핑한다.
  - 그 후 변환한 결과를 하나의 리스트로 합친다.
- `flatten()`
  - 변환할 것이 없고 평평한 컬렉션으로 만들 때

지연 계산 컬렉션 연산: 시퀀스

- map이나 filter 컬렉션 함수는 결과 컬렉션을 즉시 생성한다.
- 컬렉션 함수를 연쇄하면?
  - 매 단계마다 계산 중간 결과를 새로운 컬렉션에 임시로 담는다는 의미
- 시퀀스는 자바8의 스트림과 비슷하게 중간 임시 컬렉션을 사용하지 않고 컬렉션 연산을 연쇄하는 방법을 제공한다.

```
fun main() {
    listOf(1, 2, 3, 4)
        .asSequence() // 원본 컬렉션을 시퀀스로 변환
    .map {
        print("map($it) ") // 시퀀스도 같은 api 제공
        it * it
    }
    .filter {             // 시퀀스도 같은 api 제공
        print("filter($it) ")
```

```
        it % 2 == 0
    }
    .toList() // 결과 시퀀스를 다시 리스트로 변환
}
```

```
public interface Sequence<out T> {
    /**
     * Returns an [Iterator] that returns the values from the sequence.
     *
     * Throws an exception if the sequence is constrained to be iterated once a
     */
    public operator fun iterator(): Iterator<T>
}
```

- 중간 결과를 저장하는 컬렉션이 생기지 않기 때문에 원소가 많은 경우 성능이 좋아진다.
- 지연 계산 시퀀스는 `Sequence` 인터페이스에서 시작
  - 한 번에 하나씩 열거될 수 있는 원소의 시퀀스를 표현
  - iterator() 메소드를 통해 시퀀스에서 원소 값들을 얻을 수 있다.
- `Sequence` 인터페이스의 장점: 지연 계산
  - 시퀀스의 원스는 필요할 때 게르을게 계산된다.
  - 따라서 중간 처리 결과를 저장할 컬렉션을 만들지 않고도 연산을 연쇄적으로 적용해서 효율적으로 수행할 수 있다.
- 왜 시퀀스를 다시 컬렉션으로 되돌려야할까?
  - 항상 시퀀스가 컬렉션보다 좋은 것은 아니기 때문
  - 시퀀스 원소를 이덱스를 사용해 접근하는 등 다른 api 메소드를 호출해야한다면 리스트로 변환해야 한다.

시퀀스 연산 실행: 중간 연산과 최종 연산

```
sequence.map { ... }.filter { ... }.toList()
        // ------ 중간연산 -------  --최종 연산--
```

- 시퀀스에 대한 연산은 중간 연산과 최종 연산으로 나뉜다.

- 중간 연산: 다른 시퀀스를 반환
  - 최소 시퀀스의 원소를 변환하는 방법을 안다.
  - 항상 지연 계산된다.
- 최종 연산: 결과를 반환
  - 결과는 최초 컬렉션에 대해 변환을 적용한 시퀀스에서 일련의 계산을 수행해 얻을 수 있는 컬렉션이나 원소, 수, 또는 다른 객체

```
fun main() {
    listOf(1, 2, 3, 4)
        .asSequence() // 원본 컬렉션을 시퀀스로 변환
        .map {
            print("map($it) ") // 시퀀스도 같은 api 제공
            it * it
        }
        .filter {             // 시퀀스도 같은 api 제공
            print("filter($it) ")
            it % 2 == 0
        }
        .toList() // 결과 시퀀스를 다시 리스트로 변환, 최종연산
}
```

- 최종 연산이 호출되지 않으면 아무 내용도 출력되지 않는다.
- 시퀀스의 경우 모든 연산은 각 원소에 대해 순차적으로 적용된다.
  - 첫 번째 원소가 처리되고, 다시 두 번째 원소 처리
- 따라서 map, filter 와 filter, map 결과는 변환의 전체 횟수는 다르다.
  - 따라서 대략적으로, 연쇄적인 연산에서 더 빨리 원소들을 제거하면 할수록 성능이 더 좋아진다.

시퀀스 만들기

```
fun main() {
    val naturalNumbers = generateSequence(0) { it + 1 }
```

```
    val numbersTo100 = naturalNumbers.takeWhile { it <= 100 }
    println(numbersTo100.sum()) // 지연 계산 수행시점
}
```

- asSequence() 말고 `generateSequence` 메서드를 호출해 시퀀스를 만들 수 있다.

```
fun File.isInsideHiddenDirectory() =
        generateSequence(this) { it.parentFile }.any { it.isHidden }

fun main() {
    val file = File("/Users/svtk/.HiddenDir/a.txt")
    println(file.isInsideHiddenDirectory())
}
```

- 일반적인 용례중 하나: 객체의 조상들로 이뤄진 시퀀스를 만들어내는 것
- 파일의 상위 디렉터리를 조사하면서 숨김 속성을 가진 디렉터리가 있는지 검사
    - 조건을 만족하는 디렉터리르 찾은 뒤에는 더 이상 상위 디렉터리를 뒤지지 않는다.

요약

- 컬렉션 원소들을 직접 순회하면서 처리하는 대신, 대부분의 일반적인 표준 라이브러리 함수와 작성한 람다를 조합해 처리할 수 있다.
- filter, map 함수는 컬렉션을 다루는 기본, 술어에 따라 원소를 걸러내거나 새로운 형태로 변환할 수 있다.
- reduce와 fold 연산을 사용하면 컬렉션으로부터 정보를 종합할 수 있다.
    - 이를 통해 한 가지 값을 계산해낼 수 있다.
- aasociate와 groupBy 함수를 사용하면 평평한 리스트를 맵으로 바꿀 수 있다.
- 데이터 컬렉션이 인덱스와 연관이 있는 경우에는 chunked, windowed, zip 함수를 사용해 컬렉션 원소의 하위 그룹을 만들거나 여러 컬렉션을 하나로 합칠 수 있다.
- Boolean을 반환하는 람다 함수인 술어를 사용하면 all, any, none과 그 형제 함수들을 활용해 어떤 불변 조건이 컬렉션에 대해 성립하는지 여부를 검사할 수 있다.

- 내포된 컬렉션을 처리해야 할 경우 flattend 함수를 써너 내포된 원소를 꺼낼 수 있으며, flatMap을 통해 변환과 펼쳐내기를 한꺼번에 수행할 수 있다.
- 시퀀스를 사용하면 중간 결과를 담는 컬렉션을 생성하지 않고도 컬렉션에 대한 여러 연산을 지연 계산해서 조합할 수 있어 코드를 더 효율적으로 작성할 수 있다.
    - 컬렉션 조작을 위해 사용한 함수와 똑같은 함수를 시퀀스에도 적용할 수 있다.