아이템 50: 컬렉션 처리 단계 수를 제한하라

> 표준 컬렉션 처리는 element를 활용해 반복을 하고, 계산을 위해 **추가적인 컬렉션을 만들어 사용합니다**. 시퀀스 처리도 시퀀스 전체를 **wrap하는 객체가 만들어지며, 조작을 위해서 또 다른 추가적인 객체를** 만들어냅니다. 두 처리 모두 element 수가 많다면 비용이 꽤 크게 들어가므로, 적절한 메서드를 사용해 단계 수를 제한하는 것이 좋습니다.

시퀀스의 경우도 연산 내용이 시퀀스 객체로 전달되므로, 인라인으로 사용할 수 없습니다. 따라서 람다 표현식을 객체로 만들어서 사용해야 합니다.

- 왜 인라인으로 전달할 수 없도록 설계했을까요?

```
public fun <T> Sequence<T>.filterIndexed(predicate: (index: Int, T) -> Boolean): Seque
nce<T> {
    // TODO: Rewrite with generalized MapFilterIndexingSequence
    return TransformingSequence(FilteringSequence(IndexingSequence(this), true, { pred
icate(it.index, it.value) }), { it.value })
}

public inline fun <T> Iterable<T>.filterIndexed(predicate: (index: Int, T) -> Boolea
n): List<T> {
    return filterIndexedTo(ArrayList<T>(), predicate)
}
```

참고: Sequence가 지원하는 연산 중, 시퀀스를 wrap 하는 경우는 inline을 지원하지 않지만 나머지는 지원한다.

예제: single()

```
.filter {it != null}
.map {it!!}

=> .filterNotNull{}
```

```
.sortedBy{<key2>}
.sortedBy{<key1>}
=> .sortWith{compareBy({<key1>}, {<key
```

```
.map {<Transform>}
.filterNotNull

=> .mapNotNull {<Transfrom>}

.map {<TransForm>}
.joinToString()
=> joinToString{<Transform}

.filter{<pred1>}
.filter{<pred2>}
=> .filter{<pred1> && <pred2>}

.filter{it is Type}
.map{it as Type}
=> .filterIsInstance<Type>()
```

```
2>})}

listOf(...)
.flterNotNull()
=> listOfNotNull(...)

.withIndex()
.filter{(index,elem) ->
  <predicate using index>
}.map{it.value}

=> .filterIndexed{ index, elem ->
  <predicate using index>
}
```

정리

적절한 함수 사용을 통해 컬렉션 처리 단계를 줄여서

1. 전체 컬렉션에 대한 반복을 줄이자
2. 중간 컬렉션 생성 비용을 줄이자