Kotlin의 Collection 함수

hongbeom
Jun 3, 2020 · 24 min read

코틀린의 컬렉션 함수들을 그림과 함께 쉽게 알아봅니다.

Photo by Marc Reichelt on Unsplash

> 이 글은 *Elye*님의 동의 하에 번역하여 작성한 글입니다. 오역이 있을 수 있는 점 미리 말씀드립니다.

원문 링크👇

Kotlin Collection Functions Cheat Sheet

Enable learning and finding relevant collection function easier

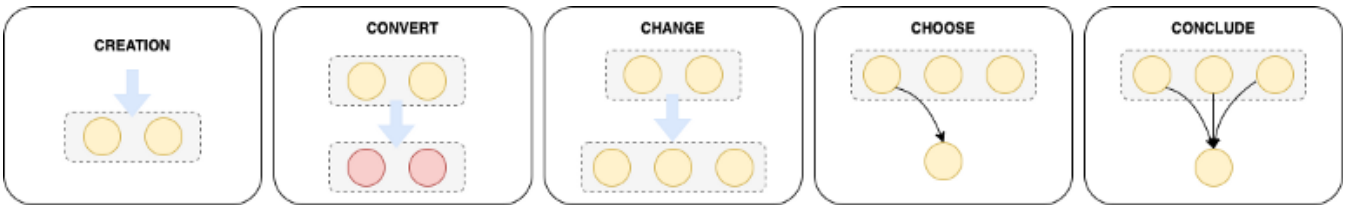
medium.com

배경

코틀린의 컬렉션은 풍부한 기능들을 제공합니다. 약 200개가 존재하는데 이것들을 우리가 사용하기 위해선 먼저 어떤 것들이 있는지 알 필요가 있습니다. 물론 이것들 중 하나를 찾기 위해서 200개를 모두 훑어보는 것은 어렵기 때문에 함수에 대해 기능별로 분류하여 정리해보았습니다.

크게 5가지 카테고리로 나누어보았습니다.

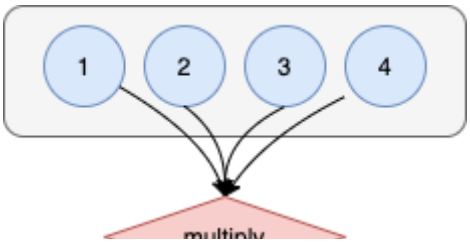
1. **Creation** — 컬렉션을 생성하는 함수 (ex : listOf)
2. **Convert** — 다른 유형으로 캐스팅하는 함수 (ex: asMap)
3. **Change** — 내용을 변환하는 함수 (ex: map)
4. **Choose** — 항목 중 하나에 접근하는 함수(ex : get)
5. **Conclude** — 항목에서 무언가를 생성하는 함수(ex: sum)

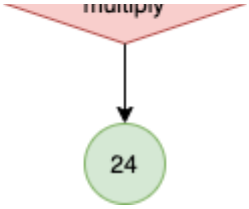


이 글에서는 쉽게 이해하기 어려운 기능 중 일부에 대해 몇 가지 예를 제공하고 이름으로 쉽게 알기 쉬운 함수들은 공식 문서에 대한 링크를 제공합니다.

예제 1

리스트에 있는 모든 정수 값의 곱셈을 수행하는 함수 찾기





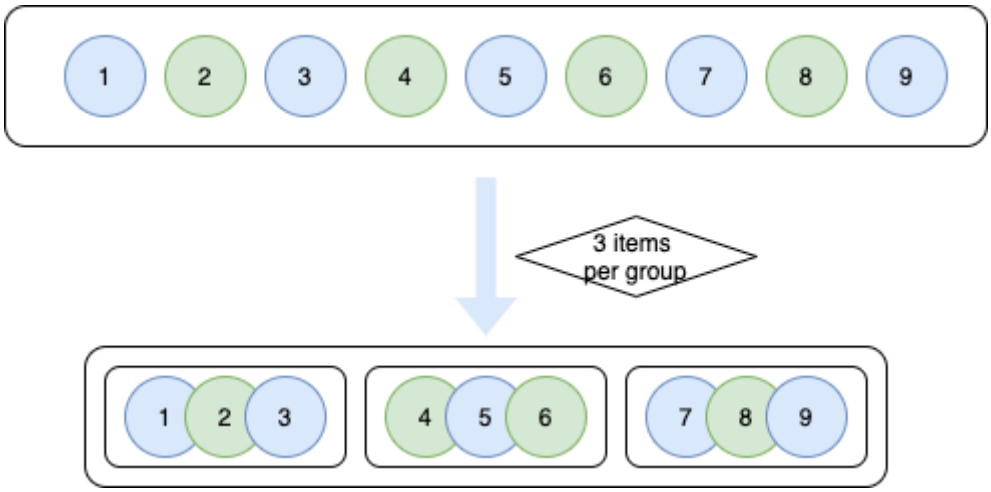
최종 결과 값은 모든 항목에서 생성된 단일 값이기 때문에 Conclude 카테고리에서 관련 함수를 찾아볼 수 있습니다.

`reduce` 함수를 통해 이를 수행할 수 있다는 것을 찾을 수 있습니다.

```
list.reduce{ result, item -> result * item }
```

예제 2

리스트를 더 작은 크기의 여러 하위 리스트로 나누는 함수



최종 결과는 기존 리스트를 다른 형태의 컬렉션으로 변환하는 것입니다. 이런 함수는 Change 카테고리에서 찾아볼 수 있습니다.

`chunk` 함수로 이를 수행할 수 있다는 것을 찾아볼 수 있습니다.

```
list.chunked(3)
```

카테고리 관의 관계

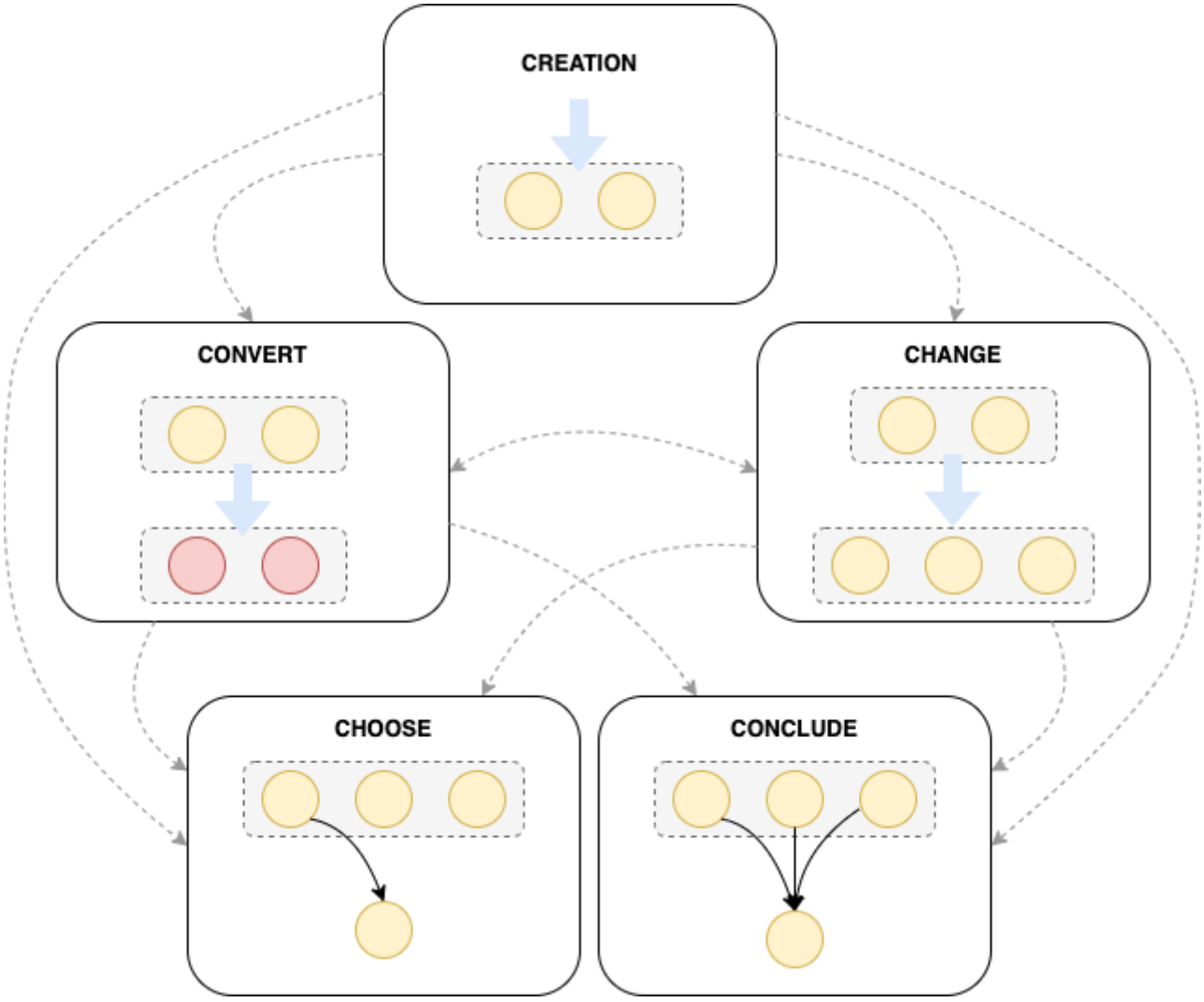
카테고리 별로 살펴보면, 컬렉션을 만들 때 Creation이 첫 번째 상태이고, Change와 Convert는 중간 상태이며, Close 또는 Choose는 최종 상태에 해당하는 것을 알 수 있습니다.

예시

```
listOf(1, 2, 3)    // Creation
.map { it * 2 }    // Change
.sum()             // Conclude
```

물론 중간 상태를 건너뛰고 최종 상태로 갈 수도 있습니다.

```
listOf(1, 2, 3)    // Creation
.sum()             // Conclude
```



참고로 일부 함수에는 `By`, `To`, `With` 이 추가된 변형된 함수가 있습니다. 이에 대한 설명은 여기서 확인할 수 있습니다.

Kotlin Collection's appended To, By, and With

Understand of Collection function variants by the name
proandroiddev.com

또한 원어와 과거 분사형 함수의 경우 아래 블로그에서 확인해 볼 수 있습니다.

Understand Kotlin Past Participle Named Collection Function

Enable faster learning of Kotlin collection function by name

levelup.gitconnected.com

Creation 카테고리

이 카테고리에서는 더 쉽게 함수를 찾을 수 있도록 하위 함수로 분류합니다.

1. Creation **Compose** — 새로운 컬렉션 인스턴스화
2. Creation **Copy** — 컬렉션 복사
3. Creation **Catch** — try-catch와 유사하게 다른 무언가를 생성

1. Creation Compose

이것들은 우리가 컬렉션을 직접적으로 인스턴스화하는 것을 돕는 함수들 입니다. 이 부분은 모두 정식 코틀린 문서가 링크되어 있습니다.

```
// 비어있는 컬렉션
emptyList, emptyMap, emptySet

// 읽기 전용 컬렉션
listOf, mapOf, setOf

// 변경 가능한 컬렉션
mutableListOf, mutableMapOf, mutableSetOf, arrayListOf

// 다른 소스를 첨가할 수 있는 빌드 컬렉션
buildList, buildMap, buildSet

// Linked 컬렉션
linkedMapOf, linkedSetOf (더 많은 설명은 이곳에서 stackOverflow)

// 정렬된 컬렉션
sortedMapOf, sortedSetOf (더 많은 설명은 이곳에서 stackOverflow)
```

```
// Hash 컬렉션
hashMapOf, hashSetOf (더 많은 설명은 이곳에서 stackOverflow)

// 프로그래밍 방식으로 컬렉션 만들기
List, MutableList, Iterable
```

2. Creation Copy

주로 배열을 복사하기 위해 쓰입니다.

```
copyInto      // 배열로 복사
copyOfRange   // 부분 복사
copyOf        // 전부 복사
toCollection  // 컬렉션으로 복사
```

3. Creation Catch

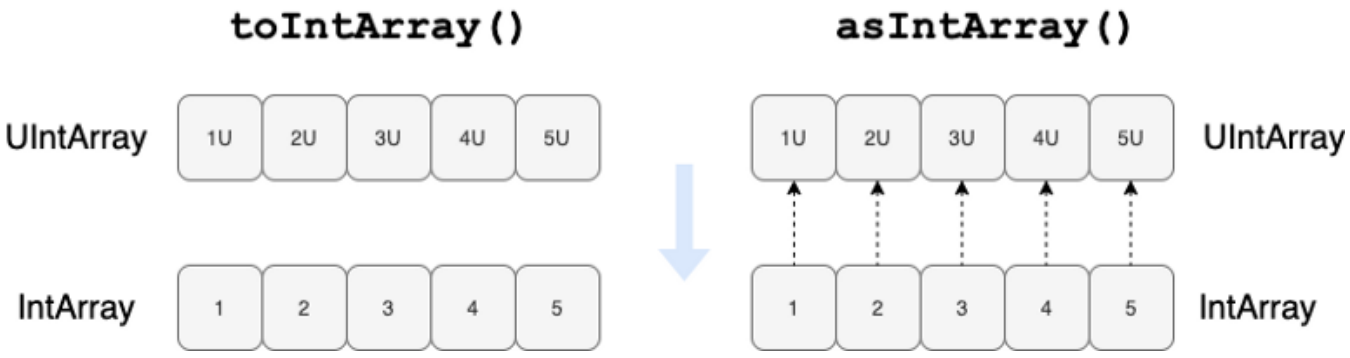
```
ifEmpty          // 비어 있으면 기본값을 준다.
orEmpty          // null을 빈 값으로 변경
requireNoNulls   // 원소가 null인 경우 에러
listOfNotNull    // null인 원소를 제거
```

Conversion 카테고리

이 카테고리는 함수가 컬렉션을 다른 타입으로 변경하려고 시도하는 함수들에 대한 카테고리입니다. 더 쉽게 찾을 수 있도록 하위 카테고리로 나누어보겠습니다.

1. Conversion **Copy** — 다른 타입의 새 컬렉션으로 변환
2. Conversion **Cite** — 원래 값에 대한 참조가 있는 다른 타입으로 변환

한 가지 좋은 예시로 `toIntArray(Copy)` 와 `asIntArray(Cite)` 가 있습니다.



```
// toIntArray 예시 (새로운 복사)
val uIntArray = UIntArray(3) { 1U }
val toIntArray = uIntArray.toIntArray()

toIntArray[1] = 2
println(toIntArray.toList())  // [1, 2, 1]
println(uIntArray.toList())   // [1, 1, 1]

// asIntArray 예시 (참조 복사)
val uIntArray = UIntArray(3) { 1U }
val asIntArray = uIntArray.asIntArray()
asIntArray[1] = 2
println(asIntArray.toList())  // [1, 2, 1]
println(uIntArray.toList())   // [1, 2, 1]
```

1. Conversion Copy

```
// 배열 타입으로
toBooleanArray, toByteArray, toCharArray, toDoubleArray,
toFloatArray, toIntArray, toLongArray, toShortArray, toTypedArray,
toUByteArray, toUIntArray, toULongArray, toUShortArray

// 읽기 전용 컬렉션으로
toList, toMap, toSet

// 변경 가능한 컬렉션으로
toMutableList, toMutableMap, toMutableSet, toHashSet

// 정렬된 컬렉션으로
toSortedMap, toSortedSet

// 엔트리를 쌍으로 변환
toPair        // 아래에 추가 설명이 있습니다.

// 맵을 프로퍼티로 변환
toProperties // 아래에 추가 설명이 있습니다.
```

`toPair` 를 사용하여 `Map` 의 엔트리를 `Pair` 로 변환할 수 있습니다. 아래는 예시 코드입니다.

```
map.entries.map { it.toPair() }

// 보통 이렇게 사용하여 변경합니다.
map.toList()

// 엔트리를 모두 Pair로 변환할 때,
// Map의 `toList`는 기본적으로 `toPair()`를 사용합니다.
```

`toProperties` 는 `Map` 을 `Properties` 로 변환합니다(Java의 네이티브 클래스). 이것은 `Map<String, String>` 의 하위 클래스입니다.

```
val map = mapOf("x" to "value A", "y" to "value B")
val props = map.toProperties()

println(props.getProperty("x"))         // value A
println(props.getProperty("z"))         // null
println(props.getProperty("y", "fail")) // value B
println(props.getProperty("z", "fail")) // fail

println(map.get("x"))                   // value A
println(map.get("z"))                   // null
println(map.getOrDefault("y", "fail"))  // value B
println(map.getOrDefault("z", "fail"))  // fail
```

2. Conversion Cite

```
// 배열 타입으로 변환
asByteArray, asIntArray, asLongArray, asShortArray, asUByteArray,
asUIntArray, asULongArray, asUShortArray,

// 컬렉션 타입으로 변환. list와 sequestion에 대한 비교는 여기서 확인할 수 있습니다.
asIterable, asList, asSequence

// 인덱스가 있는 반복자로 변환
withIndex    // 아래에 추가 설명이 있습니다.

// 사용자가 지정한 기본값을 사용하여 Map으로 변환
withDefault  // 아래에 추가 설명이 있습니다.
```

`withIndex` 를 사용하여 리스트를 IndexedValue 로 변환할 수 있습니다.(인덱스가 있는 반복자)

```
val list = listOf("A", "B", "C")
val indexed = list.withIndex()

println(list)  // [A, B, C]

println(indexed.toList())
// [IndexedValue(index=0, value=A),
//  IndexedValue(index=1, value=B),
//  IndexedValue(index=2, value=C)]
```

`withDefault` 는 `Map` 을 일치하는 키가 없는 경우 사용자가 지정한 식을 사용하여 해당 값을 리턴하는 `Map` 으로 변환할 수 있습니다.

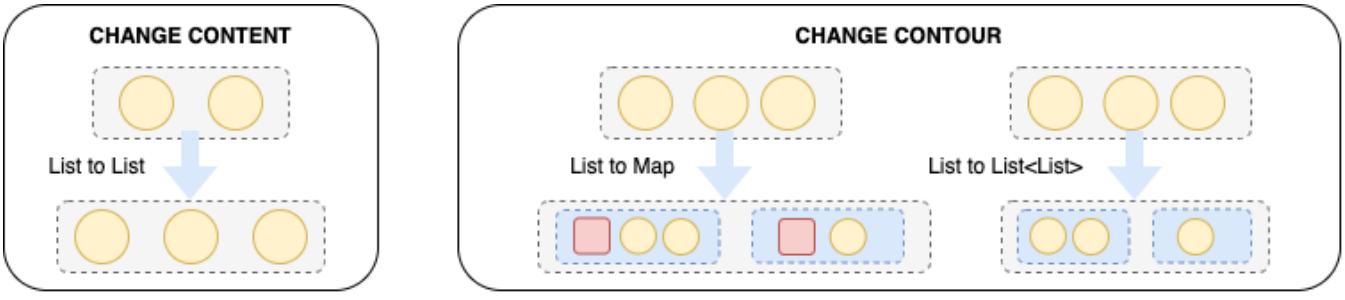
```
val map = mutableMapOf(1 to 1)

// 일치하는 값이 없을 때 식을 사용하여 값 반환 (key x 2)
val default = map.withDefault { k -> k * 2 }

println(default.getValue(10)) // 10이라는 키가 없기 때문에 key에 곱하기 2를 한 값인 20을 반환
println(map.getValue(10))     // 10이라는 키가 없어서 에러
```

Change Category

이 카테고리는 함수가 컬렉션의 내용을 변경하거나 컬렉션을 다른 구조로 만들 수 있는 함수들에 대한 카테고리입니다. 마찬가지로 하위 카테고리로 나누었습니다.

1. Change **Content** — 함수의 결과는 원래 컬렉션 타입(ex : `List` 를 출력하는 `list` 함수)을 유지하며 요소에 대한 타입도 유지됩니다.(ex : 요소 타입이 `Int` 인 경우 결과 요소 타입도 `Int`). 예를 들자면 `filter` 함수가 있습니다.

2. Change **Contour** — 컬렉션 타입을 변경하거나(ex : `groupBy` 함수에 따라 `Map` 을 출력하는 `List` 함수) 결과 요소 타입이 변경되는 함수 (ex : `Int` 에서 `List<Int>` 로 변경하는 `chunked` 함수)



1. Change Content

이 부분은 두 가지 종류로 나눌 수 있습니다.

- 새 컬렉션을 생성하고 내용을 변경한 후 리턴
- 아무것도 리턴하지 않고 내용을 변경하여 스스로 변환

간단한 예시로 `add` VS `plus` 가 있습니다.

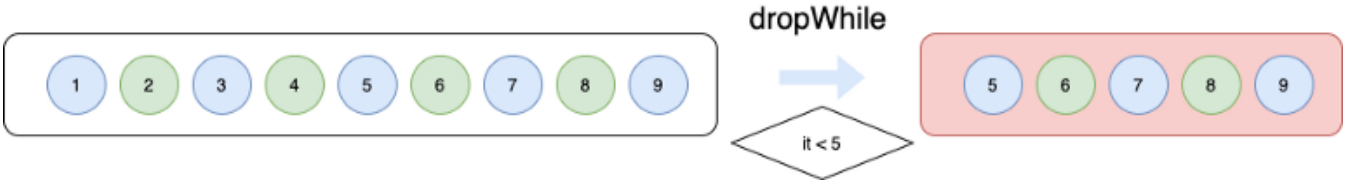
```
val list = listOf(1)
val mutableList = mutableListOf(1)

println(list)           // [1]
println(mutableList)    // [1]

val newList = list.plus(2)
mutableList.add(2)

println(list)           // [1]
println(newList)        // [1, 2]
println(mutableList)    // [1, 2]
```

이 함수들은 같은 목적을 가지고 있지만 위의 코드처럼 다른 결과가 나왔습니다. 자기 자신이 변경되는 `add` 같은 함수는 기울임체로 작성하고, 새로운 리스트를 생성하는 `plus` 같은 함수는 기울이지 않은 글씨로 작성하겠습니다.

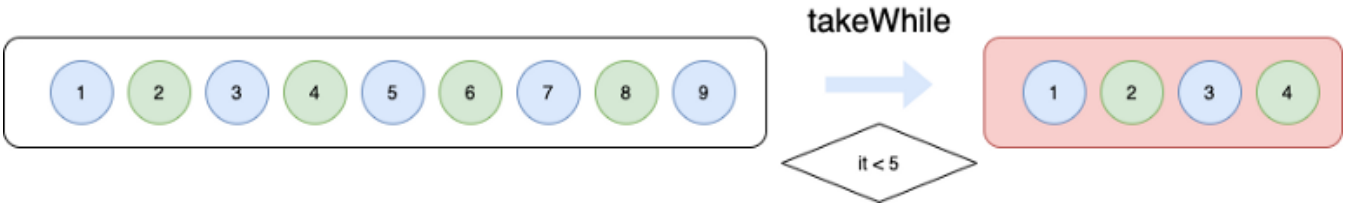
plus, *add*

이제 해당 함수들을 살펴보겠습니다.

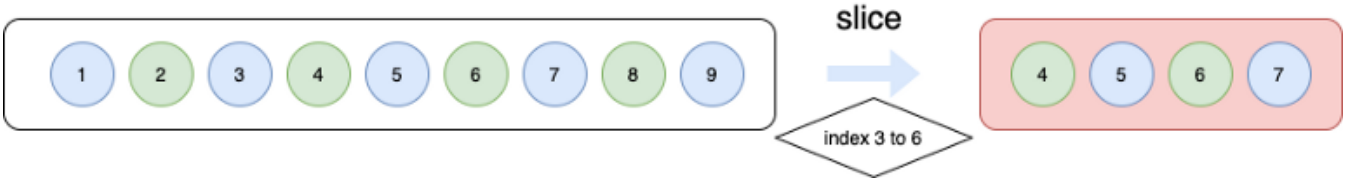
```
// 원소 변경
set, setValue // (자세한 내용은 여기에서 stackoverflow)

// 원소 추가
plus, plusElement, //(자세한 내용은 여기에서 stackoverflow)
plusAssign, //(자세한 내용은 여기에서 stackoverflow)
add, addAll, put, putAll

// 원소 삭제
minus, minusElement, //(자세한 내용은 여기에서 stackoverflow)
minusAssign, //(자세한 내용은 여기에서 stackoverflow)
remove

// 컬렉션의 앞 또는 뒤에서 제거
drop, dropLast, dropLastWhile, dropWhile,
removeFirst, removeFirstOrNull, removeLast, removeLastOrNull,
```



// 컬렉션의 앞쪽 또는 뒤쪽에서 가져오기
take, **takeLastWhile**, **takeLast**, **takeWhile**,

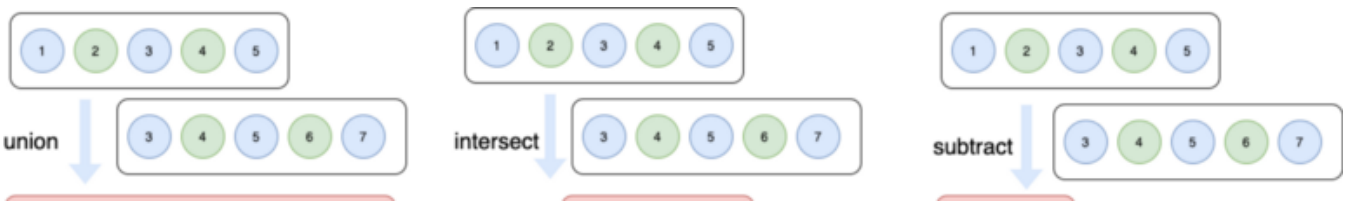


// 컬렉션의 일정 범위에서 가져오기
slice, **sliceArray**



// 중복 제거
distinct, **distinctBy**

// 두 개의 컬렉션에서 지정된 벤 다이어그램의 작업 수행두 개의 컬렉션
union, **intersect** ***retainAll***, **subtract** ***removeAll***



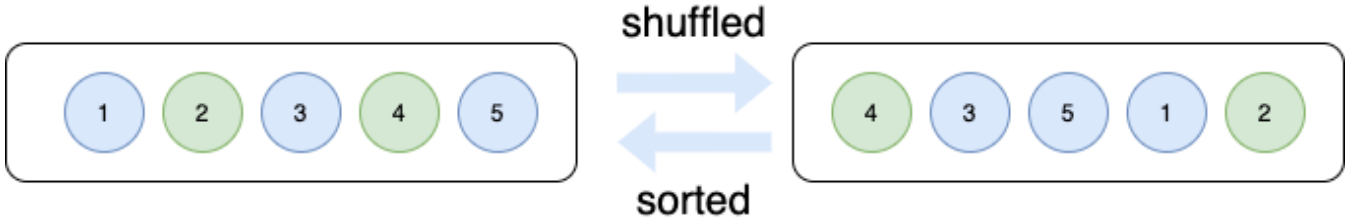
```
// 원소를 다른 값으로 변환
map, mapTo, mapIndexed, mapIndexedTo, mapKeys, mapKeysTo, mapValues,
mapValuesTo, replaceAll, fill

// 원소를 변환하며 모든 null값을 제거
// non-nullable한 결과를 가집니다.
mapNotNull, mapNotNullTo, mapIndexedNotNull, mapIndexedNotNullTo
```

> 참고: *Map* 함수는 리스트를 다른 폼 또는 다른 타입으로 변경할 수도 있습니다.

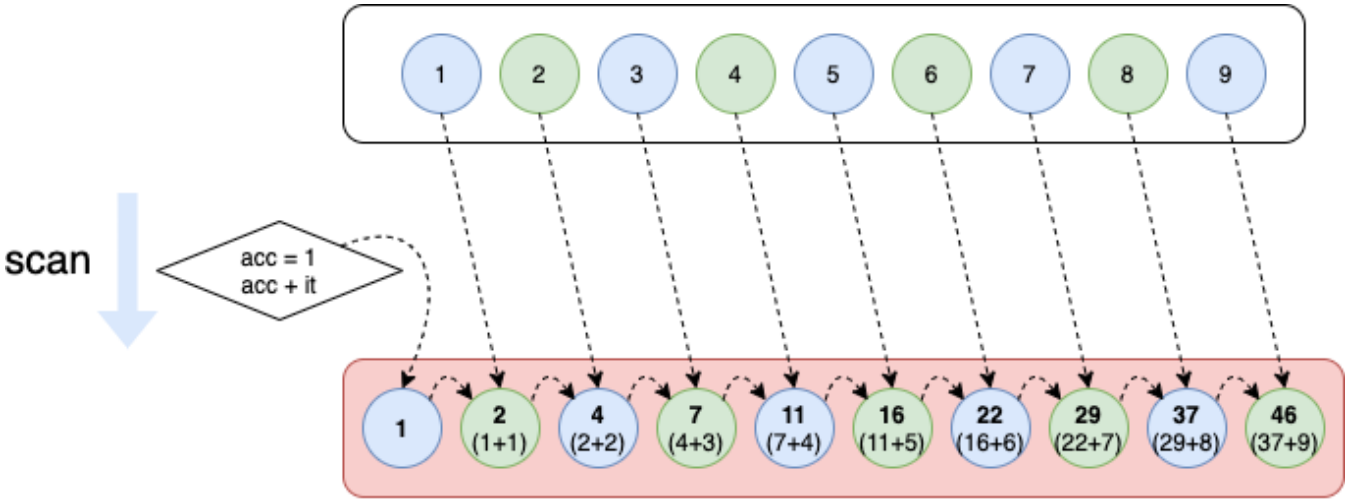
```
// 일부 원소를 필터링
filter, filterIndexed, filterIndexedTo, filterIsInstance,
filterIsInstanceTo, filterKeys, filterNot, filterNotNull,
filterNotNullTo, filterNotTo, filterTo, filterValues
```


```
// 원소 값을 뒤집기
// 여기서 이들의 차이점을 확인할 수 있습니다.
reversed, reversedArray, reverse, asReversed
```


```
// 원소를 정렬
sorted, sortedArray, sortedArrayDescending, sortedArrayWith,
sortedBy, sortedByDescending, sortedDescending, sortedWith sort,
sortBy, sortByDescending, sortDescending, sortWith

// 원소의 순서를 랜덤화
shuffle, shuffled
```




```
// fold나 reduce 와 비슷하지만, 이 함수들은 각 원소에서 순차적으로 수행됩니다.
scan, scanIndexed, scanReduce, scanReduceIndexed
```

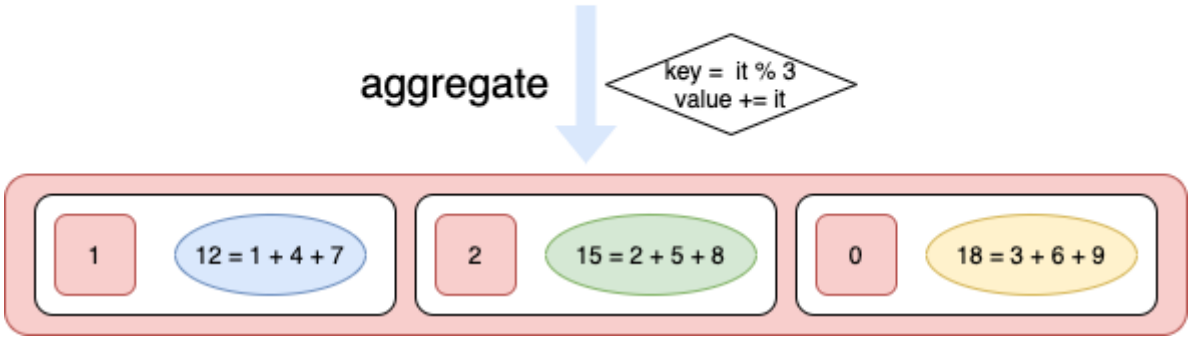


2. Change Contour

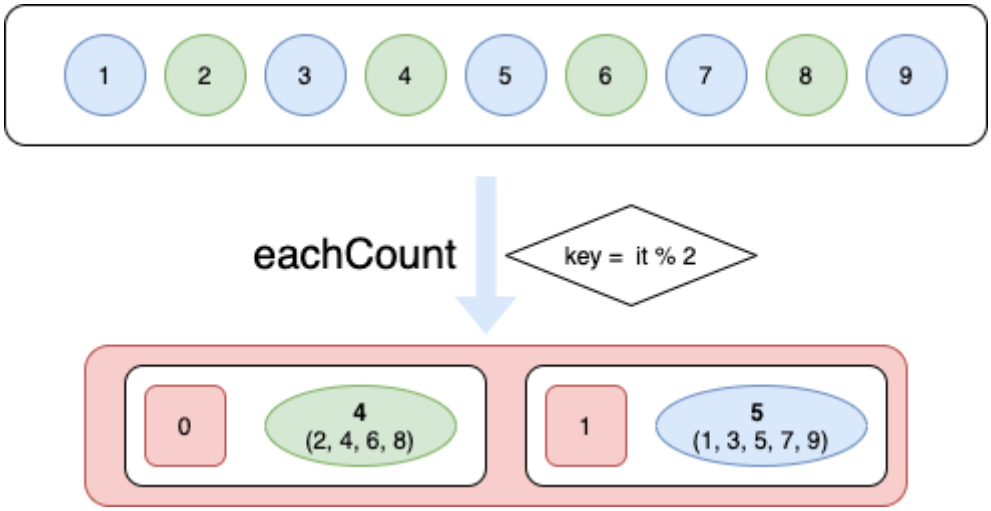
이 카테고리에서 함수의 결과는 구조를 변경합니다. (ex : List에서 Map으로 변경이나 원소의 구조 변경 (List<String>에서 List<Map>))

이제 함수를 살펴봅시다.

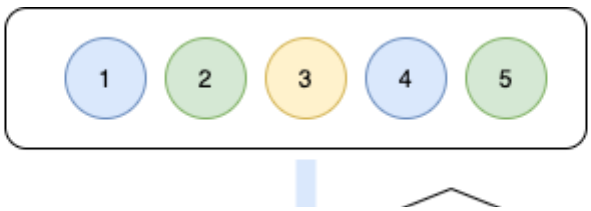
```
// 집계된 원소를 map 형태로 변환
aggregate, aggregateTo //(groupingBy를 필요로 함)

// 예시
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)
val aggregated = numbers.groupingBy { it % 3 }
    .aggregate { key, accumulator: Int?, element, first ->
        if (first) element else accumulator?.plus(element)
    }
println(aggregated) // {1=12, 2=15, 0=18}
```

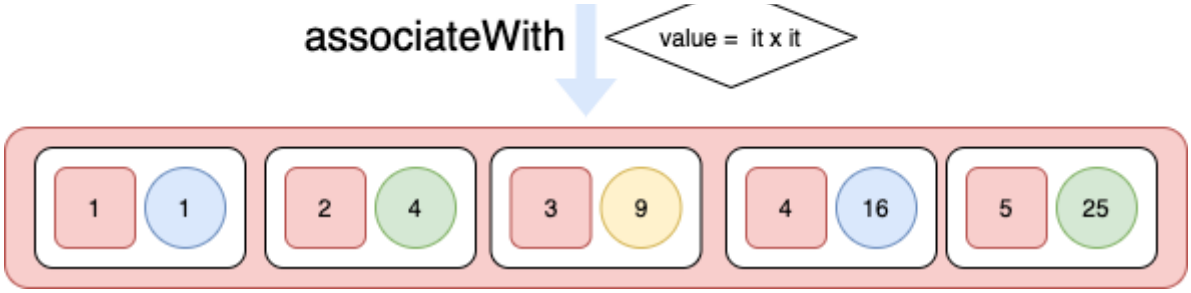



```
// 집계된 원소의 수를 map 형태로 변환
eachCount, eachCountTo //(groupingBy를 필요로 함)

// 예시
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)
val eachCount = numbers.groupingBy { it % 2 }.eachCount()
println(eachCount) // {1=5, 0=4}
```

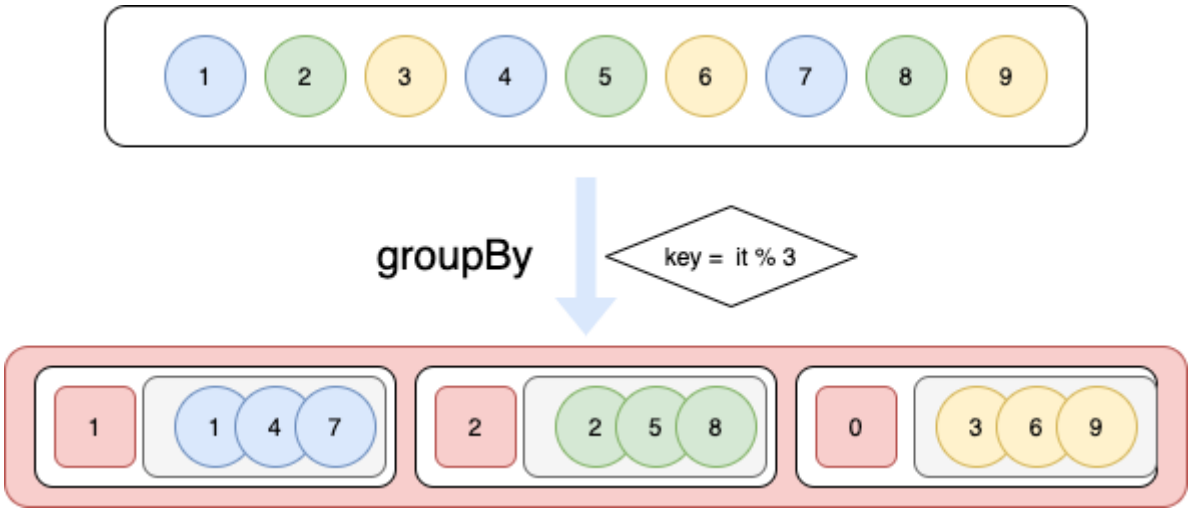



```
// 각 원소를 map의 key에 연결
associate, associateBy, associateByTo, associateTo, associateWith,
associateWithTo //(여기에서 차이점을 확인할 수 있습니다)

// 예시
val list = listOf(1, 2, 3, 4, 5)
val associate = list.associateWith { it * it }
println(associate) // {1=1, 2=4, 3=9, 4=16, 5=25}
```

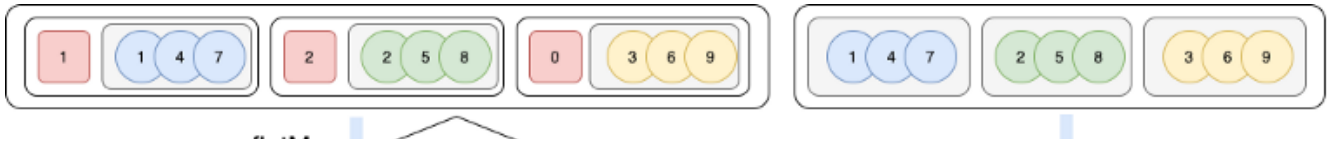


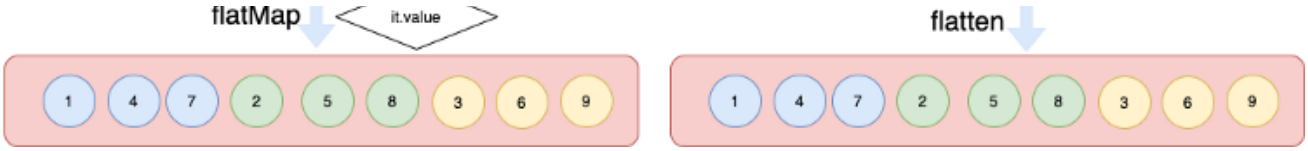



```
// 관련된 원소를 리스트로 묶어서 map으로 변환
groupBy, groupByTo

// 예시
val list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)
val groupBy = list.groupBy{it % 3}
println(groupBy) // { 1=[1,4,7], 2=[2,5,8], 0=[3,6,9]}
```

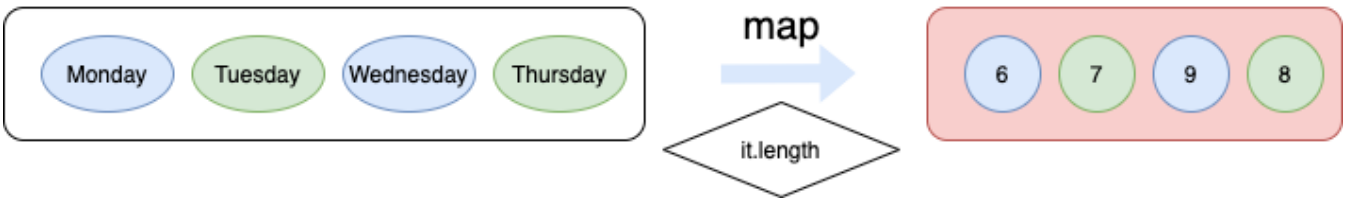



```
// 단일 리스트로 평탄하게 변환
flatMap, flatMapTo, flatten

// 예시
val map = mapOf(1 to listOf(1, 2), 2 to listOf(2, 4))
val flatMap = map.flatMap { it.value }
println(flatMap)  // [1, 2, 2, 4]

// 예시
val list = listOf(listOf(1, 2), listOf(2, 4))
val flatten = list.flatten()
println(flatten)  // [1, 2, 2, 4]
```

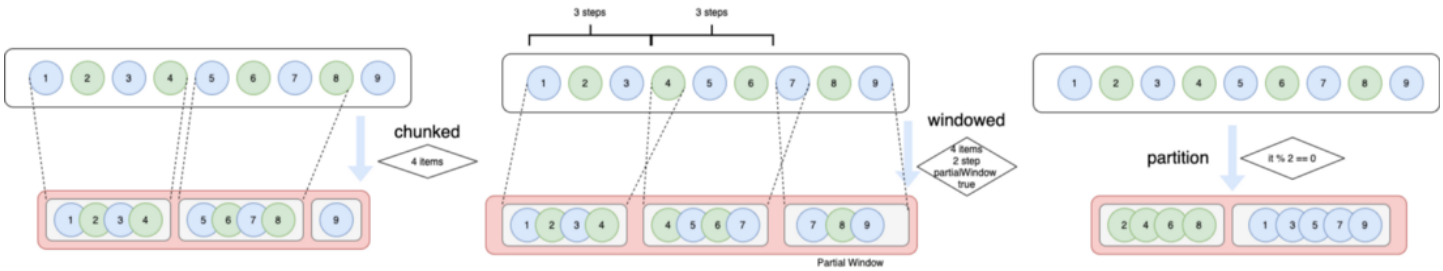





```
// 원소를 다른 타입 값으로 변환
map, mapTo, mapIndexed, mapIndexedTo, mapKeys, mapKeysTo, mapValues,
mapValuesTo

// 원소를 변환하면서 null인 값 제거
// non-nullable한 결과값 반환
mapNotNull, mapNotNullTo, mapIndexedNotNull, mapIndexedNotNullTo

// 예시
val days = listOf("Monday", "Tuesday", "Wednesday", "Thursday")
val daysLength = days.map { it.length }
println(daysLength) // [6, 7, 9, 8]
```



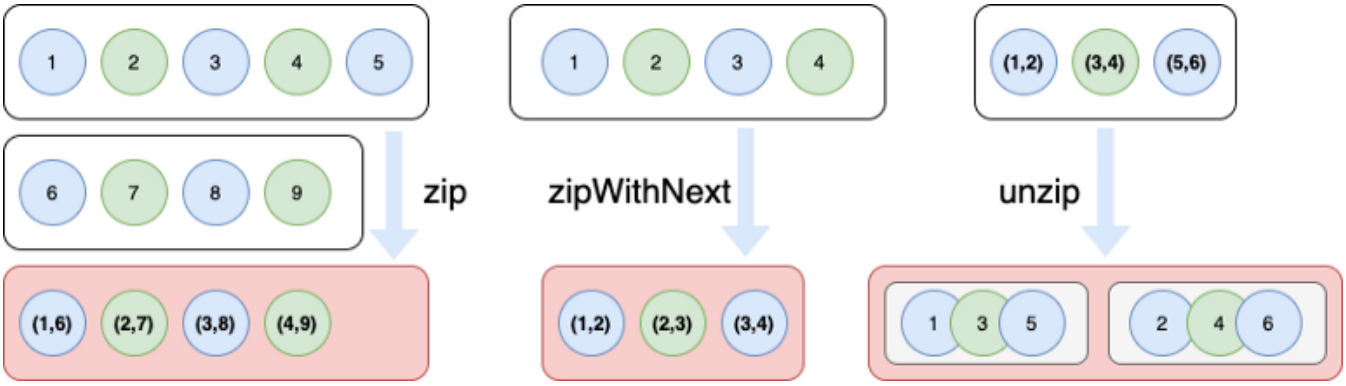

```
// 카테고리별로 항목을 묶어서 다른 리스트로 변환
chunked, partition, windowed

// 예시
val list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)
val chunked = list.chunked(4)
val windowed = list.windowed(4, 3, true)
val partition = list.partition { it % 2 == 0 }
println(chunked)  // [[1, 2, 3, 4], [5, 6, 7, 8], [9]]
println(windowed) // [[1, 2, 3, 4], [4, 5, 6, 7], [7, 8, 9]]
println(partition)// ([2, 4, 6, 8], [1, 3, 5, 7, 9])
```




```
// 두 개의 항목를 결합하거나 결합 해제
```

zip, **zipWithNext**, **unzip**

```
// 예시
val list = listOf(1, 2, 3, 4, 5)
val list2 = listOf(6, 7, 8, 9)
val zip = list.zip(list2)
println(zip) // [(1, 6), (2, 7), (4, 8)]

// 예시
val list = listOf(1, 2, 3, 4)
val zipWithNext = list.zipWithNext()
println(zipWithNext) // [(1, 2), (2, 3), (3, 4)]

// 예시
val list = listOf(1 to 2, 3 to 4, 5 to 6)
val unzip = list.unzip()
println(unzip) // [1, 3, 5], [2, 4, 6]
```



Choose Category

이 카테고리는 함수가 컬렉션의 특정 원소에 접근하는 함수들로 이루어져 있습니다. 마찬가지로 하위 카테고리로 나누었습니다.

1. Choose **Certain** —요소를 구체적으로 바로 검색할 수 있는 함수
2. Choose **Clue** — 단서가 주어졌을 때 요소를 검색하는 함수

결과값은 컬렉션 내의 요소 중 하나일 것입니다.

1. Choose Certain

이 함수들 중 몇몇은 매우 비슷하게 보일 수 있습니다. 아래 블로그에서 차이점을 확인할 수 있습니다.

> Kotlin has 4 ways to access the collection element!
>
> You have [ ], get, elementAt, and componentN. Which one to use?

levelup.gitconnected.com

```
// 대부분의 Map 및 List용
get, getOrDefault, getOrElse, getOrNull, getOrPut,
getValue // (여기에서 자세히 알아볼 수 있습니다 - stackoverflow. 그리고
withDefault를 다시 확인해 보세요)

// sequence와 set에서 주로 사용
elementAt, elementAtOrElse, elementAtOrNull

// 구조 분해 할당을 위해 사용
component1, component2, component3, component4, component5

// 랜덤하게 아무 원소나 얻어오기
random, randomOrNull

// 수동적으로 반복이 필요한 경우
iterator

// 단 하나의 요소만 가져오기
single, singleOrNull
```

2. Choose Clue

우리는 우리가 찾고자 하는 것에 대한 함수를 직접 제공해주고, 그 함수는 그것과 일치하는 요소를 검색합니다.

```
// 첫 번째 요소부터 검색하기
find, first, firstOrNull

// 뒤에서 첫 번째 요소부터 검색하기
findLast, last, lastOrNull

// 인덱스 검색하기
indexOf, lastIndexOf, indexOfFirst, indexOfLast

// 정렬된 컬렉션에서 검색
binarySearch, binarySearchBy
```

Conclude Category

이 카테고리는 모든 관련된 요소를 색출하여 어떤 결과를 도출하는 기능입니다. 마찬가지로 하위 카테고리로 나누었습니다.

1. Conclude **Choice** —함수가 Boolean값을 발생시키는 경우 (ex : `isEmpty`)

2. Conclude **Compute** — 모든 요소의 결과를 계산하는 함수(ex: `average`)

3. Conclude **Combine** — 요소들을 하나로 결합하는 함수 (ex: string, hash code)

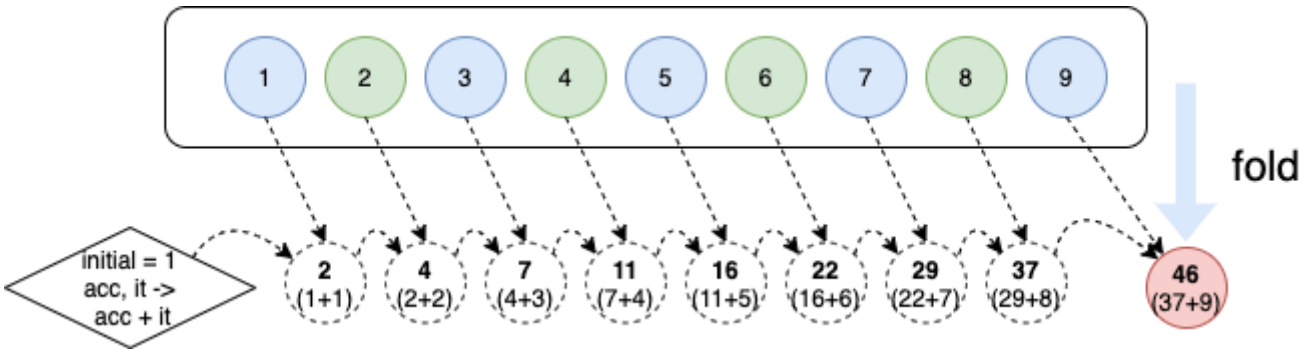
4. Conclude **Carryover** —모든 요소를 반복할 수 있으며, 각 요소에 대해 원하는 작업을 수행하는 함수(ex : `forEach`)

1. Conclude Choice

```
// 존재 여부 확인
all, any, none
contains, containsAll, containsKey, containsValue
isEmpty, isNotEmpty, isNullOrEmpty (여기를 확인해보세요.)

// 비교
contentEquals, contentDeepEquals
```

2. Conclude Compute

```
// 통계 관련
average, count, max, maxBy, maxWith, min, minBy, minWith
sum, sumBy, sumByDouble (double float 타입)

// 연쇄적인 연산 (scan과 비슷함)
fold, foldIndexed, foldRight, foldRightIndexed, foldTo,
reduce, reduceIndexed, reduceOrNull, reduceRight,
reduceRightIndexed, reduceRightOrNull, reduceTo
```



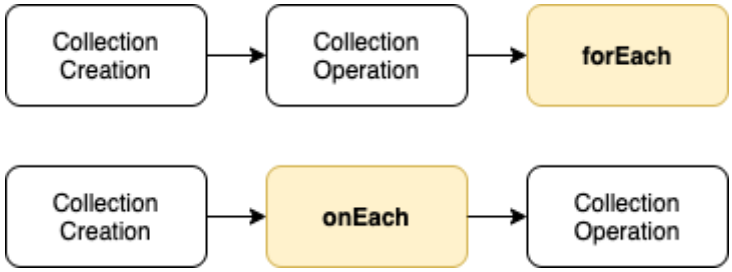
3. Conclude Combine

```
// 해시 코드 생성
contentHashCode, contentDeepHashCode
```

```
// 문자열 생성
contentToString, contentDeepToString,
joinTo, joinToString, subarrayContentToString
```

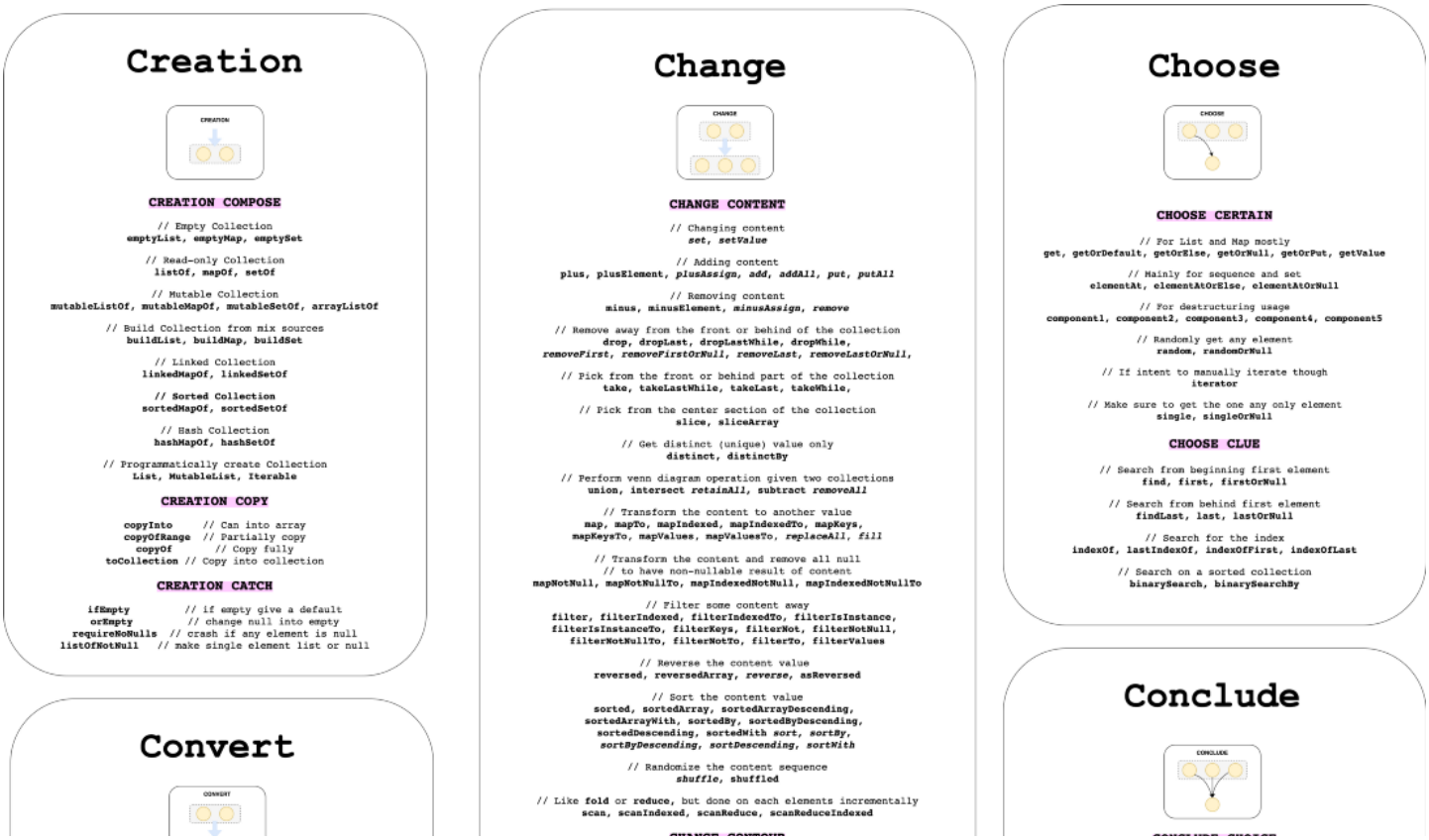
3. Conclude Carryover

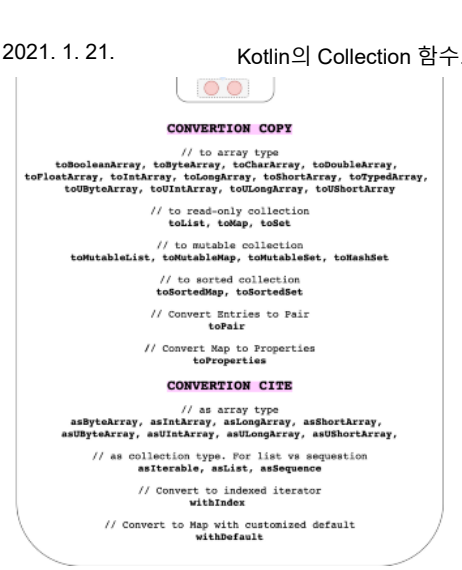
```
// 마지막 루프를 통과
forEach, forEachIndexed

// 이것은 중간 루프를 통과합니다.
// 그리고 컬렉션 자체를 반환합니다.
onEach
```



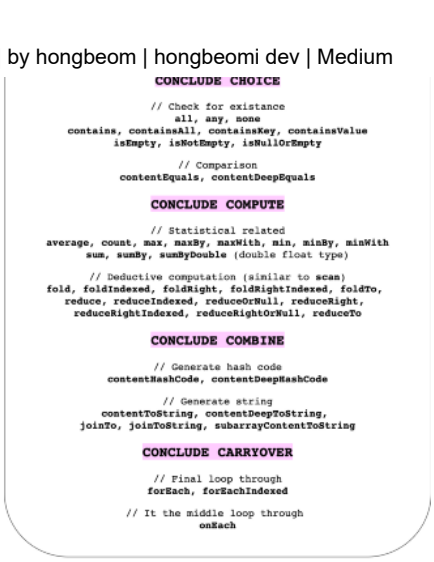
한 장으로 정리한 문서

위의 모든 함수가 설명되어 있는 한 장으로 정리된 문서입니다. 인쇄하여 벽에 붙일 수 있습니다(?)👀





https://medium.com/@elye.project



읽어주셔서 감사합니다👏

Kotlin Kotlin Collection Collection Kotlin Functions