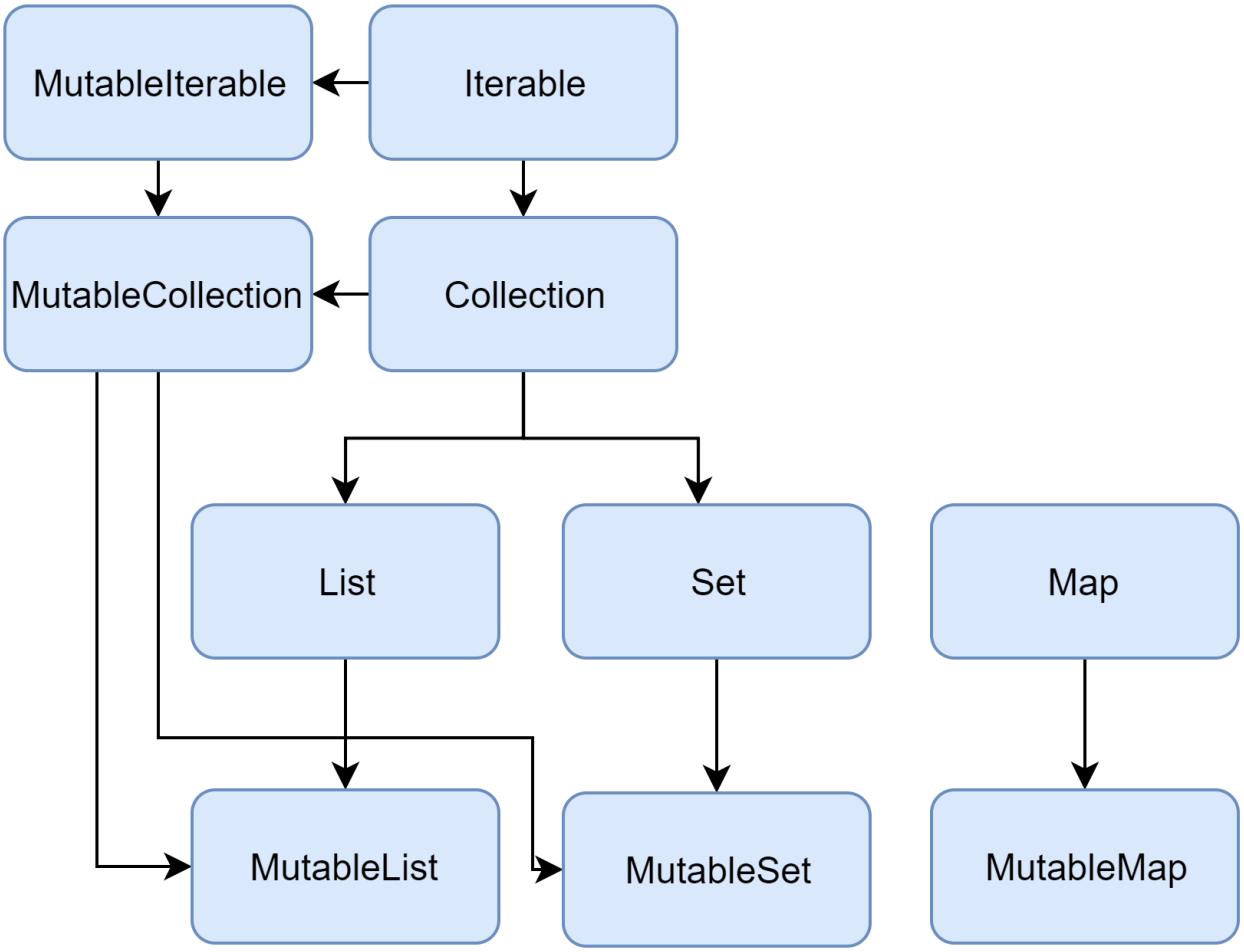
Chapter 02.코틀린 고급 - 01.컬렉션

컬렉션

1. 컬렉션 타입

- 코틀린 표준 라이브러리는 기본 컬렉션 타입인 `List`, `Set`, `Map` 을 제공한다
- 컬렉션은 두가지 종류로 나뉜다
  - 불변 컬렉션(Immutable) : 읽기 전용 컬렉션
  - 가변 컬렉션(Mutable) : 삽입, 수정, 삭제와 같은 쓰기 작업이 가능한 컬렉션

- 컬렉션 계층 다이어그램



2. 컬렉션 생성 방법

- 컬렉션을 생성할때 가장 일반적인 방법은 표준 라이브러리 함수를 사용하는 것이다
- Immutable 리스트 생성

```
val currencyList: List<String> = listOf("달러", "유로", "원")
```

- Mutable 리스트 생성

```
val mutableCurrencyList = mutableListOf<String>()
mutableCurrencyList.add("달러")
```

```
mutableCurrencyList.add("유로")
mutableCurrencyList.add("원")
```

- `apply` 함수를 사용하면 가독성이 좋아진다

```
val mutableCurrencyList = mutableListOf<String>().apply {
    add("달러")
    add("유로")
    add("원")
}
```

- Immutable 세트 생성

```
val numberSet = setOf(1, 2, 3, 4)
```

- Mutable 세트 생성

```
val mutableSet = mutableSetOf<Int>()
```

- Immutable 맵 생성
- `to` 라는 중위 함수로 키-밸류 구조를 전달한다

```
val numberMap = mapOf("one" to 1, "two" to 2)
```

- Mutable 맵 생성

```
val mutableNumberMap = mutableMapOf<String, Int>()
mutableNumberMap["one"] = 1
mutableNumberMap["two"] = 2
mutableNumberMap["three"] = 3
```

- `컬렉션 빌더` 를 사용하여 컬렉션을 생성할 수 있다
- buildList, buildSet, buildMap 3종류를 제공
- build 내부에선 Mutable 즉 `가변`이고 반환시엔 Immutable `불변`이다

```
val numberList: List<Int> = buildList {
    // buildMap 내부엔 MutableMap
    add(1)
    add(2)
    add(3)
 }
```

- 특정 구현체를 사용하고 싶은 경우 구현체의 생성자를 사용한다

```
val linkedList = LinkedList<Int>().apply {
        add(1)
        add(2)
        add(3)
    }
```

---

3. 컬렉션 반복하기

- 코틀린의 컬렉션은 `Iterable` 의 구현체이므로 순차적 반복이 가능하다

```
val iterator = currencyList.iterator()
while (iterator.hasNext()) {
  println(iterator.next())
}
```

- 자바에선 `foreach` 를 사용하면 Iterable을 구현한 컬렉션을 반복할 수 있다
- 코틀린은 앞서 학습한 `for loop` 를 사용하면 암시적으로 이터레이터를 사용하기 때문에 좀 더 쉽게 반복할 수 있다

```
for (currency in currencyList) {
  println(currency)
}
```

- 또한 코틀린 표준 라이브러리에는 컬렉션 사용시 자주 사용되는 패턴인 `forEach`, `map` , `filter` 와 같은 유용한 `인라인 함수`를 제공한다

```
currency.forEach {
  println(it)
}
```

- for loop를 map으로 변환하기 전

```
val lowerList = listOf("a", "b", "c", "d")
val upperList = mutableListOf<String>()

for (lowerCase in lowerList) {
  upperList.add(lowerCase.uppercase())
}

println(upperList)
// // [A, B, C, D]
```

- foor loop를 map으로 변환 후

```
val upperList = lowerList.map { it.uppercase() }
println(upperList)
// [A, B, C, D]
```

- for loop를 filter로 변환하기 전

```
val filteredList = mutableListOf<String>()
for (upperCase in upperList) {
  if (upperCase == "A" || upperCase == "C") {
    filteredList.add(upperCase)
  }
}
println(filteredList)
// [A, C]
```

- foor loop를 filter로 변환 후

```
val filteredList = upperList.filter { it == "A" || it == "C" }
println(filteredList)
// [A, C]
```

- 자바8의 스트림과 비교

```
val filteredList = upperList.stream().filter { it == "A" || it == "C" }

println(filteredList)
// java.util.stream.ReferencePipeline$2@77b52d12
```

- 자바8의 스트림은 중간 연산자(map, filter, flatMap 등)만 사용했을땐 아무런 동작도 하지 않는다
- 값을 얻고 싶으면 최종 연산자(terminal operator)를 사용해야한다

```
val filteredList = upperList.stream()
        .filter { it == "A" || it == "C" }
        .collect(Collectors.toList())

println(filteredList)
// [A, C]
```

- 코틀린에서는 `sequence` 를 사용해 자바8 스트림과 같이 Lazy하게 동작시킬 수 있다

```
val filteredList = upperList
        .asSequence()
        .filter { it == "A" || it == "C" }


println(filteredList)
// kotlin.sequences.FilteringSequence@7f560810
```

- 시퀀스 API도 최종 연산자를 사용해야 중간 연산자가 동작한다

```
val filteredList = upperList
        .asSequence()
        .filter { it == "A" || it == "C" }
        .toList()

println(filteredList)
// [A, C]
```

- 일반적으로 인라인 함수는 각각 함수가 `동작할때마다` 조건에 맞는 컬렉션을 생성한다
- 시퀀스 API는 각각의 함수가 동작할때 시퀀스를 생성하고 최종 연산자를 호출할때 `1` 개의 컬렉션을 생성한다
- 벤치마크상으로 일반적으론 인라인 함수가 빠르기 때문에 인라인 함수를 쓰고 대량의 데이터를 다룰때는 시퀀스 API를 사용하길 추천
- 람다식과 함수형프로그래밍에 대한 부분은 이후에 학습할 `람다로 프로그래밍하기` 챕터에서 다룰 예정이므로 이런게 있다 정도로 넘어가겠습니다