# 7장 컬렉션과 I/O 자세히 알아보기

이번 장에서는 코틀린 표준 라이브러리를 구성하는 중요한 두가지 요소에 대해 살펴본다. 첫번째 부분은 컬렉션 API이다. 이 부분에서는 일반적인 컬렉션 타입과 기본 연산에 대해 살펴보고 컬렉션이나 컬렉션에 들어있는 원소를 조작하는 다양한 방법, 즉 원소 접근, 컬렉션에 대한 술어 검사, 걸러내기, 컬렉션의 일부를 추출하는 방법, 컬렉션을 집계해 다른 값을 만들어내는 방법, 변환, 순서 지정 등에 대해 전체적으로 살펴본다. 두번째 부분에서는 I/O API에 초점을 맞춰서 I/O 스트림을 생성하고 I/O 스트림의 데이터에 접근하는 방법을 단순화해주는 유틸리티아 공통적인 파일시스템 연산에 대해 이야기한다.

## 구조

* 컬렉션
* 파일과 I/O 스트림

## 목표

코틀린 컬렉션 타입을 이해하고 코틀린 표준 라이브러리를 사용해 컬렉션 데이터를 간결하고 코틀린답게 조작하는 방법을 배우며, I/O 스트림 API 확장에 대해 이해한다.

## 컬렉션

컬렉션은 엘리먼트들로 이뤄진 그룹을 저장하기 위해 설계된 객체다. *2장. 코틀린 언어 기초*에서 이런 객체를 이미 살펴봤다. 같은 타입에 속하는 개수가 정해진 여러 원소를 함께 저장하는 배열이 바로 이런 컬렉션 객체였다. 하지만 코틀린 표준 라이브러리는 훨씬 더 풍부한 컬렉션 기능을 제공한다. 이런 기능에는 여러가지 데이터 구조(배열, 연결 리스트, 해시 테이블 등)에 기반한 다양한 컬렉션 라이브러리와 컬렉션과 컬렉션 안에 저장된 데이터를 조작하기 위한 종합적인 API가 포함된다. 걸러내기, 집계하기, 변환, 순서지정 등의 데이터 조작이 가능하다. 이번 절에서는 컬렉션 라이브러리가 코틀린 개발자들에게 제공하는 기능에 대해 자세히 다룬다.

컬렉션을 조작하는 모든 연산이 인라인 함수라는 점을 언급해둘만한 가치가 있다. 따라서 이런 연산을 사용해도 함수 호출이나 람다 호출에 따른 부가비용이 들지 않는다.

### 컬렉션 타입들

코틀린 컬렉션 타입은 기본적으로 4가지 분류로 나눌 수 있다. 배열, 이터러블(iterable), 시퀀스(sequence), 맵(map)이 이 4가지 분류다. *2장. 코틀린 언어 기초*의 주요 주제가 배열이었다. 따라서 이번 장에서는 나머지 세가지 분류에 대해 주로 초점을 맞춘다.

배열과 비슷하게 컬렉션 타입도 제네릭 타입이다. 컬렉션의 타입을 구체적으로 지정하려면 원소의 타입을 지정해야만 한다. 예를 들어 List<String>는 문자열로 이뤄진 리스트를 뜻하고 Set<Int>는 Int 타입의 값들로 이뤄진 집합을 뜻한다.

기본적인 컬렉션 타입의 개요는 다음 그림과 같다(그림 7.1).

##### 그림 7.1: 코틀린의 컬렉션 타입들

[편집자께: 그림의 모든 내용을 다 코드체로 써주세요. 따로 번역할 내용은 없습니다. 다 코틀린 클래스나 인터페이스, 메서드 이름입니다.]

### 이터러블

이터러블(iterable)은 Iterable<T> 타입으로 표현되며, 일반적으로 즉시 계산(eager)되는 상태가 있는(stateful) 컬렉션을 표현한다. 상태가 있다는 말은 컬렉션이 원소를 필요할 때 생성하는 제네레이터 함수를 유지하지 않고 원소를 저장한다는 뜻이다. 한편 즉시계산이라는 말은 나중에 어느 필요한 시점에 원소가 초기화되지 않고 컬렉션을 최초로 생성할 때 초기화 된다는 뜻이다.

이터러블 타입 자체는 자바의 이터러블 타입과 상당히 비슷하다. 이터러블 타입은 원소를 순회(traverse)할 수 있는 iterator()라는 메서드를 제공한다. 코틀린 for 루프에서는 이 메서드를 통해 모든 이터러블 객체를 활용할 수 있다.

val list = listOf(“red”, “green”, “blue”) // Create new list  
  
for (item in list) {  
 print(item + “ ”)  
} // red green blue 출력함

**자바와 코틀린의 차이**: 코틀린 Iterator 타입은 기본적으로 자바와 동일하다. 내부에는 이터레이터가 컬렉션의 끝에 도달했는지를 판단하는 hasNext()와 컬렉션의 다음 원소를 반환하는 next() 메서드가 들어있다. 유일한 차이는 remove()가 없다는 점인데, 이 메서드는 MutableIterator로 옮겨졌다.

자바와 비교할 때 코틀린 이터러블의 중요한 특징은 불변 컬렉션과 가변 컬렉션을 구분한다는 점이다. 불변 컬렉션은 생성한 다음에 내용을 바꿀 수 없지만 가변 컬렉션은 어느때나 원소를 추가하거나 삭제할 수 있다. 컬렉션의 변경 가능성은 컬렉션 인스턴스에 대한 참조를 저장하는 변수의 변경 가능성과는 아무 관계가 없다는 점에 유의하라. 가변 변수가 뜻하는 것은 변수가 가리키는 참조를 다른 참조로 바굴 수 있다는 것 뿐이다. 예를 들어 가변 컬렉션을 불변 변수에 저장할 수 있다. 이런 경우 이 변수가 다른 컬렉션을 가리키게 변경할 수는 없지만, 변수가 가리키는 컬렉션에 원소를 추가하거나 삭제할 수는 있다.

val list = ArrayList<String>()  
list.add(“abc”) // Ok: changing collection data  
list = ArrayList<String>() // Error: can’t reassign immutable variable

가변 이터러블의 기본 타입은 MutableIterable으로 표현된다. 이 인터페이스의 객체는 MutableIterator 인스턴스를 생성할 수 있다.

불변 컬렉션 타입에서 유용한 특징으로는 공변성(covariance)이 있다. 공변성이라는 말은 T가 U의 하위타입인 경우 Iterable<T>도 Iterable<U>의 하위타입이라는 뜻이다. Iterator, Collection, List, Set, Map 등의 컬렉션 관련 타입의 경우에도 이런 공변성이 성립한다. 공변성으로 인해 다음과 같은 코드를 작성할 수 있다.

fun processCollection(c: Iterable<Any>) {…}  
  
fun main() {  
 val list = listOf(“a”, “b”, “c”) // List<String>  
 processCollection(list) // Ok: passing List<String> as List<Any>  
}

하지만 가변 컬렉션의 경우 이런 코드가 작동하지 않는다. 만약 가변 컬렉션에 대해 이런 식의 코드를 작성할 수 있다면, 정수를 문자열 리스트에 추가하는 것과 같은 일이 벌어질 수 있다.

fun processCollection(c: MutableCollection<Any>) { c.add(123) }  
  
fun main() {  
 val list = arrayListOf(“a”, “b”, “c”) // ArrayList<String>  
 processCollection(list) // !!!  
}

*9장. 제네릭스*에서 공변성에 대해 더 자세히 다룬다.

### 컬렉션, 리스트, 집합

이터러블의 하위 분류 중에는 Collection 인터페이스로 표현되는 타입들과 Collection 타입의 하위 타입인 MutableCollection 인터페이스로 표현되는 타입들이 있다. 이들은 이터러블에 대한 여러가지 표준적인 구현을 위한 기본 클래스이다. Collection을 상속한 클래스는 크게 다음과 같이 분류할 수 있다.

* 리스트(List와 MutableList로 표현됨)는 인덱스를 통한 원소 접근이 가능한 순서가 정해진 원소의 컬렉션이다. 일반적인 리스트 구현은 인덱스를 통한 임의 접근이 가능한 ArrayList와 원소를 (끝 쪽에) 빠르게 추가하거나 삭제할 수 있지만 인덱스로 원소에 접근할 때는 선형 시간(선형 시간이라는 말은 O(n) 시간이 걸린다는 말이다)이 걸리는 LinkedList가 있다.
* 집합은 유일한 원소들로 이뤄진 컬렉션이다. 원소의 순서는 구현에 따라 다르다.
  + HashSet은 해시 테이블이 기반인 구현이며, 원소의 해시 코드에 따라 원소 순서가 정해진다. 일반적으로 이 순서는 원소 타입의 hashCode() 메서드 구현에 따라 달라지므로, 예측하기 어렵다.
  + LinkedHashSet은 해시 테이블이 기반이지만 삽입 순서를 유지하기 때문에 원소를 이터레이션하면 집합에 삽입된 순서대로 원소를 순회할 수 있다.
  + TreeSet은 이진 검색 트리(binary search tree)가 기반이며, 어떤 비교 규칙에 따라 일관성있는 원소 순서를 제공한다. (원소가 `인터페이스를 상속한 경우) 원소 타입이 이런 비교 규칙을 정의할 수 있고, 별도의Comparator` 객체를 통해 비교 규칙을 제공할 수도 있다.

JVM 플랫폼에서는 이런 코틀린 인터페이스를 구현하는 구체적인 클래스들이 JDK의 컬렉션 클래스에 해당한다. HashMap이나 ArrayList 같은 유명한 자바 클래스는 코틀린 라이브러리에 매끄럽게 통합되어 있다.

**자바와 코틀린의 차이**: 코틀린 코드에서는 java.util 패키지의 클래스를 사용할 필요가 없다. kotlin.collections 패키지에 있는 별명을 통해 ArrayList 등의 대부분의 표준 컬렉션을 쓸 수 있다. 코틀린 파일에서는 자동으로 kotlin.collections 패키지가 임포트된다.

### 시퀀스

이터러블과 비슷하게 시퀀스도 iterator() 메서드를 제공한다. 이 메서드를 통해 시퀀스의 내용을 순회할 수 있다. 하지만 시퀀스는 지연 계산을 가정하기 때문에, iterator()의 의도가 이터러블과 다르다. 대부분의 시퀀스 구현은 객체 초기화시 원소를 초기화하지 않고, 요청에 따라 원소를 계산한다. 대부분의 시퀀스 구현은 상태가 없다. 이 말은 지연 계산한 컬렉션 원소 중에 정해진 개수의 원소만 저장한다는 뜻이다. 반면 이터러블은 원소 개수에 비례해 메모리를 사용한다.

이터러블과 달리 대부분의 시퀀스 구현은 내부적이기 때문에 외부에서 직접 사용할 수가 없다. 대신에 특별한 함수를 통해 시퀀스를 만들어야 한다. 이에 대해서는 이번 장 뒤의 여러 절에서 다룬다.

**자바와 코틀린의 차이**: 자바에 익숙한 독자라면 자바 8에 도입된 스트림(stream)과 시퀀스 사이의 유사성을 인식했을 것이다. 코틀린 1.2부터 표준 라이브러리는 asSeqence()를 확장 함수로 제공하며, 이 함수는 자바 스트림을 감싸서 코틀린 시퀀스로 사용하게 해준다.

### 맵

맵은 키(key)와 값(value) 쌍으로 이뤄진 집합이다. 여기서 키는 유일해야 한다. 맵 자체는 Collection의 하위 타입이 아니지만 맵에 들어있는 원소들을 컬렉션처럼 사용할 수 있다. 구체적인 예를 들면, 맵으로부터 모든 키로 이뤄진 집합을 얻을 수 있고 모든 키-값 쌍으로 이뤄진 집합이나 모든 값으로 이뤄진 컬렉션을 얻을 수도 있다. 키-값 쌍은 Map.Entry와 MutableMap.MutableEntry 인터페이스로 표현된다.

맵에는 두가지 종류의 원소(키,값)가 들어있기 때문에 맵의 타입은 두가지 타입을 파라미터로 받는 제네릭 타입이다. 예를 들어 Map<Int, String>은 Int 키를 String 값에 연관시켜주는 맵이다.

맵의 표준 구현에는 HashMap, LinkedHashMap, TreeMap 등이 있다. 이들의 성질은 각각에 대응하는 Set 클래스의 성질과 비슷하다.

여러분이 직접 맵을 구현하고 싶다면 AbstractMap와 AbstractMutableMap 클래스를 기반 클래스로 사용할 수 있다.

### Comparable과 Comparator

자바처럼 코틀린도 Comparable과 Comparator 타입을 지원하며, 몇몇 컬렉션 연산에 이들을 활용한다. 비교가능(comparable) 인스턴스는 자연적인 순서(natural order)를 지원하며, 이런 타입의 인스턴스들은 모두 동일한 타입의 다른 인스턴스와 순서를 비교할 때 쓸 수 있는 compareTo() 메서드를 포함한다. 여러분의 타입이 Comparable을 상속하면 자동으로 <, > 등의 연산을 쓸 수 있고, 원소 타입이 비교가능한 타입인 컬렉션의 순서 연산에도 이런 연산이 쓰인다. Person 클래스에 전체 이름을 바탕으로 자연스러운 순서를 부여하고 싶다고 하자. 이런 경우 구현은 다음과 같다.

class Person(  
 val firstName: String,  
 val familyName: String,  
 val age: Int  
) : Comparable<Person> {  
 val fullNameget() = "$firstName $familyName"  
 override fun compareTo(p: Person) = fullName.compareTo(p.fullName)  
}

compareTo() 함수의 관례는 자바와 똑같다. 즉 현재 (수신객체) 인스턴스가 (인자로 받은) 상대방 인스턴스보다 더 크면 양수, 더 작으면 음수, 같으면 영(0)을 반환한다. compareTo() 구현은 equals() 함수 구현과 서로 의미가 일치해야 한다.

어떤 클래스를 여러가지 방법으로 비교해야 하는 경우도 자주 있다. 예를 들어 Person의 컬렉션을 이름(family name)이나 나이, 또는 여러가지 프로퍼티의 조합을 사용해 정렬할 수도 있다. 이런 경우에 대비해 코틀린 라이브러리는 비교기(comparator)라는 개념을 제공한다. 자바와 마찬가지로 Comparator<T> 클래스는 타입 T의 인스턴스 객체를 2개 인자로 받는 compare() 함수를 제공한다. 이 함수는 compareTo()와 마찬가지 관례를 따라 인자로 받은 두 인스턴스를 비교한 결과를 반환한다. 코틀린에서는 람다 비교 함수를 통해 비교기를 간결하게 작성할 수 있다.

val AGE\_COMPARATOR = Comparator<Person>{ p1, p2 ->  
 p1.age.compareTo(p2.age)  
}

또는 compareBy()나 compareByDescending() 함수를 통해 대상 객체 대신 사용할 수 있는 비교가능 객체를 제공하게 해서 비교기를 만들 수도 있다[[1]](#footnote-32).

val AGE\_COMPARATOR = compareBy<Person>{ it.age }  
val REVERSE\_AGE\_COMPARATOR = compareByDescending<Person>{ it.age }

fun <T> compareBy(  
 vararg selectors: (T) -> Comparable<\*>?  
): Comparator<T>

비교기 인스턴스를 sorted()나 max()처럼 순서를 인지하는 함수에 넘길 수 있다. 집계 함수와 컬렉션 순서에 대해 다루는 절에서 이와 관련있는 예제를 볼 수 있다.

## 컬렉션 생성하기

*2장. 코틀린 언어 기초*에서 이미 배열 인스턴스를 생성자를 통해 생성하거나 arrayOf() 같은 표준 함수를 통해 생성할 수 있다는 사실을 배웠다. 여러 표준 컬렉션 클래스의 인스턴스를 비슷한 방식으로 생성할 수 있다. 예를 들어, 자바와 마찬가지로 ArrayList나 LinkedHashSet 같은 클래스의 인스턴스를 생성자를 호출해 생성할 수 있다.

val list = ArrayList<String>()  
list.add(“red”)  
list.add(“green”)  
println(list) // [red, green]  
val set = HashSet<Int>()  
set.add(12)  
set.add(21)  
set.add(12)  
println(set) // [12, 21]  
val map = TreeMap<Int, String>()  
map[20] = "Twenty"  
map[10] = "Ten"  
println(map) // {10=Ten, 20=Twenty}

그리고 arrayOf()와 비슷하게 가변길이 인자를 받는 함수를 사용해 몇몇 컬렉션 클래스의 인스턴스를 생성할 수 있다.

* emptyList()/emptySet() : 불변인 빈 리스트/집합 인스턴스를 생성한다(자바 컬렉션 클래스의 emptyXXX() 메서드와 비슷함).
* listOf()/setOf(): 인자로 제공한 배열(가변 길이 인자)에 기반한 불변 리스트/집합 인스턴스를 만든다(리스트의 경우 listOf()는 기본적으로 자바의 Arrays.asList()와 같다).
* listOfNotNull(): 널인 값을 걸러내고 남은 원소들로 이뤄진 새 불변 리스트를 만든다.
* mutableListOf()/mutableSetOf(): 가변 리스트/집합의 디폴트 구현 인스턴스를 만든다(내부적으로 ArrayList와 LinkedHashSet을 사용한다).
* arrayListOf(): 새로운 ArrayList를 만든다.
* hashSetOf()/linkedSetOf()/sortedSetOf(): HashSet/LinkedHashSet/TreeSet의 새 인스턴스를 만든다.

몇가지 예제를 살펴보자.

valemptyList = emptyList<String>()  
println(emptyList) // []  
emptyList.add("abc") // Error: add is unresolved  
  
valsingletonSet = setOf("abc")  
println(singletonSet) // [abc]  
singletonSet.remove("abc") // Error: remove is unresolved  
  
valmutableList = mutableListOf("abc")  
println(mutableList) // [abc]  
mutableList.add("def")  
mutableList[0] = "xyz"  
println(mutableList) // [xyz, def]  
  
valsortedSet = sortedSetOf(8, 5, 7, 1, 4)  
println(sortedSet) // [1, 4, 5, 7, 8]  
sortedSet.add(2)  
println(sortedSet) // [1, 2, 4, 5, 7, 8]

맵을 만들 때도 비슷한 함수를 쓸 수 있다.

* emptyMap(): 빈 불변 맵을 만든다.
* mapOf(): 새 불변 맵을 만든다(내부적으로 LinkedHashMap를 만든다).
* mutableMapOf(): 가변 맵의 디폴트 구현 인스턴스를 만든다(내부적으로 LinkedHashMap를 사용한다).
* hashMapOf()/linkedMapOf()/sortedMapOf(): HashMap/LinkedHashMap/TreeMap의 새 인스턴스를 만든다.

앞에서 보여준 맵 함수들은 Pair 객체들로 이뤄진 가변 인자를 받는다는 점에 유의하라. to 중위 연산자를 사용하면 Pair 객체를 쉽게 만들 수 있다.

val emptyMap = emptyMap<Int, String>()  
println(emptyMap) // {}  
emptyMap[10] = "Ten" // Error: set is unresolved  
  
val singletonMap = mapOf(10 to "Ten")  
println(singletonMap) // {10=Ten}  
singletonMap.remove("abc") // Error: remove is unresolved  
  
val mutableMap = mutableMapOf(10 to "Ten")  
println(mutableMap) // {10=Ten}  
mutableMap[20] = "Twenty"  
mutableMap[100] = "Hundred"  
mutableMap.remove(10)  
println(mutableMap) // {20=Twenty, 100=Hundred}  
  
val sortedMap = sortedMapOf(3 to "three", 1 to "one", 2 to "two")  
println(sortedMap) // {1=one, 2=two, 3=three}  
sortedMap[0] = "zero"  
println(sortedMap) // {0=zero, 1=one, 2=two, 3=three}

또는 Pair를 생성하고 싶지 않다면, 가변 맵을 만들고 set() 메서드나 인덱스 연산자([])를 사용해 맵에 원소를 추가할 수도 있다.

배열과 비슷하게 크기를 지정하고 인덱스로부터 값을 만들어주는 함수를 제공함으로써 새 리스트를 만들 수도 있다.

println(List(5) { it\*it }) // [0, 1, 4, 9, 16]  
  
val numbers = MutableList(5) { it\*2 }  
println(numbers) // [0, 2, 4, 6, 8]  
numbers.add(100)  
println(numbers) // [0, 2, 4, 6, 8, 100]

원소가 알려진 시퀀스를 만드는 가장 단순한 방법은 sequenceOf()함수를 사용하는 것이다. 이 함수는 가변 인자를 받는다. 또는 배열, 이터러블, 맵 등의 기존 컬렉션 객체에 대해 asSequence() 함수를 호출해서 시퀀스를 얻을 수도 있다.

println(sequenceOf(1, 2, 3).iterator().next()) // 1  
println(listOf(10, 20, 30).asSequence().iterator().next()) // 10  
println(  
 mapOf(1 to "One", 2 to "Two").asSequence().iterator().next()  
) // 1=One

맵에 대해 asSequence()를 호출하면 맵 엔트리(키,값 쌍을 감싼 타입)의 시퀀스를 얻는다는 점에 유의하라.

다른 방법으로는 제네레이터 함수를 바탕으로 시퀀스를 만드는 방법이 있다. 두가지 generateSequence() 함수를 사용하면 함수를 사용해 시퀀스를 만들 수 있다. 첫번째 generateSequence()는 시퀀스의 다음 원소를 생성해주는 파라미터가 없는 함수를 인자로 받는다. 이 함수가 널을 반환할 때까지 시퀀스 원소 생성이 계속된다. 예를 들어 다음 코드는 숫자가 아닌 입력을 받거나 입력이 끝날 때까지 프로그램 입력 값을 돌려주는 시퀀스를 만든다.

val numbers = generateSequence{ readLine()?.toIntOrNull() }

두번째 generateSequence() 함수는 초깃값과 파라미터가 하나인 함수를 인자로 받는다. 이 함수는 이전 값으로부터 다음 값을 만들어낸다. 첫번째 generateSequence()와 마찬가지로 이 경우에도 제네레이터 함수가 다음 값으로 널을 반환하면 시퀀스가 끝난다.

// 무한 시퀀스(단, 값 오버플로가 발생해서 음수와 양수를 왔다갔다 함): 1, 2, 4, 8,...  
val powers = generateSequence(1) { it\*2 }  
// 유한 시퀀스: 10, 8, 6, 4, 2, 0  
val evens = generateSequence(10) { if (it >= 2) it - 2 else null }

코틀린 1.3부터 특별한 빌더를 사용해 시퀀스를 만드는 방법이 추가됐다. 이때 빌더는 시퀀스 원소를 부분부분 지정한다. SequenceScope이 수신 객체 타입인 확장 람다를 받는 sequence() 함수를 통해 빌더를 구현할 수 있다. SequenceScope 타입은 확장 람다 본문안에 시퀀스 뒤에 값을 추가할 수 있는 다음과 같은 함수를 제공한다.

* yield(): 원소를 하나 시퀀스에 추가한다
* yieldAll(): 지정한 이터레이터, 이터러블, 시퀀스에 들어있는 모든 원소를 시퀀스에 추가한다

원소가 지연 계산된다는 점에 유의하라. 시퀀스에서 각 부분에 속한 원소에 접근하는 경우에만 yield(),yieldAll()가 호출된다. 다음 코드를 보자.

val numbers = sequence {  
 yield(0)  
 yieldAll(listOf(1, 2, 3))  
 yieldAll(intArrayOf(4, 5, 6).iterator())  
 yieldAll(generateSequence(10) { if (it < 50) it\*3 else null })  
}  
  
println(numbers.toList()) // [0, 1, 2, 3, 4, 5, 6, 10, 30, 90]

sequence()/yield()/yieldAll()로 만들어진 이 시퀀스 빌더는 실제로는 유예가능 계산(suspendable computation)이라는 강력한 코틀린 기능의 예이다. 다중스레드 환경에서 이 기능이 아주 유용하다. *13장. 동시성*에서 이 주제를 다룬다.

마지막으로 컬렉션 사이의 변환을 처리하는 함수에 대해 설명한다. 예를 들어 배열 원소를 바탕으로 리스트를 만들거나, 시퀀스를 집합으로 만들 수 있다.

println(  
 listOf(1, 2, 3, 2, 3).toSet()  
) // [1, 2, 3]  
  
println(  
 arrayOf("red", "green", "blue").toSortedSet()  
) // [blue, green, red]  
  
println(  
 mapOf(1 to "one", 2 to "two", 3 to "threen").toList()  
) // [(1, one), (2, two), (3, threen)]  
  
println(  
 sequenceOf(1 to "one", 2 to "two", 3 to "threen").toMap()  
) // {1=one, 2=two, 3=threen}

변환 함수 전체 목록을 표준 라이브러리 참조 문서(http://kotlinlang.org/api/latest/jvm/stdlib)에서 볼 수 있다. 변환 함수들은 몇가지 관례를 따른다. to로 시작하는 함수(예: toList(), toMap())는 원본 컬렉션의 복사본을 생성한다. 반면 as로 시작하는 함수(예: asList())는 원본 컬렉션이 변경되는 경우 이를 반영해주는 뷰(view)를 만들어준다.

##### IDE 팁

변환 함수를 선택하거나 다른 메서드를 선택할 때 IDE의 자동완성 기능 사용을 주저하지 말라(Ctrl + Space/Cmd + Space로 사용가능). 그림 7.2를 보라.

##### 그림 7.2: 자동 완성을 사용해 함수 호출 코드 완성하기

걸러내기, 변환, 정렬 등의 연산을 통해 기존 컬렉션을 바탕으로 새로운 컬렉션을 만들 수도 있다. 이런 경우에 대해 다음에 다룬다.

## 기본 컬렉션 연산

이번 절에서는 코틀린 컬렉션 타입이 지원하는 몇가지 기본 연산에 대해 다룬다.

모든 컬렉션이 기본지원하는 공통 연산으로는 이터레이션이 있다. 배열, 이터러블, 시퀀스, 맵은 iterator() 함수를 지원한다. 이 함수가 반환하는 Iterator 객체를 사용해 컬렉션 원소를 순회할 수 있기는 하지만, 코틀린이 같은 일을 할 수 있는 더 간결한 방법을 제공하기 때문에 실무에서 이 이터레이터를 사용하는 경우는 드물다.

iterator() 함수가 있다는 사실은 *이터러블* 절에서 설명했던 것처럼 모든 컬렉션에 대해 for 루프를 쓸 수 있다는 뜻이다. 다만 map의 iterator() 함수가 반환하는 이터레이터는 Map.Entry 타입의 값을 돌려준다는 사실을 알아두라. 코틀린 맵 원소는 구조분해를 지원하므로, 다음과 같이 맵 이터레이션을 할 수 있다.

val map = mapOf(1 to "one", 2 to "two", 3 to "three")  
  
for ((key, value) in map) {  
 println("$key -> $value")  
}

Map.Entry 타입의 값을 인자로 받는 람다에서도 마찬가지 방식으로 구조분해를 쓸 수 있다.

for루프의 대안으로는 forEach() 확장 함수가 있다. 이 함수는 컬렉션의 각 원소를 제공하면서 인자로 받은 람다를 실행해준다.

intArrayOf(1, 2, 3).forEach { println(it\*it) }  
  
listOf("a", "b", "c").forEach { println("'$it'") }  
  
sequenceOf("a", "b", "c").forEach { println("'$it'") }  
  
mapOf(1 to "one", 2 to "two", 3 to "three").forEach { (key, value) ->  
 println("$key -> $value")  
}

원소 인덱스를 참조해야 한다면 forEachIndexed() 함수를 쓰면 된다.

listOf(10, 20, 30).forEachIndexed { i, n ->println("$i: ${n\*n}") }

컬렉션 타입이 제공하는 기본 기능으로는 다음과 같은 기능들이 있다.

* size 프로퍼티는 원소 개수를 돌려준다.
* isEmpty() 함수는 컬렉션에 원소가 없는지 검사한다.
* contains()/containsAll() 함수는 지정한 원소나 지정한 모든 원소들이 컬렉션에 들어있는지 검사한다.

contains() 함수 호출을 in 연산자로 대신할 수 있다.

val list = listOf(1, 2, 3)  
  
println(list.isEmpty()) // false  
println(list.size) // 3  
println(list.contains(4)) // false  
println(2 in list) // true  
println(list.containsAll(listOf(1, 2))) // true

equals()메서드를 제대로 구현해야 contains()/containsAll()가 잘 작동한다는 점에 유의하라. 여러분이 직접 작성한 클래스의 인스턴스를 컬렉션 원소로 사용하는 경우, 필요에 따라 내용을 기반으로 동등성 비교를 하는 equals()를 반드시 작성해야 한다.

MutableCollection 타입은 원소를 추가하거나 제거할 수 있는 메서드를 제공한다. 다음 예제를 보자.

val list = arrayListOf(1, 2, 3)  
list.add(4) // 원소 하나 추가: [1, 2, 3, 4]  
list.remove(3) // 원소 하나 제거: [1, 2, 4]  
list.addAll(setOf(5, 6)) // 합집합: [1, 2, 4, 5, 6]  
list.removeAll(listOf(1, 2)) // 차집합: [4, 5, 6]  
list.retainAll(listOf(5, 6, 7)) // 교집합: [5, 6]  
list.clear() // 모든 원소 제거: []

add()/remove()/addAll()/removeAll() 대신 +=나 -= 복합 연산을 사용할 수도 있다.

list += 4  
list -= 3  
list += setOf(5, 6)  
list -= listOf(1, 2)

불변 컬렉션이든 가변 컬렉션이든 +와 – 연산자를 지원한다. 두 연산자 모두 원본을 그대로 두고 새로운 컬렉션을 생성한다.

println(listOf(1, 2, 3) + 4) // [1, 2, 3, 4]  
println(listOf(1, 2, 3) - setOf(2, 5)) // [1, 3]

불변 컬렉션에서도 +=나 –=를 사용 할 수 있지만, (가변 컬렉션의 경우와) 의미가 많이 다르다. 불변 컬렉션에서 이 두 복합 연산자는 대입문을 단축한 연산으로 취급되기 때문에 가변 변수에 대해서만 +=나 -=를 쓸 수 있다.

val readOnly = listOf(1, 2, 3)  
readOnly += 4 // Error: can't assign to val  
var mutable = listOf(1, 2, 3)  
mutable += 4 // Correct

하지만 이런 코드는 대입이 일어날 때마다 암시적으로 새로운 컬렉션을 만들기 때문에 프로그램 성능에 영향을 끼칠 수 있어서, 일반적으로 이런 코드를 피해야 한다.

##### IDE 팁

인텔리J 코틀린 플러그인은 이런 대입문에 대해 불변 컬렉션 대신 가변 컬렉션을 쓰라고 권장한다(그림 7.3).

##### 그림 7.3: 불변 컬렉션을 가변 컬렉션으로 바꾸기

리스트도 배열처럼 원소를 인덱스로 접근할 수 있는 메서드를 제공한다.

val list = listOf(1, 4, 6, 2, 4, 1, 7)  
  
println(list.get(3)) // 2  
println(list[2]) // 6  
println(list[10]) // Exception  
println(list.indexOf(4)) // 1  
println(list.lastIndexOf(4)) // 4  
println(list.indexOf(8)) // -1

일반적으로 get() 메서드 호출보다 인덱스를 사용하는 쪽을 더 선호할만하다. 리스트가 가변 리스트인 경우에는 인덱스를 사용해 원소를 변경할 수도 있다.

val list = arrayListOf(1, 4, 6, 2, 4, 1, 7)  
  
list.set(3, 0) // [1, 4, 6, 0, 4, 1, 7]  
list[2] = 1 // [1, 4, 1, 0, 4, 1, 7]  
list.removeAt(5) // [1, 4, 1, 0, 4, 7]  
list.add(3, 8) // [1, 4, 1, 8, 0, 4, 7]

subList() 함수는 시작 인덱스(포함)와 끝인덱스(포함하지 않음)로 지정한 리스트의 일부분에 대한 래퍼를 만든다. 이 뷰는 원본 컬렉션의 데이터를 공유하기 때문에 가변 리스트의 경우 뷰와 원본의 변화가 서로 반영된다.

val list = arrayListOf(1, 4, 6, 2, 4, 1, 7)  
val segment = list.subList(2, 5) // [6, 2, 4, 1]  
  
list[3] = 0  
println(segment[1]) // 0  
segment[1] = 8  
println(list[3]) // 8

집합에 추가된 연산은 없다. 집합 구현은 Collection에 있는 공통 메서드만 지원한다. 집합은 컬렉션에 원소를 추가할 때 중복을 허용하지 않는다.

Map 인스턴스는 키를 사용해 값을 얻는 메서드와 모든 키나 값의 컬렉션을 돌려주는 메서드 등을 지원한다. 예제를 살펴보자.

val map = mapOf(1 to "I", 5 to "V", 10 to "X", 50 to "L")  
  
println(map.isEmpty()) // false  
println(map.size) // 4  
println(map.get(5)) // V  
println(map[10]) // X  
println(map[100]) // null  
println(map.getOrDefault(100, "?")) // ?  
println(map.getOrElse(100) { "?" }) // ?  
println(map.containsKey(10)) // true  
println(map.containsValue("C")) // false  
println(map.keys) // [1, 5, 10, 50]  
println(map.values) // [I, V, X, L]  
println(map.entries) // [1=I, 5=V, 10=X, 50=L]

MutableMap 인스턴스는 기본적인 변경 연산과 +, - 연산자를 지원한다.

val map = sortedMapOf(1 to "I", 5 to "V")  
  
map.put(100, "C") // {1=I, 5=V, 100=C}  
map[500] = "D" // {1=I, 5=V, 100=C, 500=D}  
map.remove(1) // {5=V, 100=C, 500=D}  
map.putAll(mapOf(10 to "X")) // {5=V, 10=X, 100=C, 500=D}  
map += 50 to "L" // {5=V, 10=X, 50=L, 100=C, 500=D}  
map += mapOf(2 to "II",  
3 to "III") // {2=II, 3=III, 5=V, 10=X, 50=L, 100=C, 500=D}  
map -= 100 // {2=II, 3=III, 5=V, 10=X, 50=L, 500=D}  
map -= listOf(2, 3) // {5=V, 10=X, 50=L, 500=D}

불변 컬렉션과 가변 컬렉션에 대한 +=와 -= 연산의 차이는 맵의 경우에도 해당된다. 그리고 + 연산자는 키-값 쌍을 인자로 받지만, - 연산자는 키만 받는다는 점에 유의하라.

### 컬렉션 원소에 접근하기

기본 컬렉션 연산 외에 코틀린 표준 라이브러리에는 개별 컬렉션 원소에 대한 접근을 편하게 해주는 확장 함수들이 포함됐다. 이번 절은 이런 함수에 대해 다룬다.

first()/last() 함수는 각각 주어진 컬렉션의 첫번째와 마지막 원소를 반환하며, 컬렉션이 비어있으면 NoSuchElementException를 발생시킨다. 원소가 없으면 널을 반환하는 안전한 버전인 firstOrNull()/lastOrNull()도 있다.

println(listOf(1, 2, 3).first()) // 1  
println(listOf(1, 2, 3).last()) // 3  
println(emptyArray<String>().first()) // Exception  
println(emptyArray<String>().firstOrNull()) // null  
  
val seq = generateSequence(1) { if (it > 50) null else it \* 3 }  
  
println(seq.first()) // 1  
println(seq.last()) // 81

주어진 조건을 만족하는 첫번째와 마지막 원소를 찾기 위해 이런 함수들에게 술어를 넘길 수도 있다.

println(listOf(1, 2, 3).first { it > 2 }) // 3  
println(listOf(1, 2, 3).lastOrNull { it < 0 }) // null  
println(intArrayOf(1, 2, 3).first { it > 3 }) // Exception

single() 함수는 싱글턴 컬렉션의 원소를 반환한다. 컬렉션이 비어있거나 원소가 2개 이상이면 single()은 예외를 던진다. 안전한 버전인 singleOrNull()은 동일한 경우 예외 대신 널을 반환한다.

println(listOf(1).single()) // 1  
println(emptyArray<String>().singleOrNull()) // null  
println(setOf(1, 2, 3).singleOrNull()) // null  
println(sequenceOf(1, 2, 3).single()) // Exception

[편집자께: 임의 접근 리스트가 아니라 임의 접근 컬렉션이 맞습니다. 리스트가 아닌 컬렉션에도 elementAt을 사용할 수 있기 때문입니다.]

elementAt() 함수를 사용하면 인덱스를 사용해 컬렉션 원소를 읽을 수 있다. 이 함수는 리스트의 get() 함수를 일반화한 함수로, 배열, 이터러블, 시퀀스 등에 모두 적용할 수 있다. 하지만 일반적으로 이 함수를 임의 접근 컬렉션(random access list)이 아닌 컬렉션에 사용하면 인덱스 값에 비례한 실행 시간이 걸린다는 점에 유의하라.

인덱스가 잘못된 경우 elementAt()은 예외를 던진다. 인덱스가 컬렉션의 인덱스 경계를 벗어날 때 널을 반환하는 안전한 버전인 elementAtOrNull()이나 인덱스가 경계를 벗어난 경우 지정된 람다가 반환하는 원소를 돌려주는 elementAtOrElse()도 있다.

println(listOf(1, 2, 3).elementAt(2)) // 3  
println(sortedSetOf(1, 2, 3).elementAtOrNull(-1)) // null  
println(arrayOf("a", "b", "c").elementAtOrElse(1) { "???" }) // b  
  
val seq = generateSequence(1) { if (it > 50) null else it \* 3 }  
  
println(seq.elementAtOrNull(2)) // 9  
println(seq.elementAtOrElse(100) { Int.MAX\_VALUE }) // 81  
println(seq.elementAt(10)) // 예외 발생

배열이나 리스트에 대한 구조분해를 통해 앞에서부터 최대 5개의 원소를 추출할 수 있다는 점을 알아두라. 하지만 컬렉션에 있는 원소의 수보다 더 많은 원소를 구조분해로 가져오려고 하면 예외가 발생한다.

val list = listOf(1, 2, 3)  
  
val (x, y) = list // 1, 2  
val (a, b, c, d) = list // 예외

### 컬렉션에 대한 조건 검사

어떤 컬렉션이 주어진 조건을 만족하는지 검사하는 일이 자주 있다. 이로 위해 코틀린 라이브러리는 컬렉션 원소에 대해 주어진 술어를 테스트하는 등의 기본적인 검사를 구현하는 함수를 제공한다.

all() 함수는 컬렉션의 모든 원소가 주어진 술어를 만족하면 true를 반환한다. 배열, 이터러블, 시퀀스, 맵 등의 모든 컬렉션 객체에 대해 이 함수를 적용할 수 있다. 맵의 경우 맵 엔트리가 술어에 전달된다.

println(listOf(1, 2, 3, 4).all { it < 10 }) // true  
println(listOf(1, 2, 3, 4).all { it % 2 == 0 }) // false  
println(  
 mapOf(1 to "I", 5 to "V", 10 to "X")  
 .all { it.key == 1 || it.key % 5 == 0 }  
) // true  
// 1, 3, 9, 27, 81  
  
val seq = generateSequence(1) { if (it < 50) it\*3 else null }  
  
println(seq.all { it % 3 == 0 }) // false  
println(seq.all { it == 1 || it % 3 == 0 }) // true

none() 함수는 all()과 반대다. 컬렉션에 주어진 조건을 만족하는 원소가 하나도 없을 때 true를 반환한다.

println(listOf(1, 2, 3, 4).none { it > 5 }) // true  
println(  
 mapOf(1 to "I", 5 to "V", 10 to "X").none { it.key % 2 == 0 }  
) // false  
// 1, 3, 9, 27, 81   
  
val seq = generateSequence(1) { if (it < 50) it\*3 else null }  
println(seq.none { it >= 100 }) // true

이런 종류의 함수로 any()가 더 있다. any()는 컬렉션 원소 중 적어도 하나가 주어진 술어를 만족할 때 true를 반환한다.

println(listOf(1, 2, 3, 4).any { it < 0 }) // false  
println(listOf(1, 2, 3, 4).any { it % 2 == 0 }) // true  
println(  
 mapOf(1 to "I", 5 to "V", 10 to "X").any { it.key == 1 }  
) // true  
// 1, 3, 9, 27, 81  
  
val seq = generateSequence(1) { if (it < 50) it\*3 else null }  
  
println(seq.any { it % 3 == 0 }) // true  
println(seq.any { it > 100 }) // false

빈 컬렉션의 경우 all()와 none() 함수는 true를, any() 함수는 false를 반환한다. 드 모르간(De Morgan) 법칙을 사용하면 세 함수를 다른 함수를 사용해 기술할 수 있다.

c.all{ p(it) } == c.none { !p(it) }  
c.none{ p(it) } == !c.any { p(it) }

무한 시퀀스에 all(), none(), any()를 적용하면 실행이 끝나지 않을 수도 있다. 예를 들어 다음 코드는 결코 끝나지 않는다.

// 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0,...  
val seq = generateSequence(0) { (it + 1) % 5 }  
println(seq.all { it < 5 })

any()와 none() 함수에는 파라미터를 전혀 받지 않는 오버로드된 버전이 있다. 이런 함수들은 단순히 컬렉션 객체가 비어있는지만 검사한다.

println(emptyList<String>().any()) // false  
println(emptyList<String>().none()) // true  
println(listOf(1, 2, 3).any()) // true  
println(listOf(1, 2, 3).none()) // false

이런 오버로드된 함수는 배열이나 Collection, Map 타입의 인스턴스에 대해 사용할 수 있는 isNotEmpty()/isEmpty() 함수를 일반화하며, 임의의 이터러블이나 시퀀스에도 사용할 수 있다.

## 집계(aggregation)

컬렉션 원소의 합계를 계산하거나 최댓값을 찾는 것처럼 컬렉션 내용으로부터 한 값을 계산해내는 경우를 집계라고 부른다. 코틀린 라이브러리는 이런 목적에 사용할 수 있는 함수를 몇가지 제공한다. 앞 절에서는 any()나 all() 처럼 컬렉션에 대한 조건을 검사하는 함수를 살펴봤다. 이런 함수를 Boolean 값을 계산해내는 특별한 집계 함수라고 생각할 수도 있다.

일반적으로는 결코 반환값을 돌려받을 수 없기 때문에 무한 시퀀스에 대해 집계 함수를 적용할 수 없다(다음에 볼 count()는 예외다).

집계 함수를 세가지 기본 그룹으로 나눌수 있다. 첫번째 그룹은 합계, 최솟값, 최댓값 등 자주 쓰이는 집계값을 계산한다. 먼저 이런 함수에 대해 살펴보자.

count()는 컬렉션의 원소 개수를 반환한다. 배열, 이터러블, 시퀀스, 맵을 포함하는 모든 컬렉션 객체에 이 함수를 적용할 수 있다. 따라서 이 함수는 배열, 맵, Collection 인스턴스가 제공하는 size 프로퍼티를 일반화한 함수다.

println(listOf(1, 2, 3, 4).count()) // 4  
println(mapOf(1 to "I", 5 to "V", 10 to "X").count()) // 3  
  
// 1, 3, 9, 27, 81  
val seq = generateSequence(1) { if (it < 50) it\*3 else null }  
  
println(seq.count()) // 5

원소 개수가 Int.MAX\_VALUE보다 크면 count()는 예외를 던진다. 특히 무한 시퀀스에 대해 count()를 적용할 때 이런 일이 발생한다.

// 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0,. ..  
val seq = generateSequence(0) { (it + 1) % 5 }  
// Int.MAX\_VALUE 보다 많은 원소를 방문하자마자 예외를 던짐  
println(seq.count())

count() 함수에는 컬렉션 원소에 대한 술어를 적용하는 오버로드된 버전이 있다. 이 경우 count()는 주어진 조건을 만족하는 원소의 개수를 반환한다.

println(listOf(1, 2, 3, 4).count { it < 0 }) // 0  
println(listOf(1, 2, 3, 4).count { it % 2 == 0 }) // 2  
println(  
mapOf(1 to "I", 5 to "V", 10 to "X").count { it.key == 1 }  
) // 1  
  
// 1, 3, 9, 27, 81  
val seq = generateSequence(1) { if (it < 50) it\*3 else null }  
  
println(seq.count { it % 3 == 0 }) // 4  
println(seq.count { it > 100 }) // 0

sum() 함수는 수로 이뤄진 배열, 이터러블, 시퀀스의 산술 합계를 구한다.

println(listOf(1, 2, 3, 4).sum()) // 10  
println(doubleArrayOf(1.2, 2.3, 3.4).sum()) // 6.9  
  
// 1, 3, 9, 27, 81의 합계 구하기  
val seq = generateSequence(1) { if (it < 50) it\*3 else null }  
  
println(seq.sum) // 121

일반적인 + 연산과 비슷하게 반환 값 타입은 원 컬렉션의 원소 타입에 따라 달라진다. 예를 들어 바이트로 이뤄진 컬렉션의 합계는 Int이지만, LongArray에 대해 sum()을 적용하면 Long 값이 나온다.

원소 타입을 수로 변환할 수만 있다면 어떤 타입의 원소로 이뤄진 컬렉션에 대해서도 합계를 구할 수 있다. sumBy()와 sumByDouble()로 이런 합계를 구할 수 있다. 두 함수 모두 변환 함수를 파라미터로 받는다. sumBy()는 컬렉션 원소 값을 Int 값으로 변환하지만(부호가 없는 정수의 경우는 UInt로 변환) sumByDouble()은 Double로 변환한다는 점이 다르다.

println(listOf(1, 2, 3, 4).sumByDouble { it/4.0 }) // 2.5  
println(arrayOf("1", "2", "3").sumBy { it.toInt() }) // 6  
  
// X, XX, XXX, XXXX, XXXXX  
val seq = generateSequence("X") {  
 if (it.length>= 5) null else it + "X"  
}  
  
println(seq.sumBy { it.length }) // 15

비슷하게 average() 함수는 수로 이뤄진 배열, 이터러블, 시퀀스의 산술 평균을 구한다. 결과는 항상 Double이다.

println(listOf(1, 2, 3, 4).average()) // 2.5  
println(doubleArrayOf(1.2, 2.3, 3.4).average()) // 2.3000000000000003  
  
// 1, 3, 9, 27, 81의 평균 구하기  
val seq = generateSequence(1) { if (it < 50) it\*3 else null }  
  
println(.average()) // 24.2

컬렉션이 비어있으면 average() 함수는 항상 Double.NaN을 반환한다. 비어있지 않은 컬렉션 c에 대해 c.average()는 근본적으로 c.sum().toDouble()/c.count()와 같다. count() 함수와 비슷하게 average() 함수도 컬렉션에 Int.MAX\_VALUE보다 더 많은 원소가 들어있으면 예외를 발생시킨다.

main()과 max()는 비교 가능한(Comparable) 타입의 값이 들어있는 배열, 이터러블, 시퀀스의 최솟값과 최댓값을 계산한다.

println(intArrayOf(5, 8, 1, 4, 2).min()) // 1  
println(intArrayOf(5, 8, 1, 4, 2).max()) // 8  
println(listOf("abc", "w", "xyz", "def", "hij").min()) // abc  
println(listOf("abc", "w", "xyz", "def", "hij").max()) // xyz  
  
// 1, -3, 9, -27, 81  
val seq = generateSequence(1) { if (it < 50) -it \* 3 else null }  
  
println(seq.min()) // -27  
println(seq.max()) // 81

합계와 비슷하게 비교가능한 값으로 원소를 변환하는 함수를 제공함으로써 비교할수 없는 원소들로 이뤄진 컬렉션의 최댓값/최솟값을 구할 수도 있다. minBy()와 maxBy() 함수가 이런 동작을 구현한다.

class Person(val firstName: String,  
 val familyName: String,  
 val age: Int) {  
 override fun toString() = "$firstName $familyName: $age"  
}  
  
fun main() {  
 val persons = sequenceOf(  
 Person("Brook", "Watts", 25),  
 Person("Silver", "Hudson", 30),  
 Person("Dane", "Ortiz", 19),  
 Person("Val", "Hall", 28)  
 )  
   
 println(persons.minBy { it.firstName }) // Brook Watts: 25  
 println(persons.maxBy { it.firstName }) // Val Hall: 28  
 println(persons.minBy { it.familyName }) // Val Hall: 25  
 println(persons.maxBy { it.familyName }) // Brook Watts: 28  
 println(persons.minBy { it.age }) // Dane Ortiz: 19  
 println(persons.maxBy { it.age}) // Silver Hudson: 30  
}

또는 minWith()와 maxWith()를 사용할 수도 있다. 이 두 함수는 변환 함수 대신 비교기(Comparator)를 받는다. 다음 예제는 성 다음에 이름을 넣어서 비교하는 비교기와 이름 다음에 성을 넣어서 비교하는 비교기를 통해 서로 다른 순서를 지정하는 코드를 보여준다.

class Person(val firstName: String,  
 val familyName: String,  
 val age: Int) {  
 override fun toString() = "$firstName $familyName: $age"  
}  
  
val Person.fullNameget() = "$firstName $familyName"  
val Person.reverseFullNameget() = "$familyName $firstName"  
  
val FULL\_NAME\_COMPARATOR = Comparator<Person>{ p1, p2 ->  
 p1.fullName.compareTo(p2.fullName)  
}  
val REVERSE\_FULL\_NAME\_COMPARATOR = Comparator<Person>{ p1, p2 ->  
 p1.reverseFullName.compareTo(p2.reverseFullName)  
}  
  
fun main() {  
 val persons = sequenceOf(  
 Person("Brook", "Hudson", 25),  
 Person("Silver", "Watts", 30),  
 Person("Dane", "Hall", 19),  
 Person("Val", "Ortiz", 28)  
 )  
   
 // Brook Hudson: 25  
 println(persons.minWith(FULL\_NAME\_COMPARATOR))  
 // Val Ortiz: 28  
 println(persons.maxWith(FULL\_NAME\_COMPARATOR))  
 // Dane Hall: 19  
 println(persons.minWith(REVERSE\_FULL\_NAME\_COMPARATOR))  
 // Silver Watts: 30  
 println(persons.maxWith(REVERSE\_FULL\_NAME\_COMPARATOR))  
}

빈 컬렉션에 대해 적용하는 경우 min/max 집계 함수의 모든 변형은 널을 반환한다.

두번째 집계 함수 그룹은 컬렉션 원소를 문자열로 엮는 일을 담당한다. 기본 함수는 joinToString()이며, 가장 단순한 형태로 사용할 때 이 함수는 아무 인자도 받지 않는다.

println(listOf(1, 2, 3).joinToString()) // 1, 2, 3

디폴트로 각 원소를 toString() 메서드로 문자열로 변환한 다음에, 구분 문자열로 콤마와 공백(,)을 사용한다. 하지만 커스텀 변환 함수를 제공해 원소를 다른 형태인 문자열로 변환할 수도 있다. 람다 파라미터를 통해 이런 변환을 제공한다. 다음은 컬렉션 원소를 이진수로 바꿔서 표현하는 경우를 보여준다.

println(listOf(1, 2, 3).joinToString { it.toString(2) }) // 1, 10, 11

이 밖에도 다음과 같은 파라미터를 선택적으로 지정할 수도 있다.

* separator: 인접한 두 원소 사이에 들어갈 구분문자열(디폴트는 ", ")
* prefix와 postfix: 결과 문자열의 맨 앞(prefix)과 맨 뒤(postfix)에 들어갈 문자열(디폴트는 빈 문자열인 "")
* limit: 최대로 보여줄 수 있는 원소의 개수(디푤트는 -1로 개수 제한이 없다는 뜻임)
* truncated: limit가 양수인 경우, 컬렉션의 원소를 모두 표현하지 못할 때 이 파라미터를 뒤에 추가함(디폴트는 "...")

joinToString() 함수를 아무 배열, 이터러블, 시퀀스에 사용할 수 있다. 다음은 여러가지 옵션을 사용하는 예를 보여준다.

val list = listOf(1, 2, 3)  
println(list.joinToString(prefix = "[", postfix = "]")) // [1, 2, 3]  
println(list.joinToString(separator = "|")) // 1|2|3  
println(list.joinToString(limit = 2)) // 1, 2, ...  
println(list.joinToString(  
 limit = 1,  
 separator = " ",  
 truncated = "???"  
)) // 1 ???

코틀린 라이브러리는 더 일반적인 joinTo() 함수를 제공한다. 이 함수는 문자열을 새로 생성하는 대신 파라미터로 받은 Appendable 객체 뒤에 덧붙여준다. StringBuilder가 Appendable 객체에 속한다.

import java.lang.StringBuilder  
  
fun main() {  
 val builder = StringBuilder("joinTo: ")  
 val list = listOf(1, 2, 3)  
   
 println(list.joinTo(builder, separator = "|")) // joinTo: 1|2|3  
}

세번째 집계 함수 그룹은 두 값을 조합하는 함수를 활용해 여러분이 원하는 임의의 집계 방식을 구현하게 해주는 함수다. 이 그룹은 fold()/reduce() 함수와 이들의 변형으로 이뤄진다.

reduce() 함수는 파라미터가 2개인 함수를 받는다. 이 함수의 첫번째 인자는 누적된 값이고 두번째 인자는 컬렉션의 현재 값이다. 집계 과정은 다음과 같다.

1. 누적값은 최초에 컬렉션의 첫번째 원소로 초기화된다.
2. 컬렉션의 매 원소에(2번째 원소부터) 대해 현재 누적값과 현재 원소를 파라미터로 받은 함수에 적용하고 이 함수 적용의 결과를 누적값에 대입한다.
3. 누적의 결과를 반환한다.

컬렉션이 비어있으면 누적값을 초기화할 수 없으므로 reduce() 함수는 예외를 던진다.

예제를 살펴보자. 다음 코드에서는 reduce() 함수를 사용해 여러 수의 곱을 계산하고 여러 문자열을 서로 연결한다.

println(intArrayOf(1, 2, 3, 4, 5).reduce { acc, n -> acc \* n }) // 120  
println(listOf("a", "b", "c", “d”).reduce { acc, s -> acc + s }) // abcd

집계 규칙이 원소의 인덱스에 따라 달라진다면 reduceIndexed()를 사용할 수 있다. 이 함수는 현재 인덱스를 집계 연산의 첫번째 파라미터로 전달해준다. 앞에서 본 예제를 홀수 인덱스에 있는 원소에 대해서만 적용하게 변경하고 싶다고 하자.

// 8  
println(intArrayOf(1, 2, 3, 4, 5)  
 .reduceIndexed { i, acc, n -> if (i % 2 == 1) acc \* n else acc })  
// abd  
println(listOf("a", "b", "c", “d”)  
 .reduceIndexed { i, acc, s -> if (i % 2 == 1) acc + s else acc })

인덱스가 홀수인 원소만 처리하고 싶었지만, 첫번째 원소가 여전히 처리된다는 사실에 유의하라. 누적값의 초깃값을 원하는대로 지정하고 싶다면 reduce()/reduceIndexed() 대신 fold()/foldIndexed()를 사용할 수 있다. 추가로, fold()/foldIndexed()를 사용하면 누적값을 컬렉션 원소의 타입과 다른 타입의 값으로 만들 수도 있다.

println(  
 intArrayOf(1, 2, 3, 4).fold("") { acc, n -> acc + ('a' + n - 1) }  
) // abcd  
  
println(  
 listOf(1, 2, 3, 4).foldIndexed("") { i, acc, n ->  
 if (i % 2 == 1) acc + ('a' + n - 1) else acc  
 }  
) // bd

프로그래머가 항상 누적값의 초깃값을 지정하기 때문에, 리스트가 비어있어도 fold()는 reduce()와 달리 예외를 발생시키지 않는다.

배열, 이터러블, 시퀀스에 대해 reduce()/reduceIndexed()나 fold()/foldIndexed() 함수를 쓸 수 있다. 이 네 함수에 대해 컬렉션의 마지막 원소부터 반대 방향으로 계산을 수행해주는 함수도 있다. 배열이나 리스트의 맨 뒤(오른쪽)부터 처리한다는 사실을 알려주기 위해 이런 함수의 이름 뒤에는 Right가 붙는다.

println(  
 arrayOf("a", "b", "c", "d").reduceRight { s, acc -> acc + s }  
) // dcba  
  
println(  
 listOf("a", "b", "c", "d").reduceRightIndexed { i, s, acc ->  
 i % 2 == 0) acc + s else acc  
   
) // dca  
  
println(  
 intArrayOf(1, 2, 3, 4).foldRight("") { n, acc -> acc + ('a' + n - 1) }  
) // dcba  
  
println(  
 listOf(1, 2, 3, 4).foldRightIndexed("") { i, n, acc ->  
 if (i % 2 == 0) acc + ('a' + n - 1) else acc  
 }  
) // ca

fold/reduce 함수의 오른쪽과 왼쪽 버전에 전달하는 람다의 파라미터 순서에 유의하라. 왼쪽 버전에서는 누적값이 현재 원소 앞(왼쪽)에 오지만 오른쪽 버전에서는 반대다.

## 걸러내기

코틀린 표준 라이브러리는 컬렉션에서 조건을 만족하지 못하는 원소를 걸러내 버리고, 원하는 원소만 남기는 여러 확장 함수를 제공한다. 걸러내기 연산은 원본 컬렉션을 변경하지 않는다. 대신 전혀 새로운 컬렉션을 만들거나 원본 컬렉션과 구별되는 기존 가변 컬렉션에 선택된 원소를 추가한다.

가장 기본적인 걸러내기 연산은 filter() 함수에 의해 제공된다. 이 함수에 전달되는 술어는 현재 원소를 인자로 받아서 원소를 컬렉션에 유지해야 하는 경우에는 true를 반환하고 버려야 하는 경우에는 false를 반환한다. filter() 함수를 배열, 이터러블, 맵, 시퀀스에 적용할 수 있으며, 각각의 반환 타입은 다음과 같이 결정된다.

* Array<T>나 Iterable<T>를 거르면 List<T>를 얻는다.
* Map<K, V>를 거르면 Map<K, V>를 얻는다.
* Sequence<T>를 거르면 Sequence<T>를 얻는다.

filter()를 IntArray와 같은 원시 타입의 배열에도 적용할 수 있다. 다만, 반환 타입은 List<Int>처럼 원시 타입이 아닌 박싱된 타입이 원소인 리스트다. filter()를 원시 타입 배열에 적용할 때는 강제로 방식이 이뤄진다는 점에 유의하라.

filter()를 여러 컬렉션 객체에 적용하는 예를 살펴보자.

// List: [green, blue, green]  
println(  
 listOf("red", "green", "blue", "green").filter { it.length> 3 }  
)  
  
// List: [green, blue]  
println(setOf("red", "green", "blue", "green").filter { it.length> 3 })  
  
// List: [green, blue, green]  
println(  
 arrayOf("red", "green", "blue", "green").filter { it.length> 3 }  
)  
  
// List: [2, 4]  
println(byteArrayOf(1, 2, 3, 4, 5).filter { it % 2 == 0 })  
  
// Map: {X=10, L=50}  
println(  
 mapOf("I" to 1, "V" to 5, "X" to 10, "L" to 50)  
 .filter { it.value> 5 }  
)  
  
// Sequence  
val seq = generateSequence(100) {  
 if (it != 0) it/3 else null  
}.filter { it > 10 }  
  
// 리스트로 변환: [100, 33, 11]  
println(seq.toList())

맵의 경우 술어 파라미터가 맵 엔트리 타입(Map.Entry)을 인자로 받는다는 점에 유의하라. 키나 값만 걸러내고 싶다면 filterKeys()나 filter(Values) 함수를 사용하면 된다.

val map = mapOf("I" to 1, "V" to 5, "X" to 10, "L" to 50)  
  
println(map.filterKeys { it != "L" }) // {X=10, V=5, X=10}  
println(map.filterValues { it >= 10 }) // {X=10, L=50}

filterNot() 함수는 조건을 부정해 걸러낼 수 있게 해준다. 다른말로 하면 술어가 false를 반환할 때만 원소를 남긴다.

// [red]  
println(listOf("red", "green", "blue").filterNot { it.length> 3 })  
  
// {I=1, V=5}  
println(  
 mapOf("I" to 1, "V" to 5, "X" to 10, "L" to 50)  
 .filterNot { it.value> 5 }  
)

filterKeys()와 filterValues()에는 filterNot() 같은 부정 변형이 없음에 유의하라.

걸러내는 조건이 원소의 값 뿐 아니라 인덱스와도 관련 있다면, 람다가 인덱스 값을 추가로 받는 fitlerIndexed() 함수를 사용하라. 배열, 이터러블, 시퀀스는 filterIndexed()를 제공하지만 맵은 제공하지 않는다.

val list = listOf("red", "green", "blue", "orange")  
  
// [green, blue]  
println(  
 list.filterIndexed { i, v ->v.length> 3 && i<list.lastIndex }  
)  
  
val seq = generateSequence(100) { if (it != 0) it/3 else null }  
  
// [33, 11, 3, 1]  
println(seq.filterIndexed { i, v -> v > 0 && i> 0 }.toList())

표준 라이브러리에는 몇가지 일반적인 조건을 사용해 원소를 걸러내는 함수도 들어있다. 이런 함수 중에 filterNotNull()이 있다. 이 함수는 널인 원소를 걸러내 버린다. 이 함수는 항상 널이 될 수 없는 타입이 원소인 컬렉션을 만들어준다.

val list = listOf("red", null, "green", null, "blue")  
  
// Error: it이 널이 될 수 있음  
list.forEach { println(it.length) }  
  
// Ok: it이 널이 될 수 없음  
list.filterNotNull().forEach { println(it.length) }

##### IDE 팁

인텔리J는 컬렉션 원소 타입이 널이 될 수 없는 타입인데 filterNotNull()을 쓸 데 없이 쓰는 경우를 잡아내는 코드 검사 기능을 기본제공한다. Alt + Enter 메뉴를 사용해 이런 불필요한 걸러내기 함수를 제거할 수 있다(그림 7.4).

##### 그림 7.4: 쓸모없는 filter 제거하기

filterIsInstance()가 원소 중 특정 타입만 남기는 다른 일반적인 경우를 처리한다. 이 함수가 반환하는 컬렉션은 함수의 타입 인자로 넘긴 타입과 동일한 타입의 원소들로만 구성된다.

val hotchpotch = listOf(1, "two", 3, "four", 5, "six")  
val numbers = hotchpotch.filterIsInstance<Int>()  
val strings = hotchpotch.filterIsInstance<String>()  
  
println(numbers.filter { it > 2 }) // [3, 5]  
println(strings.filter { it != "two" }) // [four, six]

지금까지 살펴본 걸러내기 함수들은 호출될 때마다 새로운 불변 컬렉션을 만들어낸다. 걸러낸 결과를 이미 존재하는 가변 컬렉션에 집어넣고 싶으면 어떻게 할까? 이런 경우를 위해 남겨야 할 원소를 추가하는 대상 컬렉션을 추가 인자로 받는 특별한 걸러내기 함수들이 존재한다. 이런 함수는 이름 끝에 To가 붙어있다.

val allStrings = ArrayList<String>()  
  
// green, blue 추가됨  
listOf("red", "green", "blue").filterTo(allStrings) { it.length> 3 }  
  
// one, two, three 추가됨  
arrayOf("one", null, "two", null, "three").filterNotNullTo(allStrings)  
  
// abcde, bcde, cde, de, e,  
val seq = generateSequence("abcde") {  
 if (it.isNotEmpty()) it.substring(1) else null  
}  
  
// abcde, bcde, cde 추가됨  
seq.filterNotTo(allStrings) { it.length< 3 }  
  
// [green, blue, one, two, three, abcde, bcde, cde]  
println(allStrings)

filter(), filterNot(), filterIndexed(), filterIsInstance(), filterNotNull() 함수에 대해 To가 붙은 버전이 존재한다. 일반적으로 원본 컬렉션을 이런 함수의 대상 컬렉션으로 지정하면 컬렉션을 순회하는 중에 컬렉션 내용이 바뀌기 때문에 ConcurrentModificationException 예외가 발생한다.

val list = arrayListOf("red", "green", "blue")  
list.filterTo(list) { it.length> 3 } // Exception

여러가지 걸러내기 외에도 코틀린 표준 라이브러리는 원래 컬렉션을 주어진 술어를 만족하는 부분 컬렉션과 만족하지 않는 부분 컬렉션의 쌍으로 만들어주는 partition() 함수를 제공한다[[2]](#footnote-45). 다음 예제를 보라.

val (evens, odds) = listOf(1, 2, 3, 4, 5).partition { it % 2 == 0 }  
println(evens) // [2, 4]  
println(odds) // [1, 3, 5]

filter() 종류의 함수와 달리 partition()은 시퀀스에 적용한 경우에도 리스트의 쌍을 반환한다.

val seq = generateSequence(100) { if (it == 0) null else it/3 }  
  
val (evens, odds) = seq.partition { it % 2 == 0 }  
  
println(evens) // [100, 0]  
println(odds) // [33, 11, 3, 1]

맵은 partition()을 제공하지 않는다는 점에 유의하라.

## 변환

코틀린 표준 라이브러리는 컬렉션의 모든 원소를 주어진 규칙에 따라 변경한 다음, 이를 정해진 방식으로 조합해서 새로운 컬렉션을 만들어내는 변환 함수를 제공한다. 이런 함수를 매핑, 평평하게 하기(flattening), 연관짓기(association)라는 세가지 유형으로 나눌 수 있다.

매핑 변환은 주어진 함수를 원본 컬렉션의 각 원소에 적용한다. 그리고 함수를 적용한 결과는 새로운 컬렉션의 원소가 된다. 이 유형에 속하는 기본 함수는 map()이다. 배열, 이터러블, 시퀀스 맵을 포함하는 모든 컬렉션 객체가 이 함수를 제공한다. map()의 결과는 시퀀스에 적용한 경우에는 시퀀스이고 다른 경우에는 리스트이다.

println(setOf("red", "green", "blue").map { it.length }) // [3, 5, 4]  
println(listOf(1, 2, 3, 4).map { it\*it }) // [1, 4, 9, 16]  
println(byteArrayOf(10, 20, 30).map { it.toString(16) }) // [a, 14, 1e]  
  
// 50, 16, 5, 1, 0  
val seq = generateSequence(50) { if (it == 0) null else it / 3 }  
  
println(seq.map { it\*3 }.toList()) // [150, 48, 15, 3, 0]

변환시 인덱스를 고려해야 하는 경우 mapIndexed()를 사용할 수 있다.

// [(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]  
println(List(6) { it\*it }.mapIndexed { i, n ->i to n })

map()과 mapIndexed() 함수도 널이 아닌 값만 선택해 변환해주는 변종이 있다. 의미상 이 함수는 map이나 mapIndexed를 호출한 다음 filterNotNull()을 하는 것과 비슷하다.

println(  
 arrayOf("1", "red", "2", "green", "3").mapNotNull { it.toIntOrNull() }  
) // [1, 2, 3]  
  
println(  
 listOf("1", "red", "2", "green", "3").mapIndexedNotNull { i, s ->  
 s.toIntOrNull()?.let { i to it }  
 }  
) // [(0, 1), (2, 2), (4, 3)]

#### IDE 팁:

인텔리J는 불필요한 mapNotNull()과 mapIndexedNotNull() 호출을 map()이나 mapIndexed() 호출로 대신하라고 제안해준다(그림 7.5를 보라). 그 외에도 map()과 filterNotNull() 호출을 함께 사용하는 경우 코드를 mapNotNull()을 통해 단순화할 수 있다고 알려준다. 이에 대한 예제를 그림 7.6에서 볼 수 있다.

[편집자께: 그림과 그림 제목이 달라서 그림 제목을 그림에 맞게 바꿨습니다.]

##### 그림 7.5: 불필요한 mapNotNull() 함수를 map()으로 바꾸기

##### 그림 7.6: map과 filterNotNull 연쇄 호출을 합치기

map() 함수를 맵에 적용할 수도 있다. 이런 경우 변환 함수는 맵 원소를 입력으로 받고 리스트를 출력으로 내놓는다. 추가로 키와 값만 변환한 새 맵을 돌려주는 mapKeys()와 mapValues() 함수도 있다.

val map = mapOf("I" to 1, "V" to 5, "X" to 10, "L" to 50)  
  
// [I 1, V 5, X 10, L 50]  
println(map.map { "${it.key} ${it.value}" })  
  
// {i=1, v=5, x=10, l=50}  
println(map.mapKeys { it.key.toLowerCase() })  
  
// {I=1, V=5, X=a, L=32}  
println(map.mapValues { it.value.toString(16) })

각 mapXXX() 함수에는 새 컬렉션을 만들지 않고 기존 컬렉션에 원소를 추가해주는 변종이 있다. 필터와 마찬가지로 이런 함수들의 이름 끝에는 To가 붙는다.

val result = ArrayList<String>()  
  
listOf(1, 2, 3).mapTo(result) { it.toString() }  
  
arrayOf("one", "two", "three").mapIndexedTo(result) { i, s ->  
 "${i + 1}: s"  
}  
  
sequenceOf("100", "?", "101", "?", "110").mapNotNullTo(result) {  
 it.toIntOrNull(2)?.toString()  
}  
  
println(result) // [1, 2, 3, 1: s, 2: s, 3: s, 4, 5, 6]

평평하게 하는 연산은 원래 컬렉션의 각 원소를 새로운 컬렉션으로 변환한 다음, 이렇게 만들어진 컬렉션의 원소 한 컬렉션으로 합쳐준다. 이런 변환은 flatMap() 함수를 통해 구현된다. flatMap()은 시퀀스에 적용하면 시퀀스를, 다른 컬렉션에 적용하면 리스트를 만든다.

// [a, b, c, d, e, f, g, h, i]  
println(setOf("abc", "def", "ghi").flatMap { it.asIterable() })  
  
// [1, 2, 3, 4]  
println(listOf(1, 2, 3, 4).flatMap { listOf(it) })  
  
// [1, 1, 2, 1, 2, 3]  
Array(3) { it + 1 }.flatMap { 1..it }

flatten() 함수는 원소가 컬렉션인 모든 컬렉션에 적용할 수 있고 각각의 컬렉션을 이어붙인 한 컬렉션을 내놓는다. 이 함수를 flatMap()에 단순 변환(원소를 그대로 돌려주는 함수)을 적용한 것으로 생각할 수도 있다.

println(  
 listOf(listOf(1, 2), setOf(3, 4), listOf(5)).flatten()  
) // [1, 2, 3, 4, 5]  
  
println(Array(3) { arrayOf("a", "b") }.flatten()) // [a, b, a, b, a, b]  
  
println(  
 sequence {  
 yield(sequenceOf(1, 2))  
 yield(sequenceOf(3, 4))  
 }.flatten().toList()  
) // [1, 2, 3, 4]

##### IDE 팁

인텔리J 코틀린 플러그인은 flatMap에 대한 뻔한 호출을 인식해서 flatten으로 변환하라고 제안해준다(그림 7.7).

##### 그림 7.7: 뻔한 flatMap() 호출을 flatten()으로 바꾸기

map()과 비슷하게 flatMap() 함수에도 결과 원소를 기존 컬렉션에 추가하는 버전이 있다.

val result = ArrayList<String>()  
  
listOf(listOf("abc", "def"), setOf("ghi"))  
 .flatMapTo(result) { it }  
   
sequenceOf(sequenceOf(1, 2), sequenceOf(3, 4))  
 .flatMapTo(result) { it.map { "$it" } }  
   
println(result) // [abc, def, ghi, 1, 2, 3, 4]

이번 절에서 다루려는 또 다른 유형의 변환으로, 주어진 변환 함수를 바탕으로 원본 컬렉션 원소를 맵의 키나 맵의 값으로 만들 수 있는 변환이 있다. 첫번째 경우는 associateWith()로 구현하며 이 함수는 원래 컬렉션을 키의 근원으로 사용해 새로운 맵을 만들어준다.

println(  
 listOf("red", "green", "blue").associateWith { it.length }  
) // {red=3, green=5, blue=4}  
  
println(  
 generateSequence(1) { if (it > 50) null else it\*3 }  
 .associateWith { it.toString(3) }  
) // {1=1, 3=10, 9=100, 27=1000, 81=10000}

배열에는 associateWith()를 적용할 수 없다는 점에 유의하라.

associateBy() 함수는 associateWith()와 비슷하지만 컬렉션 원소를 값으로 취급하고 변환 함수를 통해 키를 얻는다는 점이 다르다. 어떤 키에 대해 같은 값이 여럿 존재하는 경우, 오직 한 값만 반환되는 맵에 남는다는 점에 유의하라.

// {3=red, 5=green, 4=blue}  
println(listOf("red", "green", "blue").associateBy { it.length })  
  
// {1=15, 2=25, 3=35}  
println(intArrayOf(10, 15, 20, 25, 30, 35).associateBy { it/10 })  
  
// {1=1, 10=3, 100=9, 1000=27, 10000=81}  
println(  
 generateSequence(1) { if (it > 50) null else it\*3 }  
 .associateBy { it.toString(3) }  
)

마지막으로 associate() 함수는 컬렉션의 원소를 사용해서 키와 값을 만들어낸다.

println(  
 listOf("red", "green", "blue")  
 .associate { it.toUpperCase() to it.length }  
) // {RED=3, GREEN=5, BLUE=4}  
  
println(  
 int ArrayOf(10, 15, 20, 25, 30, 35).associate { it to it/10 }  
) // {10=1, 15=1, 20=2, 25=2, 30=3, 35=3}  
  
println(  
 generateSequence(1) { if (it > 50) null else it\*3 }  
 .associate {  
 val s = it.toString(3)  
 "3^${s.length - 1}" to s  
 }  
) // {3^0=1, 3^1=10, 3^2=100, 3^3=1000, 3^4=10000}

이와 같은 결과를 associateBy() 함수를 오버로딩한 함수를 사용해 달성할 수도 있다. 이 오버로딩한 associateBy() 함수는 키와 값을 변환해주는 변환 함수를 별도로 받는다.

println(  
 listOf("red", "green", "blue").associateBy(  
 keySelector = { it.toUpperCase() },  
 valueTransform = { it.length }  
 )  
) // {RED=3, GREEN=5, BLUE=4}

맵을 만드는 연관(association) 함수에도 기존의 가변 맵에 결과 원소를 추가해주는 To가 붙은 변종(예: associateByTo())이 있다.

## 하위 컬렉션 추출

걸러내기 절에서 원 컬렉션의 원소 중에 특정 조건을 만족하는 원소만 남긴 부분 컬렉션을 추출하는 함수에 대해 배웠다. 이번 절에서는 비슷한 용도로 쓰이지만 다른 기준에 따라 컬렉션의 일부를 추출하는 함수를 살펴본다.

*기본 연산* 절에서 리스트의 세그먼트에 대한 뷰를 돌려주는 subList() 함수에 대해 살펴봤다. slice() 함수도 비슷한 일을 하지만 정수 쌍 대신 정수 범위를 사용해 세그먼트의 경계를 표현한다. 그리고 slice() 함수는 리스트 뿐 아니라 배열에도 적용할 수 있다.

// 0, 1, 4, 9, 16, 25  
println(List(6) { it\*it }.slice(2..4)) // [4, 9, 16]  
  
// 0, 1, 8, 27, 64, 125  
println(Array(6) { it\*it\*it }.slice(2..4)) // [8, 27, 64]

리스트의 경우 slice()의 동작은 subList()와 비슷하며,원 컬렉션을 반영할 수 있는 래퍼 객체를 만들어낸다. 배열의 경우, slice()는 지정한 인덱스들에 속하는 배열 원소가 들어있는 새로운 리스트이다.

배열 원소를 다른 배열로 추출하고 싶다면 sliceArray()를 써야 한다.

val slice = Array(6) { it\*it\*it }.sliceArray(2..4).contentToString()

정수의 이터러블을 인자로 받아서 선택할 원소의 인덱스를 결정하는 slice()/sliceArray()를 더 일반화한 버전도 있다. 이 버전을 사용하면 원본 배열이나 리스트의 임의의 하위 시퀀스를 추출할 수 있다.

println(List(6) { it\*it }.slice(listOf(1, 2, 3))) // [1, 4, 9]  
println(Array(6) { it\*it\*it }.slice(setOf(1, 2, 3))) // [1, 8, 27]  
println(  
 Array(6) { it\*it\*it }.sliceArray(listOf(1, 2, 3)).contentToString()  
) // [1, 8, 27]

take()와 takeLast() 함수는 이터러블이나 배열에서 원소를 주어진 개수만큼 추출한다. take()는 맨 앞에서부터, takeLast()는 맨 뒤에서부터 개수를 센다.

println(List(6) { it\*it }.take(2)) // [0, 1]  
println(List(6) { it\*it }.takeLast(2)) // [16, 25]  
println(Array(6) { it\*it\*it }.take(3)) // [0, 1, 8]  
println(Array(6) { it\*it\*it }.takeLast(3)) // [27, 64, 125]

take() 함수를 시퀀스에 적용할 수도 있다. 이 경우 이 함수는 원래 시퀀스의 앞부분에 위치한 원소들이 들어있는 새로운 시퀀스를 반환한다.

val seq = generateSequence(1) { if (it > 100) null else it\*3 }  
println(seq.take(3).toList()) // [1, 3, 9]

drop()/dropLast() 함수는 take()/takeLast()를 반전시킨 연산이라고 생각할 수 있다. 이들은 주어진 개수만큼 원소를 제거한 나머지 원소들을 돌려준다.

println(List(6) { it\*it }.drop(2)) // [4, 9, 16, 25]  
println(List(6) { it\*it }.dropLast(2)) // [0, 1, 4, 9]  
  
println(Array(6) { it\*it\*it }.drop(3)) // [27, 64, 125]  
println(Array(6) { it\*it\*it }.dropLast(3)) // [0, 1, 8]  
  
val seq = generateSequence(1) { if (it > 100) null else it\*3 }  
println(seq.drop(3).toList()) // [27, 81, 243]

take/drop 연산도 수 대신 술어를 받는 버전이 있다. 이런 버전은 주어진 조건을 만족하지 못하는 첫번째 원소를 발견할 때까지 원소를 남기거나 제거한다.

val list = List(6) { it \* it }  
  
println(list.takeWhile { it < 10 }) // [0, 1, 4, 9]  
println(list.takeLastWhile { it > 10 }) // [16, 25]  
println(list.dropWhile { it < 10 }) // [16, 25]  
println(list.dropLastWhile { it > 10 }) // [0, 1, 4, 9]  
  
val seq = generateSequence(1) { if (it > 100) null else it\*3 }  
  
println(seq.takeWhile { it < 10 }.toList()) // [1, 3, 9]  
println(seq.dropWhile { it < 10 }.toList()) // [27, 81, 243]

코틀린 1.2에 추가된 chunked() 함수를 사용하면 이터러블이나 시퀀스를 주어진 개수를 넘지 않는 작은 리스트(청크(chunk)고 부름)들로 나눠준다. chunked()의 가장 간단한 형태는 그냥 최대 청크 크기를 인자로 받는다.

// 0, 1, 4, 9, 16, 25, 36, 49, 64, 81  
val list = List(10) { it\*it }  
  
println(list.chunked(3)) // [[0, 1, 4], [9, 16, 25], [36, 49, 64], [81]]  
  
// 1, 3, 9, 27, 81, 243, 729  
val seq = generateSequence(1) { if (it > 300) null else it\*3 }  
  
println(seq.chunked(3).toList()) // [[1, 3, 9], [27, 81, 243], [729]]

chunked()를 이터러블에 적용하면 리스트를 반환하고, 시퀀스에 적용하면 시퀀스를 반환한다는 점에 유의하라.

더 일반적인 버전은 각각의 청크를 임의의 값으로 변환하는 함수를 지정하게 허용한다. 결과는 변환된 결과들로 이뤄진 리스트나 시퀀스이다. 다음 코드는 앞의 예제에서 만든 청크를 각 청크의 합계로 변환한다.

// 0, 1, 4, 9, 16, 25, 36, 49, 64, 81  
val list = List(10) { it\*it }  
  
println(list.chunked(3) { it.sum() }) // [5, 50, 149, 81]  
  
// 1, 3, 9, 27, 81, 243, 729  
val seq = generateSequence(1) { if (it > 300) null else it\*3 }  
  
println(seq.chunked(3) { it.sum() }.toList()) // [13, 351, 729]

마찬가지로 코틀린 1.2에 도입된 windowed() 함수를 사용하면 일정한 간격으로 청크를 연속적으로 얻어낸 슬라이딩 윈도우(sliding window)를 얻을 수 있다. chunked()와 마찬가지로 windowed()도 이터러블에서는 리스트를 만들고 시퀀스에서는 리스트의 시퀀스를 만든다.

// 0, 1, 4, 9, 16, 25  
val list = List(6) { it\*it }  
  
// [[0, 1, 4], [1, 4, 9], [4, 9, 16], [9, 16, 25]]  
println(list.windowed(3))  
  
// 1, 3, 9, 27, 81, 243  
val seq = generateSequence(1) { if (it > 100) null else it\*3 }  
  
// [[1, 3, 9], [3, 9, 27], [9, 27, 81], [27, 81, 243]]  
println(seq.windowed(3).toList())

chunked()와 마찬가지로 각 윈도우에 속한 원소를 집계하는 변환 함수를 전달할 수도 있다.

// 0, 1, 4, 9, 16, 25  
val list = List(6) { it\*it }  
  
println(list.windowed(3) { it.sum() }) // [5, 14, 29, 50]  
  
// 1, 3, 9, 27, 81, 243  
val seq = generateSequence(1) { if (it > 100) null else it\*3 }  
  
println(seq.windowed(3) { it.sum() }.toList()) // [13, 39, 117, 351]

추가로 슬라이딩 윈도우를 생성하는 규칙을 정하는 선택적인 파라미터를 지정할수도 있다.

* step: 서로 인접한 윈도우의 첫번째 원소 사이의 거리(디폴트는 1)
* partialWindows: 컬렉션의 마지막 부분에서 지정한 윈도우 크기보다 작은 크기의 윈도우를 포함시킬지 여부(디폴트는 false)

이런 옵션을 사용한 예제를 살펴보자.

// 0, 1, 4, 9, 16, 25  
val list = List(6) { it\*it }  
  
// 짝수 인덱스(0, 2) 원소만 윈도우 시작 원소가 된다  
// [[0, 1, 4], [4, 9, 16]]  
println(list.windowed(3, step = 2))  
  
// 맨 뒤에 2가지 부분 윈도우(크기가 3보다 작음)를 포함시킨다  
// [[0, 1, 4], [1, 4, 9], [4, 9, 16], [9, 16, 25], [16, 25], [25]]  
println(list.windowed(3, partialWindows = true))

원소가 2개뿐인 윈도우를 만드는 zipWithNext()라는 별도의 함수가 있다. windowed()와 달리 zipWithNext()는 리스트가 아니라 쌍(Pair)의 시퀀스나 리스트를 만들어낸다.

// 0, 1, 4, 9, 16, 25  
val list = List(6) { it\*it }  
  
// [(0, 1), (1, 4), (4, 9), (9, 16), (16, 25)]  
println(list.zipWithNext())  
  
// 1, 3, 9, 27, 81, 243  
val seq = generateSequence(1) { if (it > 100) null else it\*3 }  
  
// [(1, 3), (3, 9), (9, 27), (27, 81), (81, 243)]  
println(seq.zipWithNext().toList())

이런 타입 차이에 따라 집계 함수를 적용하는 버전도 리스트가 아니라 쌍을 인자로 받는 람다를 취한다.

// [0, 4, 36, 144, 400]  
println(List(6) { it\*it }.zipWithNext { a, b -> a \* b })

## 순서

표준 라이브러리에는 컬렉션 원소를 정해진 순서에 따라 정렬(sorting)하는 함수가 들어있다. 가장 간단한 형태는 sorted() 함수로, 원소 타입이 비교가능(Comparable)한 배열/이터러블/시퀀스에 적용할 수 있고, 원소들의 자연스러운 순서에 따라 오름차순으로(ascending) 원소를 정렬한다. sortedDescending() 함수는 sorted()와 비슷하지만 오름차순이 아니라 역순인 내림차순(descending)으로 원소를 정렬한다.

println(intArrayOf(5, 8, 1, 4, 2).sorted()) // [1, 2, 4, 5, 8]  
  
println(  
 int ArrayOf(5, 8, 1, 4, 2).sortedDescending()  
) // [8, 5, 4, 2, 1]  
  
println(  
 listOf("abc", "w", "xyz", "def", "hij").sorted()  
) // [abc, def, hij, w, xyz]  
  
println(  
 listOf("abc", "w", "xyz", "def", "hij").sortedDescending()  
) // [xyz, w, hij, def, abc]  
  
// 1, -3, 9, -27, 81  
val seq = generateSequence(1) { if (it < 50) -it \* 3 else null }  
  
println(seq.sorted().toList()) // [-27, -3, 1, 9, 81]  
println(seq.sortedDescending().toList()) // [81, 9, 1, -3, -27]

시퀀스에 이런 함수를 적용하면 시퀀스를 반환한다. 하지만 반환되는 시퀀스는 상태가 있는 시퀀스이며 원소에 최초로 접근할 때 전체 컬렉션을 정렬한다는 점에 유의하라.

배열이나 이터러블에 대해 정렬 함수를 적용하면 항상 List가 나온다. 배열의 경우 배열을 반환하는 sortedArray()/sortedArrayDescending()와 유사한 함수가 있다.

컬렉션 원소가 비교가능하지 않아도, 순서를 제공함으로써 sorted()를 대체할 수 있는 sortedBy()나 sortedWith() 함수를 통해 컬렉션을 정렬할 수 있다. 이 경우 관례는 min()이나 max() 집계 함수와 비슷하다. sortedBy()는 컬렉션 원소를 비교가능한 타입의 값으로 변환하는 함수를 인자로 받고, sortedWith()는 비교기(Comparator) 인스턴스를 인자로 받는다. 그리고 sortedBy()와 반대 방향으로 정렬을 해주는 함수를 sortedByDescending()이라고 부른다.

class Person(val firstName: String,  
 val familyName: String,  
 val age: Int) {  
 override fun toString() = "$firstName $familyName: $age"  
}  
  
val Person.fullNameget() = "$firstName $familyName"  
val Person.reverseFullNameget() = "$familyName $firstName"  
  
val FULL\_NAME\_COMPARATOR = Comparator<Person>{ p1, p2 ->  
 p1.fullName.compareTo(p2.fullName)  
}  
val REVERSE\_FULL\_NAME\_COMPARATOR = Comparator<Person>{ p1, p2 ->  
 p1.reverseFullName.compareTo(p2.reverseFullName)  
}  
  
fun main() {  
 val persons = listOf(  
 Person("Brook", "Hudson", 25),  
 Person("Silver", "Watts", 30),  
 Person("Dane", "Hall", 19),  
 Person("Val", "Ortiz", 28)  
 )  
   
 println(persons.sortedWith(FULL\_NAME\_COMPARATOR))  
 println(persons.sortedWith(FULL\_NAME\_COMPARATOR))  
 println(persons.sortedWith(REVERSE\_FULL\_NAME\_COMPARATOR))  
 println(persons.sortedWith(REVERSE\_FULL\_NAME\_COMPARATOR))  
 println(persons.sortedBy { it.age })  
 println(persons.sortedByDescending { it.age })  
}

지금까지 살펴본 모든 정렬 함수는 원본 컬렉션을 그대로 두고 새로운 컬렉션을 반환한다. 하지만 배열이나 가변 리스트의 경우에는 원본 컬렉션을 변경할 수 있기 때문에 제자리(in place) 정렬을 수행할 수 있다. sort()와 sortDescending() 함수는 제자리 정렬을 수행해 준다.

val array = intArrayOf(4, 0, 8, 9, 2).apply { sort() }  
println(array.contentToString()) // [0, 2, 4, 8, 9]  
  
val list = arrayListOf("red", "blue", "green").apply { sort() }  
println(list) // [blue, green, red]

이터러블이나 배열을 역순으로 뒤집어주는 함수들이 있다. 원본 컬렉션의 원소를 역순으로 나열한 새 리스트를 반환하는 reversed() 함수가 가장 기본적인 경우를 처리해준다.

println(intArrayOf(1, 2, 3, 4, 5).reversed()) // [5, 4, 3, 2, 1]  
println(listOf("red", "green", "blue").reversed()) // [blue, green, red]

배열에서는 reversedArray()를 쓸 수 있다. 이 함수는 배열 대신 리스트를 반환한다.

새 컬렉션을 만들지 않고 제자리에서 가변 리스트나 배열의 원소 순서를 뒤집을 때는 reverse()를 쓸 수 있다(sort()와 sorted()/sortedArray()의 관계를 떠올려라).

val array = intArrayOf(1, 2, 3, 4, 5).apply { reverse() }.contentToString()  
println(array) // [5, 4, 3, 2, 1]  
  
val list = arrayListOf("red", "green", "blue").apply { reverse() }  
println(list) // [blue, green, red]

asReversed() 함수는 새로운 리스트를 반환한다는 점에서 reversed()와 비슷하다. 하지만 asReversed()가 만들어내는 리스트는 원본에 대한 래퍼일 뿐이다. 원본과 반환된 리스트가 같은 데이터를 공유하기 때문에 asReversed()가 메모리를 더 효율적으로 활용한다. 가변 컬렉션에 asReversed()를 적용하면 가변 래퍼를 돌려준다. 원본이나 래퍼 중 어느 쪽을 변경해도 다른쪽에 변경 내용이 반영된다(reversed() 함수로 만든 컬렉션은 그렇지 않다).

val list = arrayListOf("red", "green", "blue")  
val reversedCopy = list.reversed()  
val reversedMirror = list.asReversed()  
  
list[0] = "violet"  
  
println(list) // [violet, green, blue]  
println(reversedCopy) // [blue, green, red]  
println(reversedMirror) // [blue, green, violet]

asReversed()를 리스트에만 적용할 수 있다는 점에 유의하라.

마지막으로 shuffled()을 언급하고 싶다. 이터러블에 shuffled()를 적용하면 원본의 원소를 임의의 순서로 재배치한 새 리스트를 돌려준다.

println(listOf(1, 2, 3, 4, 5).shuffled())

가변 리스트에는 제자리에서 원소를 섞어주는 shuffle()이 있다.

arrayListOf(1, 2, 3, 4, 5).shuffle()

시퀀스나 배열은 이 두 함수를 모두 지원하지 않는다는 점에 유의하라.

## 파일과 I/O 스트림

이번 절에서는 입력/출력 연산을 처리하는 코틀린 표준 라이브러리 기능에 대해 다룬다. 여기서 다룰 기능은 자바에 있는 파일과 I/O 스트림, URL 관련 API를 기반으로 만들어졌다. 이런 측면에서 볼 때 코틀린 표준 라이브러리는 JDK에 이미 있는 I/O 관련 클래스를 더 쉽게 사용할 수 있게 해주는 확장 함수와 확장 프로퍼티를 제공한다고 할 수 있다.

### 스트림 유틸리티

코틀린 표준 라이브러리는 자바 I/O 스트림에 대한 도우미 확장을 다수 제공한다. 이런 도우미 확장 함수들은 스트림 컨텐트에 대한 접근을 단순화시켜주고, 복사, 자동 스트림 정리 등의 복잡한 사용 패턴을 구현해준다. 이번 절에서는 이런 기능에 대해 자세히 살펴본다.

다음 함수를 사용하면 스트림의 전체 컨텐트를 읽어올 수 있다.

fun InputStream.readBytes(): ByteArray  
fun Reader.readText(): String  
fun Reader.readLines(): Line<String>

끝의 두 함수와 BufferedReader 클래스에 있는 readLine() 메서드의 차이를 기억하라. readLine()은 스트림에서 한 줄을 가져오지만 readText()/readLines()는 스트림 끝까지 컨텐트를 읽어서 전체를 한 문자열이나 각 줄을 나타내는 문자열의 리스트로 반환한다. 다음 예제를 보자.

import java.io.\*  
  
fun main() {  
 FileWriter("data.txt").use { it.write("One\nTwo\nThree") }  
   
 // One  
 FileReader("data.txt").buffered().use { println(it.readLine()) }  
   
 // One Two Three  
 FileReader("data.txt").use { println(it.readText().replace('\n', ' ')) }  
   
 // [One, Two, Three]  
 println(FileReader("data.txt").readLines())  
}

readText()와 달리 readLines() 함수는 값을 반환하면서 스트림을 닫아준다.

코틀린은 버퍼가 있는 스트림에 대한 직접 이터레이션을 허용한다. 다만 이진 데이터와 텍스트 데이터에 대한 API는 약간 다르다. BufferedOutputStream의 경우에는 iterator()가 있기 때문에 for 루프에서 스트림을 사용해 개별 바이트에 대해 이터레이션할 수 있다.

FileInputStream("data.bin").buffered().use {  
 var sum = 0  
 for (byte in it) sum += byte  
}

반면 BufferedReader에는 각 줄로 이뤄진 시퀀스를 돌려주는 lineSequence()가 있다.

FileReader("data.bin").buffered().use {  
 for (line in it.lineSequence()) println(line)  
}

좀 더 간접적인 형태이긴 하지만, 임의의 Reader 인스턴스에 대해 비슷한 기능을 제공한다. forEachLine()과 useLines() 함수는 줄단위 이터레이션을 허용한다. 이 두 함수는 스트림을 자동으로 닫기 때문에 스트림을 닫는 일을 신경쓸 필요가 없다.

import java.io.\*  
  
fun main() {  
 FileWriter("data.txt").use { it.write("One\nTwo\nThree") }  
   
 // One, Two, Three  
 FileReader("data.txt").useLines { println(it.joinToString()) }  
   
 // One/Two/Three  
 FileReader("data.txt").forEachLine { print("$it/") }  
}

둘의 차이는 forEachLine()에 전달되는 람다는 이터레이션이 진행되면서 한줄씩 데이터를 인자로 전달받는 반면, useLines()에 전달되는 람다는 모든 라인에 대한 시퀀스를 인자로 받는다는 점에 있다.

copyTo() 함수를 사용하면 한 스트림에서 다른 스트림으로 데이터를 전달할 수 있다. 이 함수는 이진 데이터와 텍스트 데이터를 처리하는 오버로딩된 버전이 존재한다.

fun InputStream.copyTo(  
 out: OutputStream,  
 bufferSize: Int = DEFAULT\_BUFFER\_SIZE  
): Long  
  
fun Reader.copyTo(out: Writer,  
 bufferSize: Int = DEFAULT\_BUFFER\_SIZE): Long

두 함수의 반환 값은 실제 복사된 바이트나 문자의 수를 뜻한다. 다음 예제는 copyTo() 사용법을 보여준다.

import java.io.\*  
  
fun main() {  
 FileWriter("data.txt").use { it.write("Hello") }  
   
 val writer = StringWriter()  
 FileReader("data.txt").use { it.copyTo(writer) }  
 println(writer.buffer) // Hello  
   
 val output = ByteArrayOutputStream()  
 FileInputStream("data.txt").use { it.copyTo(output) }  
 println(output.toString("UTF-8")) // Hello  
}

이번 절에서 다룰 다른 함수로, 명시적으로 정리를 해야 하는 스트림이나 다른 자원을 안전하게 처리할 수 있는 방법을 제공하는 use() 함수가 있다. use() 함수는 자바 java.io.Closeable 타입의 값에 대해 호출할 수 있으며(코틀린 1.2부터는 java.lang.AutoCloseable에도 적용할 수 있다), 인자로 전달받은 람다를 실행한 다음에 마지막에 자원을 적절히 정리해준 다음(람다에서 예외가 발생하든 하지 않든 자원 정리를 실행해준다), 람다의 결과를 돌려준다.

val lines = FileReader("data.bin").use { it.readLines() }

**자바와 코틀린의 차이**: 이 함수는 자바 7에 도입된 try-with-resources와 같은 역할을 한다.

방금 본 코드는 대략 다음과 같은 try 블럭과 같다.

val reader = FileReader("data.bin")  
  
val lines = try {  
 reader.readLines()  
} finally {  
 reader.close()  
}

##### IDE 팁

인텔리J는 본문과 같은 일을 하는 try 블럭을 감지해 use() 함수 호출로 바꾸라고 제안해준다. try 키워드에 캐럿을 위치시키고 Alt + Enter 메뉴를 사용하면 코드를 변화할 수 있다(그림 7.8).

##### 그림 7.8: 명시적 try 블럭을 use() 함수로 바꾸기

### 스트림 생성

코틀린 표준 라이브러리는 자바 I/O 스트림을 편하게 만들 수 있는 함수들을 제공한다. 이번 절에서는 몇가지 기본적인 경우를 살펴본다.

bufferedReaders()/bufferedWriter() 확장 함수를 사용하면 지정한 File 객체에 대해 BufferedReader/BufferedWriter 인스턴스를 만들 수 있다.

import java.io.File  
  
fun main() {  
 val file = File("data.txt")  
   
 file.bufferedWriter().use { it.write("Hello!") }  
 file.bufferedReader().use { println(it.readLine()) } // Hello!  
}

비슷한 함수로 reader()/writer() 확장 함수들도 있다. 이들은 각각 버퍼가 없는 FileReader/FileWriter 객체를 만든다.

printWriter() 함수는 형식화된 출력에 적합한 PrintWriter 인스턴스를 만든다.

리더(reader)/라이터(writer) 관련 함수를 사용할 때는 선택적으로 인코딩 문자 집합(디폴트는 UTF-8)을 지정할 수도 있다. 버퍼를 사용하는 버전에는 버퍼 크기를 지정하는 선택적인 추가 파라미터도 있다. 디폴트 버퍼 크기는 DEFAULT\_BUFFER\_SIZE 상수에 들어 있는데, 현재는 8 킬로바이트이다.

file.writer(charset = Charsets.US\_ASCII).use { it.write("Hello!") }  
  
file.bufferedReader(  
 charset = Charsets.US\_ASCII,  
 bufferSize = 100  
).use { println(it.readLine()) }

Charsets 객체에는 US-ASCII나 UTF의 변종 등의 여러 표준 문자 집합 상수가 정의되어 있다.

이진 파일을 처리하고 싶다면 inputStream()/outputStream()를 사용해 적절한 스트림을 생성하면 된다.

import java.io.File  
  
fun main() {  
 val file = File("data.bin")  
 file.outputStream().use { it.write("Hello!".toByteArray()) }  
 file.inputStream().use {  
 println(String(it.readAllBytes()))  
 } // Hello!  
}

String이나 ByteArray의 내용에 대한 I/O 스트림을 만드는 함수도 몇가지 있다. byteInputStream() 함수는 주어진 문자열을 원본으로 하는 ByteArrayInputStream 인스턴스를 만든다.

println("Hello".byteInputStream().read().toChar()) // H  
println("Hello".byteInputStream(Charsets.US\_ASCII).read().toChar()) // H

이와 비슷하게 reader() 함수는 StringReader 인스턴스를 만든다.

println("One\nTwo".reader().readLines()) // [One, Two]

비슷하게 inputStream() 함수는 주어진 바이트 배열을 원본으로 하는 ByteArrayInputStream 인스턴스를 만든다.

println(byteArrayOf(10, 20, 30).inputStream().read())

inputStream()이 오버로딩된 버전을 사용하면 오프셋과 크기를 지정해서 바이트 배열 중 일부분만 사용해 스트림을 만들 수도 있다.

val bytes = byteArrayOf(10, 20, 30, 40, 50)  
  
println(  
 bytes.inputStream(2, 2).readBytes().contentToString()  
) // [30, 40]

코틀린 표준 라이브러리에는 스트림의 연결을 단순화시켜주는 몇가지 확장 함수도 들어있다. 다음과 같은 함수를 사용하면 일반적인 InputStream 클래스의 인스턴스를 바탕으로 Reader, BufferedReader, BufferedInputStream 객체를 만들 수 있다.

fun InputStream.reader(  
 charset: Charset = Charsets.UTF\_8  
): InputStreamReader  
  
fun InputStream.bufferedReader(  
 charset: Charset = Charsets.UTF\_8  
): BufferedReader  
  
fun InputStream.buffered(  
 bufferSize: Int = DEFAULT\_BUFFER\_SIZE  
): BufferedInputStream

OutputStream을 Wrtier, BufferedWriter, BufferedOutputStream에 연결할 때 사용할 수 있는 비슷한 함수(이름은 writer(), bufferedWriter(), buffered())도 있다. 다음 예제는 이런 함수를 어떻게 사용하는지를 맛보게 해준다.

import java.io.FileInputStream  
import java.io.FileOutputStream  
  
fun main() {  
 val name = "data.txt"   
 FileOutputStream(name).bufferedWriter().use { it.write("One\nTwo") }  
 val line = FileInputStream(name).bufferedReader().use {  
 it.readLine()  
 }  
   
 println(line) // One  
}

Reader와 Writer에 대해 정의된 buffered() 함수도 있다.

fun Reader.buffered(bufferSize: Int = DEFAULT\_BUFFER\_SIZE): BufferedReader  
  
fun Writer.buffered(bufferSize: Int = DEFAULT\_BUFFER\_SIZE): BufferedWriter

### URL 유틸리티

코틀린 표준 라이브러리는 URL 객체의 주소로부터 네트워크 연결을 통해 데이터를 읽어오는 몇가지 도우미 함수를 제공한다.

fun URL.readText(charset: Charset = Charsets.UTF\_8): String  
  
fun URL.readBytes(): ByteArray

readText() 함수는 URL 인스턴스에 해당하는 입력 스트림의 컨텐트를 전부 읽어온다. readBytes() 함수도 비슷하게 입력 이진 스트림의 컨텐트를 바이트 배열로 읽어온다.

두 함수 모두 전체 스트림 컨텐트를 읽어오는 작업이 완료될 때까지 스레드를 블럭시키므로, 큰 파일을 다운로드 할 때 이 두 함수를 사용하면 안된다.

### 파일 컨텐트 접근하기

코틀린 표준 라이브러리는 명시적으로 I/O 스트림을 쓰지 않고도 파일 컨텐트를 읽을 수 있는 특별한 함수를 제공한다. 이런 함수는 전체 파일을 읽고 쓰거나, 데이터를 기존 파일 뒤에 추가하거나, 한줄씩 파일을 처리해야 할 때 유용하다.

텍스트 컨텐트를 처리할 때는 다음 함수를 쓸 수 있다.

* readText(): 파일 컨텐트 전부를 한 문자열로 읽어온다
* readLines(): 파일 컨텐트 전부를 줄 구분 문자를 사용해 줄 단위로 나눠서 읽어서 문자열의 리스트를 반환한다
* writeText(): 파일 컨텐트를 주어진 문자열로 설정한다. 필요하면 파일을 덮어쓴다
* appendText(): 주어진 문자열을 파일의 컨텐트 뒤에 추가한다

이런 함수를 사용하는 방법을 다음 예제에서 볼 수 있다.

import java.io.File  
  
fun main() {  
 val file = File("data.txt")  
   
 file.writeText("One")  
 println(file.readText()) // One  
   
 file.appendText("\nTwo")  
 println(file.readLines()) // [One, Two]  
   
 file.writeText("Three")  
 println(file.readLines()) // [Three]  
}

텍스트와 연관된 각 함수는 텍스트 인코등 방법을 지정하는 Charset 타입의 값을 선택적인 파라미터로 받을 수 있다.

이진 파일의 경우에도 텍스트 파일과 비슷한 방식으로 접근이 가능하지만, 문자열 대신 바이트 배열을 사용한다는 점이 다르다.

import java.io.File  
  
fun main() {  
 val file = File("data.bin")  
   
 file.writeBytes(byteArrayOf(1, 2, 3))  
 println(file.readBytes().contentToString()) // [1, 2, 3]  
   
 file.appendBytes(byteArrayOf(4, 5))  
 println(file.readBytes().contentToString()) // [1, 2, 3, 4, 5]  
   
 file.writeBytes(byteArrayOf(6, 7))  
 println(file.readBytes().contentToString()) // [6, 7]  
}

파일 컨텐트를 처리하게 해주는 다른 함수들은 파일을 전부 읽지 않고 블럭 안에서 조금씩 처리할 수 있게 해준다. 메모리에 모두 다 읽을 수 없는 큰 파일을 다루는 경우 이런 함수가 유용하다.

forEachLine() 함수를 사용하면 파일 전체를 읽지 않고 텍스트 컨텐트를 한줄씩 처리할 수 있다. 다음 예제는 이 함수의 동작을 보여준다.

import java.io.File  
  
fun main() {  
 val file = File("data.txt")  
   
 file.writeText("One\nTwo\nThree")  
 file.forEachLine { print("/$it") } // /One/Two/Three  
}

useLines() 함수는 주어진 람다에 줄의 시퀀스를 전달해준다. 람다는 이렇게 받은 시퀀스를 사용해 어떤 결과를 계산하고, 이 결과는 다시 useLines()의 결과로 반환된다.

import java.io.File  
  
fun main() {  
 val file = File("data.txt")  
   
 file.writeText("One\nTwo\nThree")  
 println(file.useLines { lines ->lines.count { it.length> 3 } }) // 1  
}

다른 텍스트 관련 파일 함수와 마찬가지로 forEachLine()과 useLines()에도 선택적인 Charset 파라미터를 전달할 수 있다.

이진 파일을 처리하기 위해서는 forEachBlock() 함수를 사용한다. 이 함수가 받는 람다는 ByteArray 버퍼와 현재 이터레이션에서 몇 바이트가 버퍼 안에 들어있는지 알려주는 정수를 파라미터로 받는다. 예를 들어 다음 코드는 data.bin 파일에 있는 모든 바이트의 합계를 출력한다.

import java.io.File  
  
fun main() {  
 val file = File("data.bin")  
 var sum = 0  
   
 file.forEachBlock { buffer, bytesRead ->  
 (0 until bytesRead).forEach { sum += buffer[it] }  
 }  
 println(sum)  
}

디폴트 버퍼 크기는 구현에 따라 다르지만, 선택적으로 blockSize 파라미터를 지정할 수 있다. 구현에 따라서는 버퍼 크기를 일정 크기 이상으로 제한할 수도 있다는 점에 유의하라. 코틀린 1.3에서는 디폴트 크기는 4096 바이트, 최소 크기는 512 바이트이다.

## 파일 시스템 유틸리티

이번 절에서는 파일 복사, 삭제, 디렉토리 계층 구조 순회 등을 쉽게 해주는 코틀린 표준 라이브러리 함수에 대해 다룬다.

deleteRecursively() 함수를 사용하면 파일이나 디렉터리를 자신에게 포함된 자손들까지 포함해 쉽게 지울 수 있다. 이 함수는 삭제가 성공하면 true를, 그렇지 않으면 false를 반환한다. 이 함수가 false를 반환한 경우 자식 디렉터리는 안지워지는 등 파일 중 일부만 지워졌을 수도 있다. 이 함수는 자바 API가 제공하는 mkdirs() 메서드의 역이라 할 수 있다.

import java.io.File  
  
fun main() {  
 File("my/nested/dir").mkdirs()  
 val root = File("my")  
   
 println("Dir exists: ${root.exists()}") // true  
 println("Simple delete: ${root.delete()}") // false  
 println("Dir exists: ${root.exists()}") // true  
 println("Recursive delete: ${root.deleteRecursively()}") // true  
 println("Dir exists: ${root.exists()}") // false  
}

copyTo() 함수는 자신의 수신 객체를 다른 파일에 복사하고 복사본을 가리키는 파일 객체를 돌려준다.

import java.io.File  
  
fun main() {  
 val source = File("data.txt")  
 source.writeText("Hello")  
   
 val target = source.copyTo(File("dataNew.txt"))  
 println(target.readText()) // Hello  
}

디폴트로 대상 파일을 덮어쓰지는 않는다. 따라서 대상 파일이 이미 존재하는 경우 copyTo() 함수는 FileAlreadyExistsException를 발생시킨다. 하지만 파일을 강제로 복사하도록 overwrite(덮어쓰기) 파라미터를 지정할 수도 있다.

import java.io.File  
  
fun main() {  
 val source = File("data.txt").also { it.writeText("One") }  
 val target = File("dataNew.txt").also { it.writeText("Two") }  
 source.copyTo(target, overwrite = true)  
 println(target.readText()) // One  
}

copyTo() 함수를 디렉터리에도 적용할 수 있다. 하지만 디렉터리의 경우 copyTo()는 하위 디렉터리나 디렉터리에 들어있는 파일을 복사하지는 않고 대상 경로에 맞춰 빈 디렉터리만 만들어준다. 디렉터리와 내용물을 모두 복사하고 싶다면 copyRecursively() 함수를 사용해야 한다.

import java.io.File  
  
fun main() {  
 File("old/dir").mkdirs()  
 File("old/dir/data1.txt").also { it.writeText("One") }  
 File("old/dir/data2.txt").also { it.writeText("Two") }  
   
 File("old").copyRecursively(File("new"))  
   
 println(File("new/dir/data1.txt").readText()) // One  
 println(File("new/dir/data2.txt").readText()) // Two  
}

copyTo()와 마찬가지로 copyRecursively()에도 overwrite 파라미터를 통해(true나 false로 지정) 덮어쓸지 여부를 정해줄 수 있다. 추가로 어떤 파일을 복사하다가 IOException가 발생하면 호출할 액션을 설정할 수도 있다. 선택적인 OnError 파라미터를 통해 (File, IOException) -> OnErrorAction 타입의 람다를 넘기면 된다. 람다의 결과 값은 copyRecursively() 함수가 문제가 되는 파일을 어떻게 처리할지를 결정한다.

* SKIP: 파일을 무시하고 복사를 계속 진행한다
* TERMINATE: 복사를 중단한다

마지막 파라미터이기 때문에 onError 람다를 인자 목록 괄호 다음에 위치시킬 수 있다.

File("old").copyRecursively(File("new")) { file, ex -> OnErrorAction.SKIP }

디폴트 동작은 잡아낸 IOException 인스턴스를 copyRecursively()를 호출한 호출자쪽으로 재발생시키는 것이다.

walk() 함수는 깊이 우선(depth first) 디렉터리 구조 순회를 구현한다. 다음과 같이 순회 방향을 결정하는 선택적인 파라미터가 있다.

* TOP\_DOWN: 자식보다 부모를 먼저 방문(디폴트 값)
* BOTTOM\_UP: 자식을 부모보다 먼저 방문

반환값은 File 인스턴스의 시퀀스이다. 다음 예제는 여러 순회 방식의 사용법을 보여준다.

import java.io.File  
import kotlin.io.FileWalkDirection.\*  
  
fun main() {  
 File("my/dir").mkdirs()  
 File("my/dir/data1.txt").also { it.writeText("One") }  
 File("my/dir/data2.txt").also { it.writeText("Two") }  
   
 println(File("my").walk().map { it.name }.toList())  
 println(File("my").walk(TOP\_DOWN).map { it.name }.toList())  
 println(File("my").walk(BOTTOM\_UP).map { it.name }.toList())  
}

그리고, walk(TOP\_DOWN) 대신 walkTopDown()를 쓸 수 있고, walk(BOTTOM\_UP) 대신 walkBottomUp()를 쓸 수 있다.

walk() 함수가 반환하는 시퀀스는 특별한 FileTreeWalk 클래스에 속한다. 이 클래스는 일반적인 시퀀스 기능과 더불어 별도의 순회 옵션을 지정할 수 있게 해준다. maxDepth() 함수는 순회할 하위 트리의 최대 깊이를 지정한다.

println(  
File("my").walk().maxDepth(1).map { it.name }.toList()  
) // [my, dir]

onEnter()와 onLeave() 함수는 순회가 디렉터리에 들어가거나 디렉터리에서 나올 때 호출할 동작을 지정한다. onEnter() 호출은 (File) -> Boolean 람다를 파라미터로 받고, 이 람다의 반환 값은 디렉터리(그리고 이 디렉터리의 자식)를 방문할지 여부를 결정한다. onLeave() 호출은 (File) -> Unit 람다를 받는다. onFail() 함수를 통해 디렉터리의 자식에 접근할 때 IOException이 발생하는 경우에 호출될 액션을 정할 수 있다. 이 액션은 (File, IOException) -> Unit 타입의 람다이며, 문제가 되는 디렉터리와 발생한 예외를 인자로 받는다.

이 네 함수가 모두 FileTreeWalk의 현재 인스턴스를 반환하기 때문에, 다음 예제처럼 각 함수 호출을 서로 연쇄할 수 있다.

println(  
 File("my")  
 .walk()  
 .onEnter { it.name != "dir" }  
 .onLeave { println("Processed: ${it.name}") }  
 .map { it.name }  
 .toList())

이 코드는 다음과 같은 내용을 출력한다.

Processed: my  
[my]

이유는 onEnter() 함수 안에서 dir이라는 이름의 디렉터리 방문을 거부하기 때문이다.

디폴트 액션은 다음과 같다. onEnter()는 항상 true를 반환하며, onLeave()는 아무 일도 하지 않고, onFail()은 예외를 다시 던진다. 최대 트리 깊이는 Int.MAX\_VALUE이며 실질적으로 제한이 없다고 볼 수 있다.

createTempFile()/createTempDir() 함수를 사용해 임시 파일이나 디렉터리를 만들 수 있다.

valtmpDir = createTempDir(prefix = “data”)  
valtmpFile = createTempFile(directory = tmpDir)

두 함수 모두 같은 순서와 타입으로 이뤄진 파라미터 목록을 받는다.

fun createTempDir(  
 prefix: String = "tmp",  
 suffix: String? = null,  
 directory: File? = null  
): File

## 결론

이번 절에서는 코틀린 표준 라이브러리에서 컬렉션을 다루는 중요한 부분을 살펴봤다. 배열, 이터러블, 시퀀스, 맵 등의 컬렉션 타입에 대해 설명했다. 그리고 이런 컬렉션 타입의 기본 API와 연산을 살펴보고, 원소 접근, 하위 컬렉션, 걸러내기, 집계, 변환, 정렬 등 여러가지 컬렉션의 용례에 대해 살펴봤다. 이번 장의 뒷부분에서는 스트림 생성, 스트림 데이터 접근, 삭제 복사 등의 일반적인 파일 시스템 연산 등을 더 쉽게 해주는 I/O 유틸리티에 대해 다뤘다.

다음 장에서는 객체지향 프로그래밍이라는 주제로 다시 돌아가서 클래스 상속과 위임을 코틀린 애플리케이션에서 사용하는 방법에 대해 배운다.

## 문제

1. 코틀린 컬렉션 타입의 개요를 설명하라. 자바 컬렉션 라이브러리와 코틀린 컬렉션 라이브러리의 가장 중요한 차이는 무엇인가?
2. 컬렉션 타입이 제공하는 기본 연산은 무엇인가?
3. 컬렉션 원소를 이터레이션하는 여러가지 방법을 설명하라.
4. 컬렉션 원소에 접근할 때 쓸 수 있는 공통 함수를 나열하고 설명하라.
5. 코틀린 라이브러리가 제공하는 공통 집계 함수에 대해 설명하라.
6. fold()와 reduce() 연산에 대해 설명하라.
7. all()/any()/none() 함수의 목적은 무엇인가?
8. 컬렉션의 걸러내기 함수에 대해 설명하라.
9. 하위 컬렉션을 추출하는 방법에 대해 설명하라.
10. 컬렉션에 적용할 수 있는 표준 변환 함수들에 대해 설명하라. 매핑(예: map 등), 평평하게 하기(예: flatten 등), 연관짓기(예: associateBy 등)에 대해 설명하라.
11. 코틀린 표준 라이브러리가 제공하는 컬렉션 순서 지정 및 정렬 유틸리티에 대해 설명하라.
12. 스트림 생성과 변환 유틸리티에 대해 설명하라.
13. 파일이나 I/O 스크림의 컨텐트에 접근하기 위해 사용할 수 있는 함수를 나열하고 설명하라.
14. 파일 시스템 유틸리티 함수에 대해 설명하라.

1. 옮긴이: compareBy<Person>{ it.age } 같은 식에서 compareBy의 타입 파라미터로 사용되는 Person은 정렬 대상 객체의 타입이며 Comparable이 아닐 수도 있다. 반면 람다식은 이 Person 객체를 받아서 다른 타입을 반환해야 하는데, 반드시 Comparable의 하위 타입에 속하는 객체를 반환해야 한다. compareBy의 타입을 보면 이를 더 확실히 알 수 있다. [↑](#footnote-ref-32)
2. 옮긴이 - 이산 수학이나 집합론에서는 어떤 집합을 둘 이상의 부분 집합으로 나누되, 각 부분 집합 사이에 교집합이 없고(서로 소) 부분집합을 모두 합하면 원래의 집합과 같은 경우를 분할(partition)이라고 부른다. [↑](#footnote-ref-45)