

안드로이드 프로그래밍

2025년 2학기

Kotlin 컬렉션의 기초

컬렉션 인터페이스: 핵심 계약

- 객체 그룹을 저장하고 관리하기 위한 강력한 도구 집합
- 프레임워크의 기반은 인터페이스로, 각 컬렉션 유형이 준수해야 하는 특정 "계약"을 정의
- Collection<T>는 모든 컬렉션의 최상위 인터페이스, 하위 인터페이스인 List<T>, Set<T>, Map<K, V>는 각각 고유한 목적
 - 내부 구현의 세부 사항으로부터 추상화
 - 예를 들어, List와 MutableList는 개념적으로 다른 두 인터페이스이지만, 실제로는 동일한 구현(ArrayList)을 사용할 수 있음

List: 순서가 있고, 인덱스로 접근 가능하며, 유연한 컬렉션

- List는 요소의 순서를 유지하고, 0부터 시작하는 정수형 인덱스를 통해 요소에 접근할 수 있는 컬렉션
- 동일한 요소가 여러 번 중복될 수 있다
 - 예를 들어, 전화번호나 쇼핑 목록처럼 요소의 순서가 중요하고 중복이 허용되는 데이터에 적합
- List의 기본 구현은 Array-based list이며, 이는 내부적으로 ArrayList를 감싸고 있는 형태
 - ArrayList는 동적으로 크기를 조절할 수 있는 배열로, 인덱스를 통한 요소 접근(get)은 상수 시간 복잡도($O(1)$)
 - 그러나, 중간에 요소를 추가하거나 제거하는 작업은 그 뒤의 모든 요소를 이동시켜야 하므로 선형 시간 복잡도($O(n)$)
 - 반면에, 연결 리스트(Linked List)는 중간 삽입/삭제가 상수 시간 복잡도($O(1)$), 특정 인덱스에 접근하려면 처음부터 순차적으로 탐색해야 하므로 선형 시간 복잡도($O(n)$)

List: 순서가 있고, 인덱스로 접근 가능하며, 유연한 컬렉션

```
// 불변 리스트 생성
val orderedList = listOf("apple", "banana", "cherry", "apple")

// 인덱스를 통한 접근
val firstElement = orderedList // apple (0(1))

// 중복 허용 확인
val containsDuplicate = orderedList.contains("apple") // true

// 가변 리스트 사용 예제
val shoppingList = mutableListOf("milk", "bread")
shoppingList.add("eggs") // [milk, bread, eggs]
shoppingList.removeAt(0) // [bread, eggs]
```

Set: 고유함의 힘

- 중복되지 않는 고유한 요소만 저장하는 컬렉션
- 일반적으로 요소의 순서를 보장하지 않음
 - 예를 들어, 로또 당첨 번호처럼 요소가 고유하고 순서가 중요하지 않은 경우에 유용
- 기본적으로 `setOf()` 함수는 `LinkedHashSet`을 반환: 해시 테이블과 이중 연결 리스트를 결합한 자료 구조
 - 해시 테이블을 사용하여 요소의 고유성을 보장하고 상수 시간 복잡도($O(1)$)로 빠른 검색을 지원
 - 동시에 연결 리스트를 통해 요소가 삽입된 순서를 유지
- 반면, `hashSetOf()` 함수는 `HashSet`을 반환: 순서를 유지하기 위한 연결 리스트를 사용하지 않음
 - 따라서 `HashSet`은 `LinkedHashSet`보다 더 적은 메모리를 사용하고, 더 빠를 수 있음
 - 순서에 의존하는 작업에는 적합하지 않음

Set: 고유함의 힘

```
// setOf()는 기본적으로 LinkedHashSet을 반환하여 삽입 순서를 유지합니다.  
val uniqueSet = setOf("apple", "banana", "apple", "cherry") // 중복된 "apple"은  
자동으로 제거됩니다.
```

```
println(uniqueSet) // [apple, banana, cherry]
```

```
// 해시 기반의  $O(1)$  탐색
```

```
val hasBanana = "banana" in uniqueSet // true
```

Map: 효율적인 키-값 연관 관계

- 고유한 키와 해당 키에 연결된 값으로 구성된 컬렉션
 - 각 키는 반드시 고유해야 하지만, 값은 중복될 수 있음
- 객체 간의 논리적 연결을 저장하는 데 매우 유용
 - 예를 들어, 도시 이름과 인구, 또는 직원 ID와 직책을 매핑하는 경우에 사용
- Map의 기본 구현 역시 LinkedHashMap이며, 삽입 순서를 유지
 - 순서를 보장하지 않는 더 빠른 대안으로는 HashMap이 있음
 - 두 구현 모두 키를 이용한 값 검색은 평균적으로 상수 시간 복잡도($O(1)$)

Map: 효율적인 키-값 연관 관계

```
// 불변 맵 생성 (키가 중복되면 마지막 값이 사용됩니다)
val cityPopulation = mapOf("Seoul" to 9.7, "Tokyo" to 14.0, "Seoul" to 9.8)
println(cityPopulation) // {Seoul=9.8, Tokyo=14.0}

// 키를 이용한 O(1) 탐색
val tokyoPop = cityPopulation // 14.0

// 가변 맵 사용 예제
val mutableMap = mutableMapOf("a" to 1, "b" to 2)
mutableMap["c"] = 3 // 새로운 키-값 쌍 추가
mutableMap.remove("a") // 키 "a"와 값 제거
```

숙련된 Kotlin 개발의 초석: 불변성과 가변성

중요한 구분: 읽기 전용 뷰와 진정한 불변 데이터

- 표준 라이브러리의 컬렉션에서 `List<T>`, `Set<T>`, `Map<K, V>`와 같은 타입은 '불변' 컬렉션으로 알려짐
- 객체 자체가 변경 불가능하다는 것을 의미하는 것이 아니라, 해당 변수가 제공하는 인터페이스가 '읽기 전용' 기능만 허용한다는 것을 의미
- '불변성의 환상'
 - 예를 들어, `val list = mutableListOf("A", "B")`와 같이 가변 리스트를 선언, 이를 `val readOnlyList: List<String> = list`와 같이 읽기 전용 List 인터페이스에 할당할 수 있음
 - `readOnlyList` 변수 자체는 `add`나 `remove`와 같은 수정 메서드를 호출할 수 없음. 그러나 `list` 변수를 통해 원본 컬렉션에 요소를 추가하면(`list.add("C")`), `readOnlyList`가 참조하는 객체도 변경됨
- 특히 다중 스레드 환경에서 객체의 동일한 인스턴스를 여러 곳에서 참조할 때, 한 곳에서 발생한 의도치 않은 변경이 다른 곳에 영향을 미침

표준 라이브러리 함수: listOf vs. mutableListOf

- 가변성을 제어하는 두 가지 주요 요소
 - 변수의 불변성(val vs. var)과 컬렉션 객체의 불변성(List vs. MutableList)
 - 이 두 개념은 독립적이며, 이들의 조합을 이해해야
- val은 변수 참조가 한 번 할당되면 변경될 수 없음을 의미하고, var은 참조를 재할당할 수 있음을 의미
- 반면, List는 콘텐츠를 수정할 수 없는 인터페이스를 제공하고, MutableList는 콘텐츠를 추가, 제거 또는 변경할 수 있는 메서드를 제공

표준 라이브러리 함수: listOf vs. mutableListOf

```
// 1. 불변 변수, 불변 컬렉션
val listA = listOf(1, 2, 3)
// listA.add(4) // 컴파일 오류: 불변 컬렉션
// listA = listOf(4, 5) // 컴파일 오류: 불변 변수

// 2. 가변 변수, 불변 컬렉션
var listB = listOf(1, 2, 3)
// listB.add(4) // 컴파일 오류: 불변 컬렉션
listB = listOf(4, 5) // 성공: 변수 재할당 가능

// 3. 불변 변수, 가변 컬렉션
val listC = mutableListOf(1, 2, 3)
listC.add(4) // 성공: 컬렉션 내용 변경 가능
// listC = mutableListOf(4, 5) // 컴파일 오류: 불변 변수

// 4. 가변 변수, 가변 컬렉션
var listD = mutableListOf(1, 2, 3)
listD.add(4) // 성공: 컬렉션 내용 변경 가능
listD = mutableListOf(4, 5) // 성공: 변수 재할당 가능
```

kotlinx.collections.immutable 라이브러리: 진정한 불변성

- kotlinx.collections.immutable 라이브러리를 도입: 진정한 불변성을 보장하여, 일단 생성되면 절대 수정될 수 없는 컬렉션 타입을 제공
- 구조적 공유(Structural Sharing)
 - 일반적으로 내용이 변경될 때마다 전체 컬렉션의 복사본을 생성, 대규모 컬렉션에서 매우 비효율적
 - 그러나 kotlin의 영속적인(persistent) 컬렉션은 변경되지 않은 원본 컬렉션의 부분을 재사용하여 새로운 컬렉션을 생성
- 불변 컬렉션의 이점
 - 다중 스레드 안전성: 경쟁 상태(race conditions)가 발생하지 않아 복잡한 동기화 메커니즘이 필요 없음
 - Jetpack Compose 상태 관리: 데이터가 실제로 변경되었을 때만 재구성을 트리거하여 UI 성능을 최적화

함수형 API: 데이터 변환 및 집계

map 함수: 데이터 변환의 예술

- map 함수는 컬렉션의 각 요소에 변환 함수(transform function)를 적용하고, 그 결과로 구성된 **새로운** 컬렉션을 반환
 - 원본 컬렉션을 변경하지 않고 데이터를 변환하는 데 사용
- filter나 reduce와 같은 다른 컬렉션 함수와 연속적으로 연결(chaining)하여 복잡한 데이터 처리 파이프라인을 구축할 수 있음
 - forEach와 같은 부수 효과를 위한 함수와 구별되는 중요한 특징

map 함수: 데이터 변환의 예술

```
// 정수 리스트를 문자열 리스트로 변환
val numbers = listOf(1, 2, 3, 4)
val doubled = numbers.map { it * 2 } // [1, 2, 3, 4]
println(doubled)

// 맵의 각 엔트리를 포매팅된 문자열로 변환
val peopleToAge = mapOf("Alice" to 20, "Bob" to 21)
val descriptions = peopleToAge.map { (name, age) -> "$name is $age years old" }
println(descriptions)
```

filter 함수: 정밀한 요소 선택


- filter 함수는 주어진 조건(predicate)을 만족하는 요소들로 구성된 **새로운 컬렉션**을 반환
 - 새로운 컬렉션을 반환하기 때문에 다른 컬렉션 함수와 연결하여 사용할 수 있음
- filter 함수는 데이터를 즉시 평가(eagerly evaluated)하며, 이는 전체 컬렉션을 순회하여 조건을 만족하는 모든 요소를 새로운 컬렉션에 추가하는 것을 의미
 - 여러 번 연속적으로 사용되면 매 단계마다 중간 리스트가 생성되어 메모리 오버헤드가 발생

filter 함수: 정밀한 요소 선택

```
val words = listOf("apple", "banana", "kiwi", "grape")
val shortWords = words.filter { it.length <= 4 }
println(shortWords) // [kiwi]

// filter를 활용한 예시
val data = listOf(1, 10, 20, 30, 45, 55, 60, 70)
val filteredData = data.filter { it > 40 }.map { it * 10 }
println(filteredData) //
```

forEach 함수: 반복과 부수 효과

- forEach 함수는 컬렉션의 각 요소에 대해 지정된 동작을 실행
- forEach는 아무것도 반환하지 않고 Unit을 반환  forEach는 연속적으로 연결될 수 없음
- forEach의 주요 목적은 각 요소를 사용하여 로깅, 출력, 또는 외부 상태 변경과 같은 부수 효과(side effects)를 일으키는 것

forEach 함수: 반복과 부수 효과

```
// 각 요소를 출력하는 부수 효과
val fruits = listOf("apple", "banana", "cherry")
fruits.forEach { println("Fruit: $it") }

// forEach는 Unit을 반환하므로 연결할 수 없습니다.
// fruits.forEach {... }.map {... } // 컴파일 오류
```

특징	map	forEach
반환 값	변환된 요소들로 구성된 새로운 컬렉션	Unit (반환 값 없음)
용도	데이터를 변환하거나 매핑하는 작업	부수 효과(출력, 로깅 등)를 위한 반복 작업
연결 가능성	가능 (새로운 컬렉션을 반환하므로)	불가능 (Unit을 반환하므로)

데이터 집계: reduce vs. fold

- reduce와 fold는 모두 컬렉션을 단일 값으로 집계하는 강력한 함수
- reduce
 - 동작 방식: 컬렉션의 첫 번째 요소가 누적자(accumulator)의 초기값, 이후 누적된 값과 다음 요소를 결합하는 연산을 반복하여 최종 결과 도출
 - 타입 제한: 누적자와 컬렉션 요소의 타입이 동일해야 합니다.
 - 안전성 문제: 빈 컬렉션에 대해 reduce를 호출하면 초기값이 없기 때문에 RuntimeException이 발생
- fold
 - 동작 방식: fold는 개발자가 명시적인 초기값을 인자로 전달, 이 초기값을 시작으로 누적자와 다음 요소를 결합하는 연산을 반복
 - 타입 유연성: 누적자의 타입이 컬렉션 요소의 타입과 달라도 됨. 예를 들어 정수 리스트를 문자열로 변환하는 등의 복잡한 변환이 가능
 - 안전성: 빈 컬렉션에 대해 fold를 호출하면, 예외를 던지는 대신 제공된 초기값을 반환. 빈 컬렉션 상황에서도 안전

데이터 집계: reduce vs. fold

```
val numbers = listOf(1, 2, 3, 4, 5)

// reduce 예시: 합계 계산
val sum = numbers.reduce { acc, next -> acc + next }
println("Sum with reduce: $sum") // 15

// 빈 리스트에 reduce를 사용하면 예외 발생
// emptyList<Int>().reduce { acc, next -> acc + next } // throws
// RuntimeException

// fold 예시: 초기값 0으로 합계 계산
val sumWithFold = numbers.fold(0) { acc, next -> acc + next }
println("Sum with fold: $sumWithFold") // 15

// fold 예시: 결과 타입이 다른 경우
val sentence = numbers.fold("Numbers: ") { acc, next -> "$acc$next" }
println("Sentence with fold: $sentence") // Numbers: 12345
```

아키텍처 고려 사항 및 최적화

성능의 이면: 자료 구조 분석

- HashSet과 LinkedHashMap과 같은 해시 기반 컬렉션은 키를 이용한 빠른 탐색($O(1)$)이 필요한 경우에 적합
- 반면, ArrayList는 인덱스를 통해 빠르게 요소에 접근해야 할 때 이상적
- LinkedHashMap과 LinkedHashSet은 순서 유지를 위해 추가적인 메모리와 오버헤드를 감수
 - 예측 가능한 반복 순서를 제공

성능의 이면: 자료 구조 분석

자료 구조	인덱스 접근(get)	끝에 추가/제거	중간에 추가/제거	값으로 탐색(find)
ArrayList	$O(1)$	$O(1)$ (평균)	$O(n)$	$O(n)$
ArrayDeque	$O(n)$	$O(1)$	$O(n)$	$O(n)$
HashSet	$O(n)$	$O(1)$ (평균)	N/A	$O(1)$ (평균)
LinkedHashMap	N/A	N/A	N/A	$O(1)$ (평균)

즉시 평가 vs. 지연 평가: Sequence의 힘

- map이나 filter와 같은 표준 컬렉션 함수는 즉시 평가(eager evaluation) 방식
 - 즉, 각 함수가 실행될 때마다 전체 컬렉션을 처리하고 중간 결과를 담은 새로운 컬렉션을 생성
 - 대규모 데이터셋에 대해 여러 함수를 연쇄적으로 호출할 경우, 불필요한 중간 컬렉션 생성으로 인해 심각한 메모리 및 성능 병목 현상을 초래
- Kotlin은 지연 평가(lazy evaluation) 방식을 사용하는 Sequence를 제공
 - Sequence는 컬렉션이 아니라 일련의 요소를 나타내는 스트림
 - Sequence를 사용한 연산은 toList()나 first()와 같은 종료 연산(terminal operation)이 호출될 때까지 실제 계산을 시작하지 않음
 - 가장 중요한 점은, Sequence는 각 요소를 순서대로 처리하여 체인 내의 모든 연산을 한 요소에 대해 먼저 적용한 후 다음 요소로 넘어간다는 점

즉시 평가 vs. 지연 평가: Sequence의 힘

```
// 즉시 평가 방식의 리스트
val list = (1..1_000_000).toList()
val listResult = list
    .filter { it % 2 == 0 } // 첫 번째 중간 리스트 생성 (50만개)
    .filter { it % 11 == 0 } // 두 번째 중간 리스트 생성 (~4.5만개)
    .take(3) // 최종 결과 (3개)

// 지연 평가 방식의 시퀀스
val sequenceResult = list.asSequence()
    .filter { it % 2 == 0 } // 조건이 만족되면 바로 다음 연산으로
    넘어감
    .filter { it % 11 == 0 }
    .take(3) // 3개를 찾으면 즉시 중단
    .toList() // 최종 결과를 리스트로 변환

// 결과: sequence는 3개의 요소를 찾는 순간 연산을 멈추므로, 리스트
// 방식보다 훨씬 빠르고 메모리 효율적입니다.
```

즉시 평가 vs. 지연 평가: Sequence의 힘

- 그러나 작은 데이터셋의 경우, Sequence를 설정하는 데 드는 약간의 오버헤드가 지연 평가의 이점을 상쇄
- first()와 같이 조기 종료(early termination)가 가능한 연쇄 작업에 가장 큰 성능 이점

불변성 패러다임: 애플리케이션 개발 모범 사례

- 불변성은 코드의 안정성과 유지보수성을 극적으로 향상시키는 아키텍처적 패러다임
- 핵심적인 모범 사례는 "기본적으로 불변, 필요할 때만 가변"으로 요약
 - 개발자는 `val`과 `listOf()`, `setOf()`, `mapOf()`와 같은 불변 컬렉션 생성 함수를 기본값으로 사용해야
 - 인플레이스(`in-place`) 수정이 필수적이고, 그 부수 효과를 명확하게 관리할 수 있는 경우에만 `var` 또는 `mutable*Of()`와 같은 가변 컬렉션을 사용해야

개발자를 위한 전략적 의사결정 매트릭스

요구사항	최적의 선택	이유
순서가 있고 중복이 허용되는 데이터	List (listOf로 생성)	순서 기반 접근이 필요하고, 중복을 관리하기 위해
고유한 요소만 필요	Set (setOf로 생성)	데이터 무결성을 보장하고 중복을 자동으로 제거하기 위해
키-값 쌍의 관계	Map (mapOf로 생성)	키를 통해 효율적인 데이터 검색 및 관리 가능
빈번한 인플레이스(in-place) 수정	MutableList / MutableSet / MutableMap	성능 저하 없이 동적으로 요소 추가/제거가 필요할 때
다중 스레드 환경에서 데이터 공유	kotlinx.collections.immutable 라이브러리	복잡한 동기화 없이 진정한 스레드 안전성 및 데이터 무결성을 보장하기 위해
효율적인 키 기반 탐색	HashSet 또는 LinkedHashSet (Set)	해시 기반 자료 구조가 평균 상수 시간($O(1)$) 탐색을 제공하기 위해
긴 연산 체인과 대규모 데이터셋	Sequence (asSequence로 변환)	불필요한 중간 컬렉션 생성을 피하고, 메모리와 성능을 최적화하기 위해
간단한 반복과 부수 효과	forEach	반환 값이 필요 없는 부수 효과를 위한 선언적인 반복을 위해
간단한 데이터 집계 (비어있지 않음)	reduce	명시적인 초기값이 필요 없는, 동일 타입의 단순 집계를 위해
복잡한 데이터 집계 (타입 변화, 안전성)	fold	초기값 지정, 타입 유연성, 빈 컬렉션에 대한 안전성을 보장하기 위해