10. 컬렉션 프레임워크

List: `ArrayList`, `LinkedList`

🟦 1. `List` 인터페이스란?

- **순서가 있는 데이터 집합**
- **중복 요소 허용**
- 인덱스를 이용한 요소 접근 가능

주요 구현체:

- `ArrayList` → **배열 기반**
- `LinkedList` → **이중 연결 리스트 기반**
- `Vector`, `Stack` 등도 존재하지만 실무에선 잘 안 씀

🟩 2. ArrayList

> 동적 배열(Dynamic Array) 기반 리스트

📌 주요 특징

- 요소 추가 시 내부 배열이 **용량(capacity) 초과되면 크기를 1.5배 확장**
- 인덱스를 통한 **빠른 접근**이 가능
- 중간 삽입/삭제는 느림 (배열 복사 필요)

🔧 기본 사용 예시

```
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");
list.add("C");

System.out.println(list.get(1)); // B
```

✅ 시간 복잡도

연산	시간 복잡도
get(index)	O(1)
add(맨 뒤에)	평균 O(1)
add/remove(중간)	O(n)

🟧 3. LinkedList

> **이중 연결 리스트(Doubly Linked List)** 구조 기반

📌 주요 특징

- 각 요소는 **노드(Node)** 형태 (데이터 + 앞/뒤 참조)
- **삽입/삭제**는 빠름 (링크만 수정)
- **접근 속도는 느림** (순차 탐색 필요)

🔧 기본 사용 예시

```
List<String> list = new LinkedList<>();
list.add("A");
list.add("B");
list.add("C");

System.out.println(list.get(1)); // B (하지만 내부적으로 순차 탐색)
```

✅ 시간 복잡도

연산	시간 복잡도
get(index)	O(n)
add/remove(앞/중간/뒤)	O(1) ~ O(n) (노드 탐색 필요)

🟨 4. 공통 메서드

메서드	설명
`add(E e)`	요소 끝에 추가
`add(int i, E e)`	지정 위치에 삽입
`get(int i)`	i번째 요소 조회
`set(int i, E e)`	i번째 요소 수정
`remove(int i)`	i번째 요소 제거
`contains(Object o)`	포함 여부 확인
`size()`	크기 반환

🟥 5. ArrayList vs LinkedList 비교 요약

항목	ArrayList	LinkedList
내부 구조	동적 배열	이중 연결 리스트
인덱스 접근 속도	매우 빠름 (O(1))	느림 (O(n))
중간 삽입/삭제	느림 (O(n), 배열 이동)	빠름 (O(1) 링크 수정)
메모리 사용량	적음	많음 (노드마다 포인터 2개 추가)
순차 접근 성능	빠름	나쁠 수 있음
용도	접근/읽기 중심	삽입/삭제 중심

🟦 6. 선택 기준

상황	추천
데이터 접근이 많음 (`get(i)`)	✅ `ArrayList`
삽입/삭제가 빈번함	✅ `LinkedList`
리스트 끝에 추가만 함	`ArrayList`가 효율적
큐/스택처럼 앞뒤 입출력 위주	`LinkedList`가 유리 (`Deque` 지원)

🔧 실무 팁

- `LinkedList`는 `Queue` 또는 `Deque` 인터페이스로 사용할 경우 유용
- 대부분의 경우 성능과 메모리 효율 때문에 `ArrayList`가 기본 선택
- 반복문 내에서 `get(i)`을 자주 호출하면 `LinkedList` 성능 **치명적**

Set: `HashSet`, `TreeSet`, `LinkedHashSet`

🟦 1. `Set` 인터페이스 개요

- **중복을 허용하지 않는 자료구조**
- 순서 보장 여부는 구현체마다 다름
- 내부적으로는 각기 다른 구조(Hash Table, Tree, Linked List)를 가짐

주요 구현체:

구현체	내부 구조	정렬 여부	순서 유지
`HashSet`	해시 테이블	❌	❌

구현체	내부 구조	정렬 여부	순서 유지
`LinkedHashSet`	해시 테이블 + 링크드리스트	❌	✅
`TreeSet`	이진 탐색 트리(Red-Black Tree)	✅ (정렬)	✅ (정렬 순서)

🟩 2. HashSet

> 가장 일반적인 Set 구현체. 내부적으로 `HashMap`을 사용.

📌 특징

- 요소의 **중복을 허용하지 않음**
- **순서를 보장하지 않음**
- 가장 빠른 성능 제공 (검색/삽입/삭제 O(1))

🔧 사용 예시

```
Set<String> set = new HashSet<>();
set.add("apple");
set.add("banana");
set.add("apple"); // 중복, 무시됨

System.out.println(set); // 순서 보장 X
```

🟧 3. LinkedHashSet

> `HashSet`의 특성을 유지하면서 **입력 순서를 유지**함

📌 특징

- `HashSet` + **삽입 순서 기억**
- 순서가 중요한 경우 유용

🔧 사용 예시

```
Set<String> set = new LinkedHashSet<>();
set.add("apple");
set.add("banana");
set.add("cherry");

System.out.println(set); // [apple, banana, cherry]
```

🟧 4. TreeSet

> 내부적으로 **이진 탐색 트리(Red-Black Tree)** 구조를 사용

📌 특징

- 자동으로 **정렬된 상태**로 저장됨
- 중복 허용 X, **정렬 기준 필요**
- 기본적으로 `Comparable`, 또는 `Comparator` 필요

🔧 사용 예시

```
Set<String> set = new TreeSet<>();
set.add("banana");
set.add("apple");
set.add("cherry");

System.out.println(set); // [apple, banana, cherry] (정렬됨)
```

📌 사용자 정의 객체 사용 시:

```
Set<User> users = new TreeSet<>(Comparator.comparing(User::getAge));
```

🟥 5. 성능 비교 요약

연산	HashSet	LinkedHashSet	TreeSet
검색/삽입/삭제	O(1)	O(1)	O(log n)
순서 유지	❌	✅	✅(정렬)
정렬된 순서 제공	❌	❌	✅
메모리 사용량	적음	중간	높음

🧪 6. 정리된 예제 코드 비교

```
Set<Integer> hashSet = new HashSet<>();
Set<Integer> linkedSet = new LinkedHashSet<>();
Set<Integer> treeSet = new TreeSet<>();

for (int val : new int[]{5, 3, 7, 1}) {
    hashSet.add(val);
    linkedSet.add(val);
    treeSet.add(val);
}

System.out.println("HashSet:       " + hashSet);
System.out.println("LinkedHashSet: " + linkedSet);
System.out.println("TreeSet:       " + treeSet);
```

📌 출력 결과 (예시):

```
HashSet:       [1, 3, 5, 7]   // 순서 랜덤
LinkedHashSet: [5, 3, 7, 1]   // 입력 순서
TreeSet:       [1, 3, 5, 7]   // 정렬 순서
```

✅ 요약 정리

용도/조건	추천 Set 유형
순서 불필요, 빠른 성능	✅ `HashSet`
입력 순서 유지 필요	✅ `LinkedHashSet`
정렬된 데이터가 필요할 때	✅ `TreeSet`
사용자 정의 객체 정렬	`TreeSet` + `Comparator`

Map: `HashMap`, `TreeMap`, `LinkedHashMap`

🟦 1. Map 인터페이스 개요

- **키-값 쌍(key-value pair)** 구조
- 키는 **중복 불가**, 값은 **중복 허용**
- 컬렉션 프레임워크의 핵심 자료구조
- 주요 구현체:
  - `HashMap`: 일반적인 해시 테이블 기반 맵
  - `LinkedHashMap`: 입력 순서 유지
  - `TreeMap`: 자동 정렬된 맵

🟩 2. HashMap

> 가장 널리 쓰이는 일반 맵

📌 특징

- **순서 없음**
- **null 키 1개**, **null 값 여러 개** 허용
- 내부적으로 **배열 + 연결 리스트 + 트리** 구조 (Java 8부터)

🔧 기본 예제

```
Map<String, Integer> map = new HashMap<>();
map.put("apple", 3);
map.put("banana", 2);
map.put("cherry", 5);

System.out.println(map.get("banana")); // 2
```

✅ 시간 복잡도

연산	평균 시간 복잡도
삽입/검색/삭제	O(1)

🟧 3. LinkedHashMap

> `HashMap` + **입력 순서 유지**

📌 특징

- **입력된 순서 유지**
- LRU 캐시 등에서 사용 가능 (`accessOrder = true`)
- 내부적으로 **해시 테이블 + 더블 링크드 리스트**

🔧 예제

```
Map<String, Integer> map = new LinkedHashMap<>();
map.put("apple", 3);
map.put("banana", 2);
map.put("cherry", 5);

System.out.println(map); // {apple=3, banana=2, cherry=5}
```

🟧 4. TreeMap

> 자동으로 정렬되는 맵 (기본 정렬 또는 사용자 지정 정렬)

📌 특징

- **키 기준 정렬**
- 기본적으로 `Comparable`, 아니면 `Comparator` 필요
- 내부적으로 **Red-Black Tree** 사용

🔧 예제

```
Map<String, Integer> map = new TreeMap<>();
map.put("banana", 2);
map.put("apple", 3);
map.put("cherry", 5);

System.out.println(map); // {apple=3, banana=2, cherry=5}
```

정렬 기준을 바꾸려면?

```
Map<String, Integer> map = new TreeMap<>(Comparator.reverseOrder());
```

🟥 5. 주요 기능 비교

기능/구현체	HashMap	LinkedHashMap	TreeMap
내부 구조	해시 테이블	해시 테이블 + 링크드 리스트	레드블랙 트리
키 정렬	❌	❌	✅
순서 유지	❌	✅ (입력 순서)	✅ (정렬 순서)
null 키 허용	✅ (1개)	✅ (1개)	❌ (예외 발생)
동기화 여부	❌	❌	❌
평균 접근 시간	O(1)	O(1)	O(log n)
메모리 사용량	낮음	중간	높음

🔧 정리된 예제

```
Map<String, Integer> hashMap = new HashMap<>();
Map<String, Integer> linkedMap = new LinkedHashMap<>();
Map<String, Integer> treeMap = new TreeMap<>();

for (String fruit : new String[]{"banana", "apple", "cherry"}) {
    hashMap.put(fruit, 1);
    linkedMap.put(fruit, 1);
    treeMap.put(fruit, 1);
}

System.out.println("HashMap:       " + hashMap);
System.out.println("LinkedHashMap: " + linkedMap);
System.out.println("TreeMap:       " + treeMap);
```

출력 예시 (정렬 또는 순서 확인):

```
HashMap:       {cherry=1, banana=1, apple=1}
LinkedHashMap: {banana=1, apple=1, cherry=1}
TreeMap:       {apple=1, banana=1, cherry=1}
```

✅ 실무에서의 선택 기준

요구 조건	추천 Map 타입
가장 빠른 성능이 필요할 때	✅ `HashMap`
순서 보장이 필요한 경우	✅ `LinkedHashMap`
정렬된 데이터를 유지해야 할 때	✅ `TreeMap`

Queue: `PriorityQueue`, `Deque`

🟦 1. `Queue` 인터페이스 개요

- 선입선출(FIFO: First-In-First-Out) 자료구조
- `java.util.Queue` 인터페이스는 여러 구현체 존재:
  - `LinkedList`: 일반 큐, 양방향 큐로도 사용 가능
  - `PriorityQueue`: 우선순위 큐
  - `ArrayDeque`: 빠른 비선형 큐 (스택/큐 대체)
  - `ConcurrentLinkedQueue`: 멀티스레드 환경

이번엔 그중에서도 중요한 두 가지:

🟩 2. `PriorityQueue`

> 내부적으로 **최소 힙(Min-Heap)** 구조로 구성된 **우선순위 큐**

📌 특징

- 기본적으로 **오름차순 정렬** (최소값 우선)
- 정렬 기준은 `Comparable` 또는 `Comparator`
- 삽입(`offer`)과 삭제(`poll`)의 시간 복잡도는 **O(log n)**
- null 요소 저장 ❌ 불가

🔧 기본 사용

```
Queue<Integer> pq = new PriorityQueue<>();
pq.offer(30);
pq.offer(10);
pq.offer(20);

while (!pq.isEmpty()) {
    System.out.println(pq.poll()); // 10, 20, 30
}
```

🔧 사용자 정의 정렬 (내림차순)

```
Queue<Integer> maxHeap = new PriorityQueue<>(Comparator.reverseOrder());
maxHeap.offer(30);
maxHeap.offer(10);
maxHeap.offer(20);

System.out.println(maxHeap.poll()); // 30
```

🔧 객체 우선순위 예제

```
class Task {
    String name;
    int priority;
    public Task(String name, int priority) {
        this.name = name;
        this.priority = priority;
    }
}
Queue<Task> taskQueue = new PriorityQueue<>(
    Comparator.comparingInt(t -> t.priority)
);
```

🟧 3. `Deque` (`ArrayDeque`, `LinkedList`)

> **Double Ended Queue** — 양쪽에서 삽입/삭제 가능한 자료구조

📌 특징

- `Queue` + `Stack`의 기능을 동시에 제공
- 구현체로는 `ArrayDeque` 또는 `LinkedList`
- **ArrayDeque**는 스택/큐 모두보다 빠름 (성능 우수)
- null 저장 ❌ 불가

📌 주요 메서드

방향	삽입	삭제	조회
앞	`addFirst`, `offerFirst`	`removeFirst`, `pollFirst`	`peekFirst`
뒤	`addLast`, `offerLast`	`removeLast`, `pollLast`	`peekLast`

🔧 기본 예제

```
Deque<String> deque = new ArrayDeque<>();
deque.addFirst("a"); // 앞에 삽입
deque.addLast("b");  // 뒤에 삽입
deque.addLast("c");

System.out.println(deque.pollFirst()); // a
System.out.println(deque.pollLast());  // c
```

🔧 스택처럼 사용

```
Deque<String> stack = new ArrayDeque<>();
stack.push("first");
stack.push("second");
System.out.println(stack.pop()); // second
```

🟥 4. `PriorityQueue` vs `Deque` 비교

비교 항목	PriorityQueue	Deque (`ArrayDeque`)
구조	최소 힙 (기본 정렬)	원형 배열
정렬 기능	✅ (자동 정렬됨)	❌
삽입/삭제 위치	중간 기준 없음 (우선순위 기준)	앞/뒤 모두 조작 가능
null 허용 여부	❌	❌
스택처럼 사용	불가	가능 (`push`, `pop`)

비교 항목	PriorityQueue	Deque (`ArrayDeque`)
순서 유지 여부	❌	✅
시간 복잡도	O(log n)	O(1) (평균)

✅ 실무 활용 요약

상황	추천 자료구조
우선순위에 따라 정렬된 처리 필요 시	✅ `PriorityQueue`
일반적인 큐 구조, 스택도 함께 필요할 때	✅ `ArrayDeque`
양쪽에서 자유롭게 조작해야 할 때	✅ `Deque` (`LinkedList` or `ArrayDeque`)

반복자 (`Iterator`, `ListIterator`)

🟦 1. Iterator 인터페이스

> 컬렉션 프레임워크의 **기본 반복자**

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    default void remove(); // 선택적으로 지원
}
```

✅ 사용 가능한 컬렉션

- `List`, `Set`, `Queue`, `Deque`, `Map.keySet()`, `Map.values()` 등 대부분

🔧 사용 예제

```
List<String> list = List.of("A", "B", "C");
Iterator<String> iter = list.iterator();

while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

🔧 remove() 예제

```
List<String> list = new ArrayList<>(List.of("a", "b", "c"));
Iterator<String> it = list.iterator();

while (it.hasNext()) {
    if (it.next().equals("b")) {
        it.remove(); // 안전한 삭제
    }
}
System.out.println(list); // [a, c]
```

❌ 주의: for-each 에서 remove 불가

```
for (String s : list) {
    list.remove(s); // ConcurrentModificationException 발생 가능
}
```

🟧 2. ListIterator 인터페이스

> `List` 전용 양방향 반복자

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void set(E e);    // 마지막 반환 요소를 교체
    void add(E e);    // 현재 위치에 요소 삽입
}
```

✅ 사용 가능한 컬렉션

- `List` 계열만 (`ArrayList`, `LinkedList`, `Vector` 등)

🔧 기본 예제

```
List<String> list = new ArrayList<>(List.of("one", "two", "three"));
ListIterator<String> iter = list.listIterator();

while (iter.hasNext()) {
    System.out.print(iter.next() + " ");  // one two three
}
while (iter.hasPrevious()) {
    System.out.print(iter.previous() + " ");  // three two one
}
```

🔧 set(), add() 예제

```
List<String> list = new ArrayList<>(List.of("a", "b", "d"));
ListIterator<String> iter = list.listIterator();

while (iter.hasNext()) {
    String val = iter.next();
    if (val.equals("b")) {
        iter.set("B");         // b → B
        iter.add("c");         // add "c" after "B"
    }
}
System.out.println(list); // [a, B, c, d]
```

🟥 3. Iterator vs ListIterator 비교

기능	`Iterator`	`ListIterator`
지원 컬렉션	모든 컬렉션	오직 `List` 계열
순회 방향	단방향 (→)	양방향 (↔)
삭제 (`remove`)	✅ 지원	✅ 지원
삽입 (`add`)	❌	✅ 지원
수정 (`set`)	❌	✅ 지원
인덱스 접근	❌	✅ (`nextIndex()`, `previousIndex()`)

🟩 실무 팁 요약

- 컬렉션 전체 순회만 하면 `Iterator`로 충분함
- 리스트에서 **중간 삽입/삭제/수정** 필요하면 `ListIterator` 써야 함
- `for-each` 문은 내부적으로 `Iterator`를 쓰지만 **remove 불가**
- `Map`은 직접 `Iterator`를 제공하진 않지만, `entrySet()` 등으로 가능

```
Map<String, Integer> map = new HashMap<>();
map.put("a", 1);
map.put("b", 2);

Iterator<Map.Entry<String, Integer>> iter = map.entrySet().iterator();
while (iter.hasNext()) {
    Map.Entry<String, Integer> entry = iter.next();
    System.out.println(entry.getKey() + " = " + entry.getValue());
}
```

컬렉션 정렬 (`Comparable`, `Comparator`)

🟦 1. 정렬이 필요한 이유

Java의 `Collections.sort()`, `Arrays.sort()` 등 정렬 메서드는
객체의 **정렬 기준**이 명확해야 동작할 수 있어.
→ 이를 위해 두 가지 인터페이스를 제공함:

목적	인터페이스	정렬 기준 제공 방식
기본 정렬 기준	`Comparable<T>`	객체 자체가 비교 기준을 가짐
외부 정렬 기준 제공	`Comparator<T>`	정렬 시점에 외부에서 제공

🟩 2. `Comparable<T>`: 객체 스스로 비교

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

✅ 규칙

- `a.compareTo(b)` 결과:
  - 음수: a < b
  - 0: a == b
  - 양수: a > b

🔧 예제

```
class Student implements Comparable<Student> {
    String name;
    int score;

    Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    @Override
    public int compareTo(Student other) {
        return this.score - other.score; // 오름차순
    }
}
```

```
List<Student> list = new ArrayList<>();
list.add(new Student("Alice", 90));
list.add(new Student("Bob", 80));

Collections.sort(list); // Comparable 기준 사용
```

🟧 3. `Comparator<T>`: 외부에서 비교 기준 제공

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

🔧 예제 1: 이름순 정렬

```
Comparator<Student> byName = new Comparator<>() {
    public int compare(Student a, Student b) {
        return a.name.compareTo(b.name);
    }
};
Collections.sort(list, byName);
```

🔧 예제 2: 람다로 작성

```
Collections.sort(list, (a, b) -> b.score - a.score); // 점수 내림차순
```

🔧 예제 3: 메서드 체이닝

```
list.sort(
    Comparator.comparing(Student::getName)
              .thenComparingInt(Student::getScore)
);
```

🟥 4. 정리 비교

비교 항목	`Comparable`	`Comparator`
비교 기준 위치	클래스 내부	클래스 외부
구현 위치	객체 자체에 구현	별도 클래스 or 람다
하나의 기준만 사용 가능	✅ 한 가지만 정의 가능	✅ 여러 기준 정의 가능
정렬 기준 변경	어렵다	유연하게 바꿀 수 있다
기본 메서드	`compareTo(T o)`	`compare(T o1, T o2)`
사용 예시	`Collections.sort(list)`	`Collections.sort(list, comp)`

🟪 5. 실무 활용 패턴 요약

객체 정렬 기준이 고정되어 있고, 한 가지뿐이라면?

→ `Comparable` 사용

다양한 기준으로 정렬하거나 기준을 동적으로 바꾸고 싶다면?

→ `Comparator` 사용

✅ 부가 정보

Java 8+에서의 정렬 문법

```
list.sort(Comparator.comparing(Student::getScore));
```

역순 정렬

```
Comparator.reverseOrder();
Comparator.comparing(Student::getScore).reversed();
```

null-safe 정렬

```
Comparator.nullsFirst(Comparator.comparing(Student::getName));
```