8.1 컬렉션 팩토리

기존의 적은 요소를 포함하는 리스트를 만드는 코드

```
List<String> friends = new ArrayList<>();

friends.add("Raphael");

friends.add("Olivia");

friends.add("Thibaut");
```

다음처럼 Arrays.asList() 팩토리 메서드를 이용하면 코드를 간단하게 줄일 수 있다.

```
List<String> friends = Arrays.asList("Raphael", "Olivia", "Thibaut");
```

고정 크기의 리스트를 만들었으므로 요소를 갱신할 순 있지만 새 요소를 추가하거나 요소를 삭제할 순 없다.

예를 들어 요소를 갱신하는 작업은 괜찮지만 요소를 추가하려 하면 UnsupportedOperationException이 발생한다.

```
List<String> friends = Arrays.asList("Raphael", "Olivia", "Thibaut");
friends.set(0, "Richard");
try {
        friends.add("Thibaut");
} catch (UnsupportedOperationException e) {
        e.printStackTrace();
}
```

내부적으로 고정된 크기의 변환할 수 있는 배열로 구현되었기 때문에 이와 같은 일이 일어납니다.

집합의 경우

```
Set<String> friends = new HashSet<>(Arrays.asList("Raphael", "Olivia",
"Thibaut"));
Set<String> friends = Stream.of("Raphael", "Olivia", "Thibaut")
.collect(Collectors.toSet());
```

하지만 두 방법 모두 매끄럽지 못하며 내부적으로 불필요한 객체 할당을 필요로 합니다.

그리고 결과는 변환할 수 있는 집합이라는 사실에 주목해봅니다.

Java 9에서 작은 리스트, 집합, 맵을 쉽게 만들 수 있도록 팩토리 메서드를 제공합니다.

8.1.1 리스트 팩토리

List.of 팩토리 메서드를 이용해서 간단하게 리스트를 만들 수 있습니다.

```
List<String> friends = List.of("Raphael", "Olivia", "Thibaut");
```

friends 리스트에 요소를 추가하면, UnsupportedOperationException이 발생합니다. 변경할 수 없는 리스트가 만들어졌기 때문입니다.

컬렉션이 의도치 않게 변하는 것을 막을 수 있기 때문입니다. 리스트를 바꿔야 하는 상황이라면 직접 리스트를 만들면 됩니다.

마지막으로 null 요소는 금지하므로 의도치 않은 버그를 방지하고 조금 더 간결한 내부 구현을 달성했습니다.

Collectors.toList() 컬렉터로 스트림을 리스트로 변환할 수 있다.

데이터 처리 형식을 설정하거나 데이터를 변환할 필요가 없다면 사용하기 간편한 팩토리 메서드를 이용할 것을 권장한다.

오버로딩 vs 가변 인수

List.of의 다양한 오버로드 버전이 있다는 사실을 알 수 있습니다.

```
static <E> List<E> of(E e1, E e2, E e3, E e4)
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5)
```

아마 여러분은 왜 다음처럼 다중 요소를 받을 수 있도록 자바 API를 만들지 않은 것인지 궁금할 것입니다.

```
static <E> List<E> of(E...elements)
```

내부적으로 가변 인수 버전은 **추가 배열을 할당해서 리스트로 감쌉니다.** 따라서 배열을 할당하고 초기화하며 나중에 가비지 컬렉션을 하는 비용을

지불해야 합니다. 고정된 숫자의 요소(최대 열개까지)를 API로 정의하므로 이런 비용을 제거할 수 있습니다.

List.of로 열 개 이상의 요소를 가진 리스트를 만들 수도 있지만 이 때는 가변 인수를 이용하는 메서드가 사용됩니다.

8.1.2 집합 팩토리

List.of와 비슷한 방법으로 바꿀 수 없는 집합을 만들 수 있습니다.

```
Set<String> friends = Set.of("Raphael", "Olivia", "Thibaut");
System.out.println(friends);
```

중복된 요소를 제공해 집합을 만들려고 하면 Olivia라는 요소가 중복되어 있다는 설명과 함께 IllegalArgumentException이 발생합니다.

집합은 오직 고유의 요소만 포함할 수 있다는 원칙을 상기시킵니다.

8.1.3 맵 팩토리

맵을 만드는 것은 조금 복잡한데 키와 값이 있어야 하기 때문입니다.

Java 9에서는 두 가지 방법으로 바꿀 수 없는 맵을 초기화할 수 있습니다.

Map.of 팩토리 메서드에 키와 값을 번갈아 제공하는 방법으로 맵을 만들 수 있습니다.

```
Map<String, Integer> ageOfFriends = Map.of("Raphael", 30, "Olivia", 25,
"Thibaut", 26);
System.out.println(ageOfFriends);
```

열 개 이하의 키와 값 쌍을 가진 작은 맵을 만들 때는 이 메서드가 유용합니다.

그 이상의 맵에서는 `Map.Entry<K, V>` 객체를 인수로 받으며 가변 인수로 구현된 Map.ofEntries 팩토리 메서드를 이용하는 것이 좋습니다.

이 메서드는 키와 값을 감쌀 추가 객체 할당을 필요로 합니다.

```
Map<String, Integer> ageOfFriends = Map.ofEntries(entry("Rapheal", 30),
entry("Olivia", 25),
entry("Thibaut", 26));
System.out.println(ageOfFriends);
```

Map.entry는 Map.Entry 객체를 만드는 새로운 팩터리 메서드입니다.

8.2 리스트와 집합 처리

Java 8에서는 List, Set 인터페이스에 다음과 같은 메서드를 추가했습니다.

- removeIf : 프레디케이트를 만족하는 요소를 제거한다. List나 Set을 구현하거나 그 구현을 상속받은 모든 클래스에서 이용할 수 있다.
- replaceAll : 리스트에서 이용할 수 있는 기능으로 UnaryOperator (T -> T)를 이용해 요소를 바꾼다.
- sort : List 인터페이스에서 제공하는 기능으로 리스트를 정렬한다.

이들 메서드는 호출한 컬렉션 결과를 바꾼다.

새로운 결과를 만드는 스트림 동작과 달리 이들 메서드는 기존 컬렉션을 바꾼다. 컬렉션을 바꾸는 동작은 에러를 유발하며 복잡함을 더한다.

Java 8에 removeIf와 replaceAll를 추가한 이유가 바로 이 때문이다.

8.2.1 removeIf 메서드

다음은 숫자로 시작되는 참조 코드를 가진 트랜잭션을 삭제하는 코드입니다.

```
for (Transaction transaction : transactions) {
if (Character.isDigit(transaction.getReferenceCode().charAt(0))) {
        transactions.remove(transaction);
}
}
```

위 코드는 ConcurrentModificationException을 일으킵니다. 내부적으로 for-each 루프는 Iterator 객체를 사용하므로 위 코드는 다음과 같이 해석됩니다.

```
for (Iterator<Transaction> iterator = transactions.iterator();
iterator.hasNext();) {
        Transaction transaction = iterator.next();
        if (Character.isDigit(transaction.getReferenceCode().charAt(0))) {
                transactions.remove(transaction);
        }
}
```

두 개의 개별 객체가 컬렉션을 관리한다는 사실을 주목해봅니다.

- Iterator 객체, next(), hasNext()를 이용해 소스를 질의합니다.
- Collection 객체 자체, remove()를 호출해 요소를 삭제합니다.

결과적으로 반복자의 상태는 컬렉션의 상태와 서로 동기화되지 않습니다. Iterator 객체를 명시적으로 사용하고 그 객체의 remove()

메서드를 호출함으로 이 문제를 해결할 수 있다.

```
for (Iterator<Transaction> iterator = transactions.iterator();
iterator.hasNext();) {
        Transaction transaction = iterator.next();
if (Character.isDigit(transaction.getReferenceCode().charAt(0))) {
        iterator.remove();
}
}
```

이 코드 패턴은 Java 8의 removeIf 메서드로 바꿀 수 있습니다.

removeIf는 삭제할 요소를 가리키는 프레디케이트를 인수로 받습니다.

```
transactions.removeIf(transaction ->
Character.isDigit(transaction.getReferenceCode().charAt(0)));
```

replaceAll 메서드

List 인터페이스의 replaceAll 메서드를 이용해 리스트의 각 요소를 새로운 요소로 바꿀 수 있다.

스트림 API를 사용하면 다음처럼 문제를 해결할 수 있었다.

```
List<String> referenceCodes = Arrays.asList("a12", "C14", "b13");

referenceCodes.stream()
                                .map(code ->
Character.toUpperCase(code.charAt(0)) + code.substring(1))
                                .collect(Collectors.toList())
                                .forEach(System.out::println);
```

하지만 이 코드는 새 문자열 컬렉션을 만듭니다. 우리가 원하는 것은 기존 컬렉션을 바꾸는 것입니다.

```
for (ListIterator<String> iterator =
referenceCodes.listIterator();iterator.hasNext();) {
        String code = iterator.next();
        iterator.set(Character.toUpperCase(code.charAt(0)) +
code.substring(1));
}
System.out.println(referenceCodes);
```

코드가 조금 복잡해졌습니다. Java 8의 기능을 이용하면 다음처럼 간단하게 구현할 수 있습니다.

```
referenceCodes.replaceAll(code ->
        Character.toUpperCase(code.charAt(0)) + code.substring(1));
```

8.3 맵 처리

디폴트 메서드 : 기본적인 구현을 인터페이스에 제공하는 기능

8.3.1 forEach 메서드

```
ageOfFriends.forEach((friendName, age) -> System.out.println(friendName + "
is " + age + " years old"));
```

BiConsumer, 즉 (T,U) -> void 키와 값을 인수로 받는 forEach 메서드를 지원하므로 코드를 조금 더 간단하게 구현할 수 있다.

8.3.2 정렬 메서드

- Entry.comparingByValue
- Entry.comparingByKey

를 이용하면 맵의 항목을 값 또는 기준으로 정렬할 수 있다.

```
Map<String, String> favouriteMovies = Map.ofEntries(
entry("Raphael", "Star Wars"),
entry("Cristina", "Matrix"),
entry("Olivia", "James Bond"));
favouriteMovies.entrySet()
.stream()
.sorted(Entry.comparingByKey())
.forEachOrdered(System.out::println);
```

HashMap 성능

Java8에서는 HashMap의 내부 구조를 바꿔 성능을 개선했다.

기존의 맵의 항목은 키로 생성한 해시코드로 접근할 수 있는 버켓에 저장했다.

많은 키가 같은 해시코드를 반환하는 상황이 되면 O(n)의 시간이 걸리는 LinkedList로 버킷을 반환해야 하므로 성능이 저하된다.

최근에는 버킷이 너무 커질 경우 이를 O(log(n))의 시간이 소요되는 정렬된 트리를 이용해 동적으로 치환해 충돌이 일어나는 요소 반환 성능을 개선했다.

하지만 키가 String, Number 클래스 같은 Comparable의 형태여야만 정렬된 트리가 지원된다.

요청된 키가 맵에 존재하지 않을 때 이를 어떻게 처리하느냐도 흔히 발생하는 문제. 새로 추가된 getOrDefault 메서드를 이용하면 이를 쉽게 해결할 수 있습니다.

8.3.3 getOrDefault 메서드

기존에는 찾으려는 키가 존재하지 않으면 null이 반환되므로 NullPointerException을 방지하려면 요청 결과가 널인지 확인해야 한다. 기본값을 반환하는 방식으로 이 문제를 해결할 수 있다.

이 메서드는 첫 번째 인수로 키를, 두 번째 인수로 기본값을 받으며 맵에 키가 존재하지 않으면 두 번째 인수로 받은 기본값을 반환한다.

```
Map<String, String> favouriteMovies = Map.ofEntries(
entry("Raphael", "Star Wars"),
entry("Cristina", "Matrix"),
entry("Olivia", "James Bond"));
System.out.println(favouriteMovies.getOrDefault("Olivia", "Matrix"));
System.out.println(favouriteMovies.getOrDefault("Thibaut", "Matrix"));
```

키가 존재하더라도 값이 널인 상황에서는 getOrDefault가 널을 반환할 수 있다.

즉, 키가 존재하느냐의 여부에 따라서 두 번째 인수가 반환될지 결정된다.

8.3.4 계산 패턴

키가 존재하는지 여부에 따라 어떤 동작을 실행하고 결과를 저장해야 하는 상황이 필요한 때가 있다.

예를 들어 키를 이용해 값비싼 동작을 실행해서 얻을 결과를 캐시하려 한다. 키가 존재하면 결과를 다시 계산할 필요가 없다.

- computeIfAbsent : 제공된 키에 해당하는 값이 없으면(값이 없거나 널), 키를 이용해 새 값을 계산하고 맵에 추가한다. 키가 존재하면 기존 값을 반환한다.
- computeIfPresent : 제공된 키가 존재하면 새 값을 계산하고 맵에 추가한다.
- compute : 제공된 키로 새 값을 계산하고 맵에 저장한다.

정보를 캐시할 때 computeIfAbsent를 활용할 수 있다. 파일 집합의 각 행을 파싱해 SHA-256을 계산한다고 가정하자. 기존에 이미 데이터를 처리했다면 이 값을 다시 계산할 필요가 없다.

```
lines.forEach(line ->
```

```
dataToHash.computeIfAbsent(line, this::calculateDigest));

private byte[] calculateDigest(String key) {

return messageDigest.digest(key.getBytes(StandardCharsets.UTF_8));

}
```

여러 값을 저장하는 맵을 처리할 때도 이 패턴을 유용하게 활용할 수 있다. `Map<K, List<V>>`에 요소를 추가하려면 항목이 초기화되어 있는지 확인해야 한다.

```
Map<String, List<String>> friendsToMovies = new HashMap<>();


System.out.println("--> Adding a friend and movie in a verbose way");
String friend = "Raphael";
List<String> movies = friendsToMovies.get(friend);
if (movies == null) {
movies = new ArrayList<>();
friendsToMovies.put(friend, movies);
}
movies.add("Star Wars");
System.out.println(friendsToMovies);
```

computeIfAbsent는 키가 존재하지 않으면 값을 계산해 맵에 추가하고 키가 존재하면 기존 값을 반환한다.

```
friendsToMovies.computeIfAbsent("Raphael", name -> new ArrayList<>())
.add("Star Wars");
```

8.3.5 삭제 패턴

Java 8에서는 키가 특정한 값과 연관되었을 때만 항목을 제거하는 오버로드 버전 메서드를 제공한다.

```
favouriteMovies.remove(key, value);
```

8.3.6 교체 패턴

맵의 항목을 바꾸는 데 사용할 수 있는 두 개의 메서드가 맵에 추가되었다.

- replaceAll : BiFunction (T,U) -> R 을 적용한 결과로 각 항목의 값을 교체한다. 이 메서드는 이전에 살펴본 List의 replaceAll과 비슷한 동작을 수행한다.

- Replace : 키가 존재하면 맵의 값을 바꾼다. 키가 특정 값으로 매핑되었을 때만 값을 교체하는 오버로드 버전도 있다.

```
favouriteMovies.replaceAll((friend, movie) -> movie.toUpperCase());
```

지금까지 배운 replace 패턴은 한 개의 맵에만 적용할 수 있다. 두 개의 맵에서 값을 합치거나 바꿔야 한다면 merge 메서드를 이용해야 한다.

8.3.7 합침

다음처럼 putAll 을 사용해서 합칠 수 있다.

```
Map<String, String> family = Map.ofEntries(
entry("Teo", "Star Wars"),
entry("Cristina", "James Bond"));
Map<String, String> friends = Map.ofEntries(entry("Raphael", "Star Wars"));


System.out.println("--> Merging the old way");
Map<String, String> everyone = new HashMap<>(family);
everyone.putAll(friends);
System.out.println(everyone);
```

중복된 키가 없다면 위 코드는 잘 동작한다. 값을 좀 더 유연하게 합쳐야 한다면 새로운 merge 메서드를 이용할 수 있다.

이 메서드는 중복된 키를 어떻게 합칠지 결정하는 BiFunction을 인수로 받는다.

forEach와 merge를 이용해 충돌을 해결할 수 있다.

```
friends2.forEach((k, v) -> everyone2.merge(k, v, (movie1, movie2) -> movie1
+ " & " + movie2));
```

merge는 널값과 관련된 복잡한 상황도 처리한다.

```
지정된 키와 연관된 값이 없거나 값이 널이면 merge는 키는 널이 아닌 값과 연결한다.

아니면 merge는 연결된 값을 주어진 매핑 함수의 결과 값으로 대치하거나 결과가 널이면 항목을 제거한다.
```

merge를 이용해 초기화 검사를 구현할 수도 있다.

영화를 몇 회 시청했는지 기록하는 맵이 있다고 가정하자. 해당 값을 증가시키기 전에 영화가 이미 맵에 존재하는지 확인해야 한다.

```
Map<String, Long> moviesToCount = new HashMap<>();

String movieName = "JamesBond";

Long count = moviesToCount.get(movieName);

if (count == null) {

moviesToCount.put(movieName, 1L);

}

else {

moviesToCount.put(movieName, count + 1);

}
```

위 코드를 다음처럼 구현할 수 있습니다.

```
moviesToCount.merge(movieName, 1L, (key, count) -> count + 1L);
```

위 코드에서 merge의 두 번째 인수는 1L이다.

자바독에 따르면 이 인수는

1. 키와 연관된 기존 값에 합쳐질 널이 아닌 값
2. 값이 없거나
3. 키에 널 값이 연관되어 있다면 이 값을 키와 연결

키의 반환값이 null이므로 처음에는 1이 사용된다. 그 다음에는 값이 1로 초기화되어 있으므로 BiFunction을 적용해 값이 증가된다.

```
default V merge(K key, V value,
BiFunction<? super V, ? super V, ? extends V> remappingFunction) {
Objects.requireNonNull(remappingFunction);
Objects.requireNonNull(value);
V oldValue = get(key);
V newValue = (oldValue == null) ? value :
remappingFunction.apply(oldValue, value);
if (newValue == null) {
remove(key);
} else {
put(key, newValue);
}
return newValue;
}
```

책에 나오는 설명으로는 이해하기 어렵다. 따라서 구현으로 들어가보면 다음과 같다.

get(key)로 꺼낸 value가 null이라면? -> 파라미터의 value를 newValue로 사용한다.

null이 아니라면, (map에 뭔가가 있다면?) -> BiFunction Funtional Interface를 구현한 apply 메서드를 실행해서 newValue로 받는다.

newValue가 null이면 key를 삭제한다.

null이 아니라면 put(key, newValue).

8.4 개선된 ConcurrentHashMap

ConcurrentHashMap은 동시성 친화적이며 최신 기술을 반영한 HashMap 버전이다.

ConcurrentHashMap은 내부 자료구조의 특정 부분만 잠궈 동시 추가, 갱신 작업을 허용한다.

따라서 동기화된 Hashtable 버전에 비해 읽기 쓰기 연산 성능이 월등하다.(참고로 표준 HashMap은 비동기로 동작함)

ConcurrentHashMap의 상태를 잠그지 않고 연산을 수행하는 메서드들 : 이들 연산에 제공한 함수는 계산이 진행되는 동안

바뀔 수 있는 객체, 값, 순서 등에 의존하지 않아야 한다.

또한 이들 연산에 병렬성 기준값(threshold)을 지정해야 한다. 맵의 크기가 주어진 기준값보다 작으면 순차적으로 연산을 실행한다.

기준값을 1로 지정하면 공통 스레드 풀을 이용해 병렬성을 극대화한다.

Long.MAX_VALUE를 기준값으로 설정하면 한 개의 스레드로 연산을 실행한다.

```
ConcurrentHashMap<String, Long> map = new ConcurrentHashMap<>();

map.put("5", 5L);

long parallelismThreshold = 1;

Optional<Long> maxValue =
Optional.ofNullable(map.reduceValues(parallelismThreshold, Long::max));

System.out.println(maxValue.get());
```

ConcurrentHashMap 클래스는 맵의 매핑 개수를 반환하는 mappingCount 메서드를 제공한다.

long을 반환. 그래야 매핑의 개수가 long의 범위를 넘어서는 이후의 상황을 대처할 수 있기 때문이다.

ConcurrentHashMap은 Map에서 상속받은 새 디폴트 메서드를 지원함과 동시에 스레드 안전성도 제공한다.