2. 스트림(stream)

스트림이란?

- JAVA 8.0 부터 추가된 인터페이스
- 컬렉션 또는 배열의 **요소(데이터소스)를 추상화**하여, 요소에 관계 없이 반복된 요소를 처리 할 수 있도록 만들어 놓은 클래스
- 기존 컬렉션 또는 배열을 사용하며 **불편했던 점을 개선**하기 위해 만들어진 **반복자**
- 반복되는 요소를 람다식으로 처리
- 데이터 소스를 감싸는 래퍼로 데이터 소스와 함께 처리되여 대량처리를 편리하고 빠르게할 수 있음
- 순차(sequential), 병렬(parallel) 두 종류가 존재 (sequential 이 기본)

Stream 특징

1. 외부 반복을 통해 작업하는 컬렉션과는 달리 내부 반복(internal iteration)을 통해 작업을 수행한다
2. 재사용이 가능한 컬렉션과는 달리 단 한번만 사용할 수 있다
    - iterator 와 같은 성질
    - 스트림은 컬렉션, 배열, I/O 자원 등의 데이터 제공 소스로 부터 데이터를 **소비(consume)**한다
3. 원본 데이터를 변경하지 않는다
4. 지연(lazy) 연산을 통해 성능을 최적화 한다
    - 중간 성능을 최적화 -> 불필요한 연산을 피하기 때문에
5. 병렬 처리
    - 한가지 작업을 서브 작업으로 나누고, 서브 작업들을 분리된 스레드에서 병렬적으로 처리한 후, 서브 작업들의 결과들을 최종 결합하는 방법
    - 자바는 ForkJoinPool 프레임워크를 이용해서 병렬 처리를 한다
    - parallelStream() 메서드를 통한 손쉬운 병렬 처리를 지원한다
    - 병렬 스트림이 순차 스트림보다 빠르지 않은 경우도 많기때문에, 사용시 주의가 필요하며 필요하다면 성능을 측정 후 사용하는걸 권장한다
    - collect() 메서드는 컬렉션을 합치는 작업의 부담이 큰, 병렬화에 적합하지 않은 대표적인 메서드이다

ex1. 기존 컬렉션과 비교

```java
List<String> names = Arrays.asList("김자바", "이자바", "박자바", "최자바", "이자바");

// collections 로 출력
for (int i = 0; i < names.size(); i++) {
  System.out.println(names.get(i));
}

// stream 로 출력
names.stream().forEach(name -> System.out.println(name));
// 또는 람다의 '메서드 참조'를 사용하여 아래와 같이 줄일 수 있다
names.stream().forEach(System.out::println);
```

```
// 실행 결과
김자바
이자바
박자바
최자바
이자바
// ... 이하 2번 더 반복
```

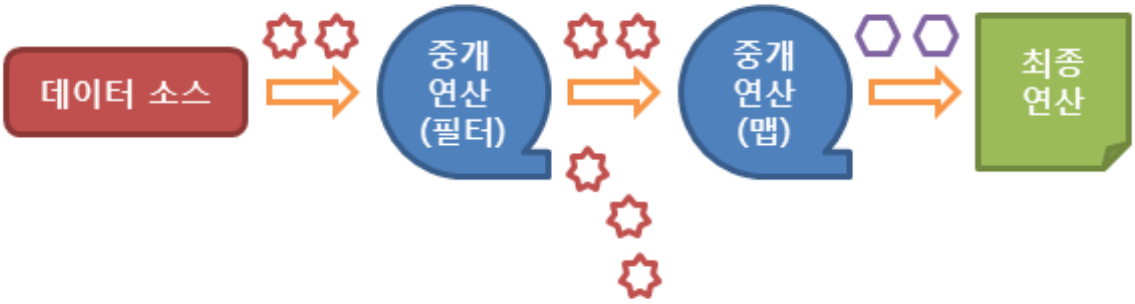
ex2. 만약 위의 예제에서 중복없이 리스트의 이름을 출력하길 원한다면

```java
List<String> names = Arrays.asList("김자바", "이자바", "박자바", "최자바", "이자바");

// collections
List<String> tempNames = new ArrayList<>();
for (int i = 0; i < names.size(); i++) {
  if (!tempNames.contains(names.get(i))) {
    tempNames.add(names.get(i));
    System.out.println(names.get(i));
  }
}

// stream
names.stream().distinct().forEach(name -> System.out.println(name));
// 또는 람다의 '메서드 참조' 사용
names.stream().distinct().forEach(System.out::println);
```

Stream 동작 흐름

1. 스트림의 생성

2. 스트림의 중간 연산 (스트림의 변환)
    - 최종처리 전에 하는 처리. 요소들의 매핑, 필터링, 정렬 등을 처리
    - 연속해서 중간 연산 처리 가능
3. 스트림의 최종 연산 (스트림의 사용)
    - 요소의 반복, 카운트, 평균, 총합 등을 처리
    - 단 한번만 가능



1. 스트림의 생성

다음과 같은 다양한 데이터 소스에서 생성할 수 있다

1. 컬렉션

   컬렉션의 최상위 클래스인 Collection 인터페이스의 stream() 메서드 사용

```
List<Integer> items = new ArrayList<>();
items.add(1);
items.add(2);
items.add(3);
items.add(4);

// 컬렉션에서 스트림 생성
Stream<Integer> integerStream = items.stream();

// 또는
List<Integer> items2 = Arrays.asList(1, 2, 3, 4);
Stream<Integer> integerStream2 = items2.stream();
```

    - parallelStream() 메서드를 사용하면 병렬 처리가 가능한 스트림을 생성 가능

2. 배열

   Arrays 클래스의 stream() 메서드 사용

```
String[] strArray = new String[]{"가", "나", "다", "라", "마"};

// 배열에서 스트림 생성
Stream<String> stream1 = Arrays.stream(strArray);

// 배열의 특정 부분만을 이용한 스트림 생성 - "나, 다"만 들어감
Stream<String> stream2 = Arrays.stream(strArray, 1, 3);
```

- Document 참고

  stream(int[] array)
  Returns a sequential **IntStream** with the specified array as its source.

  stream(int[] array, int startInclusive, int endExclusive)
  Returns a sequential **IntStream** with the specified range of the specified array as its source.

- 기본 타입인 int, long, double 형을 저장할 수 있는 스트림이 따로 제공된다
  - 오토박싱&언박싱으로 인한 비효율을 줄이기 위해 사용
  - IntStream, LongStream, DoubleStream 인터페이스 제공

3. 가변 매개변수

   Stream 클래스의 of() 메서드 사용

```
Stream<Integer> stream1 = Stream.of(3, 2, 1, 0, 8);
Stream<String> stream2 = Stream.of("가", "나", "다", "라");
```

4. 지정된 범위의 연속된 정수

   IntStream과 LongStream 클래스의 range(), rangeClosed() 메서드 사용

```
IntStream intStream1 = IntStream.range(1, 5); // 1,2,3,4
IntStream intStream2 = IntStream.rangeClosed(1, 5); // 1,2,3,4,5
```

- IntStream 에 정의된 range 메서드

  range(int startInclusive, int endExclusive)
  Returns a sequential ordered IntStream from startInclusive (inclusive) to endExclusive (exclusive) by an incremental step of 1.

  rangeClosed(int startInclusive, int endInclusive)
  Returns a sequential ordered IntStream from startInclusive (inclusive) to endInclusive (inclusive) by an incremental step of 1.

5. 특정 타입의 난수들

   Random 클래스의 ints(), longs(), doubles() 메서드 사용

```
IntStream stream1 = new Random().ints(4); // 4개의 난수 생성
IntStream stream2 = new Random().ints(); // 무한 스트림 생성, 사용시점에 limit 함수로 크기 제한
```

6. 람다 표현식

   Stream 클래스의 iterate()와 generate() 메서드 사용

   - 람다식에 의해 반환되는 값을 요소로 하는 무한 스트림을 생성한다

```
// iterate 는 람다식에 의해 계산된 결과를 다시 seed 값으로 해서 계산을 반복
Stream<Integer> stream = Stream.iterate(2, n -> n + 2); // 2, 4, 6, 8, 10,
...

// generate 는 이전 결과를 이용해서 다음요소를 계산하지 않음
// 내부적으로 Supplier<T> 로 구현되어있기 때문, Supplier 는 반환값만 있고 매개변수를 받지
않는다
Stream<Integer> stream4 = Stream.generate(() -> 1); // 1, 1, 1, 1, 1, ...
```

7. 파일

   파일의 한 행 또는 파일의 목록을 요소(데이터소스)로 하는 스트림을 반환

   - Files.lines(Path path) : 파일의 한 행(line)을 요소로 하는 스트림을 생성하여 반환
   - Files.list(Path dir) : 지정된 디렉토리에 있는 파일의 목록을 소스로 하는 스트림을 생성하여 반환

```
Stream<Path> streamFileList = Files.list(Paths.get("/Users/mj/study"));
```

8. 빈 스트림

   아무 요소도 가지지 않는 빈 스트림 생성, Stream 클래스의 empty() 메서드 사용

```
Stream emptyStream = Stream.empty();
```

   - 연산 결과값이 없을 때 null 보다는 빈 스트림을 반환할 것

2. 스트림의 중간 연산

- intermediate operation
  - API 내용 참고
- 초기 스트림은 중간 연산을 통해 또 다른 스트림으로 변환된다
- 스트림을 전달받아 스트림을 반환하므로, 연속으로 연결해서 사용할 수 있다
- 필터-맵(filter-map) 기반의 API를 사용함으로 지연(lazy) 연산을 통해 성능을 최적화할 수 있다

- 대표적인 중간 연산 메서드
  - 필터링 : filter(), distinct()
  - 변환 : map(), flatMap()

- 제한 : limit(), skip()
- 정렬 : sorted()
- 연산 결과 확인 : peek()

메소드	설명
Stream<T> filter(Predicate<? super T> predicate)	해당 스트림에서 주어진 조건(predicate)에 맞는 요소만으로 구성된 새로운 스트림을 반환함.
<R> Stream<R> map(Functoin<? super T, ? extends R> mapper)	해당 스트림의 요소들을 주어진 함수에 인수로 전달하여, 그 반환값으로 이루어진 새로운 스트림을 반환함.
<R> Stream<R> flatMap(Functoin<? super T, ? extends Stream<? extends R>> mapper)	해당 스트림의 요소가 배열일 경우, 배열의 각 요소를 주어진 함수에 인수로 전달하여, 그 반환값으로 이루어진 새로운 스트림을 반환함.
Stream<T> distinct()	해당 스트림에서 중복된 요소가 제거된 새로운 스트림을 반환함. 내부적으로 Object 클래스의 equals() 메소드를 사용함.
Stream<T> limit(long maxSize)	해당 스트림에서 전달된 개수만큼의 요소만으로 이루어진 새로운 스트림을 반환함.
Stream<T> peek(Consumer<? super T> action)	결과 스트림으로부터 각 요소를 소모하여 추가로 명시된 동작(action)을 수행하여 새로운 스트림을 생성하여 반환함.
Stream<T> skip(long n)	해당 스트림의 첫 번째 요소부터 전달된 개수만큼의 요소를 제외한 나머지 요소만으로 이루어진 새로운 스트림을 반환함.
Stream<T> sorted() Stream<T> sorted(Comparator<? super T> comparator)	해당 스트림을 주어진 비교자(comparator)를 이용하여 정렬함. 비교자를 전달하지 않으면 영문사전 순(natural order)으로 정렬함.

스트림 필터링

- filter() : 해당 스트림에서 주어진 조건(predicate)에 맞는 요소만으로 구성된 새로운 스트림을 반환
- distinct() : 해당 스트림에서 중복된 요소가 제거된 새로운 스트림을 반환
  - equals 메서드를 사용하여 요소의 중복을 비교

```
IntStream stream1 = IntStream.of(7, 5, 5, 2, 1, 2, 3, 5, 4, 6);
IntStream stream2 = IntStream.of(7, 5, 5, 2, 1, 2, 3, 5, 4, 6);

// 스트림에서 중복된 요소를 제거함
stream1.distinct().forEach(e -> System.out.print(e + " "));
System.out.println();

// 스트림에서 홀수만을 골라냄
stream2.filter(n -> n % 2 != 0).forEach(e -> System.out.print(e + " "));
```

```
// 실행 결과
7 5 2 1 3 4 6
7 5 5 1 3 5
```

스트림 변환

- map() : 해당 스트림의 요소들을 주어진 함수에 인수로 전달하여, 그 반환값들로 이루어진 새로운 스트림을 반환
- flatMap() : 스트림의 요소가 배열이거나 Object 로 감싸져 있는 모든 원소를 단일 원소 스트림으로 반환

```
// map 예제
Stream<String> stream = Stream.of("HTML", "CSS", "JAVA", "JAVASCRIPT");
stream.map(s -> s.length()).forEach(System.out::println);
```

```
// 실행 결과
4
3
4
10
```

```
// flatMap 예제
String[] arr = {"I study hard", "You study JAVA", "I am hungry"};

Stream<String> stream = Arrays.stream(arr);
stream.flatMap(s -> Stream.of(s.split(" "))).forEach(System.out::println);
```

```
// 실행 결과
I
study
hard
You
study
JAVA
I
am
hungry
```

스트림 제한

- limit() : 해당 스트림의 첫 번째 요소부터 전달된 개수만큼의 요소만으로 이루어진 새로운 스트림을 반환
- skip() : 해당 스트림의 첫 번째 요소부터 전달된 개수만큼의 요소를 제외한 나머지 요소만으로 이루어진 새로운 스트림을 반환

```
IntStream stream1 = IntStream.range(0, 10);
IntStream stream2 = IntStream.range(0, 10);
IntStream stream3 = IntStream.range(0, 10);

// skip
```

```
stream1.skip(4).forEach(n -> System.out.print(n + " "));
System.out.println();

// limit
stream2.limit(5).forEach(n -> System.out.print(n + " "));
System.out.println();

// skip & limit
stream3.skip(3).limit(5).forEach(n -> System.out.print(n + " "));
```

```
// 실행 결과
4 5 6 7 8 9
0 1 2 3 4
3 4 5 6 7
```

스트림 정렬

- sorted() : 해당 스트림을 주어진 비교자(comparator)를 이용하여 정렬
  - 비교자를 전달하지 않으면 기본적으로 natural order 로 정렬 - Arrays, Collections 모두 natural order는 ASC(오름차순)

```
Stream<Integer> stream1 = Stream.of(3, 2, 7, 0, 1);
Stream<Integer> stream2 = Stream.of(3, 2, 7, 0, 1);

stream1.sorted().forEach(s -> System.out.print(s + " "));
System.out.println();

stream2.sorted(Comparator.reverseOrder()).forEach(s -> System.out.print(s
+ " "));
```

```
// 실행 결과
0 1 2 3 7
7 3 2 1 0
```

스트림 연산 결과 확인

- peek() : 결과 스트림으로부터 요소를 소모하여 추가로 명시된 동작을 수행하여 새로운 스트림을 생성하여 반환
  - 주로 연산과 연산 사이에 결과를 확인하고 싶을 때 사용 (디버깅 용도)

```
IntStream stream = IntStream.of(7, 5, 5, 2, 1, 2, 3, 5, 4, 6);

stream.peek(s -> System.out.println("원본 스트림 : " + s))
  .skip(2)
  .peek(s -> System.out.println("skip(2) 실행 후 : " + s))
  .limit(5)
  .peek(s -> System.out.println("limit(5) 실행 후 : " + s))
  .sorted()
  .peek(s -> System.out.println("sorted() 실행 후 : " + s))
  .forEach(n -> System.out.println(n));
```

```
// 실행 결과 - 포인트는 sorted 가 상위 중간연산자가 모두 실행될때까지 지연되는걸 알 수 있다
원본 스트림 : 7
원본 스트림 : 5
원본 스트림 : 5
skip(2) 실행 후 : 5
limit(5) 실행 후 : 5
원본 스트림 : 2
skip(2) 실행 후 : 2
limit(5) 실행 후 : 2
원본 스트림 : 1
skip(2) 실행 후 : 1
limit(5) 실행 후 : 1
원본 스트림 : 2
skip(2) 실행 후 : 2
limit(5) 실행 후 : 2
원본 스트림 : 3
skip(2) 실행 후 : 3
limit(5) 실행 후 : 3
sorted() 실행 후 : 1
1
sorted() 실행 후 : 2
2
sorted() 실행 후 : 2
2
sorted() 실행 후 : 3
3
sorted() 실행 후 : 5
5
```

3. 스트림의 최종 연산

- terminal operation
- 스트림 API에서 중간 연산을 통해 변환된 스트림은 마지막으로 최종 연산을 통해 각 요소를 소모하여 결과를 표시한다

- 최종 연산이 시작되기 전까지 중간 연산은 **지연(lazy)** 된다
- **지연(lazy)**되었던 모든 중간 연산들이 최종 연산 시에 수행 되는 것
- 최종 스트림이 시작하면 컬렉션에서 요소가 하나씩 중간 스트림에서 처리되고 최종 스트림까지 오게된다

- 대표적인 최종 연산 메서드
  - 출력 : forEach()
  - 소모 : reduce()
  - 검색 : findFirst(), findAny()
  - 검사 : anyMatch(), allMatch(), noneMatch()
  - 통계 : count(), min(), max()
  - 연산 : sum(), average()
  - 수집 : collect()

메소드	설명
void forEach(Consumer<? super T> action)	스트림의 각 요소에 대해 해당 요소를 소모하여 명시된 동작을 수행함.
Optional<T> reduce(BinaryOperator<T> accumulator) T reduce(T identity, BinaryOperator<T> accumulator)	처음 두 요소를 가지고 연산을 수행한 뒤, 그 결과와 다음 요소를 가지고 또다시 연산을 수행함. 이런 식으로 해당 스트림의 모든 요소를 소모하여 연산을 수행하고, 그 결과를 반환함.
Optional<T> findFirst() Optional<T> findAny()	해당 스트림에서 첫 번째 요소를 참조하는 Optional 객체를 반환함. (findAny() 메소드는 병렬 스트림일 때 사용함)
boolean anyMatch(Predicate<? super T> predicate)	해당 스트림의 일부 요소가 특정 조건을 만족할 경우에 true를 반환함.
boolean allMatch(Predicate<? super T> predicate)	해당 스트림의 모든 요소가 특정 조건을 만족할 경우에 true를 반환함.
boolean noneMatch(Predicate<? super T> predicate)	해당 스트림의 모든 요소가 특정 조건을 만족하지 않을 경우에 true를 반환함.
long count()	해당 스트림의 요소의 개수를 반환함.
Optional<T> max(Comparator<? super T> comparator)	해당 스트림의 요소 중에서 가장 큰 값을 가지는 요소를 참조하는 Optional 객체를 반환함.
Optional<T> min(Comparator<? super T> comparator)	해당 스트림의 요소 중에서 가장 작은 값을 가지는 요소를 참조하는 Optional 객체를 반환함.
T sum()	해당 스트림의 모든 요소에 대해 합을 구하여 반환함.
Optional<T> average()	해당 스트림의 모든 요소에 대해 평균값을 구하여 반환함.
<R,A> R collect(Collector<? super T,A,R> collector)	인수로 전달되는 Collectors 객체에 구현된 방법대로 스트림의 요소를 수집함.

지연(lazy)의 의미

- stream 인터페이스의 연산을 중간 연산과 최종 연산으로 구분하여 중간 연산은 stream 을 리턴하여 지연된 연산을 적용한다
- 최종 연산이 호출되기 전까지 연산을 처리하지 않는다

```
List<String> names = Arrays.asList("김자바", "이자바", "박자바", "최자바", "이자바");
Stream<String> nameStream = names.stream()
  .filter(name -> name.startsWith("이"))
  .map(name -> name.concat("+"));

// 아래 최종연산이 호출되기 전까지 위의 코드는 실행되지 않는다
nameStream.forEach(name -> System.out.println("이름: " + name));
```

요소의 출력

- forEach() : 각 요소를 소모하여 명시된 동작을 수행

```
Stream<String> stream = Stream.of("넷", "둘", "셋", "하나");
stream.forEach(System.out::println);
```

```
// 실행 결과
넷
둘
셋
하나
```

요소의 소모

- reduce() : 첫 번째와 두 번째 요소를 가지고 연산을 수행한 뒤, 그 결과와 다음 요소를 가지고 또 다시 연산을 수행
  - 인수로 초기값을 전달하면, 초기값과 해당 스트림의 첫번째 요소와 연산을 시작

```
Stream<String> stream1 = Stream.of("넷", "둘", "셋", "하나");
Stream<String> stream2 = Stream.of("넷", "둘", "셋", "하나");

// 초기값이 없는 경우 Optional 로 결과 전달
Optional<String> result1 = stream1.reduce((s1, s2) -> s1 + "++" + s2);
result1.ifPresent(System.out::println);

// 초기값을 전달하는 reduce 는 Optional 이 아닌 T 타입
String result2 = stream2.reduce("시작", (s1, s2) -> s1 + "++" + s2);
System.out.println(result2);
```

```
// 실행 결과
넷++둘++셋++하나
시작++넷++둘++셋++하나
```

요소의 수집

- collect() : 인수로 전달되는 Collectors 객체에 구현된 방법대로 스트림의 요소를 수집
- 직접 Collector 인터페이스를 구현하여 Custom 정의 가능
- 미리 정의된 Collectors 메서드
    1. 스트림을 배열이나 컬렉션으로 변환 : toArray(), toCollection(), toList(), toSet(), toMap()
    2. 요소의 통계와 연산 메서드와 같은 동작을 수행 : counting(), maxBy(), minBy(), summingInt(), averagingInt() 등
    3. 요소의 소모와 같은 동작을 수행 : reducing(), joining()
    4. 요소의 그룹화와 분할 : groupingBy(), partitioningBy()

```
Stream<String> stream = Stream.of("넷", "둘", "하나", "셋");

// 스트림을 리스트로 변환
List<String> list = stream.collect(Collectors.toList());

// 리스트 출력
for (String item : list) {
  System.out.println(item);
}
```

```
// 실행 결과
넷
둘
하나
셋
```

Stream 장단점

- 장점
    - 내부 반복으로 반복과정을 신경쓰지 않고 요소 처리에만 집중할 수 있다
    - 간단하게 병렬 처리를 이용 가능하다 (주의 필요)
- 단점
    - 재사용이 불가하다 (매번 스트림을 생성하여 사용)
    - 디버깅이 어렵다
    - 예외처리가 필요할 경우 코드가 복잡해진다
        - 예외처리를 위한 wrapper 등을 생성하여 처리

          wrapper 예외처리 참고

스트림과 관련된 볼 만한 이야기들

[자바8 Stream API 사용시 주의사항]

[YouTube - 모던 자바 이야기(시리즈)]

Optional 클래스

java.util.Optional<T> 클래스

'T' 타입의 객체를 포장해주는 래퍼 클래스(Wrapper class)

따라서 Optional 인스턴스는 모든 타입의 참조 변수를 저장할 수 있다

예상치 못한 NullPointerException 예외를 제공되는 메서드로 간단히 회피

즉, 복잡한 조건문 없이도 null 값으로 인해 발생하는 예외처리가 가능

Optional 객체 생성

- of() : null 이 아닌 명시된 값을 가지는 Optional 객체를 반환
- ofNullable() : 명시된 값이 null 이 아니면 명시된 값을 가지는 Optional 객체를 반환, 명시된 값이 null 이면 비어있는 Optional 객체를 반환

Optional 객체 접근

- get() : Optional 객체에 저장된 값에 접근 가능
  - 만약 Optional 객체에 저장된 값이 null 이면, NoSuchElementException 예외가 발생한다. 따라서 get() 메서드를 호출하기 전에 isPresent() 메서드를 사용하여 Optional 객체에 저장된 값이 null 인지 아닌지를 먼저 확인한 후 호출해야 한다
- orElse(String other) : Optional 객체에 저장된 값이 있을때는 값을 노출하며, 저장된 값이 null 일 경우 함수의 파라미터로 전달한 내용이 반환된다

```
Optional<String> optional1 = Optional.of("옵셔널 객체");
System.out.println(optional1.orElse("없음"));

Optional<String> optional2 = Optional.empty();
System.out.println(optional2.orElse("없음"));
```

```
// 실행 결과
옵셔널 객체
없음
```

기본 타입의 Optional 클래스

IntStream 클래스와 같이 기본 타입 스트림을 위한 별도의 Optional 클래스를 제공

1. OptionalInt
2. OptionalLong
3. OptionalDouble

반환 타입이 Optional 가 아닌 해당 기본 타입

```
// 객체 접근
int getAsInt()
long getAsLong()
double getAsDouble()
```

Optional 메서드

메소드	설명
static <T> Optional<T> empty()	아무런 값도 가지지 않는 비어있는 Optional 객체를 반환함.
T get()	Optional 객체에 저장된 값을 반환함.
boolean isPresent()	저장된 값이 존재하면 true를 반환하고, 값이 존재하지 않으면 false를 반환함.
static <T> Optional<T> of(T value)	null이 아닌 명시된 값을 가지는 Optional 객체를 반환함.
static <T> Optional<T> ofNullable(T value)	명시된 값이 null이 아니면 명시된 값을 가지는 Optional 객체를 반환하며, 명시된 값이 null이면 비어있는 Optional 객체를 반환함.
T orElse(T other)	저장된 값이 존재하면 그 값을 반환하고, 값이 존재하지 않으면 인수로 전달된 값을 반환함.
T orElseGet(Supplier<? extends T> other)	저장된 값이 존재하면 그 값을 반환하고, 값이 존재하지 않으면 인수로 전달된 람다 표현식의 결괏값을 반환함.
<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)	저장된 값이 존재하면 그 값을 반환하고, 값이 존재하지 않으면 인수로 전달된 예외를 발생시킴.