EJ

이펙티브 자바 CP.6

작성 일시	@2023년 2월 14일 오후 6:57
최종 편집 일시	@2023년 2월 21일 오후 9:04
유형	이펙티브 자바
작성자	
참석자	

6️⃣ 람다와 스트림

▼ 선행 내용

- [JAVA] 람다, 스트림 기본 개념
- [JAVA] Stream 설명

▼ 42. 익명 클래스보다 람다를 사용하라.

- 익명 클래스
  - 이전 자바에서 함수 타입을 표현할 때 추상 메서드를 하나만 담은 인터페이스를 사용했다. 이런 **인터페이스의 인스턴스를 함수 객체**라고 하여, 특정 함수나 동작을 나타나는대 사용했다. JDK 1.1 등장 이후 함수 객체를 만드는 주요 수단은 익명 클래스(아이템 24)가 되었다.
  - ex) 익명 클래스의 인스턴스를 함수 객체로 사용 (문자열 정렬)

```
Collections.sort(word, new Comparator<String>() {
    pulbic int compare(String s1, String s2) {
        return Integer.compare(s1.length(), s2.length());
    }
});
```

    - 전략 패턴 처럼, 함수 객체를 사용하는 과거 객체 지향 디자인 패턴에는 익명 클래스면 충분했다.

        - `Comparator` 인터페이스가 정렬을 담당하는 추상 전략을 뜻하며, 문자열을 정렬하는 구체적인 전략을 익명 클래스로 구현했다.
        - 하지만 익명 클래스 방식은 코드가 너무 길어지기 때문에, 함수형 프로그래밍에 적합하지 않았다.
    - JDK 8 에서는 **추상 메서드 하나를 담는 인터페이스를 함수형 인터페이스**라 부르며, 이 인터페이스들의 인스턴스를 람다식(Lambda) 을 사용해 만들 수 있게 된것이다.
- 람다식
    - ex) 람다식을 함수 객체로 사용 (문자열 정렬)

```
Collections.sort(words,
    (s1, s2) -> Integer.compare(s1.length(), s2.length());
```

        - 람다, 매개변수 (`s1, s2`), 반환 값 타입을 명시하지 않았다. (`Comparator<String>`, `String`, `int`)
        - 명시하지 않았지만, 컴파일러가 대신 문맥을 살펴 **타입 추론**을 해준 것이다.
          모든 상황에 추론이 되는 것은 아님으로 명시해야할 수도 있다.
        - **타입을 명시해야 코드가 더 명확할 때만 제외하고는, 람다의 모든 매개변수 타입은 생략하자.**
          생략 후 컴파일러가 “타입을 알 수 없다” 오류를 낼 때, 타입을 명시하면 된다.

> **타입 추론**
> 아이템 26 - 제네릭 raw 타입 쓰지 말라, 아이템 29 - 제네릭을 쓰라, 아이템 30 - 제네릭 메서드를 쓰라 했었다. 이 내용들은 람다와 함께 쓸 때 두 배로 중요해진다.
> **컴파일러가 타입을 추론하는 데 필요한 타입 정보 대부분을 제네릭에서 얻기 때문이다.**
> 우리가 이 정보를 제공하지 않으면 컴파일러는 람다의 타입을 추론할 수 없게 되어, 일일이 명시해야 한다.
> 좋은 예로, 람다식 ex) 에서 `words` 라는 매개변수가 `List<String>` 이 아닌 `List (raw type)`이였다면, 컴파일 오류가 났다.

- 람다식 + 비교자 생성 메서드 (아이템 14, 43)

```
Collections.sort(words, Comparator.comparingInt(String::length));
```

- `List` 인터페이스의 `sort` 메서드

```
words.sort(Comparator.comparingInt(String::length));
```

- `::` 이중 콜론 연산자는 아이템 43에서 소개한다.

- ex) 이전 상수별 클래스 열거 타입 를 람다로 표현

```
public enum Operation {
    PLUS ("+", (x,y) -> x + y),
    MINUS ("-", (x,y) -> x - y),
    TIMES ("*", (x,y) -> x * y),
    DIVIDE ("/", (x,y) -> x / y);

    private final String symbol;
    private final DoubleBinaryOperator op;

    Operation(String symbol, DoubleBinaryOperator op) {
        this.symbol = symbol;
        this.op = op;
    }

    @Override
    public String toString() {
        return symbol;
    }

    public double apply(double x, double y) {
        return op.applyAsDouble(x, y);
    }
}
```

  - 람다를 `DoubleBinaryOperator` 인터페이스 변수에 할당했다.
    java 가 지원하는 다양한 함수 인터페이스 (아이템 44) 중 하나이다.

- 람다의 단점

  - **메서드나 클래스와 달리 람다는 이름이 없고 문서화도 할 수 없다. 따라서 코드 자체로 동작이 명확히 설명되지 않거나 코드 줄 수가 많아지면 람다를 사용하면 안된다.**

    - 람다 내용이 길거나 (3줄 이상) 읽기 어렵다면 더 줄이거나, 람다를 사용하지 않는 쪽으로 리팩터링해라.

    - 열거 타입 생성자에 넘겨지는 인수들의 타입도 컴파일 타임에 추론된다.
      따라서 열거 타입 생성자 안의 람다는 열거 타입의 인스턴스 멤버에 접근 할 수 없다.
      (인스턴스는 런타임에 만들어지기 때문임)

    - **람다는 함수형 인터페이스에서만** 쓰인다.
      추상 클래스의 인스턴스를 만들 때, 람다를 쓸 수 없으니, 익명 클래스를 써야 한다.
      추상 메서드가 여러 개인 인터페이스의 인스턴스를 만들 때도 익명 클래스를 쓸 수 있다.

    - 람다는 자신을 참조할 수 없다. **람다에서 `this` 는 바깥 인스턴스를 가르킨다.**
      익명 클래스에서 `this` 는 인스턴스 자신을 가르킴으로, 함수 객체가 자신을 참조해야 한다면 반드시 익명클래스를 써야한다.

- 람다의 주의점

  - 람다도 익명 클래스 처럼 직렬화 형태가 구현별로 다를 수있다. 따라서 람다에서 직렬화 하는 일은 삼가하자 (익명 클래스의 인스턴스도 마찬가지)

    - 직렬화해야하는 함수 객체가 있다면 (`Comparator` 처럼) `private` 정적 중첩 클래스 (아이템 24)의 인스턴스를 사용하자
- 정리
    - jdk 8 - 함수 객체를 구현하는데 람다가 도입되었다.
      람다는 함수 객체를 아주 쉽게 표현할 수 있어, 코드가 명확하고 간결해 질 수있다.
    - 익명 클래스는 (함수형 인터페이스가 아닌) 타입의 인스턴스를 만들 때만 사용하자.

▼ 43. 람다보다는 메서드 참조를 사용하라.

- 이중 콜론 연산자 (::)
    - java 8 에서 추가된 메서드 참조 연산자이다.
    - 람다식에서 파라미터를 중복해서 사용하고 싶지 않을 때, 사용하고 람다식과 동일한 처리 방법을 갖지만, 이름으로 기존 메소드를 참조함으로써 더욱 간결하다.
    - 사용 방법
        - `[인스턴스]::[메소드명(or new)]`
        - `User::getId`
        - 람다 표현식이 () → {} 에서만 가능하다
        - static 메서드인 경우 인스턴스 대신 클래스 이름으로도 사용할 수 있다
- 메서드 참조 (Method Reference)
    - 함수 객체를 람다보다도 더 간결하게 만들수 있다.
    - ex) 임의의 키와 `Integer` 값의 매핑을 관리하는 프로그램 (`merge` 메서드를 잘쓴 예)
      이때 값이 키의 인스턴스 개수로 해석된다면, 이 프로그램은 멀티셋을 구현한게 된다.

      ```
      map.merge(key, 1, (count, incr) -> count + incr);
      ```

        - 키가 맵안에 없다면 키와 숫자 1을 매핑하고, 이미 있다면 기존 매핑 값을 증가 시킨다.
        - 자바 8 때 `Map` 에 추가된 `merge` 메서드를 사용했다.
          `merge` 메서드는 키, 값, 함수를 인수로 받으며, 주어진 키가 맵 안에 아직 없다면 주어진 {키, 값} 쌍을 그대로 저장한다.
          반대로 키가 이미 있다면 (세 번째 인수로 받은) 함수를 현재 값과 주어진 값에 적용한 다음, 그 결과로 현재 값을 덮어쓴다.
          즉, 맵에 {키, 함수의 결과} 쌍을 저장한다.
        - 매개변수 count, incr 은 크게 할 일 없어 보인다.
          사실 이 람다는 두 인수의 합을 단순히 반환할 뿐이다.
          자바 8 때 Integer 클래스 (와 모든 기본 타입의 박싱 타입)는 이 람다와 기능이 같은 정적 메서드 sum 을 제공하기 시작했다. 따라서 더 간결하게 작성 가능하다.

          ```
          map.merge(key, 1, Integer::sum);
          ```

        - 또한 매개변수 수가 늘어날수록 메서드 참조로 제거할 수 있는 코드양도 늘어난다.
    - 어떤 람다에서는 매개변수의 이름 자체가 프로그래머에게 좋은 가이드가 되기도 한다.
      이런 람다는 길이는 더 길지만, 메서드 참조보다 읽기 쉽고 유지보수가 쉬울 수 있다.
    - 람다로 할 수 없는 일이라면, 메서드 참조로도 할 수 없다.
      그렇더라도 메서드 참조를 사용하는 편이 보통 더 짧고 간결함으로, 람다로 구현했을 때 너무 길거나 복잡하면 메서드 참조가 좋은 대안이 되어준다.
- 람다가 더 좋을 때
    - ex) `GoshThisClassNameIsHumongous` 클래스

```
// 메서드 참조
service.execute(GoshThisClassNameIsHumongous::action);
// 람다
service.execute(() -> action());
```

        - 메서드 참조 쪽은 더 짧지도, 명확하지도 않다. 람다쪽이 보다 좋다.
- **메서드 참조 유형 5가지**
    1. 정적 메서드를 가리키는 메서드 참조
    2. 인스턴스 메서드를 참조하는 유형
        a. 수신 객체 (참조 대상 인스턴스)를 특정하는 한정적 인스턴스 메서드 참조
            - 근복적으로 정적 참조와 비슷하다.
            - 즉, 함수 객체가 받는 인수와 참조되는 메서드가 받는 인수가 똑같다.
        b. 수신 객체 (참조 대상 인스턴스)를 특정하지 않는 비한정적 인스턴스 메서드 참조
            - 함수 객체를 적용하는 시점에 수신 객체를 알려준다.
            - 이를 위해 수신 객체 전달용 매개변수가 매개변수 목록의 첫 번째로 추가되며, 그 뒤로는 참조되는 메서드 선언에 정의된 매개변수들이 뒤따른다.
            - 주로 스트림 파이프라인에서의 매핑과 필터 함수로 쓰인다. (아이템 45)
    3. 클래스 생성자를 가르키는 메서드 참조
    4. 배열 생성자를 가르키는 메서드 참조

메서드 참조 유형	람다	메서드 참조
정적	str -> Integer.parseInt(str)	Integer::parseInt
한정적 (인스턴스)	Instant then = Instant.now(); t -> then.isAfter(t)	Instant.now()::isAtfer
비한정적 (인스턴스)	str -> str.toLowerCase()	String::toLowerCase
클래스 생성자	() -> new TreeMap<K, V>()	TreeMap<K,V>::new
배열 생성자	len -> new int[len]	int[]::new

- 정리

- 메서드 참조는 람다의 간단명료한 대안이 될 수 있다.
- 메서드 참조 쪽이 짧고 명확하면, 메서드 참조, 그 반대이면 람다를 쓰자.
- 람다를 쓰고 Intellij 에서는 Replace Lambda with Method reference 되기 때문에, 람다를 작성하고 해당 IDE 기능을 사용해서 보다 나은게 어떤건지 확인후 적용하자.

▼ 44. 표준 함수형 인터페이스를 사용하라.

- 람다 지원 후 API 작성 변경
  - 상위 클래스의 기본 메서드를 재정의해 원하는 동작을 구현하는 템플릿 메서드 패턴이 줄었다.
  - 대신, 같은 효과의 함수 객체를 받는 정적 팩터리나 생성자를 제공
  - 일반화 해서 말하자면, 함수 객체를 매개변수로 받는 생성자와 메서드를 더 많이 만들어야 한다. 이때 함수형 매개변수 타입을 올바르게 선택해야 한다.
- `LiskedHash`
  - 이 클래스의 `protected` 메서드인 `removeEldestEntry` 를 재정의하면 캐시로 사용할 수 있다. `put` 메서드는 `removeEldestEntry`를 호출하여 `true`가 반환되면 맵에서 가장 오래된 원소를 제거한다.

```
@Override
protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
    return size() > 100;
}

// 람다 적용 - 불필요한 함수형 인터페이스
@FunctinalInterface
interface EldestEntryRemovalFunction<K, V> {
    boolean remove(Map<K,V> map, Map.Entry<K,V> eldest);
}
```

    - `removeEldestEntry`는 인스턴스 메서드임으로, 람다로 작성시 (팩터리나 생성자를 호출할 때는, 맵의 인스턴스가 존재하지 않음), 맵은 자기 자신도 함수 객체에 건네줘야 함으로 `Map<K,V> map` 도 인수로 필요하다.
    - 람다로 적용된 메소드를 굳이 사용할 필요는 없다. 자바 표준 라이브러리에 이미 같은 모양의 인터페이스가 준비되어 있다.
    - **필요한 용도에 맞는 게 있다면, 직접 구현하지 말고 표준 함수형 인터페이스를 활용하라**
- **표준 함수형 인터페이스**
  - API가 다루는 개념의 수가 줄어들어 익히기 더 쉽다.
  - 유용한 디폴트 메서드를 많이 제공함으로, 다른 코드와의 상호운용성도 좋아진다.
    - `Predicate` 인터페이스는 `predicate` 들을 조합하는 메서드를 제공한다.
    - 위 예에서 `EldestEntryRemovalFunction` 대신 표준 인터페이스인 `BiPredicate<Map<K,V>, Map.Entry<K,V>` 를 사용 가능하다.

- `java.util.function` 패키지에는 총 43개의 인터페이스가 있다. 전부 기억할 필요 없고, 기본 인터페이스 6개만 기억하면, 나머지를 충분히 유추해낼 수 있다. 이 기본 인터페이스들은 모두 **참조 타입용**이다.

- **기본 인터페이스**

인터페이스	함수 시그니처	Ex
UnaryOperator<T>	T apply (T t)	String::toLowerCase
BinaryOperator<T>	T apply (T t1, T t2)	BigInteger::add
Predicate<T>	boolean test(T t)	Collection::isEmpty
Function<T,R>	R apply (T t)	Arrays::asList
Supplier<T>	T get()	Instant::now
Consumer<T>	void accept(T t)	System.out::println

  - `Operator` - 인수 타입과 반환 타입이 같은 함수
    - `UnaryOperator` - 인수가 1 개인 인터페이스
    - `BinaryOperator` - 인수가 2개인 인터페이스
  - `Predicate` - 인수 하나를 받아 boolean 반환하는 함수
  - `Function` - 인수와 반환 타입이 다른 함수
    - 유일하게 반환 타입만 매개변수화 된다.
    - 인터페이스의 변형은 입력과 결과의 타입이 항상 다르다.
    - 입력과 결과 타입이 모두 기본 타입이면 접두어 `SrcToResult` 를 사용한다.
      - `long` 을 받아 `int` 를 반환하면 `LongToIntFunction` 이 되는 식이다.
  - `Supplier` - 인수를 받지 않고 값을 반환(제공)하는 함수
  - `Consumer` - 인수를 하나 받고 반환 값이 없는 (인수를 소비하는) 함수
- 기본 인터페이스는 기본 타입인 `int`, `long`, `double` 용으로 각 3개씩 변형이 생기는데, 기본 인터페이스 이름 앞에 기본 타입 이름을 붙여 사용된다.
  - `Predicate` → `IntPredicate`, `BinaryOperator` → `LongBinaryOperator`
- 인수 타입을 2개 (혹은 3개)씩 받는 변형된 함수형 인터페이스
  - `Predicate<T>` → `BiPredicate<T,U>`
  - `Function<T,R>` → `BiFunction<T,U,R>`
  - `Consumer<T>` → `BiConsumer<T,U>`
- 기본 인터페이스 변형
  - 기본 타입을 반환하는 `BiFunction` 변형
    - `ToIntBiFunction<T,U>` - `int` 타입 반환
    - `ToLongBiFunction<T,U>` - `long` 타입 반환

        - `ToDoubleBiFunction<T,U>` - `double` 타입 반환
    - 기본 타입을 받는 `BiConsumer` 변형
        - `ObjIntConsumer<T>` - `int` 타입 받음
        - `ObjLongConsumer<T>` - `long` 타입 받음
        - `ObjDoubleConsumer<T>` - `double` 타입 받음
    - `boolean` 타입을 반환하는 `Supplier` 변형
        - `BooleanSupplier`
- 전부 외울 필요 없다.
  표준 함수형 인터페이스 6개를 기억하고, 범용적인 이름으로, 필요할 때 찾아 사용하자.
  기본 타입만 지원한다. 그러니 박싱된 기본 타입을 넣어 사용하지 말자. (아이템 61)
  계산량이 많을때 성능이 처참히 느려질 수 있다.
- 직접 함수형 인터페이스를 작성하는 경우
    - 표준 인터페이스 중 필요한 용도가 맞는 게 없을 때
        - ex) `Predicate` 에서 매개 변수가 3개 이상인 경우, 검사 예외를 던져야 하는 경우
    - 그런데, 구조적으로 똑같은 표준 함수형 인터페이스가 있더라도 작성해야 할 때가 있다.
        - ex) `Comparator<T>` 인터페이스
            - 구조적으로는 `ToIntBiFunction<T,U>` 와 동일하다.
            - 사용해야하는 이유
                1. API 에 자주 사용된다. 이름이 그 용도를 잘 설명한다.
                2. 구현하는 쪽에서 반드시 지켜야 할 규약을 담고 있다.
                3. 비교자들을 변환하고 조합해주는 유용한 디폴트 메서드까지 있다.
    - 고민점 (이 중 하나 이상 만족한다면, 전용 함수형 인터페이스를 구현을 고민해봐야 한다.)
        - 자주 쓰이며, 이름 자체가 용도가 명확함.
        - 반드시 따라야 하는 규약이 있다.
        - 유용한 디폴트 메서드를 제공한다.
- `@FunctionalInterface` 애너테이션
    - 인터페이스가 함수형 인터페이스로 사용됨을 알려주는 애너테이션
    - 목적
        1. 사용자에게 이 인터페이스가 람다용으로 설계된 것임
        2. 추상 메서드가 하나만 있어야 함을 컴파일되게 해준다.
        3. 유지보수 과정에서 실수로 메서드를 추가하지 못하도록 해준다.
    - 직접 만든 함수형 인터페이스에는 항상 `@FunctionalInterface` 애너테이션을 사용하자
- 함수형 인터페이스 API 사용시 주의점

    - 서로 다른 함수형 인터페이스를 같은 위치의 인수로 받는 메서드들을 다중 정의해서는 안된다.
    - 클라이언트에게 불필요한 모호함을 주며, 이 모호함이 에러를 발생하기도한다.
- 정리
    - JDK 8(람다) 이후 - API 설계 시 람다를 염두에 두고 설계해라
    - 입력값과 반환값에 함수형 인터페이스 타입을 활용해라
    - 보통 표준 함수형 인터페이스를 사용하는게 베스트이다.
    - `Comparator` 처럼 직접 새로운 함수형 인터페이스를 만드는 편이 좋을 수 있다.

▼ 45. 스트림은 주의해서 사용하라.

- 스트림 API
    - 다량의 데이터 처리 작업을 도와줌
    - 추상 개념 (스트림) - 데이터 원소의 유한 혹은 무한 시퀀스를 뜻
    - 추상 개념 (스트림 파이프 라인) - 이 원소들로 수행하는 연산 단계를 뜻
    - 데이터 원소들은 객체 참조나 기본 타입(int, long, double) 값이다.
    - 다재다능하여, 사실상 어떠한 계산이라도 할 수 있다.
    - 제대로 사용한다면 프로그램이 짧고 깔끔해지지만,
      잘못 사용하면 읽기 어렵고 유지보수도 힘들어진다.
- **스트림 파이프 라인**
    - 단계
        - 소스 스트림 (생성하기)
        - 중간 연산 (가공하기)
            - 한 스트림을 다른 스트림으로 변환하는데, 원소 타입이 다를 수도 있다.
        - 종단 연산 (결과 만들기)
            - 원소를 컬렉션에 담거나, 특정 원소를 선택하거나, 출력하거나 등.
    - 특징
        - **지연 평가**된다. 평가는 종단 연산이 호출될 때, 이뤄지며 종단 연산에 쓰이지 않는 데이터 원소는 계산에 쓰이지 않는다. 지연 평가로 인해 무한 스트림을 다룰 수 있다.
          종단 연산이 없는 경우는 아무 일도 하지 않는 명령어와 같으니, 반드시 추가하자.
        - 메서드 연쇄를 지원하는 플루언트 API (fluent API) 다.
          파이프라인 하나를 구성하는 모든 호출을 연결하여 단 하나의 표현식으로 완성 할 수 있다. 파이프 라인 여러 개를 연결해 하나의 표현식으로 만들 수 있다.
        - 기본적으로 스트림 파이프라인은 순차적으로 수행된다. 병렬 처리를 위해 스트림 중 하나에서 `parallel` 메서드를 호출 하면되나, 효과를 볼 수 있는 상황이 많지 않다.(아이템 48)
- ex) 스트림 적용 전 예제 (`Anagrams`)
  아나그램 - 철자를 구성하는 알파벳은 같고 순서가 다르다. (ex - read, dear, dare)

```
public class Anagrams {
    public static void main(String[] args) throws IOException {
      File dictionary = new File(args[0]);
      int minGroupSize = Interger.parseInt(args[1]);

      Map<String, Set<String>> groups = new HashMap<>();
      try(Scanner s = new Scanner(dictionary)){
        while(s.hasNext()){
          String word = s.next();
            groups.computeIfAbsent(alphabetize(word), (unused) -> new TreeSet<>()).add(word);
        }
      }

      for(Set<String> group : groups.values())
        if(group.size() >= minGroupSize)
          System.out.println(group.size() + ":" + group);
    }

    public static String alphabetize(String s){
      char[] a = s.toCharArray();
        Arrays.sort(a);
        return new String(a);
    }
}
```

  - `computeIfAbsent` 는 첫 번째 인수는 `Key`를 뜻하고 해당 `Key`가 있으면 해당 `Value` 를 반환한다. 없다면, 두 번째 인수의 작업을 동작하고 계산된 값을 반환한다.

- ex) 스트림 적용 후 예제 (`Anagrams`) - 과하게 사용

```
public class StreamAnagrams {
    public static void main(String[] args) throws IOException {
        Path dictionary = Paths.get(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);

        //사전 파일을 제대로 닫기 위해 try-with-resources 활용
        try (Stream<String> words = Files.lines(dictionary)) {
            words.collect(
                    groupingBy(word -> word.chars().sorted()
                            .collect(StringBuilder::new,
                                    (sb, c) -> sb.append((char) c),
                                    StringBuilder::append).toString()))
                    .values().stream()
                    .filter(group -> group.size() >= minGroupSize)
                    .map(group -> group.size() + ": " + group)
                    .forEach(System.out::println);
        }
    }
}
```

  - 스트림을 과하게 사용하면 프로그램이 읽거나 유지보수하기 어려워진다.

- ex) 스트림 적용 후 예제 (`Anagrams`) - 적절한 사용

```
public class HybridAnagrams {
    public static void main(String[] args) throws IOException {
        Path dictionary = Paths.get(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);

        try (Stream<String> words = Files.lines(dictionary)) {
```

```
            words.collect(groupingBy(word -> alphabetize(word)))
                    .values().stream()
                    .filter(group -> group.size() >= minGroupSize)
                    .forEach(g -> System.out.println(g.size() + ": " + g));
        }
    }
    public static String alphabetize(String s){
      char[] a = s.toCharArray();
        Arrays.sort(a);
        return new String(a);
    }
}
```

- `try-with-resources` 블록으로 사전 파일을 열고, 파일의 모든 라인으로 구성된 스트림을 얻는다. 스트림 변수의 이름을 `words` 로 지어 스트림 안의 각 원소가 단어 (`word`)을 명확히 했다.
- 스트림 파이프라인에 중간 연산은 없으며, 종단 연산에서 모든 단어를 수집해 맵으로 모은다. (`groupingBy`) 이후 맵의 `values()` 가 반환한 값으로 부터 새로운 `Stream<List<String>>` 스트림을 열고, 해당 스트림의 원소는 아나그램 리스트가 된다.
  `filter` 메서드를 통해 `true` 를 반환하는 원소만 남게되고, 종단 연산인 `forEach` 에서 리스트를 출력한다.
- `alphabetize` 메서도 스트림을 사용해 구현할 수는 있다. 하지만, 명확성이 떨어지고 잘못 구현할 가능성이 높아진다. 심지어 느려질 수도 있다.
  자바가 기본 타입인 `char` **용 스트림을 지원하지 않기 때문이다.**
  그러니, **char 값들을 처리할 때는 스트림을 삼가하는 편이 낫다.**

> 람다에서는 타입 이름을 자주 생략하므로 매개변수 이름을 잘 지저야 스트림 파이프라인의 가독성이 유지된다. `alphabetie` 메서드 처럼 로직 밖으로 빼내 전체 가독성이 높일 수 있다. (도우미 메서드), 해당 메서드는 스트림 파이프라인에서 많은 도움이 된다

- 스트림으로 적용해야 할 때
  - 기존 코드는 스트림을 사용하도록 리팩터링 하되, 새 코드가 더 나아 보일때 반영하자
  - 스트림 파이프라인은 함수 객체(람다, 메서드 참조)로 표현한다. 반면 반복 코드에서는 코드 블럭을 사용해 표현한다.
    함수 객체로는 할 수 없지만, **코드 블록에서는 할 수 있는 일**
    - 코드 블록은 범위 안의 지역변수를 읽고 수정 가능하다. 하지만 람다에서는 `final` 이거나 사실상 `final` 인 변수만 읽을 수 있고, 지역 변수를 수정이 불가능하다.
    - 코드 블록에서는 `return`문이나, `break`, `continue` 문을 사용해 코드 반복을 제어 하고검사 예외를 던질 수 도 있다. 하지만, 람다는 그 어떤 것도 할 수 없다.
  - 스트림에 안성 맞춤
    - 원소들의 시퀀스를 일관되게 변환
    - 원소들으 시퀀스 필터링
    - 원소들의 시퀀스를 하나의 연산을 사용해 결합한다. (더하기, 연결하기, 최소 값 등)

        - 원소들의 시퀀스를 컬렉션에 모은다. (공통된 속성을 기준으로 묶어가며)
        - 원소들의 시퀀스에서 특정 조건을 만족하는 원소를 찾는다.
    - 스트림으로 처리하기 어려운 일
        - 한 데이터가 파이프라인의 여러 단계를 통과할 때, 이 데이터의 각 단계에서의 값에 동시에 접근하기는 어려운 경우다. 스트림 파이프라인은 일단 한 값을 다른 값에 매핑하고 나면 원래의 값을 잃는 구조이기 때문이다.
        - 원래 값고 새로운 값의 쌍을 저장하는 객체를 사용해 매핑하는 우회 방법도 있지만 코드양이 많아지고 지저분해진다. 가능한 경우라면, 앞 단계의 값요 할 때 매핑을 거꾸로 수행하는 방법을 택하자.
- 정리
    - 반복 코드와 스트림 둘 다 과도하게 하나의 방식만 고집할게 아닌 상황에 맞게 사용하자
    - 스트림과 반복 중 어느 쪽이 나은지 확신하기 어렵다면 둘 다 해보고 더 나은쪽을 택하자

▼ 46. 스트림에서는 부작용 없는 함수를 사용하라.

- 스트림 패러다임
    - 함수형 프로그램밍에 기초한 패러다임.
    - 스트림이 제공하는 표현력, 속도, (상황에 따라) 병렬성을 얻으려면 API 와 이 패러다임을 받아들어야 한다.
    - 핵심으로는 일련의 변화으로 재구성하는 부분에 있다.
      이때 각 변환 단계는 가능한 한 이전 단계의 결과를 받아 처리하는 **순수 함수**여야 한다.
        - 순수 함수 - 오직 입력만이 결과에 영향을 주는 함수를 뜻함, 즉 다른 가변 상태를 참조하지 않고, 함수 스스로도 다른 상태를 변경하지 않는다. 이렇게 하려면 (중간, 종단) 스트림 연산에 건네는 함수 객체는 부작용이 없어야 한다.
    - ex) 텍스트 파일에서 단어별 수를 세어 빈도표를 만드는 코드 - 따라하지 말것.

```
Map<String, Long> freq = new HashMap<>();
try (Stream<String> words = new Scanner(file).tokens()) {
    words.forEach(word -> {
        freq.merge(word.toLowerCase(), 1L, Long::sum);
    });
}
```

        - 스트림, 람다, 메서드 참조를 사용했고, 결과도 올바르지만, 스트림 코드라 할 수 없다. 스트림 코드를 가장한 반복적 코드이다.
        - 스트림 API 이점을 살리지 못하여 같은 기능의 반복적 코드보다 조금 더 길고, 읽거 어렵고, 유지보수에도 좋지 않다.
        - 모든 작업은 `forEach` 에서 일어나는데, 이때 외부 상태를 수정하는 람다를 실행하면서 문제가 생긴다. `forEach` 가 그저 스트림이 수행한 연산 결과를 보여주는 일 이상을 하는 것

(람다가 상태를 수정함)을 보니 나쁜 코드의 냄새가 난다.

- ex) 텍스트 파일에서 단어별 수를 세어 빈도표를 만드는 코드 - 스트림 제대로 활용

```
Map<String, Long> freq;
try (Stream<String> words = new Scanner(file).tokens()) {
    freq = words
        .collect(groupingBy(String::toLowerCase, counting());
}
```

  - 짧고 명확하다.

- `forEach`
  - `forEach` 연산은 종단 연산 중 기능이 가장 적고 가장 '덜' 스트림 답다. 대놓고 반복적이라 병렬화도 할 수 없다.
  - `forEach` 연산은 스트림 계산 결과를 보고할 때만 사용하고, 계산에서는 사용하지 말자.
  - 가끔은 스트림 계산 결과를 기존 컬렉션에 추가하는 등의 다른 용도로 쓸 수 있다.

- **수집기 (collector)**
  - `java.util.stream.Collectors` 클래스는 메서드를 39개 가지고 있고 그 중에서는 타입 매개 변수가 5개나 되는 것도 있다.
  - 스트림의 종단 연산을 사용자가 원하는 형태로 반환할 수 있다.

    `.collect(Collectros.toList());`
  - 축소(reduction) 전략 을 캡슐화한 블랙박스(내부 구조를 알 수 없다) 객체
    여기서, 축소는 스트림의 원소들을 객체 하나에 취합한다는 뜻.
  - 종류
    - `toList()`
      - 스트림의 원소들을 하나의 List 객체에 담아 반환
    - `toSet()`
      - 스트림의 원소들을 하나의 Set 객체에 담아 반환
    - `toCollection(collectionFactory)`
      - 스트림의 원소들을 원하는 컬렉션을 생성하여 담아 반환
    - `toMap()`
      - 스트림의 원소들을 `KeyMapper`, `ValueMapper` 를 이용해 축소해 반환
      - 3, 4 번째 인수가 있는데, 선택적으로 사용한다. (자세한 내용은 본문)
    - `groupingBy()`
      - 입력으로 분류 함수를 받고 출력으로는 원소들을 카테고리별로 모아 놓은 맵을 담은 수집기를 반환한다. 그리고 이 카테고리가 해당원소의 맵 키로 쓰인다.
      - 다중정의된 `groupingBy` 중 형태가 가장 간단한 것은 분류 함수 하나를 인수로 받아 맵을 반환한다.

        - 여럿이 있지만, 대부분 스트림을 맵으로 취합하는 기능들.

- `toList`
    - ex) 빈도표에서 가장 흔한 단어 10개를 뽑아내는 파이프라인

```
List<String> topTen = freq.keySet().stream()
    .sorted(comparing(freq::get).reversed())
    .limit(10)
    // 책에서는 Collectors import 시켜 사용한다고 한데,
    // 공부하는 단계임으로, 전체적으로 명시한다. 실제 사용에는 import 하자
    .collect(Collectors.toList();
```

        - `sorted` 에는 비교자를 넘겨야 한다. (기준이 필요)
        - `comparing` 메서드는 키 추출 함수를 받는 비교자 생성 메서드이다.
        - 한정적 메서드 참조인 키 추출 함수로 쓰인 `freq::get`은 입력받은 단어 (`key`)를 빈도표에서 찾아(추출) 그 빈도를 반환한다.

- `toMap`
    - ex) `toMap` 수집기를 사용해 문자열을 열거 타입 상수에 매핑한다. (Operation 코드)

```
private static final Map<String, Operation> stringToEnum =
    Stream.of(values()).collect(toMap(Object::toString, e -> e);
```

        - 위 에시에는 각 원소가 고유한 키에 매핑되어야 있어야만 한다.
          그렇지 못하면 `IllegalStateException` 을 던질 것이다.
        - 더 복잡한 형태의 `toMap` 이나 `groupingBy` 는 이런 충돌을 다루는 다양한 전략을 제공한다.
          `toMap` 에 병합 합수까지 제공할 수 있다.
        - 병합 합수의 형태는 `BinaryOperator<U>` 이며 여기서 `U`는 해당 맵의 값 타입이다.
          병합 함수 형태가 곱셈이라면 키가 같은 모든 값을 곱한 결과를 얻는다.
    - ex) `toMap` 수집기를 이용해 각 키와 해당 키의 특정 원소를 연관 짓는 맵 생성 수집기

```
Map<Artist, Album> topHits = albums.collect(
    toMap(Album::artist, a -> a, BinaryOperator.maxBy(comparing(Album::sales))));
```

        - 결과는 음악가와 가장 많이 팔린 앨범을 매핑해서 반환한다.
        - `maxBy` 는 `Comparator<T>` 를 받아 `BinaryOpterator<T>`를 반환한다.
    - ex) `toMap` 수집기를 이용해 마지막에 쓴 값 취하는 수집기

```
toMap(keyMapper, valueMapper, (oldVal, newVal) -> newVal)
```

        - `toMap` 에 3번째 인수는 `toMap` 에서 충돌이 나면 동작을 어떻게 할지 정할 수 있다.

        - `toMap` 에 4번째 인수로 맵 팩터리를 받는다. `EnumMap`, `TreeMap` 처럼 특정 맵 구현체를 지정할 수 있다. (default `hashMap::new`)
- `groupingBy`
    - ex) 알파벳화 한 단어를 알파벳화 결과 같은 단어의 리스트로 매핑하는 맵을 생성

```
words.collect(groupingBy(word -> alphabetize(word)))
```

        - 결과로 `Map<Key, List<String>>` 형태를 반환
    - ex) 카테고리에 속하는 원소의 개수(값) 와 매핑한 맵을 얻는다.

```
Map<String, Long> freq = words
    .collect(groupingBy(String::toLowerCase, Collectors.counting());
```

        - 수집기가 리스트 외의 값을 갖는 맵을 생성하기 위해선 분류 함수와 함께 다운 스트림 수집기도 명시해야 한다.
        - 가장 쉬운 예로 `toSet()` 을 보내면 집합 (`Set`) 을 값으로 갖는 맵을 만든다.
        - `counting()` 은 각 키에 해당 카테고리에 속하는 원소의 개수(값)와 매핑한 맵을 얻는다.
    - 3번째 인수로 받는 `groupingBy` 도 있다. `toMap` 과 같이 맵 팩터리를 받으며, 특정 맵 구현체를 반환 할 수 있다.
    - 또, 동시성 문제를 해결해주는 `ConcurrentHashMap` 을 만들 수 있는 `groupingByConcurrent` 메서드도 `groupingBy` 형태가 같다
- `partitioningBy`
    - 분류 함수 자리에 `predicate`를 받고 키가 `Boolean`인 맵을 반환한다.
    - `predicate`, 다운 스트림 수집기 까지 받는 다중정의된도 있다.
- 다운 스트림 수집기 전용
    - 종류
        - `counting()`, `summing()`, `averaging()`, `summarizing()`
    - ex) `collect(counting())` 형태로 사용할 일은 전혀 없다.
- `joining`
    - (문자열 등의) `CharSequence` 인스턴스의 스트림에만 적용할 수 있다.
    - 이 중 매개변수가 없는 `joining` 은 단순히 원소들을 연결하는 수집기를 반환한다.
- 정리
    - 스트림 파이프라인 프로그래밍의 핵심은 부작용이 없는 함수 객체에 있다.
    - 종단 연산 중 `forEach`는 스트림이 수행한 계산 결과를 보고할 때만 이용하자.
    - 가장 중요한 수집기 팩토리는 `toList`, `toSet`, `toMap`, `groupingBy`, `joining`이 있다.

▼ 47. 반환 타입으로는 스트림보다 컬렉션이 낫다.

- java 8 이전
  - `Stream` 이 없었기 때문에, 원소 시퀀스를 반환하는 메서드는 `Collection Interface` (기본) 혹은, `Iterable` (`for-each` 문에서만 쓰이거나, 반환된 원소 시퀀스가 일부 `Collection` 메서드를 구현 할 수 없을 때), 배열(성능 이점)을 사용했다. 하지만 `Java 8 Stream` 이 등장하면서 선택은 복잡한 일이 되어 버렸다.
- 스트림은 반복(iteration)을 지원하지 않는다.
  - 따라서, 스트림과 반복을 알맞게 조합해야 좋은 코드가 나온다.
    API를 스트림만 반환하도록 만들면, 반환된 스트림을 `for-each`로 반복하길 원하는 사용자는 불만일 것이다.
  - `Stream` 인터페이스는 `Interable` 인터페이스가 정의한 추상 메서드를 전부 포함할 뿐만 아니라, `Iterable` 인터페이스가 정의한 방식대로 동작한다. 그럼에도 `for-each` 로 스트림을 반복할 수 없는 이유는 `Stream` **이** `Iterable` **을** `extend` **하지 않는다.**
  - ex) `Stream` 에서 `iterator` 사용하기

```
// 자바 타입 추론의 한계로 컴파일 되지 않는다.
for (ProcessHandle ph : ProcessHandle.allProcesses()::iterator) {
    ...
}

// 끔직한 우회 방법
// 작동은 하지만 너무 난잡하고 직관성이 떨어진다.
for (ProcessHandle ph : (Interable<ProcessHandle>)ProcessHandle.allProcesses()::iterator) {
    ...
}

// Stream<E>를 Iterable<E> 로 중개해주는 어뎁터
public static <E> Iterable<E> iterableOf(Stream<E> stream) {
    return stream::iterator;
}

for (ProcessHandle p : iterableOf(ProcessHandle.allProcesses()) {
    ...
}
```

  - ex) `iterator` 에서 `Stream` 사용하기

```
public static <E> Stream<E> streamOf(Iterable<E> iterable) {
    return StreamSupport.stream(iterable.spliterator(), false);
}
```

    - `Stream API`가 `Iterable`만 반환하면, 스트림 파이프라인에서 처리하는 사용자도 불만일 것이다.
    - 어뎁터 메서드로 손쉽게 구현 가능하다.
  - 객체 시퀀스 작성
    - 스트림 파이프 라인에서만 쓰인다면 `Stream` 반환

        - 반환된 객체들이 반복문에서만 쓰인다면 `Iterable` 반환
        - 공개 API를 작성한다면, 두 가지 방식 모두 적용 필요하다.
- `Collection` 으로 반환
    - **`Collection` 인터페이스는 `Iterable` 의 하위 타입이고 `stream` 메서드도 제공하니 반복과 스트림을 동시에 지원한다.** 따라서 **원소 시퀀스를 반환하는 공개 API 의 반환 타입에는 `Collection` 이나 그 하위 타입을 쓰는게 일반적으로 최선이다.**
    - `Arrays` 역시 `Arrays.asList` 와 `Stream.of` 메서드로 손쉽게 반복과 스트림을 지원한다.
- 전용 컬렉션을 구현을 검토하자
    - 데이터가 충분히 적다면 (`ArrayList`, `HashSet`) 같은 표준 컬렉션 구현체를 반환하는게 최선일 수 있으나, 데이터가 많다면 검토 해보는게 좋다.
    - ex) 주어진 집합의 멱집합 반환 하는 상황
        - 원소 개수가 n 개면 멱집합의 원소 개수는 2^n 이 된다, 그러니 표준 컬렉션 구현체에 저장하려는 생각은 위험하다. 하지만 `AbstractList` 를 이용하면 효율 좋은 컬렉션을 반환 할 수 있다.

```
public class PowerSet {
    public static final <E>Collection<Set<E>> of(Set<E> s){
        List<E> src = new ArrayList<>();
        if (src.size() > 30) {
            throw new IllegalArgumentException();
        }

        return new AbstractList<>() {
            @Override
            public int size() {
                return 1 << src.size();
            }

            @Override
            public boolean contains(Object o) {
                return o instanceof Set && src.containsAll((Set)o);
            }

            @Override
            public Set<E> get(int index) {
                Set<E> result = new HashSet<>();
                for (int i = 0; index != 0; i++, index >>= 1) {
                    if ((index & 1) == 1) {
                        result.add(src.get(i));
                    }
                }
                return result;
            }
        };
    }
}
```

        - `AbstractCollection`을 활용할 때는 `Iterable`용 메서드외 `contains`, `size` 메서드만 더 구현하면 된다.

        - `contains`, `size` 메서드를 구현하는게 불가능할 때는 컬렉션 보다는 `Stream`, `Iterable`을 반환하는 편이 낫다.
- 정리
    - **원소 시퀀스를 반환하는 메서드를 작성할 때**
        - 이를 스트림으로 처리하기, 반복으로 처리하기 양쪽 다 만족시키려 노력하자.
        - 컬렉션(그 하위 타입)을 반환할 수 있다면 그렇게 하라.
            - 반환 전부터 이미 원소들을 컬렉션에 담아 관리하고 있거나 컬렉션을 하나 더 만들어도 될 정도로 원소 개수가 적다면 `ArrayList` 같은 표준 컬렉션에 담아 반환하자.
            - 그렇지 않다면 전용 컬렉션을 구현할지 고민하라.
        - 컬렉션을 반환하는 게 불가능하면 `Stream`, `Iterable` 중 자연스러운 것을 반환하자

▼ 48. 스트림 병렬화는 주의해서 사용하라.

- Java 동시성 프로그래밍
    - 첫 릴리즈: 스레드, 동기화, `wait/notify` 지원
    - Java 5: 동시성 컬렉션인 `java.util.concurrent` 라이브러리, 실행자 프레임 워크
    - Java 7: 고성능 병렬 분해 (parallel decom-position)
    - Java 8: `parallel` 메서드로 파이프라인을 병렬 실행할 수 있는 스트림 지원
    - Java 는 동시성 프로그램을 작성하기 점점 쉬워지고 있지만, 이를 올바르고 빠르게 작성하는 일은 어려운 일이다.
    - 동시성 프로그래밍을 할 때는 안전성과 응답 가능 상태를 유지하기 위해 애써야 하는데, 병렬 스크림 파이프라인 프로그래밍에서도 다를 바 없다.
- Stream API 병렬화
    - ex) 스트림을 사용해 처음 20개의 메르센 소수를 생성하는 프로그램

```
public static void main(String [] args) {
    primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
        .filter(mersenne -> mesenne.isProbablePrime(50))
        .limit(20)
        .forEach(System.out::println);
}

static Stream<BigInteger> primes() {
    return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

        - 속도를 높이고 싶어 스트림 파이프라인의 `parallel()` 을 호출하면, 이 프로그램은 안타깝게도 아무것도 출력하지 못하면서 CPU 90%나 먹는 상태가 무한히 계속된다. (응답 불가) 상태에 빠지게된다.
        - 이렇게 느려진 원인은 스트림 라이브러리가 파이프라인을 병렬화하는 방법을 찾아내지 못했기 때문이다. 환경이 좋더라도 **데이터 소스가 `Stream.iterate` 이거나 중간 연산으로**

`limit` **을 쓰면 파이프라인 병렬화로는 성능 개선을 기대할 수 없다.**

- 스트림 파이프라인을 마구 병렬화하면 안 된다. 오히려 성능이 나빠질 수 있다.

- 병렬화 효과가 좋은 소스
  - `ArrayList` 인스턴스, 배열
  - `HashMap` 인스턴스, 배열
  - `HashSet` 인스턴스, 배열
  - `ConcurrentHashMap` 인스턴스, 배열
  - 배열, `int` 범위, `long` 범위
  - 위 자료구조들은 모두 데이터를 원하는 크기로 정확하고 손쉽게 나눌 수 있어서 일을 다수의 스레드에 분배하기에 좋다는 특징이 있다.
    - **나누는 작업은** `Spliterator` 가 담당하며, `Spliterator` 객체는 `Stream` 이나 `Iterable` 의 `spliterator` 메서드로 얻어 올 수 있다.
  - 위 자료구조는 원소들을 순차적으로 실행할 때의 **참조 지역성**이 뛰어나다.
    참조 지역성 (이웃한 원소의 참조들이 메모리에 순차적으로 저장됨)
    - 참조 지역성이 낮으면, 스레드는 데이터가 주 메모리에서 캐시 메모리로 전송되어 오기를 기다리며 대부분 시간을 멍하니 보내게 된다. 따라서 참조 지역성은 다량의 데이터를 처리하는 벌크 연산을 병렬화할 때 아주 중요한 요소로 작용한다.
- 병렬화 효과가 좋은 파이프 라인의 종단 연산
  - 파이프 라인의 종단 연산 역시 병렬화 성능에 영향을 준다.
  - 종단 연산에서 수행하는 작업량이 파이프라인 전체 작업에서 상당 비중을 차지하면서 순차적인 연산이라면 파이프라인 병렬 수행의 효과는 제한될 수 밖에 없다.
  - 병렬화에 적합한 종단 연산으로는 **축소 (`reduction`)** 이다.
    - 축소는 파이프라인에서 만들어진 모든 원소를 하나로 합치는 작업으로, `Stream` 의 `reduce` 메서드 중 하나, 혹은 `min`, `max`, `count`, `sum` 같이 완성된 형태로 제공하는 메소드 중 하나를 선택해 수행한다.
    - `anyMath`, `allMacth`, `noneMatch` 처럼 조건에 맞으면 바로 반환되는 메서드도 병렬화에 적합하다.
    - 반면, 가변 축소를 수행하는 `Stream` 의 `collect` 메서드는 병렬화에 적합하지 않다.컬렉션을 합치는 부담이 크기 때문
- 직접 구현한 `Stream`, `Iterable`, `Collection` 의 병렬화 이점을 얻고 싶은 경우 `spliterator` 메서드를 재정의한 뒤 테스트를 통해 성능 테스트를 진행해라.
- 병렬화는 주의해서 사용해야한다.
  - 스트림을 잘못 병렬화 하면 성능이 나빠질 뿐만아니라, 안전 실패(결과 자체가 잘못되거나 예상 못한 동작)를 할 수 있다. 안전 실패는 병렬화 파이프라인이 사용하는 `mappers`, `filters`, 프로그래머가 제공한 다른 함수 객체가 명시대로 동작하지 않을 때 벌어질 수 있다.

        - `Stream` 명세는 이때 사용되는 함수 객체 관한 엄중한 규약을 정의 했다.
            - `Stream` 의 `reduce` 연산에 건네지는 `accumulator`, `combiner` 함수는 반드시 결합법칙을 만족하고, 간섭받지 않고, 상태를 갖지 말아야 한다.
    - ~~이럴거면 안 쓰고 말지..~~라고 할 수 있다. 하지만, **조건이 잘 갖춰지면 `parallel` 메서드 호출 하나로 거의 프로세서 코어 수에 비례하는 성능 향상을 얻을 수 있다.**
        - ex) 소수 계산 스트림, π(n) n 보다 작거나 같은 소수의 개수를 계산하는 수

```
// 병렬화 전
static long pi(long n) {
    return LongStream.rangeClosed(2, n)
        .mapToObj(BigInteger::valueOf)
        .filter(i -> i.isProbablePrime(50))
        .count();
}

// 병렬화 후
static long pi(long n) {
    return LongStream.rangeClosed(2, n)
        .parallel()
        .mapToObj(BigInteger::valueOf)
        .filter(i -> i.isProbablePrime(50))
        .count();
```

        - 병렬화를 적용하기 위해 조건은 맞춰 졌다.
            - 무한 스트림이 아닌 유한 스트림
            - `limit` 제한을 하지 않음으로 적절히 값을 분할 가능
            - `count()` 메서드로 병렬화에 적합한 종단 연산
            - 순차적으로 정렬된 `LongStream.rangeClosed` 참조 지역성
        - 책에서는 성능이 병렬 후 3.37 배 빨라짐
- 정리
    - 계산 도 올바로 수행하고 성능도 빨라질 거라는 확신 없이는 스트림 파이프라인 병렬화를 시도조차 하지말자
    - 스트림을 잘못 병렬화하면 프로그램을 오동작하게 하거나 성능을 급격히 떨어트린다.
    - 병렬화하는 편이 낫다고 믿고라도, 수정 후의 코드가 여전히 정확한지를 확인하고 운영 환경과 유사한 조건에서 수행해보며 성능지표를 관찰해라
    - 그때, 계산도 정확하고 성능이 좋다고 확실 할 때만, 병렬화 버전 코드를 적용해라