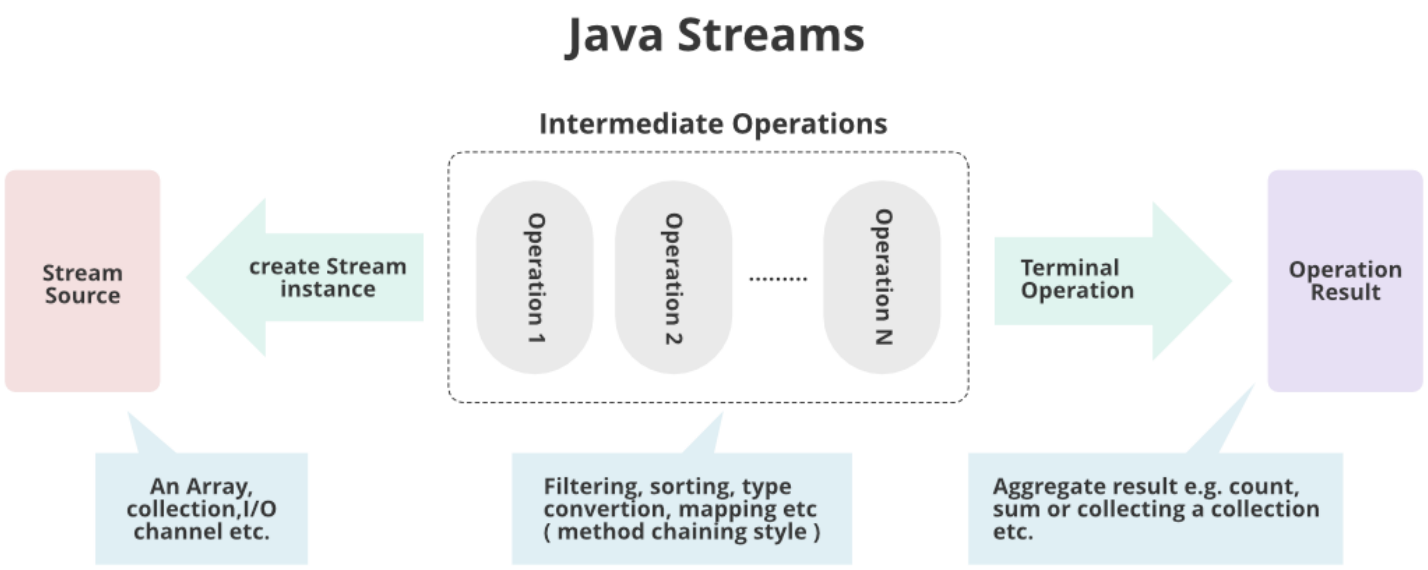
아이템 45) 스트림은 주의해서 사용하라

스트림이란

- `스트림` : 데이터 원소의 유한/무한 시퀀스
- `스트림 파이프라인` : 데이터 원소들로 수행하는 연산 단계를 표현하는 개념
  - 소스 스트림에서 시작해 종단연산으로 끝남
  - 그 사이에 하나 이상의 중간 연산이 있을 수 있음 (중간 연산 - 스트림을 어떠한 방식으로 변환)
  - 중간연산은 한 스트림 → 다른 스트림으로 변환



스트림의 종단 연산과 중간 연산

- 종단 연산
  - **forEach(Consumer<? super T> consumer)** : Stream의 요소를 순회
  - **count()** : 스트림 내의 요소 수 반환
  - **max(Comparator<? super T> comparator)** : 스트림 내의 최대 값 반환
  - **min(Comparator<? super T> comparator)** : 스트림 내의 최소 값 반환
  - **allMatch(Predicate<? super T> predicate)** : 스트림 내에 모든 요소가 predicate 함수에 만족할 경우 true
  - **anyMatch(Predicate<? super T> predicate)** : 스트림 내에 하나의 요소라도 predicate 함수에 만족할 경우 true
  - **noneMatch(Predicate<? super T> predicate)** : 스트림 내에 모든 요소가 predicate 함수에 만족하지않는 경우 true
  - **sum()** : 스트림 내의 요소의 합 (IntStream, LongStream, DoubleStream)
  - **average()** : 스트림 내의 요소의 평균 (IntStream, LongStream, DoubleStream)
- 중간 연산
  - f**ilter(Predicate<? super T> predicate)** : predicate 함수에 맞는 요소만 사용하도록 필터
  - **map(Function<? Super T, ? extends R> function)** : 요소 각각의 function 적용
  - **flatMap(Function<? Super T, ? extends R> function)** : 스트림의 스트림을 하나의 스트림으로 변환
  - **distinct()** : 중복 요소 제거
  - **sort()** : 기본 정렬
  - **sort(Comparator<? super T> comparator)** : comparator 함수를 이용하여 정렬
  - **skip(long n)** : n개 만큼의 스트림 요소 건너뜀
  - **limit(long maxSize)** : maxSize 갯수만큼만 출력

스트림 파이프라인의 지연 연산과 최적화

- `지연 평가` : 결과값이 필요할 때까지 계산을 늦추는 기법
  - 대용량 데이터에서, 실제로 필요하지 않은 데이터들을 탐색하는 것을 방지해 속도를 높일 수 있음

1. 스트림 파이프라인 실행시, JVM 은 곧바로 스트림 연산 실행 X
2. 최소한으로 필수적인 작업만 수행하고자 지연 연산을 위한 준비 작업을 수행
   (스트림 파이프라인이 **어떤 중간연산과 종단 연산으로 구성되어있는지**에 대한 검사)
3. 이를 바탕으로 JVM 은 사전에 최적화 방법을 찾아내 계획함
4. 해당 계획에 따라 개별 요소에 대한 스트림 연산을 수행함

- 종단 연산이 호출될 때 이루어지며, 종단 연산에 쓰이지 않는 데이터 원소는 계산에 쓰이지 않음 **(Short-Circuit 방식)**
  - limit(n) 연산이 내부적으로 자신에게 도달한 요소가 n 개가 되었을 때 스트림 내 다른 요소들에 대해 더 이상 순회하지 않고 탈출하도록 만들었기 때문에, 아래와 같은 출력 결과가 나옴

```
void test() {
  List<String> list = List.of("abcde", "asdfasdf", "aa", "zzzzzzzz", "bbb");
    list.stream()
      .filter(x -> x.length() >= 5)
      .peek(x -> System.out.println("intermediate : " + x));
      .limit(2)
      .forEach(x -> System.out.println("terminate : " + x));

intermediate : abcde
terminate : abcde
intermediate : asdfasdf
terminate : asdfasdf
```

- 지연평가가 무한 스트림을 다룰 수 있게 해주는 열쇠
  - 크기가 정해져있지 않으므로 중복 제거가 불가
    → limit () 과 같은 short-circuit 연산을 통해 유한 스트림으로 변환함으로써 가능해짐
  - 중복을 제거하는 `distinct()` 나, 전체 데이터를 정렬하는 `sort()` 연산들을 `Stateful` 연산이라고 함
    하지만 이는 **지연 평가를 무효화**시키고, 결과를 생성하기 전에 전체 데이터를 탐색하는 결과를 초래

```
void test() {
  List<String> list = List.of("abcde", "asdfasdf", "aa", "zzzzzzzz", "bbb");
    list.stream()
      .filter(x -> x.length() >= 5)
      .peek(x -> System.out.println("intermediate : " + x));
      .sorted()
      .limit(2)
      .forEach(x -> System.out.println("terminate : " + x));

intermediate : abcde
intermediate : asdfasdf
intermediate : zzzzzzzz
terminate : abcde
terminate : asdfasdf
```

- 종단 연산이 없는 스트림 파이프라인은 아무일도 하지 않는 명령어와 동일

Char 값을 처리할 때는 스트림 값을 삼가는 편이 나은 이유는 ?

1. **인코딩 문제**: 스트림은 기본적으로 바이트 기반이라, CHAR 값을 처리할 때 스트림을 사용하면 인코딩 문제가 발생할 수 있음
   - 특히, 문자 데이터를 바이트로 변환하고 다시 CHAR로 변환할 때
2. **텍스트 데이터의 추상화**: 자바에서는 `Reader` 및 `Writer` 클래스와 같은 텍스트 데이터를 다루기 위한 특수한 스트림 클래스가 제공되기에, 이 클래스를 사용하면 문자 데이터를 효과적으로 다룰 수 있고, 인코딩 및 디코딩 문제를 줄일 수 있음

스트림이 적합한 경우

- 원소들의 시퀀스를 일관되게 변환한다.
- 원소들의 시퀀스를 필터링한다.
- 원소들의 시퀀스를 하나의 연산을 사용해 결합한다. (더하기, 연결하기, 최소값 등..)
- 원소들의 시퀀스를 컬렉션에 모은다

- 원소들의 시퀀스에서 특정 조건을 만족하는 원소를 찾는다.

스트림이 적합하지 않은 경우

- 데이터가 파이프라인의 여러 단계(stage)를 통과할 때 이 데이터의 각 단계에서의 값들에 동시에 접근하기 어려운 경우
- 스트림 파이프라인은 한 값을 다른 값에 매핑하고 나면 원래의 값을 잃는 구조이기 때문

결론

- 스트림을 과도하게 사용하면 읽기 어렵고 유지보수가 힘든 코드가 만들어지므로, 모든 반복문을 스트림으로 바꾸기보단 둘 다 테스트해보고 더 나은 쪽을 선택하라