Java Stream 사용시 흔히 하는 실수

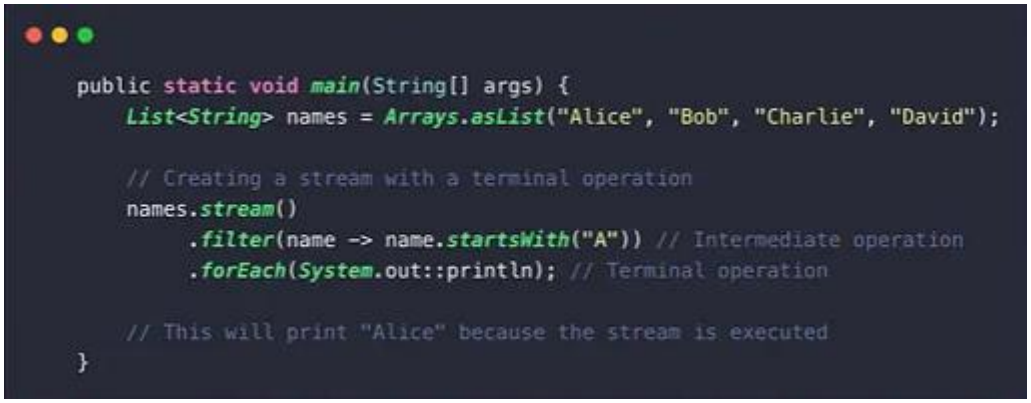
1. 터미널 연산을 사용하지 않는 경우
   collect(), forEach(), reduce()와 같은 터미널 연산을 호출하는 것을 잊으면 코드가 실행되지 않는다.

```java
public static void main(String[] args) {
    List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

    // Creating a stream without a terminal operation
    names.stream()
         .filter(name -> name.startsWith("A")); // No terminal operation is called here

    // Nothing gets printed because the stream is not executed
    System.out.println("Stream operations have not been executed.");
}
```

Stream은 forEach, collect, reduce와 같은 종료 연산이 있어야만 실행되는데 위 코드에서는 종료 연산없이 filter로 마무리되기 때문에 결과값이 출력되지 않는다.

출력

Stream operations have not been executed



```java
public static void main(String[] args) {
    List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

    // Creating a stream with a terminal operation
    names.stream()
         .filter(name -> name.startsWith("A")) // Intermediate operation
         .forEach(System.out::println); // Terminal operation

    // This will print "Alice" because the stream is executed
}
```

위와 같이 Stream은 종료 연산으로 마무리 되어야지 원하는 결과값이 출력된다.

출력

Alice

Stream 연산에는 중간 연산과 종료 연산이 있는데 중간 연산의 return 타입은 Stream이고 종료 연산의 return 타입은 Stream이 아니다.

중간연산	종료 연산
Filter	forEach
Map	Collect
Sorted	Reduce
Distinct	Count
limit	findFirst

2. 스트림 처리 중에 데이터 구조 변경 시 예상치 못한 오류가 발생할 수 있다.

```java
public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

        // Attempt to modify the source list while streaming
        names.stream()
             .filter(name -> {
                 if (name.startsWith("B")) {
                     names.remove(name); // Modifying the source list
                 }
                 return true;
             })
             .forEach(System.out::println);

        // Output might not be as expected due to concurrent modification
        System.out.println("Remaining names: " + names);
    }
```

위 코드와 같이 리스트 자료구조의 데이터를 stream 처리 중에 "B"로 시작하는 이름을 제거 후 나머지 값들을 출력하는 코드를 실행 결과 Alice만 출력되고 이후의 값들은 출력되지 않는 것을 확인할 수 있었다.

```java
public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

        // Create a new list based on filtered results
        List<String> filteredNames = names.stream()
                                          .filter(name -> !name.startsWith("B")) // Filter out names starting with 'B'
                                          .collect(Collectors.toList());

        // Display the filtered list
        System.out.println("Filtered names: " + filteredNames);
        System.out.println("Original names remain unchanged: " + names);
    }
```

위와 같은 일을 방지하기 위해서 stream 처리 중 원본 데이터의 자료구조를 조정하는 것보다는 위와 같이 filter 처리가 된 자료구조를 따로 만드는 방식으로 처리하여야 결과값 출력에 오류가 발생하지 않는다.

```java
public static void S2_new() {
    List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

    List<String> filteredNames = names.stream()
                                      .filter(name -> !name.startsWith("B"))
                                      .toList();

    System.out.println("Filtered names: " + filteredNames);
    System.out.println("Original names remain unchanged: "+ names);
}
```

수행시간을 단축시키기 위해 collect 함수 대신 바로 toList 함수를 사용해보았으나 수행시간에는 유의미한 차이가 발생하지 않았다.

3. Parallel Stream 오버헤드 무시

   Parallel Stream은 소규모 데이터 또는 가벼운 작업에서는 항상 성능 향상으로 이어지지 않는다.

Parallel Stream?

Stream 작업을 여러 스레드로 나눠서 실행하는 것으로 CPU를 동시에 써서 빠르게 작업을 처리하는 것을 목표로 한다.

```java
public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5); // Small dataset

        // Using a parallel stream on a small dataset
        numbers.parallelStream()
               .map(n -> {
                   // Simulating a lightweight operation
                   System.out.println(Thread.currentThread().getName() + " processing: " + n);
                   return n * n;
               })
               .forEach(System.out::println);

        // Output may show threads being created unnecessarily for a simple task
    }
```

위 작업은 데이터가 5개 밖에 없고 n * n의 가벼운 연산을 수행하는데 ParallelStream을 사용하여 스레드 분배, 작업 분할, 결과 병합을 하여 오히려 오버헤드가 발생하여 수행 시간이 더 늘어난다. 따라서 이런 경우는 stream을 사용하는 것이 더 빠르다.

```java
public static void main(String[] args) {
        List<Integer> numbers = IntStream.rangeClosed(1, 1_000_000) // Large dataset
                                         .boxed()
                                         .collect(Collectors.toList());

        // Using parallel stream for a CPU-bound operation on a large dataset
        List<Integer> squaredNumbers = numbers.parallelStream()
                                              .map(n -> {
                                                  // Simulating a CPU-bound operation
                                                  return n * n;
                                              })
                                              .collect(Collectors.toList());

        // Print the first 10 results
        System.out.println("First 10 squared numbers: " + squaredNumbers.subList(0, 10));
    }
```

다음과 같이 데이터의 개수가 100,000개인 경우에 parallelstream을 사용하게 되면 stream 보다 더 빠르게 작업을 수행할 수 있다.

4. 중간 연산을 과도하게 사용하는 경우 성능 저하를 발생시킬 수 있다.

```java
public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eve");

        // Overusing intermediate operations
        List<String> result = names.stream()
            .filter(name -> name.startsWith("A")) // First intermediate operation
            .filter(name -> name.length() > 3)    // Second intermediate operation
            .map(String::toUpperCase)              // Third intermediate operation
            .map(name -> name + " is a name")     // Fourth intermediate operation
            .toList();                             // Terminal operation

        // Output the result
        System.out.println(result);
    }
```

위와 같이 중간 연산 filter, map 등을 과도하게 사용하면 작업의 수행 시간이 더 오래 걸리는 것을 확인할 수 있었다. 중간 연산자를 여러 번 사용하기 보다는 ||, &&와 같은 연산자를 사용해서 중간 연산자 사용의 횟수를 줄이는 것이 성능에 더 이롭다.

(중간 연산자를 여러 번 사용 시 성능 하락 이유)
.filter() 메소드 같은 중간 연산자들은 interface이므로 이를 사용하기 위해서는 람다식을 넘겨 구현체를 생성해야 하기에 이를 여러 번 사용하면 구현체를 생성하는 횟수가 많아져 오버헤드가 증가할 것이라고 생각한다.

(실제 수행시간 비교)
약 7~10배 정도의 수행 시간 차이가 발생한다.

```
[STUDY, ZLOUD]
Intermediate Operations 과도하게 사용 시 : 0.0096651
[STUDY, ZLOUD]
Intermediate Operations 한번씩만 사용 시 : 0.0010014
```

(결론)
코드 가독성이 떨어지더라도 중간 연산자를 여러 번 사용하는 것보다 하나 안에 사용하는 것이 성능면에서 더 유리하다고 본다.

5. findFirst(), reduce()와 같은 연산을 사용할 때는 결과값이 존재하는지 확인해야 한다.

```java
public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        // Attempting to find a name starting with "Z" (which doesn't exist)
        String firstNameStartingWithZ = names.stream()
            .filter(name -> name.startsWith("Z"))
            .findFirst() // Returns an Optional
            .get(); // This will throw NoSuchElementException if the Optional is empty

        // Output the result
        System.out.println(firstNameStartingWithZ);
    }
```

findFirst()의 값이 없는 상태에서 get()을 사용하므로 오류가 발생한다.

```java
public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        // Properly handling Optional
        Optional<String> firstNameStartingWithZ = names.stream()
            .filter(name -> name.startsWith("Z"))
            .findFirst(); // Returns an Optional

        // Check if the Optional is present
        if (firstNameStartingWithZ.isPresent()) {
            System.out.println(firstNameStartingWithZ.get());
        } else {
            System.out.println("No name starts with 'Z'");
        }
    }
```

따라서 이런 함수들을 사용할 때에는 값의 존재 유무를 먼저 확인해야 한다.

```java
public static void S5_new() {
    List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

    String firstNameStartingWithZ = names.stream()
            .filter(name -> name.startsWith("Z"))
            .findFirst()
            .orElse("No name starts with Z");

    System.out.println(firstNameStartingWithZ);
}
```

수행시간을 줄이기 위해 조건문 대신 orElse() 함수를 사용해 보았으나 유의미한 변화는 없었다.

6. 스레드 안전성 무시

   ParallelStream에서 공유 가능한 자원을 사용하면 race condition이 발생하여 일관성 없는 결과가 출력된다.

```java
public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        List<Integer> results = new ArrayList<>(); // Shared mutable state

        // Using parallel stream with shared mutable state
        numbers.parallelStream().forEach(number -> {
            results.add(number * 2); // This can cause race conditions
        });

        // Output the results
        System.out.println("Results: " + results);
    }
```

parallelstream을 사용할 때 mutable한 공유 자원을 조작하여 race condition이 발생한다.

(이유)

멀티쓰레드 환경에서 ArrayList와 같은 자료구조들은 lock이 없어 경합이 발생할 가능성이 있기 때문이다.

(검증)

```
results의 예상 사이즈 : 10000
results의 실제 사이즈 : 8401
results의 예상 사이즈 : 10000
results의 실제 사이즈 : 10000
```

1000개의 데이터를 lock이 없는 ArrayList와 lock이 존재하는 CopyOrWriteArrayList를 사용하여 코드를 수행시킨 결과 lock이 없는 ArrayList에서는 경합이 발생하여 결과값의 수가 데이

터의 수와 다른 것을 확인할 수 있었고 lock이 존재하는 CopyOrWriteArrayList에서는 데이터의 개수와 같은 1000개가 출력되는 것을 확인할 수 있었다.

(결론)
멀티스레드 환경에서 자료를 다룰 때에는 멀티스레드 환경에서도 안전항 자료구조를 사용해야 한다.