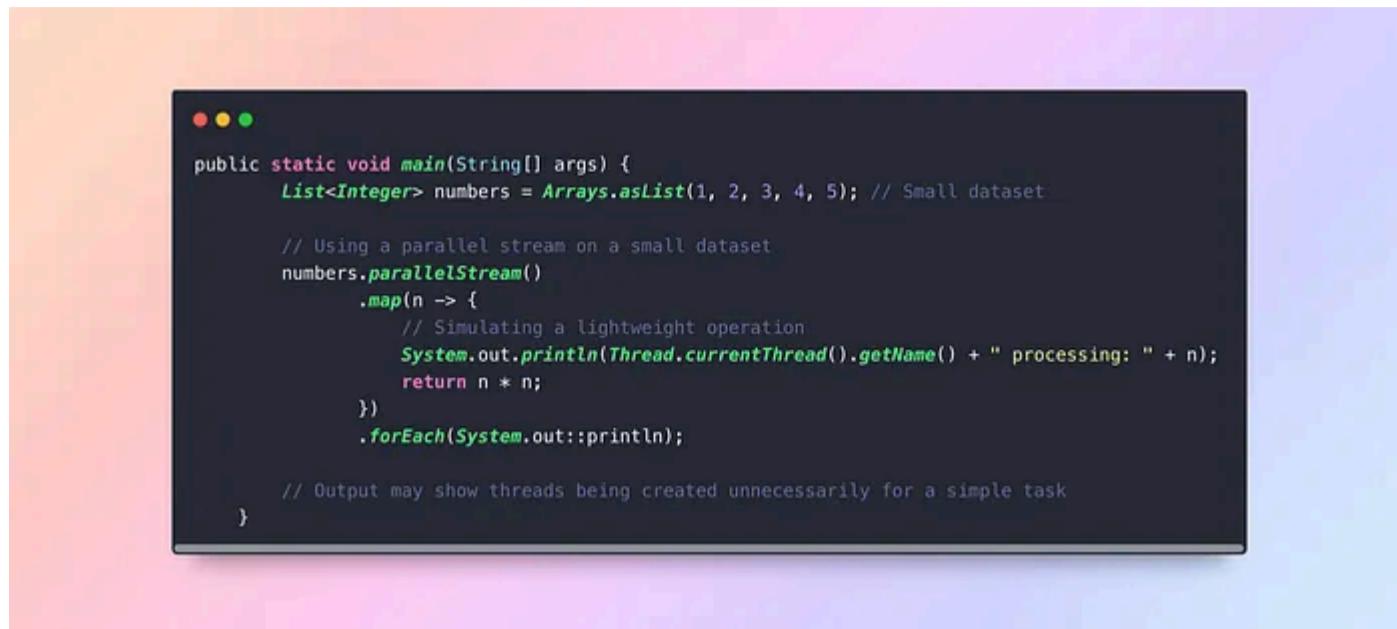


problem3

3. Ignoring Parallel Stream Overhead:Mistake: Thinking parallel streams always lead to performance improvements without considering context, e.g small data sets or lightweight operations.

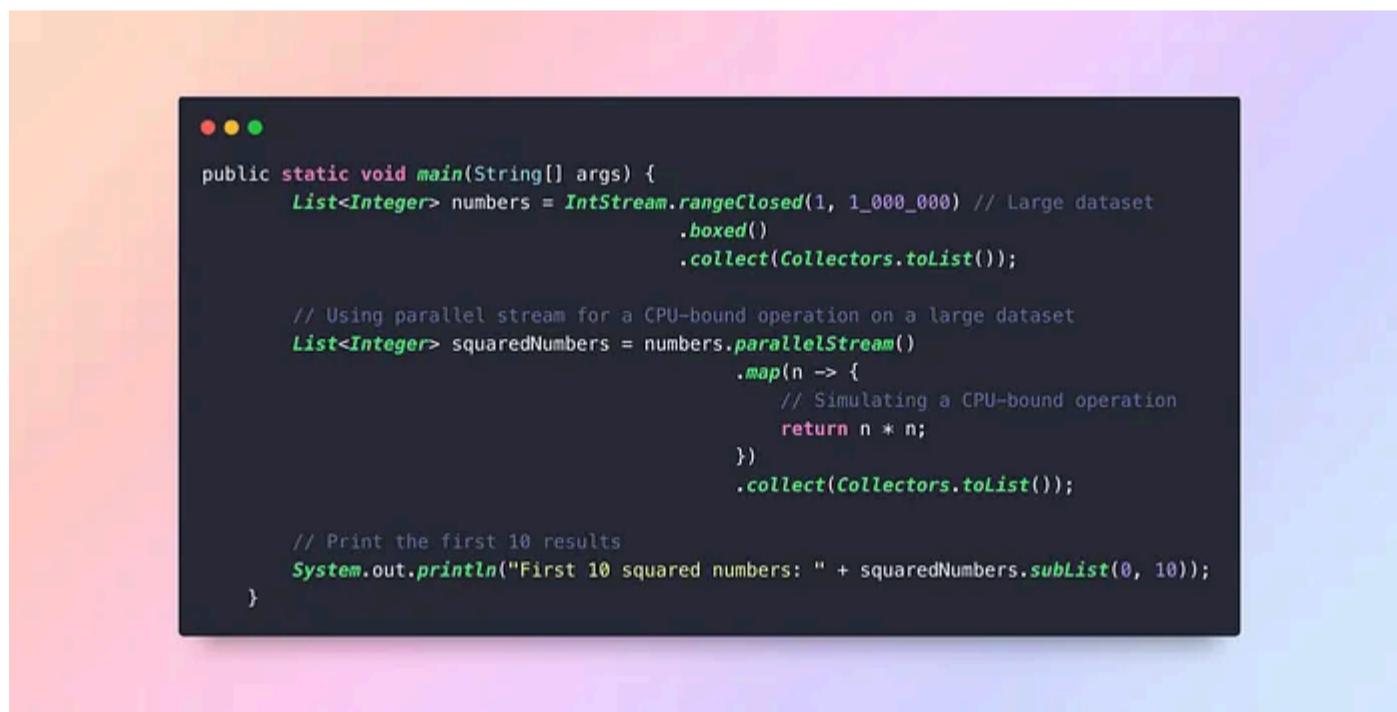


```
public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5); // Small dataset

    // 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
}
```

Solution: Be careful with parallel streams, especially for CPU-bound tasks with a large data set.



```
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));
}
```

1 parallelStream이 뭔데?

numbers.parallelStream()

- Stream 작업을 **여러 스레드**로 나눠서 실행
- 내부적으로 **ForkJoinPool(스레드 풀)** 사용
- 목표: **CPU를 동시에 써서 빠르게 처리**

2 첫 번째 코드 (X 오히려 손해인 경우)

🔍 코드 상황

List<Integer> numbers = Arrays.asList(1,2,3,4,5); // 작은 데이터

```
numbers.parallelStream()
    .map(n -> {
        System.out.println(Thread.currentThread().getName());
        return n * n;
    })
```

```
    })  
    .forEach(System.out::println);
```

! 문제점

- 데이터: **5개밖에 없음**
 - 연산: $n * n$ → **너무 가벼움**
 - 그런데도...
 - 스레드 분배
 - 작업 분할
 - 결과 병합
- 👉 **오버헤드(추가 비용)** 발생

💥 결과

| 차라리 그냥 stream()이 더 빠름

| ↪ 즉,

| “일은 작은데, 사람만 많이 불러온 상황”

3 오버헤드(Overhead)가 뭐냐면

parallelStream을 쓰면 자동으로 생기는 비용들:

- 스레드 분할
- 스레드 간 컨텍스트 스위칭
- 작업 병합
- 메모리 접근 비용

👉 이 비용이 실제 계산보다 커질 수 있음

4 두 번째 코드 (⭕ 쓸 만한 경우)

🔍 코드 상황

```
List<Integer> numbers = IntStream.rangeClosed(1,1_000_000)  
    .boxed()  
    .collect(Collectors.toList());  
  
List<Integer> squaredNumbers = numbers.parallelStream()  
    .map(n → n * n)  
    .collect(Collectors.toList());
```

✅ 왜 여긴 괜찮을까?

- 데이터: **100만 개**
- 연산: CPU 계속 사용
- 작업이 충분히 큼 → 스레드 나눌 가치 있음

| ↪ 이 경우는

| “일도 많고, 사람도 여러 명 필요한 상황”

5 문장 해석 다시 해보자 (중요)

✖ Mistake

| Thinking parallel streams always lead to performance improvements

👉 "parallelStream 쓰면 항상 빨라진다고 생각하는 것"

✖ 틀린 생각

✖ without considering context

| small data sets or lightweight operations

👉 데이터가 적거나

👉 연산이 가벼운데도 parallelStream 쓰는 경우

✓ Solution

| Be careful with parallel streams

👉 무조건 쓰지 말고 상황 봐라

| especially for CPU-bound tasks with a large data set

👉 데이터 많고

👉 CPU 계산이 많은 경우에만 효과 있음

6 실무 기준 정리 🔥

✖ parallelStream 쓰면 안 좋은 경우

- 데이터 개수 적음 (수십~수천)
- 단순 연산 (+1, 2)
- I/O 작업(DB, 파일, 네트워크)
- 순서 중요할 때

✓ parallelStream 써볼 만한 경우

- 데이터 많음 (수십만~)
- CPU 연산 위주
- 독립적인 계산
- 성능 테스트로 효과 확인한 경우