1. Map do not support Stream
2. Parallel Stream is handle in multi-thread, do not guarantee order

🔷 2. Parallel Stream

- Created with `collection.parallelStream()` or `stream().parallel()`.
- **Parallel execution**: splits elements into **multiple chunks** and processes them in **different threads** (using ForkJoinPool).
- No guarantee of order (unless you use ordered operations like `.forEachOrdered()`).
- Best for **large datasets** or CPU-heavy operations.

👉 Example:

java — Copy code

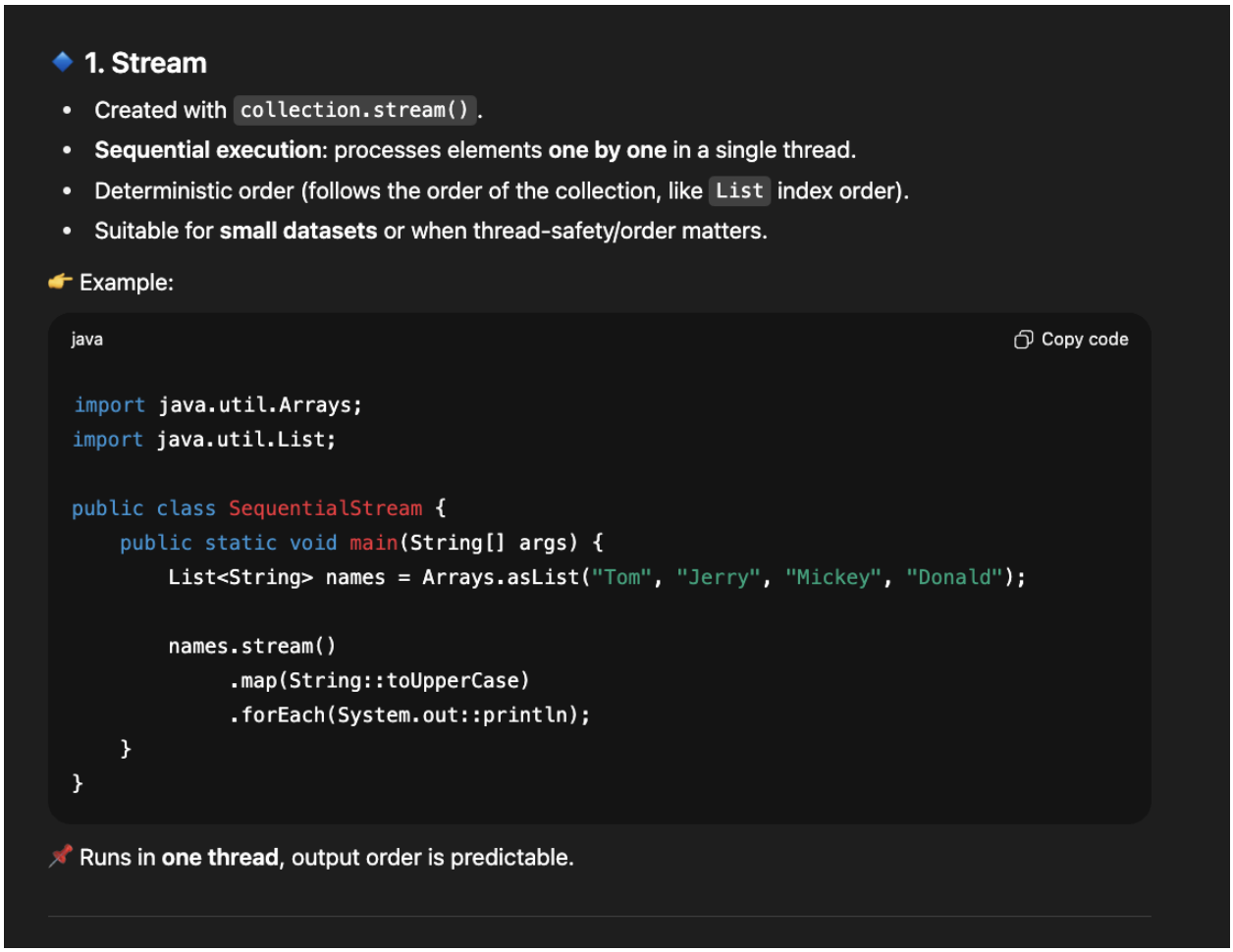
```java
import java.util.Arrays;
import java.util.List;

public class ParallelStreamDemo {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Tom", "Jerry", "Mickey", "Donald");

        names.parallelStream()
             .map(String::toUpperCase)
             .forEach(System.out::println); // Order may vary
    }
}
```

📌 May print in different orders, and runs on multiple threads.

3. Sequence Stream is in order

4. 

🔷 1. Stream

- Created with `collection.stream()`.
- **Sequential execution**: processes elements **one by one** in a single thread.
- Deterministic order (follows the order of the collection, like `List` index order).
- Suitable for **small datasets** or when thread-safety/order matters.

👉 Example:

java — Copy code

```java
import java.util.Arrays;
import java.util.List;

public class SequentialStream {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Tom", "Jerry", "Mickey", "Donald");

        names.stream()
             .map(String::toUpperCase)
             .forEach(System.out::println);
    }
}
```

📌 Runs in **one thread**, output order is predictable.