```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eve");

long count = names.stream()
        .filter(name -> name.startsWith("C")) // Intermediate: Filter names starting with "C"
        .map(String::toUpperCase) // Intermediate: Convert to uppercase
        .sorted() // Intermediate: Sort alphabetically
        .forEach(System.out::println); // Terminal: Print each name

System.out.println("Count: " + count);  // Terminal: Get the count of filtered names
```

Let's delve deeper into Java Streams. They are a powerful feature introduced in Java 8 that provide a functional way to process collections of data. Think of them as a sequence of elements that support various operations, but importantly, these operations don't modify the original data source. Streams are about *what* you want to do, not *how* to do it, allowing for more concise and often more parallelizable code.

Here's a breakdown of key concepts and characteristics:

1. What are Streams?

- **Sequences of Elements:** A stream is a sequence of elements of a particular type (e.g., `Stream<String>`, `Stream<Integer>`, `Stream<Product>`). It's not a data structure itself; it's a way to process data from a source (like a collection, array, or I/O channel).
- **Functional in Nature:** Streams promote functional programming principles. Operations on streams typically take lambda expressions as arguments, allowing you to express your data processing logic in a declarative way.
- **Immutable Data:** Streams don't modify the original data source. Operations on streams create new streams with the transformed data.

- **Lazy Evaluation:** Operations on streams are often executed lazily, meaning they are only performed when a terminal operation is encountered. This can improve efficiency by avoiding unnecessary computations.

2. Stream Operations:

Streams support a variety of operations, which can be categorized into:

- **Intermediate Operations:** These operations transform a stream into another stream. They are chained together to form a pipeline of operations. Examples include:
    - `filter(Predicate)`: Filters elements based on a condition.
    - `map(Function)`: Transforms each element using a function.
    - `distinct()`: Removes duplicate elements.
    - `sorted(Comparator)`: Sorts elements based on a comparator.
    - `limit(long)`: Limits the number of elements.
    - `skip(long)`: Skips a certain number of elements.
- **Terminal Operations:** These operations produce a result or perform a side effect. They mark the end of the stream pipeline. Examples include:
    - `forEach(Consumer)`: Performs an action for each element.
    - `collect(Collector)`: Collects elements into a collection (e.g., `List`, `Set`, `Map`).
    - `count()`: Returns the number of elements.
    - `reduce(identity, BinaryOperator)`: Combines elements into a single value using a binary operation.
    - `anyMatch(Predicate)`, `allMatch(Predicate)`, `noneMatch(Predicate)`: Match elements based on a predicate.

- `findFirst()`, `findAny()`: Find the first or any element.

3. Creating Streams:

You can create streams from various sources:

- **Collections:** `collection.stream()` or `collection.parallelStream()` (for parallel processing).
- **Arrays:** `Arrays.stream(array)`.
- **Int/Long/Double Streams:** `IntStream.range(start, end)`, `LongStream.of(longArray)`, etc.
- **Files:** `Files.lines(Path)`.
- **Other sources:** `Stream.of(elements)`, `Stream.generate()`, `Stream.iterate()`.

4. Example:

Java

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eve");

long count = names.stream()
        .filter(name -> name.startsWith("C")) // Intermediate: Filter names starting with "C"
        .map(String::toUpperCase) // Intermediate: Convert to uppercase
        .sorted() // Intermediate: Sort alphabetically
        .forEach(System.out::println); // Terminal: Print each name

System.out.println("Count: " + count);  // Terminal: Get the count of filtered names
```

Output:

```
CHARLIE
Count: 1
```

5. Parallel Streams:

Streams can be processed in parallel to take advantage of multi-core processors. Use `parallelStream()` instead of `stream()` to create a parallel stream. Be mindful of thread safety when using parallel streams.

6. Key Advantages of Streams:

- **Conciseness:** Reduces boilerplate code for data processing.
- **Readability:** Makes code more expressive and easier to understand.
- **Parallelism:** Enables easy parallel processing for performance gains.
- **Functional Programming:** Encourages a more functional style of programming.

7. When to Use Streams:

Streams are most effective when dealing with collections of data and performing operations like filtering, mapping, sorting, and aggregating. They are particularly useful when you can express your data processing logic in terms of functional operations.

8. When Not to Use Streams:

- For very simple loops where the overhead of creating a stream might outweigh the benefits.
- When you need to modify the original data source directly (streams don't do this).
- For complex algorithms that are not easily expressed with stream operations.

Streams are a powerful addition to Java and provide a more elegant and efficient way to process data. Understanding them well will significantly improve your Java coding skills.