Mastering Stream API in Java

Unlock modern, functional data processing

Stream API

Introduced in Java 8

Revolutionized collection handling and data processing.

Data Pipeline

A sequence of elements from a source (collections, arrays, etc.).

Functional Style

Supports declarative, concise operations on data.

Processes Data

Does not store data; it processes elements from its source.

Diverse Stream Creation Methods

From Collections & Arrays

```
List<String> list = Arrays.asList("A", "B");
Stream<String> stream = list.stream();

int[] numbers = {1, 2, 3};
IntStream numStream = Arrays.stream(numbers);
```

The most common way: leverage existing data structures directly.

Static Factories & Generators

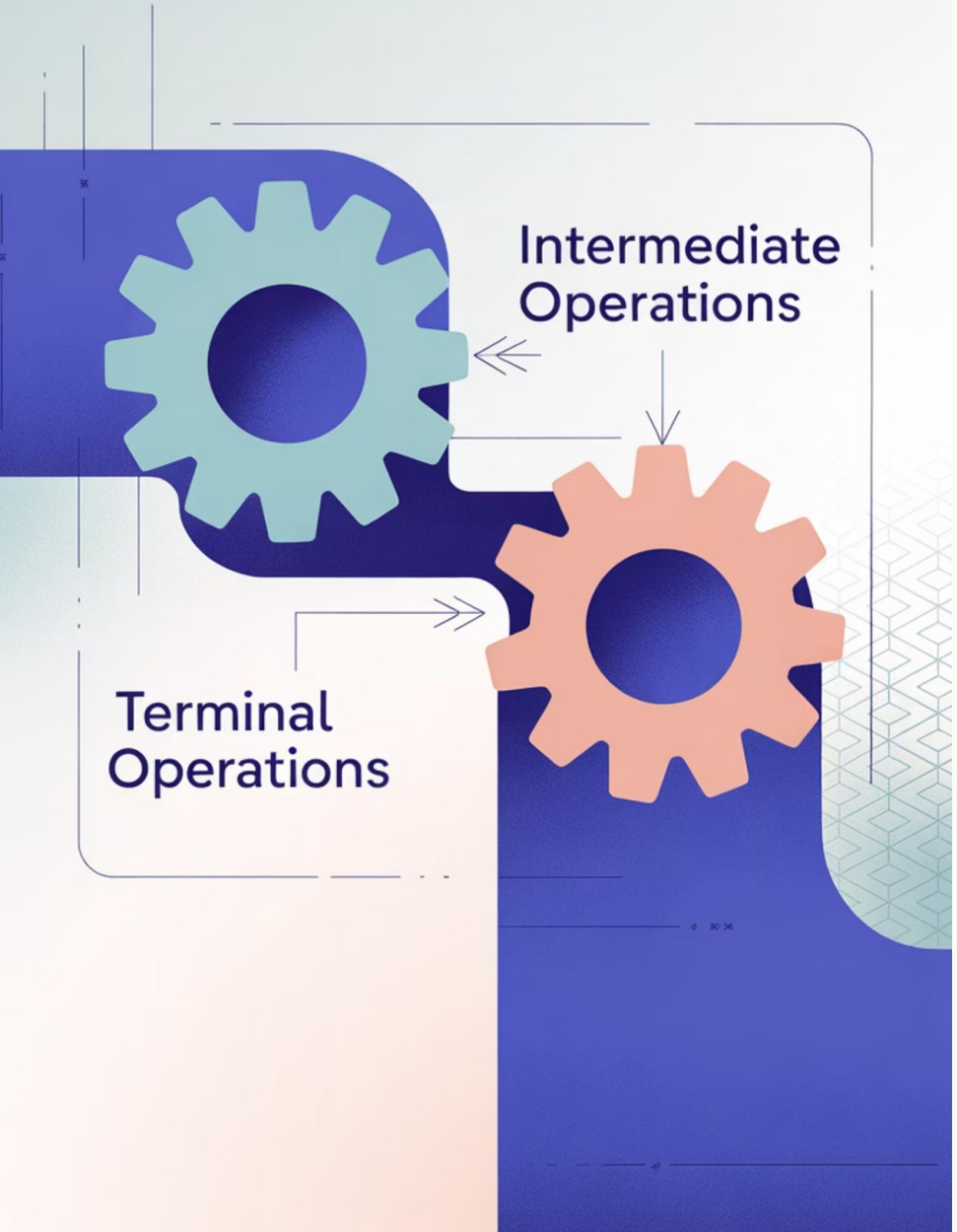
```
Stream<String> s = Stream.of("Java", "Python");

Stream<Double> randoms = Stream.generate(Math::random).limit(5);

Stream<Integer> evens = Stream.iterate(0, n -> n + 2).limit(5);
```

For arbitrary data or infinite sequences, with a `limit()` for finite results.

Streams can be generated from almost anywhere – collections, arrays or even infinite generators, providing immense flexibility for various data sources.



Stream Operations: The Pipeline

Intermediate Operations

Transform the stream: These operations are lazy and return a new `Stream`.
They build the pipeline without processing data until a terminal operation is called.

- `filter()`: Select elements based on a predicate.
- `map()`: Transform elements to a new type.
- `sorted()`: Sort elements.
- `distinct()`: Remove duplicates.
- `limit()`, `skip()`: Control stream size.

Terminal Operations

Produce the result: These operations are eager and consume the stream, triggering all intermediate operations. They produce a non-stream result.

- `collect()`: Accumulate elements into a collection.
- `forEach()`: Perform an action for each element.
- `reduce()`: Combine elements into a single result.
- `count()`: Return the number of elements.
- `anyMatch()`: Check if any element matches.

Intermediate Operations in Action: Filter & Map

```
List<String> names = Arrays.asList("John", "Jane", "Mike", "Sam");

names.stream().filter(n -> n.startsWith("J")) // Keeps "John", "Jane"
.map(String::toUpperCase)      // Transforms to "JOHN", "JANE"
.forEach(System.out::println); // prints JOHN JANE
```

Output:

```
JOHN JANE
```

This pipeline efficiently filters names starting with 'J' and then transforms them to uppercase.

Intermediate operations are **lazy** – they only execute when a terminal operation like `forEach()` is invoked

Advanced Intermediate Chaining

```
List<Integer> numbers = Arrays.asList(5, 2, 3, 2, 8, 5, 9);
numbers.stream().distinct()// Removes duplicate 2s, 5s -> [5, 2, 3, 8, 9]
.sorted()    // Sorts elements -> [2, 3, 5, 8, 9]
.skip(1)     // Skips the first element (2) -> [3, 5, 8, 9]
.limit(3)    // Takes the next 3 elements -> [3, 5, 8]
.forEach(System.out::println); // prints 3 5 8
```

Output:

```
3 5 8
```

Observe how multiple intermediate operations can be chained together to create expressive data transformation pipelines. Each step refines the stream, leading to a precise final result.

Terminal Operations: Getting Results

1

forEach()

```
list.stream().forEach(System.out::println);
```

Performs an action for **each element**. No return value.

2

collect()

```
Set<String> set = list.stream().collect(Collectors.toSet());
```

Gathers elements into a **new collection** or summary. Highly versatile.

3

reduce()

```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

Aggregates elements into **a single result** (e.g., sum, max, min).

4

count() & Matchers

```
long count = list.stream().count();
boolean hasEven = numbers.stream().anyMatch(n -> n % 2 == 0);
```

Counts **elements or checks if elements** satisfy a predicate.

Real-World Application: Employee Data Processing

```
class Employee {
String name;
int age;
double salary;

Employee(String name, int age, double salary) {
this.name = name;
this.age = age;
this.salary = salary;
}
}
List<Employee> employees = Arrays.asList(
new Employee("John", 25, 50000),
new Employee("Alice", 30, 60000),
new Employee("Bob", 35, 70000),
new Employee("Sam", 28, 45000)
);
List<String> highEarners = employees.stream()
                           .filter(e -> e.salary > 50000)
                           .map(e -> e.name)
                           .collect(Collectors.toList());

System.out.println(highEarners);
```

Output:

```
[Alice, Bob]
```

- This example demonstrates how Stream API simplifies complex data manipulation.
- We concisely filter for employees earning over 50,000, extract their names, and collect them into a new list.
- This replaces many lines of imperative loop-based code with a single, readable pipeline.

Key Benefits of Embracing Streams

Concise & Declarative

Write less code, focusing on "what" to do rather than "how" to do it.

Functional Paradigm

Aligns with modern functional programming principles, promoting purity.

Efficient Data Processing

Optimized for processing large datasets with minimal overhead.

Parallel Processing

Easily leverage multi-core processors with `parallelStream()` for enhanced performance.

Reduces Boilerplate

Eliminates verbose loops and conditional statements.

Recap & Next Steps

What is Stream API?

A functional pipeline for processing data sequences.

Creating Streams

From collections, arrays, `Stream.of()`, generators.

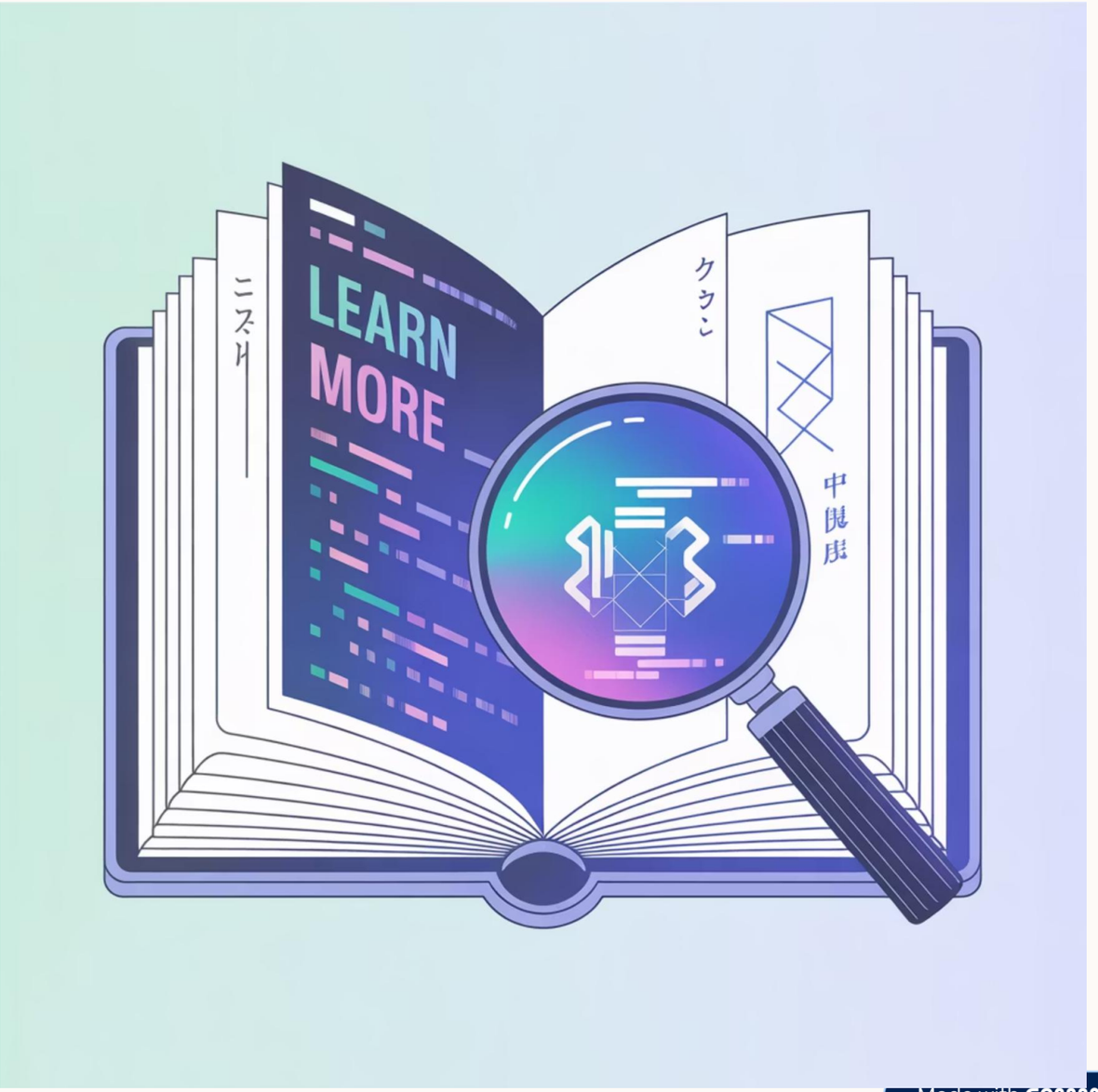
Intermediate Operations

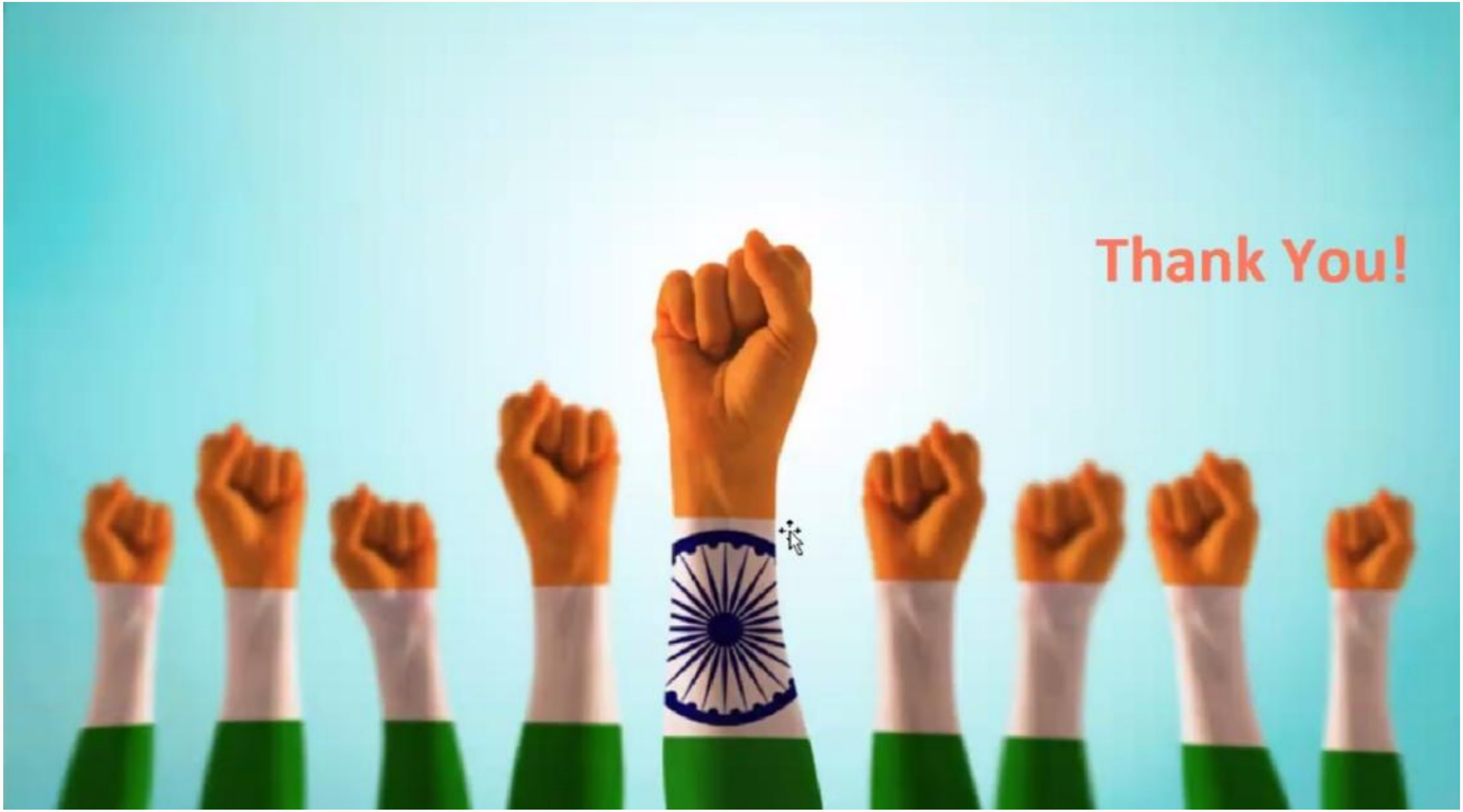
Lazy transformations that build the pipeline (e.g., filter, map).

Terminal Operations

Trigger execution and produce results (e.g., collect, forEach).

The Stream API, combined with Lambda Expressions, provides a modern, expressive, and highly efficient way to process data in Java.




Thank You!