Power of the Stream API *in Java*

What is the Stream API

The Stream API, introduced in Java 8, provides a functional programming approach to process sequences of elements in a declarative manner. It's not about storing data but rather about processing it!

Why Is the Stream API Important?

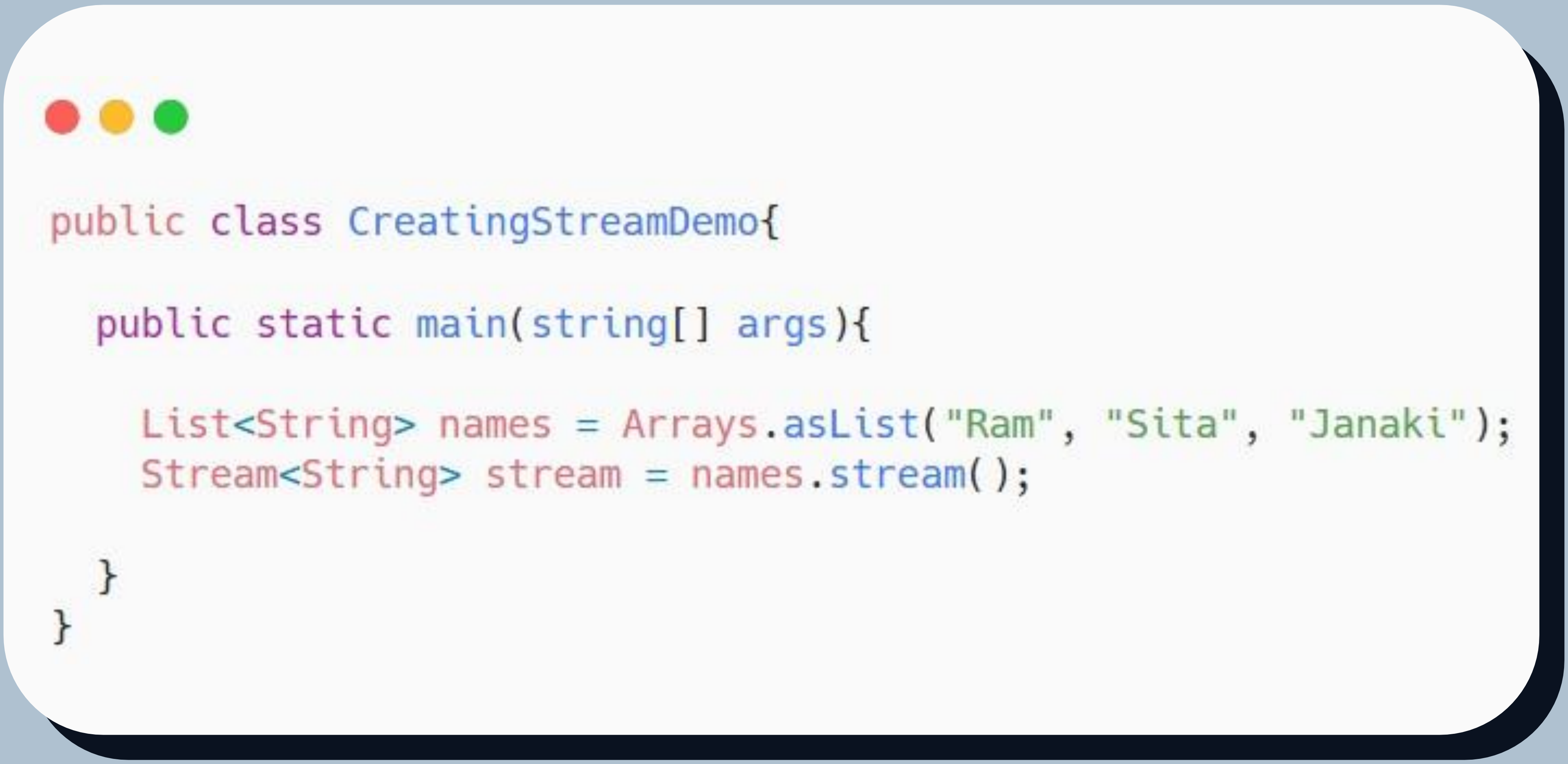
1. Simplifies data manipulation with functional operations like filtering, mapping, and reducing.

2. Enhances readability and reduces boilerplate code.

3. Supports parallel processing to leverage multi-core processors for better performance.

Creating a Stream

Streams can be created from collections, arrays, or custom sources.

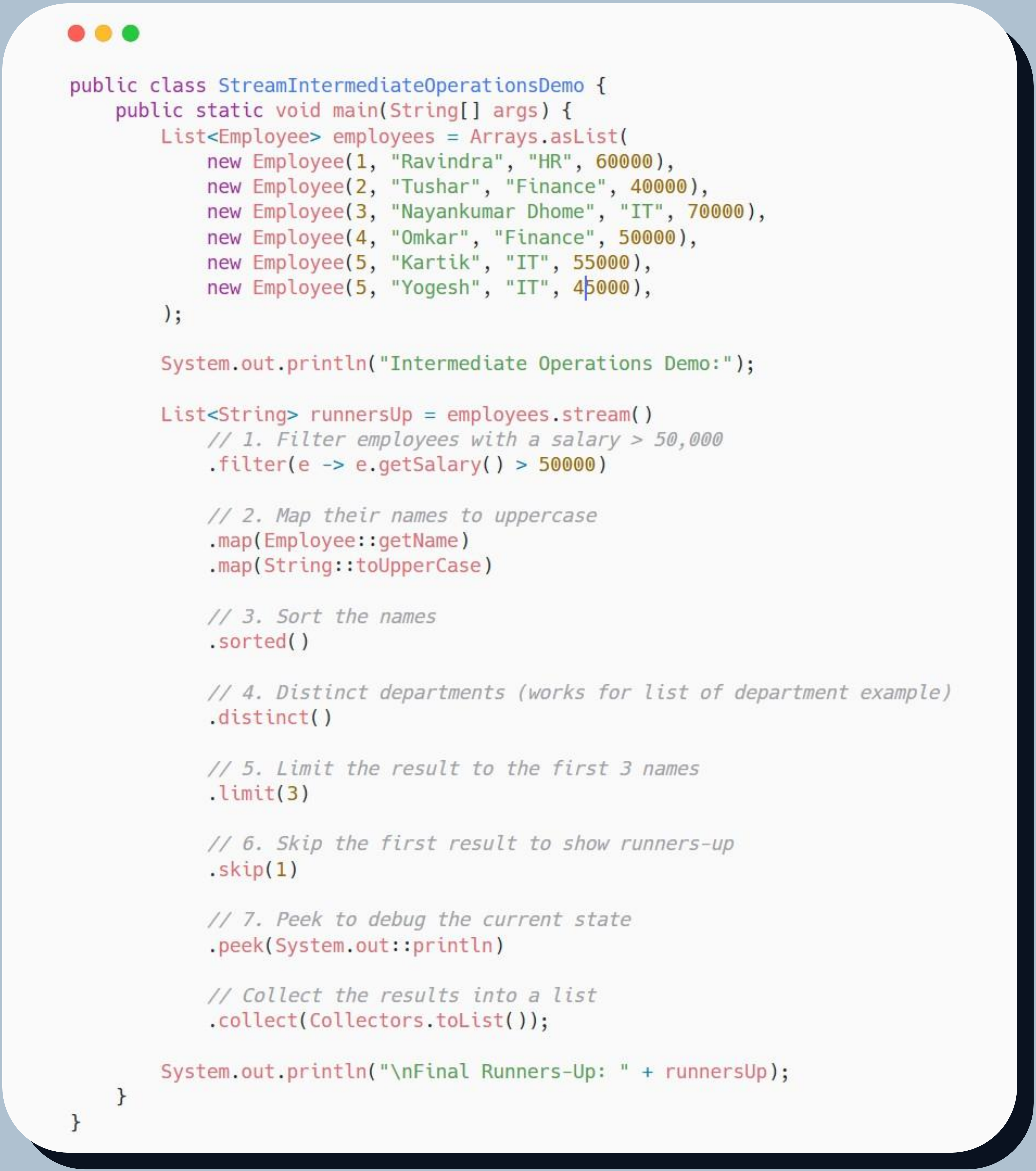


```
public class CreatingStreamDemo{

  public static main(string[] args){

    List<String> names = Arrays.asList("Ram", "Sita", "Janaki");
    Stream<String> stream = names.stream();

  }
}
```

Intermediate Operations

Used to transform or filter data; these operations are lazy and return a stream.

- filter(): Filters elements based on a condition.
- map(): Transforms each element using a function.
- sorted(): Sorts elements in natural or custom order.

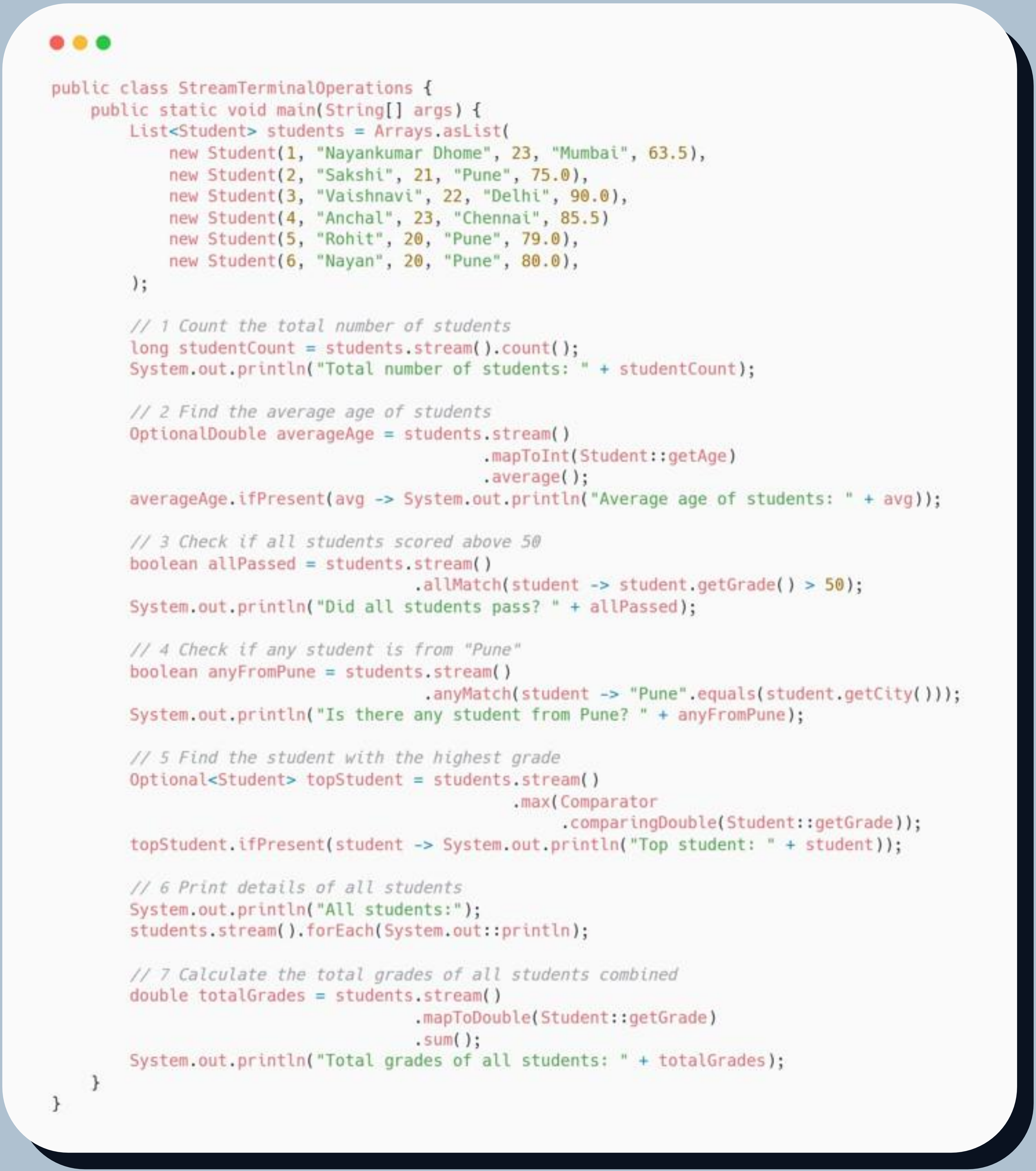


```java
public class StreamIntermediateOperationsDemo {
    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
            new Employee(1, "Ravindra", "HR", 60000),
            new Employee(2, "Tushar", "Finance", 40000),
            new Employee(3, "Nayankumar Dhome", "IT", 70000),
            new Employee(4, "Omkar", "Finance", 50000),
            new Employee(5, "Kartik", "IT", 55000),
            new Employee(5, "Yogesh", "IT", 45000),
        );

        System.out.println("Intermediate Operations Demo:");

        List<String> runnersUp = employees.stream()
            // 1. Filter employees with a salary > 50,000
            .filter(e -> e.getSalary() > 50000)

            // 2. Map their names to uppercase
            .map(Employee::getName)
            .map(String::toUpperCase)

            // 3. Sort the names
            .sorted()

            // 4. Distinct departments (works for list of department example)
            .distinct()

            // 5. Limit the result to the first 3 names
            .limit(3)

            // 6. Skip the first result to show runners-up
            .skip(1)

            // 7. Peek to debug the current state
            .peek(System.out::println)

            // Collect the results into a list
            .collect(Collectors.toList());

        System.out.println("\nFinal Runners-Up: " + runnersUp);
    }
}
```

Terminal Operations

Trigger the processing and return a non-stream result.

- forEach(): Iterates over elements.
- collect(): Gathers results into a collection.
- reduce(): Reduces elements to a single value.



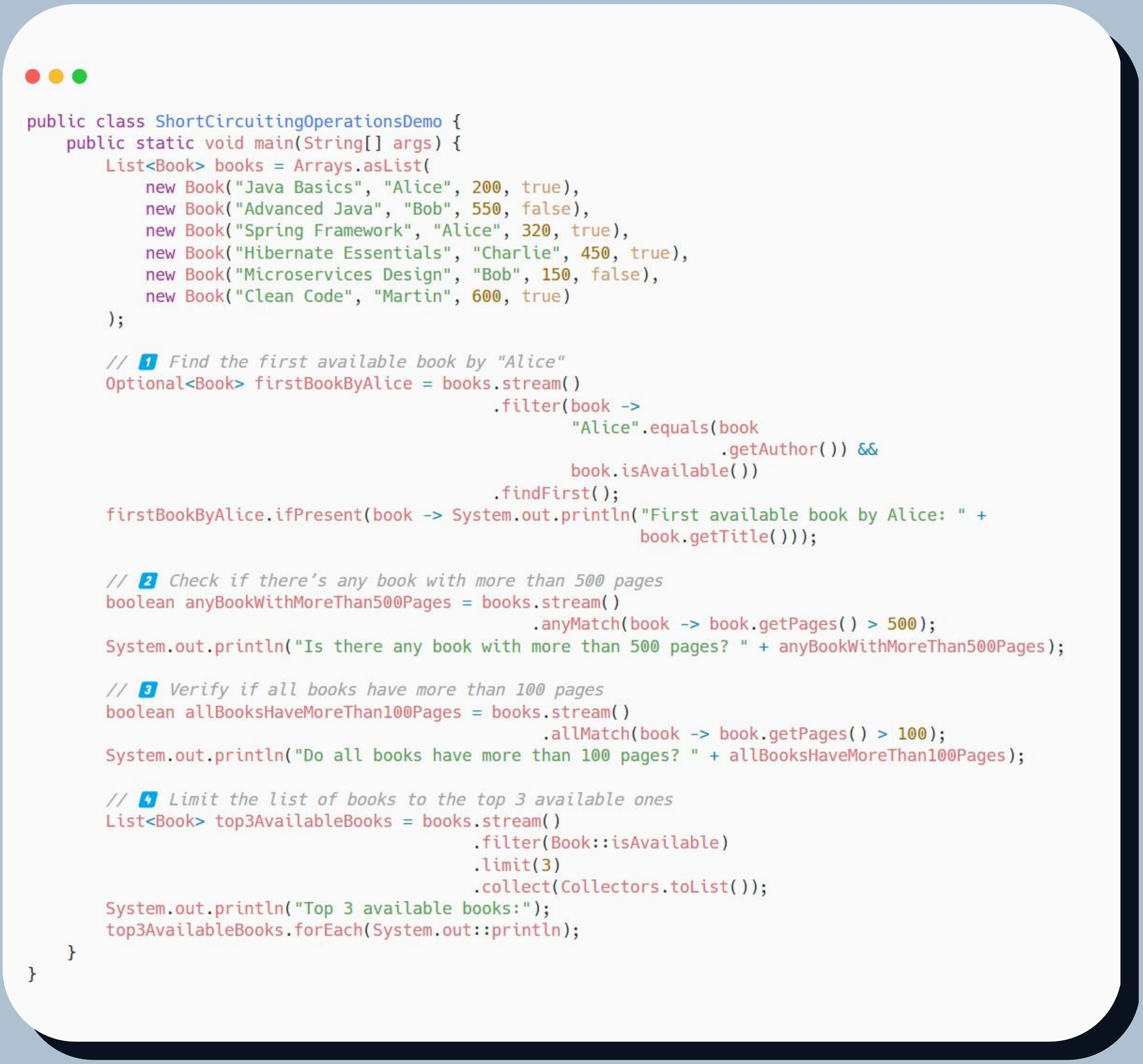
```java
public class StreamTerminalOperations {
    public static void main(String[] args) {
        List<Student> students = Arrays.asList(
            new Student(1, "Nayankumar Dhome", 23, "Mumbai", 63.5),
            new Student(2, "Sakshi", 21, "Pune", 75.0),
            new Student(3, "Vaishnavi", 22, "Delhi", 90.0),
            new Student(4, "Anchal", 23, "Chennai", 85.5)
            new Student(5, "Rohit", 20, "Pune", 79.0),
            new Student(6, "Nayan", 20, "Pune", 80.0),
        );

        // 1 Count the total number of students
        long studentCount = students.stream().count();
        System.out.println("Total number of students: " + studentCount);

        // 2 Find the average age of students
        OptionalDouble averageAge = students.stream()
                                    .mapToInt(Student::getAge)
                                    .average();
        averageAge.ifPresent(avg -> System.out.println("Average age of students: " + avg));

        // 3 Check if all students scored above 50
        boolean allPassed = students.stream()
                             .allMatch(student -> student.getGrade() > 50);
        System.out.println("Did all students pass? " + allPassed);

        // 4 Check if any student is from "Pune"
        boolean anyFromPune = students.stream()
                               .anyMatch(student -> "Pune".equals(student.getCity()));
        System.out.println("Is there any student from Pune? " + anyFromPune);

        // 5 Find the student with the highest grade
        Optional<Student> topStudent = students.stream()
                                       .max(Comparator
                                            .comparingDouble(Student::getGrade));
        topStudent.ifPresent(student -> System.out.println("Top student: " + student));

        // 6 Print details of all students
        System.out.println("All students:");
        students.stream().forEach(System.out::println);

        // 7 Calculate the total grades of all students combined
        double totalGrades = students.stream()
                             .mapToDouble(Student::getGrade)
                             .sum();
        System.out.println("Total grades of all students: " + totalGrades);
    }
}
```

Short-Circuiting Operations

Terminate the stream early for efficiency.

- limit(): Limits the number of elements.

- findFirst(): Returns the first element.



```java
public class ShortCircuitingOperationsDemo {
    public static void main(String[] args) {
        List<Book> books = Arrays.asList(
            new Book("Java Basics", "Alice", 200, true),
            new Book("Advanced Java", "Bob", 550, false),
            new Book("Spring Framework", "Alice", 320, true),
            new Book("Hibernate Essentials", "Charlie", 450, true),
            new Book("Microservices Design", "Bob", 150, false),
            new Book("Clean Code", "Martin", 600, true)
        );

        // 1️⃣ Find the first available book by "Alice"
        Optional<Book> firstBookByAlice = books.stream()
                                               .filter(book ->
                                                       "Alice".equals(book
                                                                      .getAuthor()) &&
                                                       book.isAvailable())
                                               .findFirst();
        firstBookByAlice.ifPresent(book -> System.out.println("First available book by Alice: " +
                                                        book.getTitle()));

        // 2️⃣ Check if there’s any book with more than 500 pages
        boolean anyBookWithMoreThan500Pages = books.stream()
                                                   .anyMatch(book -> book.getPages() > 500);
        System.out.println("Is there any book with more than 500 pages? " + anyBookWithMoreThan500Pages);

        // 3️⃣ Verify if all books have more than 100 pages
        boolean allBooksHaveMoreThan100Pages = books.stream()
                                                    .allMatch(book -> book.getPages() > 100);
        System.out.println("Do all books have more than 100 pages? " + allBooksHaveMoreThan100Pages);

        // 4️⃣ Limit the list of books to the top 3 available ones
        List<Book> top3AvailableBooks = books.stream()
                                             .filter(Book::isAvailable)
                                             .limit(3)
                                             .collect(Collectors.toList());
        System.out.println("Top 3 available books:");
        top3AvailableBooks.forEach(System.out::println);
    }
}
```

What Are Parallel Streams?

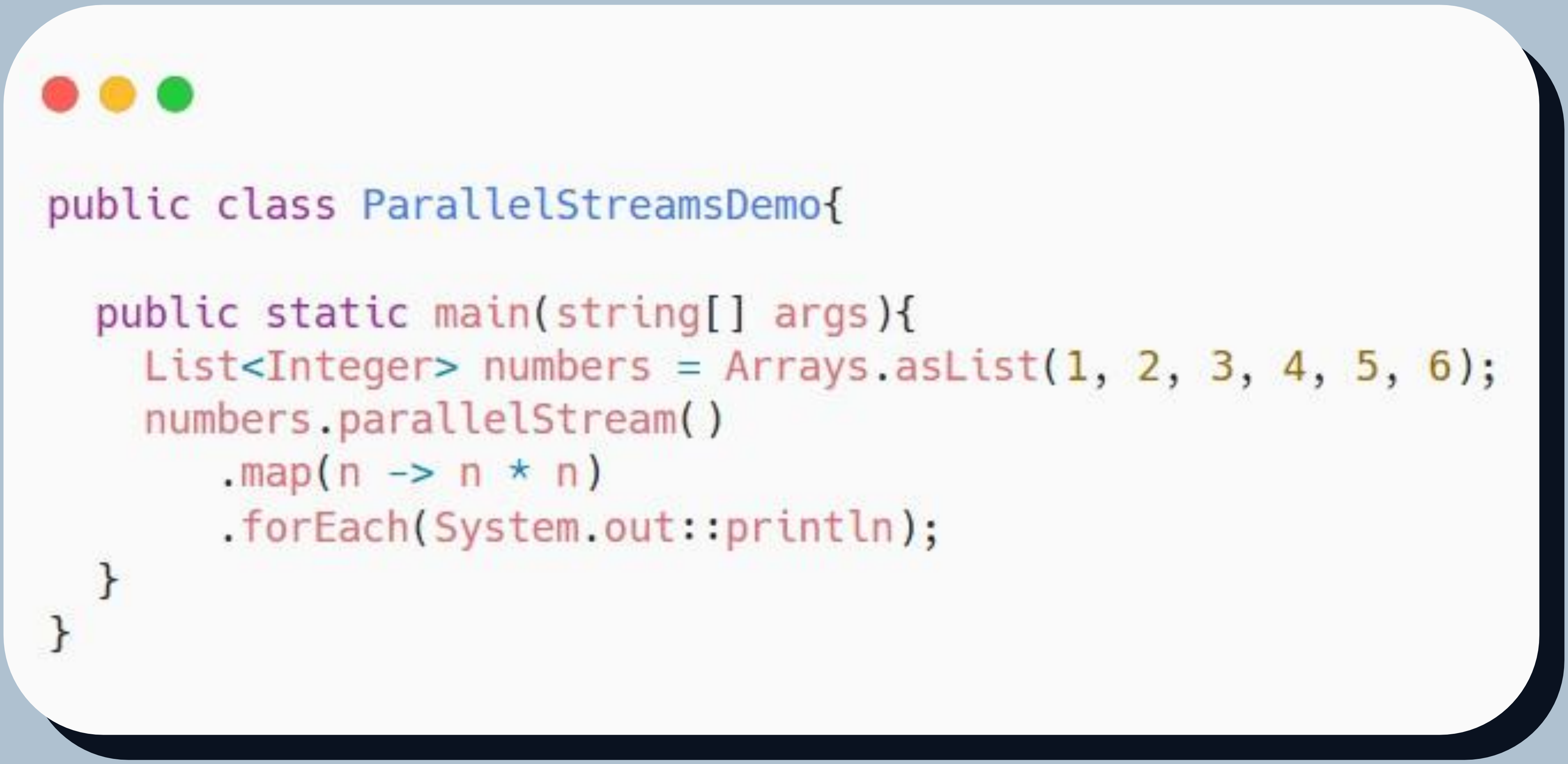
Parallel Streams split a data source into multiple chunks and process them concurrently using the Fork/Join framework. This can significantly improve performance for large data sets.

When Not to Use Parallel Streams?

- Small data sets where overhead outweighs performance gains.

- Non-thread-safe operations, as parallel streams may lead to race conditions.

- Operations with dependencies, where sequential execution is required.



```
public class ParallelStreamsDemo{

  public static main(string[] args){
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
    numbers.parallelStream()
        .map(n -> n * n)
        .forEach(System.out::println);
  }
}
```

Advantages of Parallel Streams

- Faster execution for large data sets.
- Utilizes multiple CPU cores.
- Simplifies parallel programming.

Advantages of the Stream API

- Simplifies data processing with less boilerplate.

- Supports parallel processing for better performance.

- Encourages functional programming principles.

Key Features of the Stream API

- **Functional Style**: Supports Functional programming with lambda expressions.

- **Lazy Evaluation**: Operations Are executed only when a terminal operation is invoked.

- **Parallel Streams**: Enables Efficient processing using multiple threads.

- **Immutable Operations**: Does not Modify the original data source.