Sales Data Analyzer (Java Streams Project)

This project demonstrates how to process and analyze sales data using Java functional programming techniques, especially the Streams API. It reads a CSV file with over 100 sales records and performs various analytical queries such as aggregations, filtering, sorting, and grouping using stream pipelines and lambda expressions.

Project Structure

```
sales-analyzer/
  pom.xml
  src/
      main/
          java/
              org/example/
                  Main.java              // Entry point: loads CSV and prints analysis
                  model/
                      SalesRecord.java   // Data model for each sales record
                  service/
                      SalesAnalyzer.java // Stream-based analysis logic
                  utils/
                       CSVReader.java     // CSV parsing into Java objects
          resources/
              sales_data_large.csv       // Sample dataset (100 records)

      test/
          java/
              org/example/service/
                  SalesAnalyzerTest.java // Unit tests using JUnit 5
```

Key Features and Functionalities

The application answers common analytical questions like:

- What is the total sales revenue?
- How much revenue came from each region?
- What is the average sale amount for each product category?
- What are the top N highest sales?
- Which sales occurred after a specific date?
- What is the lowest and highest sale recorded?
- How many sales came from each region?

- What are the total sales by month?
- Which category was the most frequently ordered?
- Which orders occurred between two dates?

All of these are implemented using the Streams API and functional constructs without relying on external libraries beyond JDK and JUnit.

Why Use Java Streams?

Streams provide a clean and expressive way to work with collections and perform transformations and aggregations. Instead of writing imperative loops, we use:

- `stream()` pipelines
- `map`, `filter`, `sorted`, `collect`, etc.
- `Collectors.groupingBy`, `summingDouble`, `averagingDouble`, `counting`
- `Comparator.comparing`, `reversed()`
- Lambda expressions for inline logic

This results in concise, composable, and testable code.

Explanation of Each Class

This project is modular and follows good separation of concerns. Each class has a distinct responsibility.

`Main.java`

Purpose: Acts as the entry point for the application.

- Reads the CSV file using `CSVReader`.
- Calls various analysis methods from `SalesAnalyzer`.
- Prints out results in a readable format.
- Includes sections like total sales, sales by region, top sales, filtering by date, etc.

This class is deliberately kept simple it doesn't perform calculations itself, just coordinates calls to utility and service methods.

`SalesRecord.java` (in `model` package)

Purpose: Represents a single sales transaction as a Java object.

- Fields: `orderId`, `customerName`, `region`, `category`, `amount`, `orderDate`
- Constructor initializes all fields.
- Getters are provided for each field.

- `toString()` is overridden for pretty printing of the object.

This is a basic POJO (Plain Old Java Object) used to map each row of the CSV.

`CSVReader.java` (in `utils` package)

Purpose: Loads the CSV file and parses it into a list of `SalesRecord` objects.

- Uses `BufferedReader` to read each line.
- Skips the header row.
- Splits each row by comma.
- Converts string values to appropriate types (int, double, LocalDate).
- Returns a list of `SalesRecord` objects.

This utility isolates the file-reading logic so the rest of the application can work with clean Java objects.

`SalesAnalyzer.java` (in `service` package)

Purpose: Core logic class that performs all analytical operations using Java Streams.

Contains multiple static methods, including:

- `getTotalSales(List<SalesRecord>)`: sums up all sales.
- `getSalesByRegion(...)`: groups sales by region and totals them.
- `getAverageByCategory(...)`: groups by category and calculates average amount.
- `getTopNSales(...)`: sorts records by amount and returns top N.
- `getSalesAfterDate(...)`: filters records with order date after a given date.
- `getMinSale(...)`, `getMaxSale(...)`: finds lowest and highest order.
- `getCountByRegion(...)`: counts how many orders per region.
- `getMonthlySales(...)`: aggregates sales totals month-wise.
- `getMostFrequentCategory(...)`: finds the category with the most sales.
- `getOrdersBetween(...)`: returns sales between two dates.

This class is where functional programming is showcased through stream pipelines and lambda expressions. It holds all the business logic.

`SalesAnalyzerTest.java` (in `test` package)

Purpose: Unit tests that validate each method in `SalesAnalyzer`.

- Uses JUnit 5.
- Creates sample `SalesRecord` data.
- Tests all aggregation and filtering logic with assertions.
- Ensures code correctness and acts as documentation.

These tests are important to ensure functional methods work correctly across different conditions and edge cases.

`sales_data_large.csv` (in `resources`)

Purpose: The sample dataset used for analysis.

- CSV format: `OrderID,CustomerName,Region,Category,Amount,OrderDate`
- Contains over 100 records for realistic testing.
- Can be replaced with a different file as long as the format is the same.

Summary

Class/File	Responsibility
`Main.java`	Coordinates the flow and prints output
`SalesRecord.java`	Represents a single row from the CSV
`CSVReader.java`	Converts CSV into a list of `SalesRecord` objects
`SalesAnalyzer.java`	Contains all analysis logic using Streams
`SalesAnalyzerTest`	Unit tests to validate the functionality
`sales_data_large.csv`	The dataset to be analyzed

How to Run

1. Prerequisites

Make sure you have:

- Java 17 or higher installed
- Apache Maven installed

2. Build the project

Open a terminal in the root directory and run:

```
mvn clean compile
```

alt text

Figure 1: alt text

alt text

Figure 2: alt text

3. Run the application

```
mvn exec:java -Dexec.mainClass="org.example.Main"
```

This will load the CSV file from `src/main/resources/` and print the analysis results to the console.

How to Run Tests

Tests are written using JUnit 5 and cover all analytical methods.

To execute tests:

```
mvn test
```

You'll see a summary of test results in the console. The test file `SalesAnalyzerTest.java` covers:

- Total revenue calculation
- Grouping and aggregation by region and category
- Top N sales
- Filtering by date ranges
- Minimum/maximum sales
- Monthly summaries
- Most frequent categories

Output

Running Main

Running tests

Sample CSV Format

The file `sales_data_large.csv` should have this format:

```
OrderID,CustomerName,Region,Category,Amount,OrderDate
1001,John Doe,North,Electronics,249.99,2023-01-10
...
```

Each row represents one sale record.

Assumptions

- Dates are in the format `yyyy-MM-dd`
- All fields are present and well-formed (no nulls or malformed entries)
- The amount field represents a positive number
- Basic file reading is done using core Java libraries only

Concepts Demonstrated

- Java 8+ Streams and functional programming
- Lambda expressions and comparator chaining
- CSV parsing with `BufferedReader`
- JUnit 5 unit testing
- Modular project structure (Maven)
- Good coding practices: separation of concerns, immutability, readability