SENJUTI GHOSAL
RA2111030010096
COGNIZANT WEEK 2

Exercise 1: Implementing Functional Interfaces with Lambda Expressions

Objectives:

- Understand and implement functional interfaces.
- Use lambda expressions to simplify code.

Business Scenario:

You are developing an application that performs various operations on a list of customer orders. You need to implement functional interfaces for filtering and processing the orders based on different criteria.

Tasks:

1. **Create a New Java Project:**
    - Create a new Java project named **OrderProcessing**.
2. **Define Functional Interfaces:**
    - Define a functional interface **OrderFilter** with a method boolean **filter(Order order)**.
    - Define another functional interface **OrderProcessor** with a method void **process(Order order)**.
3. **Create the Order Class:**
    - Define an **Order** class with attributes like **orderId**, **customerName**, **orderAmount**, and **status**.
4. **Implement Lambda Expressions:**
    - In the **OrderProcessing** class, create a list of **Order** objects.
    - Use lambda expressions to implement **OrderFilter** for filtering orders with an amount greater than a specified value.
    - Use lambda expressions to implement **OrderProcessor** for processing orders by changing their status.
5. **Filter and Process Orders:**
    - Write a method that takes an **OrderFilter** and processes all orders that match the filter.
    - Write a method that takes an **OrderProcessor** and applies it to all orders.
6. **Test the Application:**
    - Create sample orders and test the filtering and processing methods.
    - Print the results to verify that the orders are correctly filtered and processed.

1)

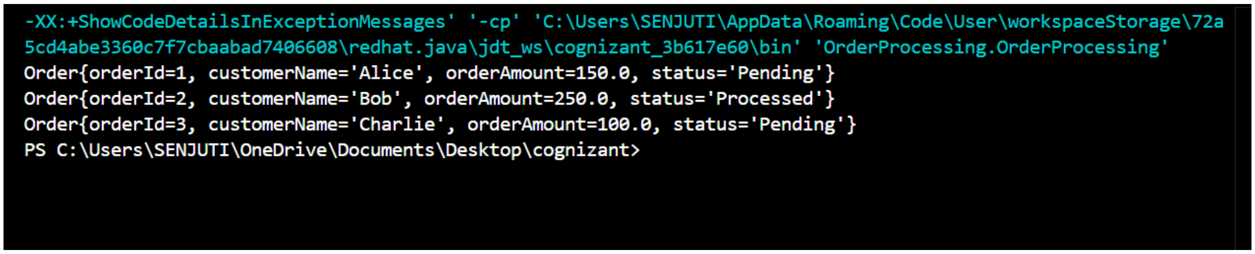
```
@FunctionalInterface
interface OrderFilter {
    boolean filter(Order order);
}

@FunctionalInterface
interface OrderProcessor {
    void process(Order order);
}
```

2)

```
public class Order {
    private int orderId;
    private String customerName;
    private double orderAmount;
    private String status;

    // Constructors, getters, and setters

    public Order(int orderId, String customerName, double orderAmount, String status) {
        this.orderId = orderId;
        this.customerName = customerName;
        this.orderAmount = orderAmount;
        this.status = status;
    }

    public int getOrderId() {
        return orderId;
    }

    public String getCustomerName() {
        return customerName;
    }

    public double getOrderAmount() {
        return orderAmount;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }

    @Override
    public String toString() {
        return "Order{" +
                "orderId=" + orderId +
                ", customerName='" + customerName + '\'' +
                ", orderAmount=" + orderAmount +
                ", status='" + status + '\'' +
                '}';
    }
}
```

3)

```
import java.util.ArrayList;
import java.util.List;

public class OrderProcessing {

    public static void main(String[] args) {
        List<Order> orders = new ArrayList<>();
        orders.add(new Order(1, "Alice", 150.0, "Pending"));
        orders.add(new Order(2, "Bob", 250.0, "Pending"));
        orders.add(new Order(3, "Charlie", 100.0, "Pending"));

        OrderFilter filter = order -> order.getOrderAmount() > 200.0;
        OrderProcessor processor = order -> order.setStatus("Processed");

        processOrders(orders, filter, processor);

        for (Order order : orders) {
            System.out.println(order);
        }
    }

    public static void processOrders(List<Order> orders, OrderFilter filter, OrderProcessor
processor) {
        for (Order order : orders) {
            if (filter.filter(order)) {
                processor.process(order);
            }
        }
    }
}
```



```
-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\SENJUTI\AppData\Roaming\Code\User\workspaceStorage\72a
5cd4abe3360c7f7cbaabad7406608\redhat.java\jdt_ws\cognizant_3b617e60\bin' 'OrderProcessing.OrderProcessing'
Order{orderId=1, customerName='Alice', orderAmount=150.0, status='Pending'}
Order{orderId=2, customerName='Bob', orderAmount=250.0, status='Processed'}
Order{orderId=3, customerName='Charlie', orderAmount=100.0, status='Pending'}
PS C:\Users\SENJUTI\OneDrive\Documents\Desktop\cognizant>
```

Exercise 2: Using Stream API for Processing Collections

Objectives:

- Use Stream API to process collections.
- Perform various operations such as filtering, mapping, and reducing on streams.

Business Scenario:

You are developing a sales analysis application that processes a list of sales records. You need to use the Stream API to analyze the sales data and generate reports.

Tasks:

1. **Create a New Java Project:**
    - Create a new Java project named **SalesAnalysis**.
2. **Define the SalesRecord Class:**
    - Define a SalesRecord class with attributes like **recordId**, **salesPerson**, **region**, **amount**, and **date**.

```
public class SalesRecord {

   private int recordId;

   private String salesPerson;

   private String region;

   private double amount;

   private String date;

   public SalesRecord(int recordId, String salesPerson, String region, double amount, String date) {

      this.recordId = recordId;

      this.salesPerson = salesPerson;
```

```
    this.region = region;
    this.amount = amount;
    this.date = date;
}

public int getRecordId() {
    return recordId;
}

public String getSalesPerson() {
    return salesPerson;
}

public String getRegion() {
    return region;
}

public double getAmount() {
    return amount;
}

public String getDate() {
    return date;
}


public String toString() {
    return "SalesRecord{" +
            "recordId=" + recordId +
                ", salesPerson='" + salesPerson + '\'' +
                ", region='" + region + '\'' +
                ", amount=" + amount +
                ", date='" + date + '\'' +
                '}';
    }
}
```

3. **Create Sample Data:**

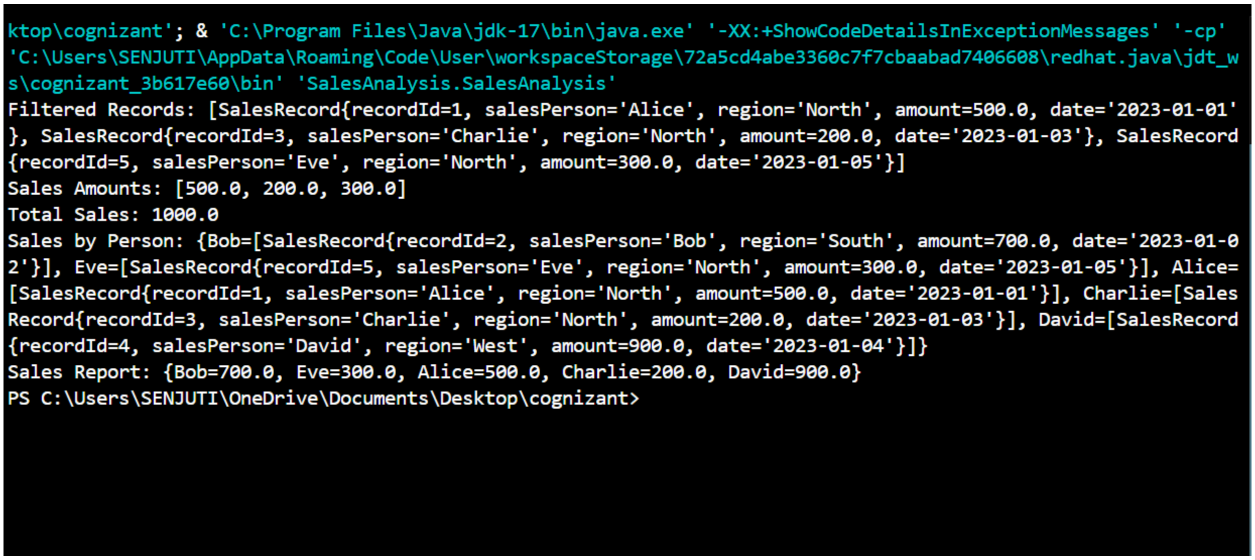
- In the **SalesAnalysis** class, create a list of **SalesRecord** objects with sample data.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class SalesAnalysis {
    public static void main(String[] args) {
        List<SalesRecord> records = new ArrayList<>();
        records.add(new SalesRecord(1, "Alice", "North", 500.0, "2023-01-01"));
        records.add(new SalesRecord(2, "Bob", "South", 700.0, "2023-01-02"));
        records.add(new SalesRecord(3, "Charlie", "North", 200.0, "2023-01-03"));
        records.add(new SalesRecord(4, "David", "West", 900.0, "2023-01-04"));
        records.add(new SalesRecord(5, "Eve", "North", 300.0, "2023-01-05"));

        // Step 4: Filter Sales Records
        List<SalesRecord> northRecords = records.stream()
                .filter(record -> "North".equals(record.getRegion()))
                .collect(Collectors.toList());
        System.out.println("Filtered Records: " + northRecords);
        // Step 5: Map and Transform Data
        List<Double> salesAmounts = northRecords.stream()
                .map(SalesRecord::getAmount)
                .collect(Collectors.toList());
        System.out.println("Sales Amounts: " + salesAmounts);

        // Step 6: Calculate Total Sales
        double totalSales = northRecords.stream()
                .mapToDouble(SalesRecord::getAmount)
                .sum();
        System.out.println("Total Sales: " + totalSales);

        // Step 7: Group Sales by SalesPerson
        Map<String, List<SalesRecord>> salesByPerson = records.stream()
                .collect(Collectors.groupingBy(SalesRecord::getSalesPerson));
        System.out.println("Sales by Person: " + salesByPerson);

        // Step 8: Generate Sales Report
        Map<String, Double> salesReport = records.stream()
                .collect(Collectors.groupingBy(SalesRecord::getSalesPerson,
Collectors.summingDouble(SalesRecord::getAmount)));
        System.out.println("Sales Report: " + salesReport);
    }
}
```

4. **Filter Sales Records:**
    - Use the **Stream API** to filter sales records for a specific region.
    - Print the filtered records.
5. **Map and Transform Data:**
    - Use the **Stream API** to extract the sales amounts from the filtered records.

- Print the sales amounts.

6. **Calculate Total Sales:**
    - Use the **Stream API** to calculate the total sales amount for the filtered records.
    - Print the total sales amount.
7. **Group Sales by SalesPerson:**
    - Use the **Stream API** to group sales records by **salesPerson**.
    - Print the grouped sales records.
8. **Generate Sales Report:**
    - Use the **Stream API** to generate a sales report that includes the total sales amount for each salesperson.
    - Print the sales report.



```
ktop\cognizant'; & 'C:\Program Files\Java\jdk-17\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp'
'C:\Users\SENJUTI\AppData\Roaming\Code\User\workspaceStorage\72a5cd4abe3360c7f7cbaabad7406608\redhat.java\jdt_w
s\cognizant_3b617e60\bin' 'SalesAnalysis.SalesAnalysis'
Filtered Records: [SalesRecord{recordId=1, salesPerson='Alice', region='North', amount=500.0, date='2023-01-01'
}, SalesRecord{recordId=3, salesPerson='Charlie', region='North', amount=200.0, date='2023-01-03'}, SalesRecord
{recordId=5, salesPerson='Eve', region='North', amount=300.0, date='2023-01-05'}]
Sales Amounts: [500.0, 200.0, 300.0]
Total Sales: 1000.0
Sales by Person: {Bob=[SalesRecord{recordId=2, salesPerson='Bob', region='South', amount=700.0, date='2023-01-0
2'}], Eve=[SalesRecord{recordId=5, salesPerson='Eve', region='North', amount=300.0, date='2023-01-05'}], Alice=
[SalesRecord{recordId=1, salesPerson='Alice', region='North', amount=500.0, date='2023-01-01'}], Charlie=[Sales
Record{recordId=3, salesPerson='Charlie', region='North', amount=200.0, date='2023-01-03'}], David=[SalesRecord
{recordId=4, salesPerson='David', region='West', amount=900.0, date='2023-01-04'}]}
Sales Report: {Bob=700.0, Eve=300.0, Alice=500.0, Charlie=200.0, David=900.0}
PS C:\Users\SENJUTI\OneDrive\Documents\Desktop\cognizant>
```

Exercise 3: Advanced Stream Operations and Parallel Streams

Objectives:

- Perform advanced operations using Stream API.
- Utilize parallel streams for improved performance.

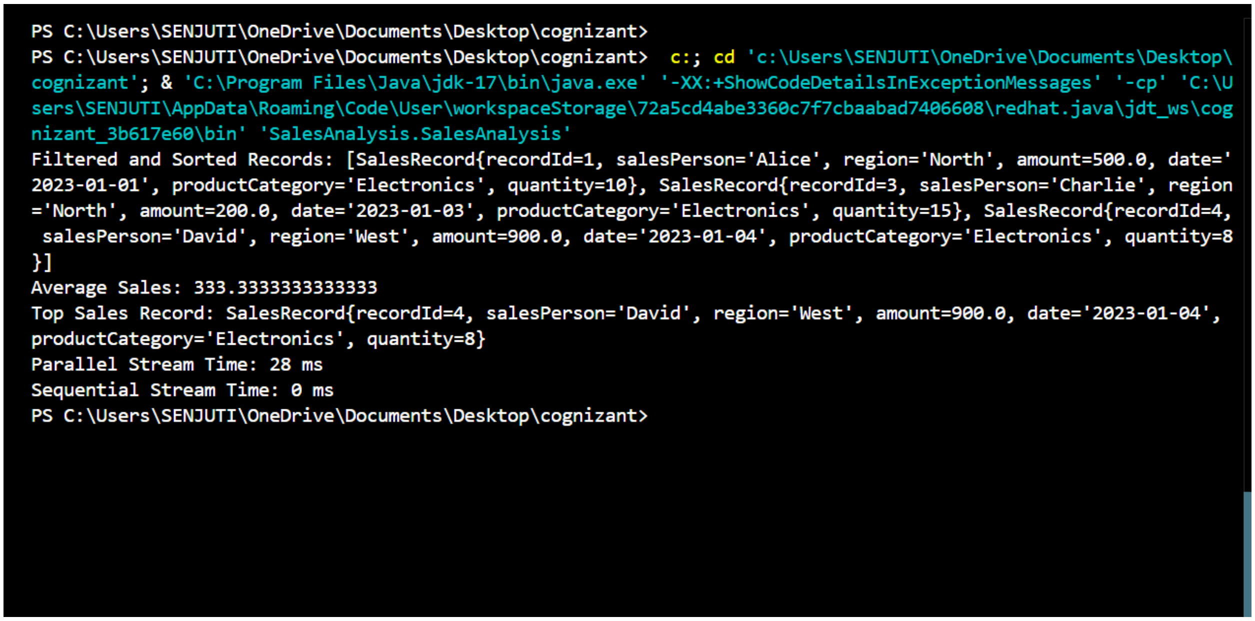
Business Scenario:

You are enhancing the sales analysis application to include more complex analysis and improve performance using parallel streams.

Tasks:

1. **Update SalesRecord Class:**
    - Add additional attributes such as **productCategory** and quantity to the **SalesRecord** class.

2. **Filter and Sort Records:**
    - Use the **Stream API** to filter sales records for a specific product category and sort them by date.
    - Print the sorted records.
3. **Calculate Average Sales:**
    - Use the **Stream API** to calculate the average sales amount for a specific region.
    - Print the average sales amount.
4. **Find Top Sales Record:**
    - Use the **Stream API** to find the sales record with the highest amount.
    - Print the top sales record.
5. **Parallel Stream Operations:**
    - Use parallel streams to perform the filtering and sorting operations for improved performance.
    - Measure and print the time taken for both sequential and parallel stream operations.



```
PS C:\Users\SENJUTI\OneDrive\Documents\Desktop\cognizant>
PS C:\Users\SENJUTI\OneDrive\Documents\Desktop\cognizant>  c:; cd 'c:\Users\SENJUTI\OneDrive\Documents\Desktop\
cognizant'; & 'C:\Program Files\Java\jdk-17\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\U
sers\SENJUTI\AppData\Roaming\Code\User\workspaceStorage\72a5cd4abe3360c7f7cbaabad7406608\redhat.java\jdt_ws\cog
nizant_3b617e60\bin' 'SalesAnalysis.SalesAnalysis'
Filtered and Sorted Records: [SalesRecord{recordId=1, salesPerson='Alice', region='North', amount=500.0, date='
2023-01-01', productCategory='Electronics', quantity=10}, SalesRecord{recordId=3, salesPerson='Charlie', region
='North', amount=200.0, date='2023-01-03', productCategory='Electronics', quantity=15}, SalesRecord{recordId=4,
 salesPerson='David', region='West', amount=900.0, date='2023-01-04', productCategory='Electronics', quantity=8
}]
Average Sales: 333.3333333333333
Top Sales Record: SalesRecord{recordId=4, salesPerson='David', region='West', amount=900.0, date='2023-01-04',
productCategory='Electronics', quantity=8}
Parallel Stream Time: 28 ms
Sequential Stream Time: 0 ms
PS C:\Users\SENJUTI\OneDrive\Documents\Desktop\cognizant>
```

SALESANALYSIS.JAVA

```
package SalesAnalysis;

import java.util.ArrayList;

import java.util.Comparator;

import java.util.List;

import java.util.Map;

import java.util.stream.Collectors;
```

```
public class SalesAnalysis {
  public static void main(String[] args) {
    List<SalesRecord> records = new ArrayList<>();
    records.add(new SalesRecord(1, "Alice", "North", 500.0, "2023-01-01", "Electronics", 10));
    records.add(new SalesRecord(2, "Bob", "South", 700.0, "2023-01-02", "Clothing", 20));
    records.add(new SalesRecord(3, "Charlie", "North", 200.0, "2023-01-03", "Electronics", 15));
    records.add(new SalesRecord(4, "David", "West", 900.0, "2023-01-04", "Electronics", 8));
    records.add(new SalesRecord(5, "Eve", "North", 300.0, "2023-01-05", "Clothing", 5));

    // Step 2: Filter and Sort Records
    List<SalesRecord> electronicsRecords = records.stream()
        .filter(record -> "Electronics".equals(record.getProductCategory()))
        .sorted(Comparator.comparing(SalesRecord::getDate))
        .collect(Collectors.toList());
    System.out.println("Filtered and Sorted Records: " + electronicsRecords);

    // Step 3: Calculate Average Sales
    double averageSales = records.stream()
        .filter(record -> "North".equals(record.getRegion()))
        .mapToDouble(SalesRecord::getAmount)
        .average()
        .orElse(0.0);
    System.out.println("Average Sales: " + averageSales);

    // Step 4: Find Top Sales Record
    SalesRecord topSalesRecord = records.stream()
        .max(Comparator.comparingDouble(SalesRecord::getAmount))
        .orElse(null);
    System.out.println("Top Sales Record: " + topSalesRecord);
```

```
        // Step 5: Parallel Stream Operations
        long startTime = System.currentTimeMillis();
        List<SalesRecord> parallelFilteredRecords = records.parallelStream()
                .filter(record -> "Electronics".equals(record.getProductCategory()))
                .sorted(Comparator.comparing(SalesRecord::getDate))
                .collect(Collectors.toList());
        long endTime = System.currentTimeMillis();
        System.out.println("Parallel Stream Time: " + (endTime - startTime) + " ms");

        startTime = System.currentTimeMillis();
        List<SalesRecord> sequentialFilteredRecords = records.stream()
                .filter(record -> "Electronics".equals(record.getProductCategory()))
                .sorted(Comparator.comparing(SalesRecord::getDate))
                .collect(Collectors.toList());
        endTime = System.currentTimeMillis();
        System.out.println("Sequential Stream Time: " + (endTime - startTime) + " ms");
    }
}
```

SALESRECORD.JAVA:

```
package SalesAnalysis;
public class SalesRecord {
    private int recordId;
    private String salesPerson;
    private String region;
    private double amount;
    private String date;
    private String productCategory;
    private int quantity;

    public SalesRecord(int recordId, String salesPerson, String region, double amount, String date, String productCategory, int quantity) {
        this.recordId = recordId;
        this.salesPerson = salesPerson;
        this.region = region;
        this.amount = amount;
        this.date = date;
        this.productCategory = productCategory;
        this.quantity = quantity;
    }

    public int getRecordId() {
        return recordId;
    }

    public String getSalesPerson() {
        return salesPerson;
    }

    public String getRegion() {
        return region;
    }

    public double getAmount() {
        return amount;
    }
    public String getDate() {
        return date;
    }

    public String getProductCategory() {
        return productCategory;
    }

    public int getQuantity() {
        return quantity;
    }

    @Override
    public String toString() {
        return "SalesRecord{" +
                "recordId=" + recordId +
                ", salesPerson='" + salesPerson + '\'' +
                ", region='" + region + '\'' +
                ", amount=" + amount +
                ", date='" + date + '\'' +
                ", productCategory='" + productCategory + '\'' +
                ", quantity=" + quantity +
                '}';
    }
}
```

Exercise 4: Combining Functional Interfaces and Streams

Objectives:

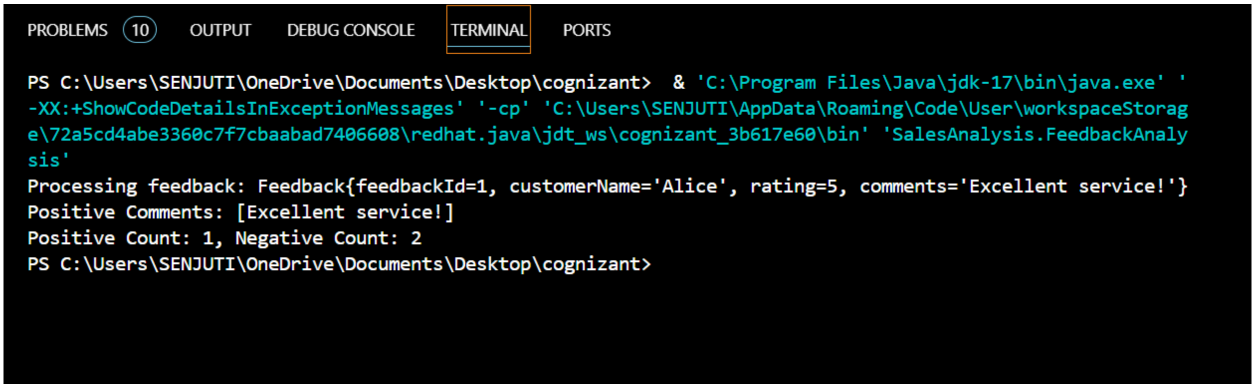
- Combine **functional interfaces** and **Stream API** for flexible and reusable code.
- Implement complex data processing pipelines.

Business Scenario:

You are tasked with developing a comprehensive data processing pipeline for customer feedback analysis. The pipeline should be flexible and reusable for different types of analysis.

Tasks:

1. **Define Functional Interfaces:**
    - Define functional interfaces **FeedbackFilter** and **FeedbackProcessor**.
2. **Create Feedback Class:**
    - Define a **Feedback** class with attributes like **feedbackId**, **customerName**, rating, and comments.
3. **Implement Data Processing Pipeline:**
    - Use the **Stream API** to create a flexible data processing pipeline that:
        - Filters feedback based on a minimum rating.
        - Maps feedback to extract customer names and comments.
        - Reduces feedback to count the number of positive and negative feedbacks.
4. **Implement Flexible Processing:**
    - Write methods that take **FeedbackFilter** and **FeedbackProcessor** as parameters to allow flexible and reusable processing.
    - Create lambda expressions to implement different filtering and processing strategies.
5. **Test the Pipeline:**
    - Create sample feedback data and test the data processing pipeline.
    - Print the results to verify the correct operation of the pipeline.



```
PS C:\Users\SENJUTI\OneDrive\Documents\Desktop\cognizant>  & 'C:\Program Files\Java\jdk-17\bin\java.exe' '
-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\SENJUTI\AppData\Roaming\Code\User\workspaceStorag
e\72a5cd4abe3360c7f7cbaabad7406608\redhat.java\jdt_ws\cognizant_3b617e60\bin' 'SalesAnalysis.FeedbackAnaly
sis'
Processing feedback: Feedback{feedbackId=1, customerName='Alice', rating=5, comments='Excellent service!'}
Positive Comments: [Excellent service!]
Positive Count: 1, Negative Count: 2
PS C:\Users\SENJUTI\OneDrive\Documents\Desktop\cognizant>
```

FEEDBACK.JAVA:

```
package SalesAnalysis;
public class Feedback {
    private int feedbackId;
    private String customerName;
    private int rating;
    private String comments;

    public Feedback(int feedbackId, String customerName, int rating, String comments) {
        this.feedbackId = feedbackId;
        this.customerName = customerName;
        this.rating = rating;
        this.comments = comments;
    }

    public int getFeedbackId() {
        return feedbackId;
    }

    public String getCustomerName() {
        return customerName;
    }

    public int getRating() {
        return rating;
    }

    public String getComments() {
        return comments;
    }

    @Override
    public String toString() {
        return "Feedback{" +
                "feedbackId=" + feedbackId +
                ", customerName='" + customerName + '\'' +
                ", rating=" + rating +
                ", comments='" + comments + '\'' +
                '}';
    }
}
```

FeedbackAnalysis.java:

```
package SalesAnalysis;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;


public class FeedbackAnalysis {

    public static void main(String[] args) {

        List<Feedback> feedbacks = new ArrayList<>();

        feedbacks.add(new Feedback(1, "Alice", 5, "Excellent service!"));

        feedbacks.add(new Feedback(2, "Bob", 3, "Average experience."));

        feedbacks.add(new Feedback(3, "Charlie", 1, "Very poor service."));


        FeedbackFilter filter = feedback -> feedback.getRating() >= 4;

        FeedbackProcessor processor = feedback -> System.out.println("Processing feedback: " + feedback);
        processFeedback(feedbacks, filter, processor);

        List<String> positiveComments = feedbacks.stream()
            .filter(feedback -> feedback.getRating() >= 4)
            .map(Feedback::getComments)
            .collect(Collectors.toList());
        System.out.println("Positive Comments: " + positiveComments);

        long positiveCount = feedbacks.stream()
            .filter(feedback -> feedback.getRating() >= 4)
            .count();
        long negativeCount = feedbacks.stream()
            .filter(feedback -> feedback.getRating() < 4)
            .count();
        System.out.println("Positive Count: " + positiveCount + ", Negative Count: " + negativeCount);
    }

    public static void processFeedback(List<Feedback> feedbacks, FeedbackFilter filter, FeedbackProcessor processor) {
        for (Feedback feedback : feedbacks) {
          if (filter.filter(feedback)) {
            processor.process(feedback);
          }
        }
    }
}
```

FeedbackFilter.java:

```
package SalesAnalysis;
```

```
public interface FeedbackFilter {
    boolean filter(Feedback feedback);
}
```

FeedbackProcessor.java:

```
package SalesAnalysis;
public interface FeedbackProcessor {
    void process(Feedback feedback);
}
```