Academic Year 2022-23

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJS22ITL306 **DATE:05/12/23**

COURSE NAME: Programing Laboratory 1 (Advanced Java)

CLASS: S.Y B. Tech IT

NAME: ANISH SHARMA

SAP.ID: 60003220045

DIV: I-1

EXPERIMENT NO. 7

CO/LO:
CO1- Modify the behaviour of methods, classes, and interfaces at runtime.

AIM / OBJECTIVE:

Use streams API to implement program logic by composing functions and executing them in a data flow.

PROBLEM STATEMENTS:

WAP to create a list of products with data fields: id, name, price. Create a stream on the product list and generate an optimized code to filter data (e.g. price>30k) based on the product prize. Display the products whose cost is 30k using forEach(). Find the total cost of all products using reduce(). Find the product with minimum and maximum cost. Fint the count of products with prize<30k. Convert the list to map(id,name).

PROGRAM:

```
import java.util.Arrays;

import java.util.List;

import java.util.Map;

import java.util.stream.Collectors;

import java.util.*;
```

```
class Product {
    private int id;
    private String name;
    private double price;

    public Product(int id, String name, double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }
}

public class StreamExample {
    public static void main(String[] args) {
        List<Product> productList = Arrays.asList(
                new Product(1, "Laptop", 35000),
                new Product(2, "Smartphone", 25000),
                new Product(3, "Headphones", 2000),
```

```
        new Product(4, "Camera", 45000),
        new Product(5, "Tablet", 30000)
    );


    List<Product> expensiveProducts = productList.stream()
        .filter(product -> product.getPrice() > 30000)
        .collect(Collectors.toList());


    System.out.println("Expensive Products (price > 30k):");
    expensiveProducts.forEach(product -> System.out.println(product.getName()));

    double totalCost = productList.stream()
        .mapToDouble(Product::getPrice)
        .sum();
    System.out.println("Total Cost of All Products: " + totalCost);


    Product minCostProduct = productList.stream()
        .min((p1, p2) -> Double.compare(p1.getPrice(), p2.getPrice()))
        .orElse(null);

    Product maxCostProduct = productList.stream()
        .max((p1, p2) -> Double.compare(p1.getPrice(), p2.getPrice()))
        .orElse(null);

    System.out.println("Product with Minimum Cost: " + minCostProduct.getName());
    System.out.println("Product with Maximum Cost: " + maxCostProduct.getName());
```

```
        long countProductsUnder30k = productList.stream()
                .filter(product -> product.getPrice() < 30000)
                .count();
        System.out.println("Count of Products with Price < 30k: " + countProductsUnder30k);

        // Convert the list to map(id, name)
        Map<Integer, String> idNameMap = productList.stream()
                .collect(Collectors.toMap(Product::getId, Product::getName));
        System.out.println("Map(id, name): " + idNameMap);
    }
}
```

OUTPUT:

```
C:\Users\user\Desktop>javac StreamExample.java

C:\Users\user\Desktop>java StreamExample
Expensive Products (price > 30k):
Laptop
Camera
Total Cost of All Products: 137000.0
Product with Minimum Cost: Headphones
Product with Maximum Cost: Camera
Count of Products with Price < 30k: 2
Map(id, name): {1=Laptop, 2=Smartphone, 3=Headphones, 4=Camera, 5=Tablet}
```

OBSERVATION:

List and explain the features of streams API.

1. **Functional Composition:**
    - Streams support functional composition, allowing you to chain multiple operations together in a concise and expressive manner.
2. **Lazy Evaluation:**
    - Operations on streams are lazily evaluated, meaning they are only executed when a terminal operation is invoked. This can lead to more efficient processing.

3. **Parallel Execution:**
    - Streams can be easily parallelized, enabling concurrent processing of elements. This is beneficial for performance improvement on multi-core systems.
4. **Intermediate and Terminal Operations:**
    - Streams consist of intermediate operations (e.g., filter, map) that transform the data, and terminal operations (e.g., forEach, reduce) that produce a result or a side-effect.
5. **Filtering and Mapping:**
    - Stream API provides convenient methods for filtering elements based on conditions and transforming elements using mapping functions.
6. **Reduction Operations:**
    - Streams support reduction operations like **reduce** and **collect** to perform computations on the elements and aggregate them into a single result.
7. **Parallel Streams:**
    - Parallel streams enable automatic parallelization of operations, distributing the workload across multiple threads for improved performance.
8. **Predicate and Consumer Support:**
    - Stream operations often take functional interfaces like **Predicate** for filtering and **Consumer** for performing actions on elements, promoting a functional programming style.
9. **Distinct and Sorted Operations:**
    - Stream API provides methods for removing duplicate elements (**distinct**) and sorting elements based on natural order or a custom comparator (**sorted**).
10. **Stream Sources:**
    - Streams can be created from various sources such as collections, arrays, I/O channels, and generators, offering flexibility in data input.

These features collectively make the Streams API a powerful tool for expressive, functional, and efficient data processing in Java.

CONCLUSION:

In this exp, using the Streams API in Java simplified the code for handling product data. It allowed easy filtering, cost calculations, and transformation, showcasing the effectiveness of the API in creating clear and concise data processing logic.