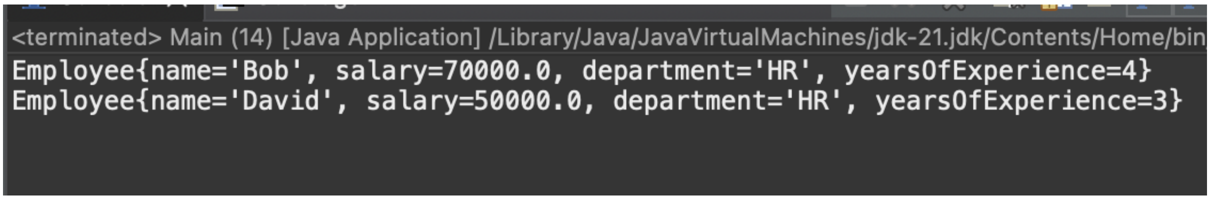
SKILL AOOP **P. ABHINAYA**
WEEK-7 **2300030499**

1) Custom Functional Interface with Complex Predicate Filtering. Filter employees based on multiple dynamic criteria such as salary range, department, and years of experience using a custom functional interface, lambda expressions, and the Stream API. 1 Define the Employee class: This class will represent each employee with fields like name, salary, department, and yearsOfExperience. 2 Create a Functional Interface: Define a functional interface that will allow for dynamic filtering based on employee attributes. 3 Implement Filtering Logic: Using the Stream API, we will dynamically filter the employee list based on the provided lambda expressions. Explanation: 1 Employee Class: Represents employees with fields like name, salary, department, and yearsOfExperience. 2 EmployeeFilter Interface: This custom functional interface is used to define the filtering logic via lambdas. 3 filterEmployees Method: This method accepts a list of employees and an EmployeeFilter functional interface. The Stream API is used to filter the employees based on the lambda expression provided as the filter. 4 Lambda Expressions: ⎫ highSalaryFilter: Filters employees with a salary greater than 80,000. ⎫ engineeringDeptFilter: Filters employees belonging to the "Engineering" department. ⎫ experiencedFilter: Filters employees with 7 or more years of experience. 5 Combining Filters: The filters are combined using logical AND (&&). This allows you to dynamically compose multiple filter conditions.
CODE:

```
public class Employee {
   private String name;
   private double salary;
   private String department;
   private int yearsOfExperience;

   public Employee(String name, double salary, String department, int
yearsOfExperience) {
      this.name = name;
      this.salary = salary;
      this.department = department;
      this.yearsOfExperience = yearsOfExperience;
   }

   public String getName() {
      return name;
   }

   public double getSalary() {
      return salary;
```

```
    }

    public String getDepartment() {
        return department;
    }

    public int getYearsOfExperience() {
        return yearsOfExperience;
    }

    @Override
    public String toString() {
        return "Employee{" +
                "name='" + name + '\'' +
                ", salary=" + salary +
                ", department='" + department + '\'' +
                ", yearsOfExperience=" + yearsOfExperience +
                '}';
    }
}
@FunctionalInterface
public interface EmployeeCriteria {
    boolean match(Employee employee);
}
import java.util.List;
import java.util.stream.Collectors;

public class EmployeeSelector {

    public static List<Employee> selectEmployees(List<Employee> employees,
EmployeeCriteria criteria) {
        return employees.stream()
                        .filter(criteria::match)
                        .collect(Collectors.toList());
    }
}
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Sample list of employees with varied salaries, departments, and experience
        List<Employee> employees = Arrays.asList(
      new Employee("Alice", 90000, "Engineering", 10),
      new Employee("Bob", 70000, "HR", 4),
      new Employee("Charlie", 60000, "Engineering", 2),
      new Employee("David", 50000, "HR", 3),
      new Employee("Eva", 45000, "Marketing", 1)
    );

    // Define new filters
    EmployeeCriteria lowSalaryFilter = employee -> employee.getSalary() < 80000;
    EmployeeCriteria hrDeptFilter = employee ->
employee.getDepartment().equals("HR");
    EmployeeCriteria lessExperienceFilter = employee ->
employee.getYearsOfExperience() < 5;

    // Combine filters using logical AND (matching all conditions)
    EmployeeCriteria combinedCriteria = employee ->
      lowSalaryFilter.match(employee) &&
      hrDeptFilter.match(employee) &&
      lessExperienceFilter.match(employee);

    // Select employees using the new filters
    List<Employee> selectedEmployees =
EmployeeSelector.selectEmployees(employees, combinedCriteria);

    // Output the selected employees
    selectedEmployees.forEach(System.out::println);
  }
}
```

OUTPUT:



```
<terminated> Main (14) [Java Application] /Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/Home/bin
Employee{name='Bob', salary=70000.0, department='HR', yearsOfExperience=4}
Employee{name='David', salary=50000.0, department='HR', yearsOfExperience=3}
```

2) Combining Stream Operations with Method References and Lambdas. You have a list of transactions, where each transaction has an amount and a category (like "groceries," "utilities," etc.). You need to generate a report that groups transactions by category and calculates the total amount spent in each category using the Stream API, combining both method references and lambda expressions. 1 Transaction Class: ⎤ This class represents each transaction with two fields: amount (of type double) and category (of type String). ⎤ There are two getter methods, getAmount() and getCategory(), used for accessing these fields. 2 Stream Operations: ⎤ The transactions.stream() method initiates a stream pipeline on the list of transactions. ⎤

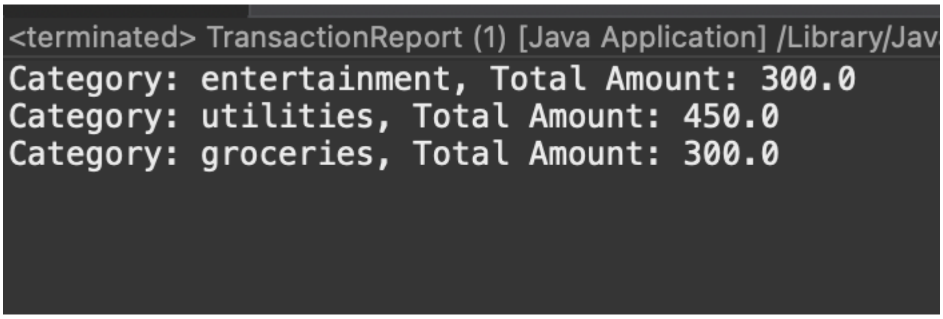
We use Collectors.groupingBy() to group the transactions by category. ⎤ The first argument to groupingBy() is a method reference Transaction::getCategory, which extracts the category of each transaction. ⎤ The second argument is a collector Collectors.summingDouble(), which sums the amounts of transactions. This is achieved using a lambda expression (t -> t.getAmount()) that extracts the amount of each transaction. 3 Result: ⎤ The transactions are grouped by their category, and the sum of the transaction amounts is computed for each category. ⎤ Finally, the forEach() method is used to print the total amount for each category. Key Concepts Used: ⎤ Method Reference: Transaction::getCategory to group by category. ⎤ Lambda Expression: (t -> t.getAmount()) to sum the transaction amounts. ⎤ Stream API: To process the transactions efficiently and perform grouping and aggregation.

CODE:

```
import java.util.Objects;

public abstract class Transaction {
    protected double amount;
    protected String category;

    public Transaction(double amount, String category) {
        this.amount = amount;
        this.category = category;
    }

    public double getAmount() {
        return amount;
    }

    public String getCategory() {
        return category;
    }

    @Override
    public String toString() {
        return this.getClass().getSimpleName() + "{category='" + category + "',
amount=" + amount + "}";
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Transaction that = (Transaction) o;
        return Double.compare(that.amount, amount) == 0 &&
                Objects.equals(category, that.category);
    }

    @Override
    public int hashCode() {
        return Objects.hash(amount, category);
    }
}
public class GroceryTransaction extends Transaction {
    public GroceryTransaction(double amount) {
        super(amount, "groceries");
    }
}
public class UtilityTransaction extends Transaction {
    public UtilityTransaction(double amount) {
        super(amount, "utilities");
    }
}
public class EntertainmentTransaction extends Transaction {
    public EntertainmentTransaction(double amount) {
        super(amount, "entertainment");
    }
}
import java.util.*;
import java.util.stream.Collectors;

public class TransactionReport {
    public static void main(String[] args) {
        // Create a list of various types of transactions
        List<Transaction> transactions = Arrays.asList(
            new GroceryTransaction(100.0),
            new GroceryTransaction(200.0),
            new UtilityTransaction(300.0),
            new UtilityTransaction(150.0),
            new EntertainmentTransaction(120.0),
            new EntertainmentTransaction(180.0)
        );

        // Use Stream API to group transactions by category and sum the amounts
        Map<String, Double> totalAmountByCategory = transactions.stream()
            .collect(Collectors.groupingBy(
                Transaction::getCategory,                // Group by category (Method
reference)
```

```
                Collectors.summingDouble(t -> t.getAmount()) // Sum the amounts
(Lambda expression)
            ));

        // Print the total amount spent per category
        totalAmountByCategory.forEach((category, totalAmount) ->
            System.out.println("Category: " + category + ", Total Amount: " +
totalAmount)
        );
    }
}
```

OUTPUT:



```
<terminated> TransactionReport (1) [Java Application] /Library/Jav
Category: entertainment, Total Amount: 300.0
Category: utilities, Total Amount: 450.0
Category: groceries, Total Amount: 300.0
```

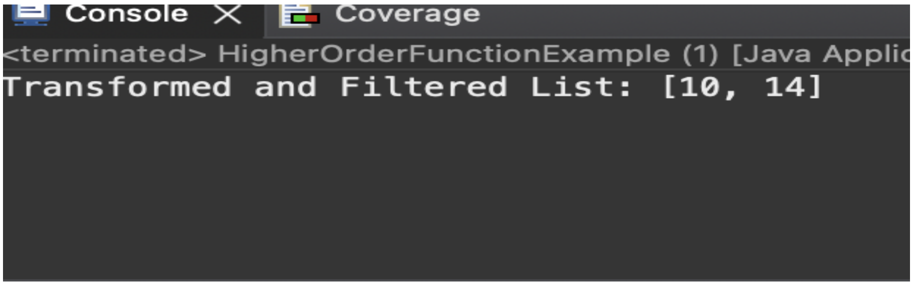
3) Higher-order Function with Functional Interfaces Create a higher-order function that takes two functional interfaces: 1. A Function to transform a list of integers. 2. A Predicate to filter the integers after transformation. We will then use the Stream API to apply these transformations and filters on a list of integers. 1 Higher-order function transformAndFilter: This method accepts a list of integers, a Function (for transformation), and a Predicate (for filtering). It uses the Stream API to apply the transformation (map) and filtering (filter) operations in sequence and returns the result as a list. 2 Lambda expressions: ⎤ The squareFunction lambda squares each integer. ⎤ The isEvenPredicate lambda filters out odd numbers and retains only even ones. 3 Stream operations: ⎤ map(transformer) applies the transformation (squaring the number). ⎤ .filter(filter) applies the filter to keep only even numbers.

CODE:

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

public class HigherOrderFunctionExample {

    // Higher-order function: takes a list, a transformer function, and a filter predicate.
    public static List<Integer> transformAndFilter(List<Integer> numbers,
                                                   Function<Integer, Integer> transformer,
                                                   Predicate<Integer> filter) {
        // Apply the transformation and filtering using Stream API
        return numbers.stream()
```

```
                .map(transformer)   // Apply the transformation (map)
                .filter(filter)     // Apply the filtering (filter)
                .collect(Collectors.toList());  // Collect the result as a list
    }

    public static void main(String[] args) {
        // Sample list of integers
        List<Integer> numbers = Arrays.asList(5, 7, 8, 9, 10, 11, 12);

        // Lambda expressions
        Function<Integer, Integer> doubleFunction = n -> n * 2;  // Lambda to double
the numbers
        Predicate<Integer> lessThanOrEqual15Predicate = n -> n <= 15;  // Lambda to
filter numbers <= 15

        // Call the higher-order function
        List<Integer> result = transformAndFilter(numbers, doubleFunction,
lessThanOrEqual15Predicate);

        // Print the result
        System.out.println("Transformed and Filtered List: " + result);
    }
}
```

OUTPUT:



```
<terminated> HigherOrderFunctionExample (1) [Java Applic
Transformed and Filtered List: [10, 14]
```

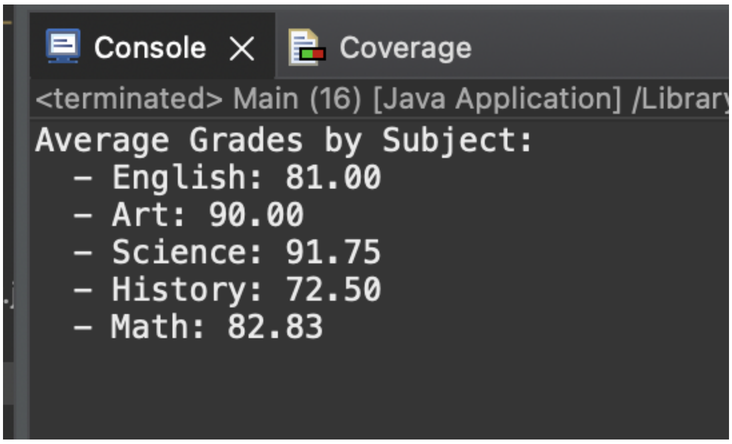
4) Parallel Stream with Custom Collector and Lambdas Calculate the average grade by subject from the large set of student grades. The process should be efficient, so you decide to use a parallel stream. Create a custom Collector that uses a lambda expression to handle the accumulation and reduction phases in parallel. Objective: Implement a custom Collector using lambdas to compute the average grade per subject in parallel using the Stream API. 1 Define the Data Structure: Create a StudentGrade class to hold the data (subject and grade). 2 Create a Custom Collector: Implement a collector to accumulate grades and calculate averages. 3 Parallel Stream Processing: Use the parallel stream to ensure the operation is performed efficiently. Explanation 1 StudentGrade Class: Holds data for each grade entry, including the subject and the grade. 2 AverageGradeCollector Class: ⎤ supplier: Provides a new HashMap to collect grades by subject. ⎤ accumulator: Adds grades to the list

corresponding to each subject. ⌉ combiner: Merges two maps by combining lists of grades. ⌉ finisher: Computes the average grade for each subject from the collected lists. 3 Parallel Stream Processing: ⌉ Use parallelStream() to process the grades list in parallel. ⌉ Collect results using AverageGradeCollector.

CODE:

```
public class StudentGrade {
    private String subject;
    private double grade;

    public StudentGrade(String subject, double grade) {
        this.subject = subject;
        this.grade = grade;
    }

    public String getSubject() {
        return subject;
    }

    public double getGrade() {
        return grade;
    }
}
import java.util.*;
import java.util.function.*;
import java.util.stream.Collector;
import java.util.stream.Collectors;

public class AverageGradeCollector implements Collector<StudentGrade,
Map<String, List<Double>>, Map<String, Double>> {

    @Override
    public Supplier<Map<String, List<Double>>> supplier() {
        // Provides an empty HashMap to collect grades by subject
        return HashMap::new;
    }

    @Override
    public BiConsumer<Map<String, List<Double>>, StudentGrade> accumulator() {
        // Adds grades to the list corresponding to each subject
        return (map, studentGrade) -> {
            map.computeIfAbsent(studentGrade.getSubject(), k -> new
ArrayList<>()).add(studentGrade.getGrade());
        };
    }

    @Override
    public BinaryOperator<Map<String, List<Double>>> combiner() {
        // Merges two maps by combining lists of grades
        return (map1, map2) -> {
            map2.forEach((subject, grades) -> {
                map1.merge(subject, grades, (grades1, grades2) -> {
                    grades1.addAll(grades2);
                    return grades1;
                });
            });
            return map1;
        };
    }

    @Override
    public Function<Map<String, List<Double>>, Map<String, Double>> finisher() {
        // Computes the average grade for each subject
        return map -> map.entrySet()
                .stream()
                .collect(Collectors.toMap(
                        Map.Entry::getKey,
                        entry -> entry.getValue().stream()
                                .mapToDouble(Double::doubleValue)
                                .average()
                                .orElse(0.0)
                ));
    }

    @Override
    public Set<Characteristics> characteristics() {
        // No characteristics are specified since we need to perform a final transformation
(finisher)
        return Collections.emptySet();
    }
}
import java.util.Arrays;
import java.util.List;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        // New sample data of student grades with different subjects and more variation
        List<StudentGrade> grades = Arrays.asList(
                new StudentGrade("Math", 75.0),
                new StudentGrade("Math", 82.5),
                new StudentGrade("Science", 95.0),
                new StudentGrade("Math", 91.0),
                new StudentGrade("Science", 88.5),
                new StudentGrade("History", 60.0),
                new StudentGrade("History", 85.0),
                new StudentGrade("English", 78.0),
                new StudentGrade("English", 84.0),
                new StudentGrade("Art", 92.0),
                new StudentGrade("Art", 88.0)
        );

        // Parallel stream processing to calculate average grades per subject
        Map<String, Double> averageGradesBySubject = grades.parallelStream()
                .collect(new AverageGradeCollector());

        // Display the results in a different format
        System.out.println("Average Grades by Subject:");
        averageGradesBySubject.forEach((subject, avgGrade) -> {
            System.out.printf("  - %s: %.2f%n", subject, avgGrade);
        });
    }
}
```

OUTPUT:



```
<terminated> Main (16) [Java Application] /Library
Average Grades by Subject:
  - English: 81.00
  - Art: 90.00
  - Science: 91.75
  - History: 72.50
  - Math: 82.83
```