**Week 8. Functional Interfaces, Lambda Expressions, Map and Stream API**

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.

Class: EmployeeFilterDemo import java.util.\*;

import java.util.function.\*;

import java.util.stream.\*;

class Employee { String name; double salary; String department;

int yearsOfExperience;

Employee(String name, double salary, String department, int yearsOfExperience) {

this.name = name; this.salary = salary;

this.department = department; this.yearsOfExperience = yearsOfExperience;

}

@Override

public String toString() {

return name + " - " + department + " - " + salary + " - " + yearsOfExperience + " years";

}

}

@FunctionalInterface interface EmployeeFilter {

boolean filter(Employee e);

}

public class EmployeeFilterDemo {

public static List<Employee> filterEmployees(List<Employee> employees, EmployeeFilter filter) {

return employees.stream().filter(filter::filter).collect(Collectors.toList());

}

public static void main(String[] args) { List<Employee> employees = Arrays.asList(

new Employee("Alice", 90000, "Engineering", 8),

new Employee("Bob", 75000, "Marketing", 6),

new Employee("Charlie", 85000, "Engineering", 10),

new Employee("David", 70000, "HR", 5),

new Employee("Eve", 95000, "Engineering", 9)

);

EmployeeFilter highSalaryFilter = e -> e.salary > 80000; EmployeeFilter engineeringDeptFilter = e ->

"Engineering".equals(e.department);

EmployeeFilter experiencedFilter = e -> e.yearsOfExperience >= 7;

List<Employee> filteredEmployees = filterEmployees(employees,

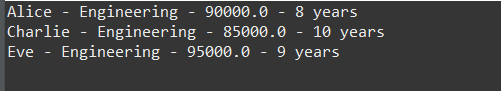
e -> highSalaryFilter.filter(e) && engineeringDeptFilter.filter(e) && experiencedFilter.filter(e));

filteredEmployees.forEach(System.out::println);

}

}

OUTPUT:



1. **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.

Class: Transaction

import java.util.\*;

import java.util.stream.Collectors;

class Transaction { private double amount; private String category;

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

this.category = category;

}

public double getAmount() { return amount;

}

public String getCategory() { return category;

}

}

public class TransactionReport {

public static void main(String[] args) { List<Transaction> transactions = Arrays.asList(

new Transaction(50.0, "groceries"), new Transaction(20.0, "utilities"), new Transaction(30.0, "groceries"),

new Transaction(10.0, "entertainment"), new Transaction(40.0, "utilities")

);

Map<String, Double> report = transactions.stream()

.collect(Collectors.groupingBy(Transaction::getCategory, Collectors.summingDouble(Transaction::getAmount)));

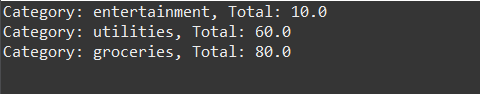
report.forEach((category, total) ->

System.out.println("Category: " + category + ", Total: " + total));

}

}

OUTPUT:



1. **Higher-order Function with Functional Interfaces**

Create a higher-order function that takes two functional interfaces:

1. A Function<Integer, Integer> to transform a list of integers.
2. A Predicate<Integer> 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<Integer, Integer> (for transformation), and a Predicate<Integer> (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.

Class: HigherOrderFunctionExample import java.util.List;

import java.util.function.Function; import java.util.function.Predicate; import java.util.stream.Collectors;

public class HigherOrderFunctionExample {

public static List<Integer> transformAndFilter(List<Integer> integers, Function<Integer, Integer> transformer, Predicate<Integer> filter)

{

return integers.stream()

.map(transformer)

.filter(filter)

.collect(Collectors.toList());

}

public static void main(String[] args) { List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);

Function<Integer, Integer> squareFunction = x -> x \* x; Predicate<Integer> isEvenPredicate = x -> x % 2 == 0;

List<Integer> result = transformAndFilter(numbers, squareFunction, isEvenPredicate); System.out.println(result);

}

}

OUTPUT:



1. **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.

Class: ParallelStreamExample

import java.util.\*;

import java.util.concurrent.ConcurrentHashMap; import java.util.function.\*;

import java.util.stream.Collector; import java.util.stream.Collectors;

class StudentGrade { String subject; double grade;

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

this.grade = grade;

}

public String getSubject() { return subject;

}

public double getGrade() { return grade;

}

}

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

@Override

public Supplier<Map<String, List<Double>>> supplier() { return ConcurrentHashMap::new;

}

@Override

public BiConsumer<Map<String, List<Double>>, StudentGrade> accumulator() {

return (map, studentGrade) -> map.computeIfAbsent(studentGrade.getSubject(), k -> new ArrayList<>()).add(studentGrade.getGrade());

}

@Override

public BinaryOperator<Map<String, List<Double>>> combiner() { return (map1, map2) -> {

map2.forEach((key, value) -> map1.merge(key, value, (l1, l2) -> { l1.addAll(l2);

return l1;

}));

return map1;

};

}

@Override

public Function<Map<String, List<Double>>, Map<String, Double>> finisher() {

return map -> map.entrySet().stream()

.collect(Collectors.toMap(Map.Entry::getKey, e -> e.getValue().stream().mapToDouble(Double::doubleValue).average().orElse (0.0)));

}

@Override

public Set<Characteristics> characteristics() { return Collections.emptySet();

}

}

public class ParallelStreamExample { public static void main(String[] args) {

List<StudentGrade> grades = Arrays.asList( new StudentGrade("Math", 85),

new StudentGrade("Math", 90),

new StudentGrade("Science", 78),

new StudentGrade("Science", 82),

new StudentGrade("English", 88),

new StudentGrade("English", 76)

);

Map<String, Double> averageGrades = grades.parallelStream().collect(new AverageGradeCollector());

averageGrades.forEach((subject, avg) -> System.out.println(subject + ": " + avg));

}

}

OUTPUT:

