import java.util.\*;

import java.util.function.Function;

import java.util.stream.Collectors;

class Employee {

private int id;

private String name;

private String email;

private String gender;

private boolean newJoining;

private double salary;

private double rating;

// Constructor

public Employee(int id, String name, String email, String gender, boolean newJoining, double salary, double rating) {

this.id = id;

this.name = name;

this.email = email;

this.gender = gender;

this.newJoining = newJoining;

this.salary = salary;

this.rating = rating;

}

// Getters

public int getId() { return id; }

public String getName() { return name; }

public String getEmail() { return email; }

public String getGender() { return gender; }

public boolean isNewJoining() { return newJoining; }

public double getSalary() { return salary; }

public double getRating() { return rating; }

// toString method for printing employee details

@Override

public String toString() {

return "Employee{id=" + id + ", name='" + name + "', email='" + email + "', gender='" + gender + "', newJoining=" + newJoining + ", salary=" + salary + ", rating=" + rating + "}";

}

}

public class EmployeeStreamExample {

public static void main(String[] args) {

// Sample list of employees

List<Employee> employees = Arrays.asList(

new Employee(1, "Alice", "alice@example.com", "Female", true, 50000, 4.2),

new Employee(2, "Bob", "bob@example.com", "Male", false, 60000, 3.8),

new Employee(3, "Charlie", "charlie@example.com", "Male", true, 45000, 4.5),

new Employee(4, "Diana", "diana@example.com", "Female", true, 70000, 4.9),

new Employee(5, "Eve", "eve@example.com", "Female", false, 55000, 4.0),

new Employee(6, "Frank", "frank@example.com", "Male", false, 40000, 3.5)

);

// i. Filter employees by gender "Female"

System.out.println("Employees with gender Female:");

employees.stream()

.filter(e -> e.getGender().equalsIgnoreCase("Female"))

.forEach(e -> System.out.println(e)); // Replaced System.out::println with lambda

// ii. Filter employees who are new joiners

System.out.println("\nNew joiners (newJoining = true):");

employees.stream()

.filter(Employee::isNewJoining)

.forEach(e -> System.out.println(e)); // Replaced System.out::println with lambda

// iii. Sort employees by rating in ascending order

System.out.println("\nEmployees sorted by rating (ascending):");

employees.stream()

.sorted(Comparator.comparingDouble(Employee::getRating))

.forEach(e -> System.out.println(e)); // Replaced System.out::println with lambda

// iv. Sort employees by both rating (ascending) and salary (ascending)

System.out.println("\nEmployees sorted by rating and salary:");

employees.stream()

.sorted(Comparator.comparingDouble(Employee::getRating).thenComparingDouble(Employee::getSalary))

.forEach(e -> System.out.println(e)); // Replaced System.out::println with lambda

// v. Retrieve employee with max salary

System.out.println("\nEmployee with maximum salary:");

employees.stream()

.max(Comparator.comparingDouble(Employee::getSalary))

.ifPresent(e -> System.out.println(e)); // Replaced System.out::println with lambda

// vi. Retrieve employee with min salary

System.out.println("\nEmployee with minimum salary:");

employees.stream()

.min(Comparator.comparingDouble(Employee::getSalary))

.ifPresent(e -> System.out.println(e)); // Replaced System.out::println with lambda

// vii. Group employees by Gender

System.out.println("\nEmployees grouped by Gender:");

Map<String, List<Employee>> employeesByGender = employees.stream()

.collect(Collectors.groupingBy(Employee::getGender));

employeesByGender.forEach((gender, empList) -> {

System.out.println(gender + ": ");

// Using a for loop to print employee details in the group

for (Employee emp : empList) {

System.out.println(emp);

}

});

}

}

The Stream API in Java, introduced in **Java 8**, is a powerful feature that allows for functional-style operations on collections. It provides a high-level abstraction for processing sequences of elements (such as collections, arrays, or I/O channels) in a declarative way. In the context of the employee data example, we use the Stream API to filter, sort, group, and aggregate data from a collection of Employee objects.

**Employee Class Design**

The first part of the program involves creating an Employee class. This class has several attributes that represent typical employee details, such as:

* **id**: A unique identifier for each employee.
* **name**: The name of the employee.
* **email**: The employee’s email address.
* **gender**: Gender of the employee (could be "Male" or "Female").
* **newJoiner**: A boolean flag to indicate whether the employee is a new joiner.
* **salary**: The salary of the employee.
* **rating**: A performance rating (e.g., out of 5).

The class also includes a constructor, getters, setters, and a toString() method to help in printing the employee details.

**2. Using Stream API Operations**

Stream API allows us to process a stream of data (in this case, a list of employees) in a functional and declarative style. We can perform operations like **filtering**, **sorting**, **mapping**, and **grouping** on the data. Here's how each operation works in the example: