Jose Eduardo Pons

Case Study: Analyzing Employee Data Using Java Streams

Introduction

This case study focuses on a Java application that processes a list of employees using the Stream API. The objective is to filter, analyze, and report on employee data, particularly focusing on those over the age of 21. The operations include filtering employees, generating reports, collecting project information, sorting employees by department, and calculating average salaries by department.

The Problem

A company's HR department wants to generate insights and reports based on the employee data. Specifically, they need to:

1. Identify employees who are older than 21.
2. Generate detailed reports for these employees.
3. List all projects involving these employees.
4. Group employees by their departments.
5. Calculate the average salary within each department.

The Dataset

The dataset consists of a list of employees, each represented by an `Employee` object. The `Employee` class contains:

- A name (`String`).
- A department (`String`).
- A salary (`double`).
- An age (`int`).
- A list of projects (`List<String>`).

The list of employees is initialized in the `main` method.

The Solution

The Java code implements the solution using the Stream API. Below, we discuss each operation performed by the code:

Filtering Employees Over 21

```
List<Employee> matureEmployees = employees.stream()
    .filter(e -> e.getAge() > 21)
    .collect(Collectors.toList());
```

This stream operation filters the employees who are older than 21 and collects them into a list.

Jose Eduardo Pons

Generating Employee Reports

```
List<String> employeeReports = matureEmployees.stream()
    .map(emp -> "Name: " + emp.getName() + ", Dept: " +
emp.getDepartment() + ", Salary: $" + emp.getSalary())
    .collect(Collectors.toList());
```

This code generates a list of string reports for each employee over the age of 21, including their name, department, and salary.

Collecting All Projects

```
List<String> allProjects = matureEmployees.stream()
    .flatMap(emp -> emp.getProjects().stream())
    .collect(Collectors.toList());
```

This operation collects all the projects associated with employees over 21. The `flatMap` is used to flatten the list of projects from each employee into a single list.

Grouping Employees by Department

```
Map<String, List<Employee>> employeesByDepartment =
matureEmployees.stream()
    .collect(Collectors.groupingBy(Employee::getDepartment));
```

The employees are grouped by their department into a `Map` where the key is the department name and the value is the list of employees in that department.

Calculating Average Salary by Department

```
Map<String, Double> averageSalaryByDepartment =
employeesByDepartment.entrySet().stream()
    .collect(Collectors.toMap(
        Map.Entry::getKey,
        entry -> entry.getValue().stream()
            .mapToDouble(Employee::getSalary)
            .average()
            .orElse(0.0)
));
```

This operation calculates the average salary for each department. The `Map` contains department names as keys and the corresponding average salary as values.

Output

The program outputs the following:

- Detailed reports of employees over the age of 21.
- A list of all projects involving these employees.
- The average salary for each department.

Jose Eduardo Pons

Conclusión

This case study demonstrates how the Java Stream API can be used to perform various data processing tasks efficiently. The operations, such as filtering, mapping, and grouping, make it easier to derive insights from the employee dataset. This approach is both concise and scalable, making it suitable for handling larger datasets and more complex analysis.