Jose Eduardo Pons Fidalgo

Case Study: Analyzing Student Performance Using Java Streams

Introduction

In this case study, we examine a Java application that processes and analyzes a list of students using the Stream API. The goal is to perform several operations, such as calculating the average class grade, identifying failing students, selecting top performers, sorting students by grades, and grouping them by grade level.

The Problem

A school administrator wants to analyze student performance in a large class. Specifically, the administrator needs to:

1. Calculate the overall class average.
2. Identify students who are failing.
3. Highlight the top students.
4. Sort students by their overall grades.
5. Group students by their grade levels.

The Dataset

The dataset consists of a list of 26 students, each represented by a `Student` object. The `Student` class contains:

- A name (`String`).
- A student ID (`String`).
- A list of grades (`List<Double>`).
- An overall grade (`double`), which represents the final grade.

The list of students 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:

Calculating the Class Average

```
double classAverage = students.stream()
    .mapToDouble(Student::getOverallGrade)
    .average()
    .orElse(0.0);
```

This stream operation calculates the average of all students' overall grades. If the list is empty, it returns `0.0`.

Identifying Failing Students

```
List<Student> failingStudents = students.stream()
    .filter(student -> student.getGradeLevel().equals("F"))
    .collect(Collectors.toList());
```

This code filters out students whose grade level is "F" (failing) and collects them into a list.

Selecting Top Students

```
List<Student> topStudents = students.stream()
    .filter(student -> student.getGradeLevel().equals("A"))
    .collect(Collectors.toList());
```

Similarly, this operation filters students with grade level "A" (top students) and collects them into a list.

Sorting Students by Overall Grade

```
List<Student> sortedStudents = students.stream()

.sorted(Comparator.comparingDouble(Student::getOverallGrade).reverse
d())
    .collect(Collectors.toList());
```

The students are sorted by their overall grades in descending order (from highest to lowest).

Grouping Students by Grade Level

```
Map<String, List<Student>> studentsByGradeLevel = students.stream()
    .collect(Collectors.groupingBy(Student::getGradeLevel));
```

This operation groups students by their grade levels into a `Map` where the key is the grade level and the value is a list of students with that grade level.

Output

The program outputs the following:

- The class average.
- A list of failing students.
- A list of top students.
- A list of all students sorted by their overall grades.
- A map of students grouped by their grade levels, with a blank line separating each grade level.

Conclusion

This case study demonstrates how the Java Stream API can be used to efficiently process and analyze data in a functional style. By leveraging streams, the code is concise, readable, and easy to maintain. The approach can be extended to handle more complex data processing tasks, making it a powerful tool for handling large datasets.

Jose Eduardo Pons Fidalgo