**Code Explanation**

1. **Class Structure and Imports**

import java.util.\*;

import java.util.concurrent.ConcurrentHashMap;

import java.util.function.Function;

import java.util.stream.Collectors;

import java.util.stream.Stream;

* **Purpose**: Imports essential utilities for collections, concurrency, functional interfaces, and stream processing
* **Key Components**:
  + ConcurrentHashMap: Thread-safe map implementation
  + Function: Core functional interface for data transformation
  + Collectors: Stream terminal operations for aggregation
  + Stream: Foundation for declarative data processing

2. **Employee Data Model**

class Employee {

private final String name;

private final int age;

private final String department;

private final double salary;

private final String employeeId;

public Employee(String name, int age, String department, double salary, String employeeId) {

*// Constructor initializing all fields*

}

*// Accessor methods (getters)*

}

* **Immutable Design**: All fields are final to prevent state modification
* **Data Attributes**:
  + employeeId: Unique identifier for tracking
  + salary: Double-precision for financial accuracy
  + department: Categorical grouping key
* **Encapsulation**: Private fields with public getters ensure data integrity

3. **Core Processing Pipeline**

public class AdvancedEmployeeProcessor {

private static final int AGE\_THRESHOLD = 30;

private static final int DATA\_SIZE = 10000;

public static void main(String[] args) {

List<Employee> employees = generateEmployeeData(DATA\_SIZE);

*// Function interface implementation*

Function<Employee, String> nameDeptMapper = emp ->

String.format("%s (%s) - %s", emp.getEmployeeId(), emp.getName(), emp.getDepartment());

*// Concurrent result collection*

Collection<String> employeeIdentifiers = Collections.newSetFromMap(new ConcurrentHashMap<>());

*// Parallel stream processing*

double averageSalary = employees.parallelStream()

.peek(emp -> employeeIdentifiers.add(nameDeptMapper.apply(emp)))

.filter(emp -> emp.getAge() > AGE\_THRESHOLD)

.collect(Collectors.teeing(

Collectors.averagingDouble(Employee::getSalary),

Collectors.groupingByConcurrent(

Employee::getDepartment,

Collectors.summarizingDouble(Employee::getSalary)

),

(avg, statsMap) -> {

*// Statistical analysis*

statsMap.forEach((dept, stats) ->

System.out.printf("%-15s | Count: %-4d | Avg: $%,8.2f | Max: $%,8.2f%n",

dept, stats.getCount(), stats.getAverage(), stats.getMax())

);

return avg;

}

));

}

}

Key Components:

1. **Function Interface**:
   * Transforms Employee → Formatted String
   * nameDeptMapper: Combines ID, name, and department
   * Demonstrates behavior parameterization
2. **Concurrent Collection**:
   * Collections.newSetFromMap(new ConcurrentHashMap<>())
   * Thread-safe storage for parallel processing
3. **Parallel Stream**:
   * .parallelStream(): Automatic workload distribution
   * Operations:
     + peek(): Side-effect to collect identifiers
     + filter(): Age-based data selection
4. **Teeing Collector**:
   * Simultaneously calculates:
     + Overall average salary
     + Departmental salary statistics
   * Collectors.summarizingDouble() provides:
     + Count, average, min, max, sum

4. **Advanced Analytics**

*// Salary distribution analysis*

Map<String, Long> salaryDistribution = employees.parallelStream()

.collect(Collectors.groupingByConcurrent(

emp -> {

double salary = emp.getSalary();

if (salary < 50\_000) return "<50K";

if (salary < 100\_000) return "50K-100K";

return ">100K";

},

Collectors.counting()

));

*// Seniority index calculation*

Map<String, Double> seniorityIndex = employees.parallelStream()

.collect(Collectors.groupingByConcurrent(

Employee::getDepartment,

Collectors.averagingInt(emp -> emp.getAge() - 22)

));

Features:

1. **Salary Distribution**:
   * Categorizes employees into salary brackets
   * Uses custom classifier lambda
   * Concurrent grouping for performance
2. **Seniority Index**:
   * Calculates average experience per department
   * Assumes 22 as typical graduation age
   * Collectors.averagingInt() for experience mean

5. **Data Generation Utility**

private static List<Employee> generateEmployeeData(int size) {

String[] departments = {"Engineering", "Marketing", "Finance", "HR", "Operations", "R&D"};

Random random = new Random();

return Stream.generate(() -> {

String name = "EMP" + String.format("%05d", random.nextInt(size));

int age = 22 + random.nextInt(40);

String dept = departments[random.nextInt(departments.length)];

double salary = 40\_000 + (random.nextDouble() \* 120\_000);

String id = "ID" + UUID.randomUUID().toString().substring(0, 8);

return new Employee(name, age, dept, salary, id);

}).limit(size).collect(Collectors.toList());

}

* **Dynamic Generation**:
  + Creates 10,000 employee records
  + Realistic data ranges:
    - Age: 22-62 years
    - Salary: $40k-$160k
* **Unique Identifiers**:
  + UUID-based employee IDs
* **Department Distribution**:
  + Even spread across 6 departments

Architectural Advantages

1. **Concurrency Model**:
   * parallelStream() auto-scales to available cores
   * Concurrent collectors prevent thread contention
   * Lock-free algorithms minimize synchronization
2. **Memory Efficiency**:
   * Stream processing avoids intermediate collections
   * Primitive specialization (averagingDouble) reduces boxing
3. **Statistical Depth**:
   * Built-in summary statistics
   * Custom aggregation functions
   * Multi-dimensional analysis
4. **Real-World Readiness**:
   * Synthetic data scalability testing
   * Production-ready concurrency handling
   * Comprehensive performance metrics

*// Performance tracking*

long startTime = System.currentTimeMillis();

*// ... processing ...*

long duration = System.currentTimeMillis() - startTime;

System.out.printf("Processed %,d records in %d ms%n", DATA\_SIZE, duration);

1. **Functional Programming Principles**:
   * Immutable data structures
   * Pure functions (no side effects)
   * Declarative transformation pipelines
   * Higher-order function usage