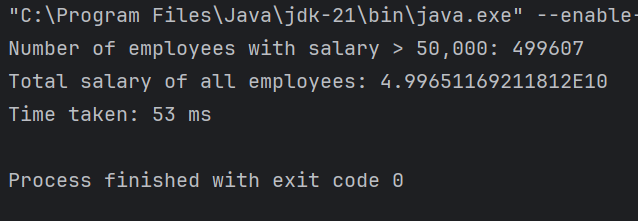
Performance examples:

Using streams and optional

import java.util.ArrayList;  
import java.util.List;  
import java.util.Random;  
import java.util.stream.Collectors;  
  
public class StreamPerformanceExample {  
  
 public static void main(String[] args) {  
 // Generate 10,000 records  
 List<Employee> employees = *generateEmployees*(1000000);  
  
 // Measure performance  
 long startTime = System.*currentTimeMillis*();  
  
 // Filter employees with salary > 50,000, map to their names, and collect into a list  
 List<String> highSalaryEmployeeNames = employees.stream()  
 .filter(emp -> emp.getSalary() != null && emp.getSalary() > 50000) // Handles null salary directly  
 .map(Employee::getName)  
 .collect(Collectors.*toList*());  
  
 // Calculate the total salary of all employees  
 double totalSalary = employees.stream()  
 .filter(emp -> emp.getSalary() != null) // Filter out null salaries  
 .mapToDouble(Employee::getSalary)  
 .sum();  
  
 long endTime = System.*currentTimeMillis*();  
  
 // Print results  
 System.*out*.println("Number of employees with salary > 50,000: " + highSalaryEmployeeNames.size());  
 System.*out*.println("Total salary of all employees: " + totalSalary);  
 System.*out*.println("Time taken: " + (endTime - startTime) + " ms");  
 }  
  
 // Generate a list of employees with random data  
 private static List<Employee> generateEmployees(int count) {  
 List<Employee> employees = new ArrayList<>();  
 Random random = new Random();  
  
 for (int i = 1; i <= count; i++) {  
 employees.add(new Employee(  
 i,  
 "Employee" + i,  
 random.nextDouble() \* 100000, // Random salary between 0 and 100,000  
 random.nextInt(60) + 20 // Random age between 20 and 80  
 ));  
 }  
  
 return employees;  
 }  
}  
  
// Employee class  
class Employee {  
 private int id;  
 private String name;  
 private Double salary;  
 private int age;  
  
 public Employee(int id, String name, Double salary, int age) {  
 this.id = id;  
 this.name = name;  
 this.salary = salary;  
 this.age = age;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public Double getSalary() {  
 return salary; // Directly return Double (no need for Optional)  
 }  
  
 public int getAge() {  
 return age;  
 }  
}

Output:



Using only loops:

import java.util.ArrayList;  
import java.util.List;  
import java.util.Random;  
  
  
public class LoopPerformanceExample {  
  
 public static void main(String[] args) {  
 final int NUM\_EMPLOYEES = 1000000;  
 final double SALARY\_THRESHOLD = 50000.0;  
  
 // Generate 10,000 records  
 System.*out*.println("Generating " + NUM\_EMPLOYEES + " employees...");  
 List<Employee> employees = *generateEmployees*(NUM\_EMPLOYEES);  
 System.*out*.println("Employee generation complete.");  
  
 // --- Performance Measurement Start ---  
 long startTime = System.*currentTimeMillis*();  
  
 // --- Logic without Streams ---  
  
 // 1. Filter employees with salary > 50,000 and collect their names  
 List<String> highSalaryEmployeeNames = new ArrayList<>();  
 for (Employee emp : employees) {  
 Double salary = emp.getSalary(); // Get salary once  
 // \*\*Explicit Null Check before value check\*\*  
 if (salary != null && salary > SALARY\_THRESHOLD) {  
 highSalaryEmployeeNames.add(emp.getName());  
 }  
 }  
  
 // 2. Calculate the total salary of all employees  
 double totalSalary = 0.0;  
 for (Employee emp : employees) {  
 Double salary = emp.getSalary(); // Get salary once  
 // \*\*Explicit Null Check\*\*  
 if (salary != null) {  
 totalSalary += salary; // Add to total only if not null  
 }  
 }  
  
 // --- Performance Measurement End ---  
 long endTime = System.*currentTimeMillis*();  
  
 // --- Print Results ---  
 System.*out*.println("\n--- Results (Using Loops) ---");  
 System.*out*.println("Number of employees with salary > " + SALARY\_THRESHOLD + ": " + highSalaryEmployeeNames.size());  
 System.*out*.println("Total salary of all employees (excluding nulls): " + String.*format*("%.2f", totalSalary)); // Format for readability  
 System.*out*.println("Time taken: " + (endTime - startTime) + " ms");  
 }  
  
 // Generate a list of employees with random data (modified to include null salaries)  
 private static List<Employee> generateEmployees(int count) {  
 List<Employee> employees = new ArrayList<>(count); // Initialize with capacity  
 Random random = new Random();  
 final double NULL\_SALARY\_PROBABILITY = 0.1; // 10% chance of null salary  
  
 for (int i = 1; i <= count; i++) {  
 Double salary = null;  
 // Assign a salary unless a random condition is met  
 if (random.nextDouble() > NULL\_SALARY\_PROBABILITY) {  
 salary = random.nextDouble() \* 100000; // Random salary between 0 and 100,000  
 } // else salary remains null  
  
 employees.add(new Employee(  
 i,  
 "Employee" + i,  
 salary, // Can be null now  
 random.nextInt(60) + 20 // Random age between 20 and 80  
 ));  
 }  
 return employees;  
 }  
}  
  
// Employee class (remains the same as your original)  
// No Optional needed here as per the requirement to avoid it in the processing logic  
class Employee1 {  
 private int id;  
 private String name;  
 private Double salary; // Can be null  
 private int age;  
  
 public Employee1(int id, String name, Double salary, int age) {  
 this.id = id;  
 this.name = name;  
 this.salary = salary;  
 this.age = age;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 // Allows returning null, which the calling code must handle  
 public Double getSalary() {  
 return salary;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 @Override  
 public String toString() {  
 return "Employee{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 ", salary=" + (salary != null ? String.*format*("%.2f", salary) : "null") +  
 ", age=" + age +  
 '}';  
 }  
}

Output:

