Java Stream API Employee Dataset - Answer Key With Code

59 active practice questions with Stream API implementations and expected outputs.

Filter	**Q1. Find all employees whose salary is greater than 60,000.**

Code	`String answer = employees.stream()`
	`  .filter(e -> e.getSalary() > 60000)`
	`  .map(e -> e.getName() + "(" + e.getId() + ", " + (int)e.getSalary() + ")")`
	`  .collect(Collectors.joining(", "));`

Output	`John(101, 70000), Emma(105, 80000), Mike(106, 75000), Sophia(107, 90000), John(109, 70000)`

Basic	**Q2. Count the total number of employees.**
Code	`long answer = employees.stream().count();`
Output	`10`

Collector	**Q3. Get a list of all employee names.**

Code	`List<String> answer = employees.stream()`
	`  .map(Employee::getName)`
	`  .collect(Collectors.toList());`

Output	`[John, Alice, Bob, David, Emma, Mike, Sophia, Chris, John, Robert]`

Collector	**Q4. Convert all employee names to uppercase.**

Code	`List<String> answer = employees.stream()`
	`  .map(e -> e.getName().toUpperCase())`
	`  .collect(Collectors.toList());`

Output	`[JOHN, ALICE, BOB, DAVID, EMMA, MIKE, SOPHIA, CHRIS, JOHN, ROBERT]`

Sorting	**Q5. Sort employees by salary in ascending order.**

Code	`String answer = employees.stream()`
	`  .sorted(Comparator.comparingDouble(Employee::getSalary))`
	`  .map(Employee::toString)`
	`  .collect(Collectors.joining(", "));`

Output	`David(104,45000), Alice(102,50000), Chris(108,55000), Bob(103,60000), Robert(110,60000),` `John(101,70000), John(109,70000), Mike(106,75000), Emma(105,80000), Sophia(107,90000)`

Sorting	**Q6. Sort employees by salary in descending order.**

Code	`String answer = employees.stream()`
	`  .sorted(Comparator.comparingDouble(Employee::getSalary).reversed())`
	`  .map(Employee::toString)`
	`  .collect(Collectors.joining(", "));`

Output	`Sophia(107,90000), Emma(105,80000), Mike(106,75000), John(101,70000), John(109,70000),` `Bob(103,60000), Robert(110,60000), Chris(108,55000), Alice(102,50000), David(104,45000)`

Basic	**Q7. Find the employee with the highest salary.**

Code	`Employee answer = employees.stream()`
	`  .max(Comparator.comparingDouble(Employee::getSalary))`
	`  .orElseThrow();`

Output	`Sophia(107, 90000)`

Basic	**Q8. Find the employee with the lowest salary.**

Code	`Employee answer = employees.stream()`
	`  .min(Comparator.comparingDouble(Employee::getSalary))`
	`  .orElseThrow();`

Output	`David(104, 45000)`

Basic	**Q9. Find the average salary of all employees.**

Code	`double answer = employees.stream()`
	`  .mapToDouble(Employee::getSalary)`
	`  .average()`
	`  .orElse(0);`

Output	`65500.0`

Basic	**Q10. Find the total salary expenditure of the company.**

Code	`int answer = employees.stream()`
	`    .mapToInt(e -> (int)e.getSalary())`
	`    .sum();`

Output	`655000`

Filter	**Q11. Find all employees belonging to the IT department.**

Code	`String answer = employees.stream()`
	`    .filter(e -> e.getDepartment().equals("IT"))`
	`    .map(Employee::toString)`
	`    .collect(Collectors.joining(", "));`

Output	`John(101,70000), Bob(103,60000), Mike(106,75000), Robert(110,60000)`

Filter	**Q12. Find employees whose names start with 'J'.**

Code	`String answer = employees.stream()`
	`    .filter(e -> e.getName().startsWith("J"))`
	`    .map(e -> e.getName() + "(" + e.getId() + ")")`
	`    .collect(Collectors.joining(", "));`

Output	`John(101), John(109)`

Filter	**Q13. Find employees whose salary is between 50,000 and 80,000.**

Code	`String answer = employees.stream()`
	`    .filter(e -> e.getSalary() >= 50000 && e.getSalary() <= 80000)`
	`    .sorted(Comparator.comparingDouble(Employee::getSalary))`
	`    .map(Employee::toString)`
	`    .collect(Collectors.joining(", "));`

Output	`Alice(102,50000), Bob(103,60000), Chris(108,55000), Robert(110,60000), John(101,70000),` `John(109,70000), Mike(106,75000), Emma(105,80000)`

Basic	**Q14. Check whether all employees earn more than 40,000.**
Code	`boolean answer = employees.stream().allMatch(e -> e.getSalary() > 40000);`
Output	`true`

Basic	**Q15. Check whether any employee belongs to Finance.**
Code	`boolean answer = employees.stream().anyMatch(e -> e.getDepartment().equals("Finance"));`
Output	`true`

Basic	**Q16. Check whether no employee has a salary less than 30,000.**
Code	`boolean answer = employees.stream().noneMatch(e -> e.getSalary() < 30000);`
Output	`true`

Filter	**Q17. Find the first employee whose salary is greater than 75,000.**

Code	`Employee answer = employees.stream()`
	`    .filter(e -> e.getSalary() > 75000)`
	`    .findFirst()`
	`    .orElseThrow();`

Output	`Emma(105, 80000)`

Filter	**Q18. Find any employee from the HR department.**

Code	`String answer = employees.stream()`
	`    .filter(e -> e.getDepartment().equals("HR"))`
	`    .findAny()`
	`    .map(e -> e.getName() + "(" + e.getId() + ")")`
	`    .orElse("Not found");`

Output	`Alice(102) or Emma(105)`

Grouping	**Q19. Count employees in each department.**
Code	`Map<String, Long> answer = employees.stream()` `    .collect(Collectors.groupingBy(Employee::getDepartment, LinkedHashMap::new, Collectors.counting()));`
Output	`IT:4, HR:2, Admin:2, Finance:2`

Grouping	**Q20. Group employees by department.**
Code	`Map<String, List<Employee>> answer = employees.stream()` `  .collect(Collectors.groupingBy(Employee::getDepartment, LinkedHashMap::new, Collectors.toList()));`
Output	`IT:[John(101), Bob(103), Mike(106), Robert(110)]; HR:[Alice(102), Emma(105)]; Admin:[David(104), John(109)]; Finance:[Sophia(107), Chris(108)]`

Grouping	**Q21. Find the average salary department-wise.**
Code	`Map<String, Double> answer = employees.stream()` `  .collect(Collectors.groupingBy(Employee::getDepartment, LinkedHashMap::new, Collectors.averagingDouble(Employee::getSalary`
Output	`IT:66250.0, HR:65000.0, Admin:57500.0, Finance:72500.0`

Grouping	**Q22. Find the highest-paid employee in each department.**

Code	`Map<String, Optional<Employee>> answer = employees.stream()`
	`  .collect(Collectors.groupingBy(Employee::getDepartment, LinkedHashMap::new,`
	`  Collectors.maxBy(Comparator.comparingDouble(Employee::getSalary))));`

Output	`IT: Mike(106,75000); HR: Emma(105,80000); Admin: John(109,70000); Finance: Sophia(107,90000)`

Grouping	**Q23. Find the lowest-paid employee in each department.**

Code	`Map<String, List<Employee>> answer = employees.stream()`
	`  .collect(Collectors.groupingBy(Employee::getDepartment, LinkedHashMap::new, Collectors.toList()));`
	`// For each department, find min salary and keep all employees matching it.`

Output	`IT: Bob(103,60000) & Robert(110,60000); HR: Alice(102,50000); Admin: David(104,45000); Finance: Chris(108,55000)`

Grouping	**Q24. Find the department with the highest average salary.**

Code	`String answer = employees.stream()`
	`  .collect(Collectors.groupingBy(Employee::getDepartment, Collectors.averagingDouble(Employee::getSalary)))`
	`  .entrySet().stream()`
	`  .max(Map.Entry.comparingByValue())`
	`  .map(e -> e.getKey() + " (" + e.getValue() + ")")`
	`  .orElse("");`

Output	`Finance (72500.0)`

Grouping	**Q25. Partition employees based on salary > 70,000.**
Code	`Map<Boolean, List<Employee>> answer = employees.stream()` `  .collect(Collectors.partitioningBy(e -> e.getSalary() > 70000));`
Output	`true: Emma(105,80000), Mike(106,75000), Sophia(107,90000); false: others`

Grouping	**Q26. Partition employees into IT and Non-IT groups.**
Code	`Map<Boolean, List<Employee>> answer = employees.stream()` `  .collect(Collectors.partitioningBy(e -> e.getDepartment().equals("IT")));`
Output	`IT:[John(101), Bob(103), Mike(106), Robert(110)]; Non-IT:[Alice(102), David(104), Emma(105), Sophia(107), Chris(108), John(109)]`

Grouping	**Q27. Group employees by salary range: Low <60k, Medium 60k-75k, High >75k.**
Code	`Map<String, List<Employee>> answer = employees.stream()` `  .collect(Collectors.groupingBy(e -> e.getSalary() < 60000 ? "Low" : e.getSalary() <= 75000 ? "Medium" : "High", LinkedHash`
Output	`Low:[David(104), Alice(102), Chris(108)]; Medium:[Bob(103), Robert(110), John(101), John(109), Mike(106)]; High:[Emma(105), Sophia(107)]`

Collector	**Q28. Find all distinct department names.**

Code	`List<String> answer = employees.stream()`
	`  .map(Employee::getDepartment)`
	`  .distinct()`
	`  .collect(Collectors.toList());`

Output	`[IT, HR, Admin, Finance]`

Collector	**Q29. Find duplicate employee names.**

Code	`Map<String, Long> freq = employees.stream()`
	`    .collect(Collectors.groupingBy(Employee::getName, LinkedHashMap::new, Collectors.counting()));`
	`String answer = freq.entrySet().stream()`
	`    .filter(e -> e.getValue() > 1)`
	`    .map(e -> e.getKey() + " (appears " + e.getValue() + " times)")`
	`    .collect(Collectors.joining(", "));`

Output	`John (appears twice)`

Collector	**Q30. Find duplicate salaries.**

Code	`Map<Double, Long> freq = employees.stream()`
	`    .collect(Collectors.groupingBy(Employee::getSalary, LinkedHashMap::new, Collectors.counting()));`
	`String answer = freq.entrySet().stream()`
	`    .filter(e -> e.getValue() > 1)`
	`    .map(e -> ((int)e.getKey().doubleValue()) + " (" + e.getValue() + ")")`
	`    .collect(Collectors.joining(", "));`

Output	`70000 (2), 60000 (2)`

Collector	**Q31. Find the frequency of employee names.**
Code	`Map<String, Long> answer = employees.stream()` `    .collect(Collectors.groupingBy(Employee::getName, LinkedHashMap::new, Collectors.counting()));`
Output	`John:2; Alice:1; Bob:1; David:1; Emma:1; Mike:1; Sophia:1; Chris:1; Robert:1`

Collector	**Q32. Find the frequency of salaries.**
Code	`Map<Double, Long> answer = employees.stream()` `    .collect(Collectors.groupingBy(Employee::getSalary, LinkedHashMap::new, Collectors.counting()));`
Output	`70000:2; 60000:2; 45000:1; 50000:1; 80000:1; 75000:1; 90000:1; 55000:1`

Grouping	**Q33. Find employees having the same salary.**

Code	`Map<Double, List<Employee>> answer = employees.stream()`
	`    .collect(Collectors.groupingBy(Employee::getSalary, LinkedHashMap::new, Collectors.toList()))`
	`    .entrySet().stream()`
	`    .filter(e -> e.getValue().size() > 1)`
	`    .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue, (a, b) -> a, LinkedHashMap::new));`

Output	`70000: John(101), John(109); 60000: Bob(103), Robert(110)`

Grouping	**Q34. Find departments having more than one employee.**

Code	`List<String> answer = employees.stream()`
	`    .collect(Collectors.groupingBy(Employee::getDepartment, LinkedHashMap::new, Collectors.counting()))`
	`    .entrySet().stream()`
	`    .filter(e -> e.getValue() > 1)`
	`    .map(Map.Entry::getKey)`
	`    .collect(Collectors.toList());`

Output	`IT, HR, Admin, Finance`

Sorting	**Q35. Find the second highest salary.**

Code	`int answer = employees.stream()`
	`    .map(e -> (int)e.getSalary())`
	`    .distinct()`
	`    .sorted(Comparator.reverseOrder())`
	`    .skip(1)`
	`    .findFirst()`
	`    .orElseThrow();`

Output	`80000`

Sorting	**Q36. Find the third highest salary.**

Code	`int answer = employees.stream()`
	`    .map(e -> (int)e.getSalary())`
	`    .distinct()`
	`    .sorted(Comparator.reverseOrder())`
	`    .skip(2)`
	`    .findFirst()`
	`    .orElseThrow();`

Output	`75000`

Sorting	**Q37. Find the Nth highest salary as distinct salaries descending.**

Code	`List<Integer> salaries = employees.stream()`
	`    .map(e -> (int)e.getSalary())`
	`    .distinct()`
	`    .sorted(Comparator.reverseOrder())`
	`    .collect(Collectors.toList());`
	`String answer = IntStream.range(0, salaries.size())`
	`    .mapToObj(i -> (i + 1) + ":" + salaries.get(i))`
	`    .collect(Collectors.joining(", ", "[", "]"));`

Output	`[1:90000, 2:80000, 3:75000, 4:70000, 5:60000, 6:55000, 7:50000, 8:45000]`

Sorting	**Q38. Find the top 3 highest salaries.**

Code	`List<Integer> answer = employees.stream()`
	`    .map(e -> (int)e.getSalary())`
	`    .distinct()`
	`    .sorted(Comparator.reverseOrder())`
	`    .limit(3)`
	`    .collect(Collectors.toList());`

Output	`[90000, 80000, 75000]`

Sorting	**Q39. Find the top 5 highest-paid employees.**

Code	`String answer = employees.stream()`
	`    .sorted(Comparator.comparingDouble(Employee::getSalary).reversed())`
	`    .limit(5)`
	`    .map(Employee::toString)`
	`    .collect(Collectors.joining(", "));`

Output	`Sophia(107,90000), Emma(105,80000), Mike(106,75000), John(101,70000), John(109,70000)`

Sorting	**Q40. Sort employees by department and then salary descending.**

Code	`Map<String, List<Employee>> answer = employees.stream()`
	`    .sorted(Comparator.comparing(Employee::getDepartment)`
	`        .thenComparing(Comparator.comparingDouble(Employee::getSalary).reversed()))`
	`    .collect(Collectors.groupingBy(Employee::getDepartment, LinkedHashMap::new, Collectors.toList()));`

Output	`IT:[Mike(106,75000), John(101,70000), Bob(103,60000), Robert(110,60000)]; HR:[Emma(105,80000), Alice(102,50000)]; Admin:[John(109,70000), David(104,45000)]; Finance:[Sophia(107,90000), Chris(108,55000)]`

Sorting	**Q41. Sort employees by name and then salary.**

Code	`List<Employee> answer = employees.stream()`
	`    .sorted(Comparator.comparing(Employee::getName).thenComparingDouble(Employee::getSalary))`
	`    .collect(Collectors.toList());`

Output	`[Alice(102,50000), Bob(103,60000), Chris(108,55000), David(104,45000), Emma(105,80000), John(101,70000), John(109,70000), Mike(106,75000), Robert(110,60000), Sophia(107,90000)]`

Sorting	**Q42. Find employees with the same salary sorted by name.**

Code	`Map<Double, List<Employee>> answer = employees.stream()`
	`    .collect(Collectors.groupingBy(Employee::getSalary, LinkedHashMap::new, Collectors.toList()))`
	`    .entrySet().stream()`
	`    .filter(e -> e.getValue().size() > 1)`
	`    .collect(Collectors.toMap(Map.Entry::getKey,`
	`        e -> e.getValue().stream().sorted(Comparator.comparing(Employee::getName)).collect(Collectors.toList()),`
	`        (a, b) -> a, LinkedHashMap::new));`

Output	`60000: [Bob(103), Robert(110)]; 70000: [John(101), John(109)]`

Collector	**Q43. Convert List<Employee> to List<String> containing names only.**

Code	`List<String> answer = employees.stream()`
	`    .map(Employee::getName)`
	`    .collect(Collectors.toList());`

Output	`[John, Alice, Bob, David, Emma, Mike, Sophia, Chris, John, Robert]`

Collector	**Q44. Convert List<Employee> to Map<Integer, Employee> using ID as key.**
Code	`Map<Integer, Employee> answer = employees.stream()` `    .collect(Collectors.toMap(Employee::getId, e -> e, (a, b) -> a, LinkedHashMap::new));`
Output	`{101:John(IT,70000), 102:Alice(HR,50000), 103:Bob(IT,60000), 104:David(Admin,45000), 105:Emma(HR,80000), 106:Mike(IT,75000), 107:Sophia(Finance,90000), 108:Chris(Finance,55000), 109:John(Admin,70000), 110:Robert(IT,60000)}`

Collector	**Q45. Convert List<Employee> to Map<String, Double> where name maps to salary list because John repeats.**

Code	`Map<String, List<Double>> answer = employees.stream()`
	`.collect(Collectors.groupingBy(Employee::getName, LinkedHashMap::new,`
	`Collectors.mapping(Employee::getSalary, Collectors.toList())));`

Output	`John:[70000,70000], Alice:[50000], Bob:[60000], David:[45000], Emma:[80000], Mike:[75000], Sophia:[90000], Chris:[55000], Robert:[60000]`

Collector	**Q46. Join all employee names into a comma-separated string.**

Code	`String answer = employees.stream()`
	`.map(Employee::getName)`
	`.collect(Collectors.joining(","));`

Output	`John,Alice,Bob,David,Emma,Mike,Sophia,Chris,John,Robert`

Collector	**Q47. Join all distinct department names separated by " | ".**

Code	`String answer = employees.stream()`
	`.map(Employee::getDepartment)`
	`.distinct()`
	`.collect(Collectors.joining(" | "));`

Output	`IT | HR | Admin | Finance`

Collector	**Q48. Create a map of Department -> Employee Count.**
Code	`Map<String, Long> answer = employees.stream()` `.collect(Collectors.groupingBy(Employee::getDepartment, LinkedHashMap::new, Collectors.counting()));`
Output	`{IT:4, HR:2, Admin:2, Finance:2}`

Collector	**Q49. Create a map of Department -> Total Salary.**

Code	`Map<String, Integer> answer = employees.stream()`
	`.collect(Collectors.groupingBy(Employee::getDepartment, LinkedHashMap::new,`
	`Collectors.summingInt(e -> (int)e.getSalary())));`

Output	`{IT:265000, HR:130000, Admin:115000, Finance:145000}`

Advanced	**Q66. Find employees whose salary is greater than their department average.**

Code	`Map<String, Double> avg = employees.stream()`
	`.collect(Collectors.groupingBy(Employee::getDepartment, Collectors.averagingDouble(Employee::getSalary)));`
	`List<Employee> answer = employees.stream()`
	`.filter(e -> e.getSalary() > avg.get(e.getDepartment()))`
	`.collect(Collectors.toList());`

Output	`John(101, IT,70000), Mike(106, IT,75000), Emma(105, HR,80000), John(109, Admin,70000), Sophia(107, Finance,90000)`

Basic	**Q67. Find employees earning the maximum salary in the company.**

Code	`double max = employees.stream().mapToDouble(Employee::getSalary).max().orElse(0);`
	`List<Employee> answer = employees.stream()`
	`.filter(e -> e.getSalary() == max)`
	`.collect(Collectors.toList());`

Output	`Sophia(107, 90000)`

Basic	**Q68. Find employees earning the minimum salary in the company.**

Code	`double min = employees.stream().mapToDouble(Employee::getSalary).min().orElse(0);`
	`List<Employee> answer = employees.stream()`
	`.filter(e -> e.getSalary() == min)`
	`.collect(Collectors.toList());`

Output	`David(104, 45000)`

Advanced	**Q69. Find departments where average salary is greater than 60,000.**

Code	`List<String> answer = employees.stream()`
	`.collect(Collectors.groupingBy(Employee::getDepartment, LinkedHashMap::new, Collectors.averagingDouble(Employee::getSalary`
	`.entrySet().stream()`
	`.filter(e -> e.getValue() > 60000)`
	`.map(Map.Entry::getKey)`
	`.collect(Collectors.toList());`

Output	`IT, HR, Finance`

Grouping	**Q70. Find the sum of salaries department-wise.**

Code	`Map<String, Integer> answer = employees.stream()`
	`    .collect(Collectors.groupingBy(Employee::getDepartment, LinkedHashMap::new,`
	`        Collectors.summingInt(e -> (int)e.getSalary())));`

Output	`IT:265000, HR:130000, Admin:115000, Finance:145000`

Advanced	**Q71. Find the employee with the longest name.**

Code	`int maxLen = employees.stream().mapToInt(e -> e.getName().length()).max().orElse(0);`
	`List<Employee> answer = employees.stream()`
	`    .filter(e -> e.getName().length() == maxLen)`
	`    .collect(Collectors.toList());`

Output	`Sophia(107) and Robert(110)`

Advanced	**Q72. Find the employee with the shortest name.**

Code	`Employee answer = employees.stream()`
	`    .min(Comparator.comparingInt(e -> e.getName().length()))`
	`    .orElseThrow();`

Output	`Bob(103)`

Grouping	**Q73. Find employees grouped by the first letter of their name.**
Code	`Map<Character, List<Employee>> answer = employees.stream()` `    .collect(Collectors.groupingBy(e -> e.getName().charAt(0), TreeMap::new, Collectors.toList()));`
Output	`A:[Alice(102)]; B:[Bob(103)]; C:[Chris(108)]; D:[David(104)]; E:[Emma(105)]; J:[John(101), John(109)]; M:[Mike(106)]; R:[Robert(110)]; S:[Sophia(107)]`

Advanced	**Q74. Create a custom collector to join employee names with " | ".**

Code	`String answer = employees.stream()`
	`    .map(Employee::getName)`
	`    .collect(Collector.of(StringBuilder::new,`
	`        (sb, name) -> { if (sb.length() > 0) sb.append(" | "); sb.append(name); },`
	`        (a, b) -> { if (a.length() > 0 && b.length() > 0) a.append(" | "); return a.append(b); },`
	`        StringBuilder::toString));`

Output	`John | Alice | Bob | David | Emma | Mike | Sophia | Chris | John | Robert`

Advanced	**Q75. Create a custom collector to calculate total salary without using Collectors.summingDouble().**

Code	`int answer = employees.stream()`
	`    .collect(Collector.of(() -> new int[1],`
	`        (sum, e) -> sum[0] += (int)e.getSalary(),`
	`        (a, b) -> { a[0] += b[0]; return a; },`
	`        sum -> sum[0]));`

Output	`655000`