🏢 Comprehensive Employee Examples

This section demonstrates various stream operations applied specifically to a list of `Employee` objects, covering common data manipulation tasks.

Additional Resources

If you found this guide helpful, you might also be interested in my other Spring Framework resources:

- Core Java & Java-8 Interview Questions
- Spring Boot Interview Questions
- Microservices with Spring Cloud Tutorials

Feel free to star and fork these repositories if you find them useful!

Employee Class Definition

- *First, let's define the `Employee` class used in these examples.*

```
import java.util.Objects;

class Employee {
    private String name;
    private int age;
    private String gender;
    private double salary;
    private String city;
    private String deptName;
    private boolean activeEmp;

    // Constructors , Getters, Setters , toStrring(),
hashCode()

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return
false;
        Employee employee = (Employee) o;
        return age == employee.age &&
Double.compare(employee.salary, salary) == 0 &&
                activeEmp == employee.activeEmp &&
Objects.equals(name, employee.name) &&
                Objects.equals(gender, employee.gender) &&
Objects.equals(city, employee.city) &&
                Objects.equals(deptName, employee.deptName);
    }

}
```

Sample Employee Data

- *We'll use this list for the examples below.*

```
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList; // Needed for some examples later

List<Employee> employeeList = Arrays.asList(
    new Employee("Alice", 30, "Female", 52000, "New York",
"IT", true),
    new Employee("Bob", 25, "Male", 60000, "San Francisco",
"Finance", false),
    new Employee("Charlie", 35, "Male", 55000, "New York",
"Marketing", true),
    new Employee("David", 40, "Male", 70000, "San Francisco",
"IT", true),
    new Employee("Emma", 28, "Female", 75000, "Los Angeles",
"HR", false),
    new Employee("Frank", 35, "Male", 58000, "New York", "IT",
true), // Duplicate name, same age as Charlie but diff salary
    new Employee("Grace", 28, "Female", 62000, "New York",
"IT", true) // Same age as Emma
);
```

Iterating Over Employees

- *Iterating using an enhanced for-loop (traditional).*

```
System.out.println("--- Iterating with Enhanced For-Loop ---
");
for (Employee emp : employeeList) {
    System.out.println(emp);
    // Access specific fields:
System.out.println(emp.getName());
}
```

- *Iterating using `forEach` with a Lambda Expression.*

```
System.out.println("\n--- Iterating with forEach + Lambda ---
");
employeeList.forEach(emp -> {
```

```
    System.out.println("Name: " + emp.getName() + ", Age: " +
emp.getAge());
});
```

- *Iterating using `forEach` with a Method Reference (prints the `toString()` representation).*

```
System.out.println("\n--- Iterating with forEach + Method
Reference ---");
employeeList.forEach(System.out::println);
// employeeList.stream().forEach(System.out::println);
```

- *Iterating using a Stream and `forEach` (equivalent to list's `forEach`).*

```
System.out.println("\n--- Iterating with Stream + forEach ---
");
employeeList.stream().forEach(emp -> {
    System.out.println("Dept: " + emp.getDeptName() + ",
Salary: " + emp.getSalary());
});
```

Collecting & Filtering Employee Data

- *Collect only the names of all employees into a new List.*

```
import java.util.stream.Collectors;

System.out.println("\n--- Collecting Employee Names ---");
List<String> employeeNames = employeeList.stream()
                                .map(Employee::getName) //
Method reference for getter
                            // .map(emp -> emp.getName())
// Equivalent lambda
                    //  .map(emp -> emp.getFirstName()+"
"emp.getLastName())

.collect(Collectors.toList());
                            // .filter(x -> x!=null)
```

```
                                //  .filter(x ->
x.startWith('a'))
                               //   .filter(name -> name != null
&& name.length() > 3)
                              //    .filter(name -> name != null
&& name.toLowerCase().startsWith("r"))
System.out.println(employeeNames);
```

> *Output: [Alice, Bob, Charlie, David, Emma, Frank, Grace]*

- *Filter employees older than 30.*

```
System.out.println("\n--- Filtering Employees Older Than 30 --
-");
// works good with the primitive type --  emp.getAge(),if it
is Integer -- NullPointerException
List<Employee> olderEmployees = employeeList.stream()
                                    .filter(emp ->
emp.getAge() > 30)

.collect(Collectors.toList());
List<Employee> olderEmployees = employeeList.stream()
                                    .filter(emp->
emp.getAge()!=null && emp -> emp.getAge() > 30)

.collect(Collectors.toList());
List<Employee> olderEmployees = employeeList.stream()
                                   .filter(emp -> {
                                       return emp.getAge() !=
null && emp.getAge() > 30;

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

olderEmployees.forEach(System.out::println);
```

> *Output: Includes Charlie, David, Frank*

- *Find unique and duplicate employee names.*

```
import java.util.Set;
import java.util.HashSet;
```

```
System.out.println("\n--- Finding Unique and Duplicate Names -
--");
Set<String> uniqueNames = new HashSet<>();
Set<String> duplicateNames = employeeList.stream()

.map(Employee::getName)
                                        .filter(name ->
!uniqueNames.add(name)) // .add returns false if element
already exists

.collect(Collectors.toSet());

System.out.println("All Unique Names Encountered: " +
uniqueNames);
System.out.println("Duplicate Names Found: " +
duplicateNames);
```

> *Output: All Unique Names Encountered: `[Alice, Bob, Charlie, David, Emma, Frank, Grace]` (order may vary) Duplicate Names Found: `[]` (In this specific dataset, names are unique) Note: If 'Frank' was named 'Charlie', Duplicates would be `[Charlie]`.*

- *Store unique employee objs.*

```
 // approach - 1
Set<Employee> uniqueEmployees = employeeList.stream()
            .collect(Collectors.toMap(
                            e -> e.getId(),             // Use
employee ID as key
                            Function.identity(),        //
Value: Employee object
                            (e1, e2) -> e1              // If
duplicate key (same ID), keep first
            )).values()
              .stream()
              .collect(Collectors.toSet());

// approach - 2
Set<Long> seen = new HashSet<>();
Set<Employee> uniqueEmployees = employeeList.stream()
```

```
                    .filter(e -> seen.add(e.getId()))  //
returns false for duplicates
                    .collect(Collectors.toSet());
```

- *Store unique employee objs on the basis of -(key,value).*

```
 // ☑ 1. Store first employee per department
    // If a department repeats, the first employee is
retained, others are ignored
    Map<String, Employee> firstEmployeeByDept =
employeeList.stream()
        .collect(Collectors.toMap(
            Employee::getDeptName,             // Key:
department name
            e -> e,                            // Value:
Employee object
            (existing, replacement) -> existing // Merge
function: keep first (ignore replacement)
        ));

    System.out.println("=== First Employee per Department
===");
    firstEmployeeByDept.forEach((dept, emp) ->
        System.out.println("Dept: " + dept + ", Employee: " +
emp)
    );

    // ☑ 2. Store latest employee per department
    // If a department repeats, the latest employee replaces
the previous one
    Map<String, Employee> latestEmployeeByDept =
employeeList.stream()
        .collect(Collectors.toMap(
            Employee::getDeptName,                // Key:
department name
            e -> e,                               // Value:
Employee object
            (existing, replacement) -> replacement // Merge
function: keep latest (override existing)
        ));

    System.out.println("\n=== Latest Employee per Department
```

```
===");
    latestEmployeeByDept.forEach((dept, emp) ->
        System.out.println("Dept: " + dept + ", Employee: " +
emp)
    );
});
```

👬 Grouping Employees

- *Group employees by their Department.*

```
import java.util.Map;

System.out.println("\n--- Grouping Employees by Department ---
");
Map<String, List<Employee>> employeesByDept =
employeeList.stream()
    .collect(Collectors.groupingBy(Employee::getDeptName));

employeesByDept.forEach((dept, emps) -> {
    System.out.println("Department: " + dept);
    emps.forEach(emp -> System.out.println("  " +
emp.getName()));
});
```

- *Group employees by Active Status (Active vs. Inactive).*

```
System.out.println("\n--- Grouping Employees by Active Status
---");
Map<Boolean, List<Employee>> employeesByActiveStatus =
employeeList.stream()
    .collect(Collectors.groupingBy(Employee::isActiveEmp));

employeesByActiveStatus.forEach((isActive, emps) -> {
    System.out.println("Active: " + isActive);
    emps.forEach(emp -> System.out.println("  " +
emp.getName()));
});
```

- *Count employees in each Department.*

```
System.out.println("\n--- Counting Employees per Department --
-");
Map<String, Long> countByDept = employeeList.stream()
    .collect(Collectors.groupingBy(Employee::getDeptName, //
Group by department
                                   Collectors.counting())); //
Count elements in each group
countByDept.forEach((dept, count) -> System.out.println(dept +
": " + count));
```

> *Output: HR: 1 Finance: 1 Marketing: 1 IT: 4*

- *Total salaries grouped by department.*

```
    // 1. Total salary of all employees
    double totalSalary = empList.stream()
        .map(Employee::getSalary)
        .reduce(0.0, Double::sum);
    System.out.println("Total Salary of All Employees: " +
totalSalary);

    // 2. Total salaries grouped by department
    Map<String, Double> totalSalariesByDept = empList.stream()
        .collect(Collectors.groupingBy(
            Employee::getDepartment,
            Collectors.reducing(0.0, Employee::getSalary,
Double::sum)
        ));

    System.out.println("Total Salaries by Department:");
    totalSalariesByDept.forEach((dept, total) ->
        System.out.println(dept + " → " + total)
    );
```

```
Total Salary of All Employees: 237000.0
Total Salaries by Department:
HR → 105000.0
IT → 122000.0
Finance → 70000.0
```

Sorting Employees

- *Sort employees by Name in ascending order.*

```
    // Approach 1: Using Collections.sort() with natural
ordering
    Collections.sort(employees); // Uses compareTo() method

    // Approach 2: Alternative using Stream API
    List<Employee> sortedEmployees = employees.stream()
                .sorted() // Uses natural ordering (compareTo)
            //
.sorted(Comparator.comparing(Employee::getName)
                .collect(Collectors.toList());

    // Approach 3:Sort employees by salary in ascending order
using anonymous class.
    List<Employee> sortedEmployees = employees.stream()
            .sorted(new Comparator<Employee>() {
                @Override
                public int compare(Employee e1, Employee e2) {
                    return Double.compare(e1.getSalary(),
e2.getSalary()); // Better than casting
                }
            })
            .collect(Collectors.toList());

    // Approach 4: Using lambda expression
    List<Employee> sorted3 = employees.stream()
            .sorted((e1, e2) ->
e2.getName().compareTo(e1.getName()))
            .collect(Collectors.toList());
    /*
    --------- Approach 5: Sorting using lambda and
Double.compare ---------
    Safer than (e1.getSalary() - e2.getSalary())
    because it avoids floating-point precision errors
    and potential overflow when working with large values.
    Returns:
     -1 if e1.getSalary() < e2.getSalary()
      0 if equal
      1 if e1.getSalary() > e2.getSalary()
    */
```

```
    List<Employee> sortedBySalary = employees.stream()
            .sorted((e1, e2) -> Double.compare(e1.getSalary(),
e2.getSalary())) // ascending order
            .collect(Collectors.toList());
```

```
Comparator.comparing(Employee::getName).reversed()
Comparator.comparingInt(Employee::getId).reversed()
Comparator.comparingDouble(Employee::getSalary).reversed()
Comparator.comparingLong(Employee::getMobile).reversed()
Comparator.comparing(Employee::getJoiningDate).reversed()
-  LocalDate
```

- *Sort employees by Name in descending order.*

```
    // Approach 1: Using Collections.sort() with reverse order
    Collections.sort(employees, Collections.reverseOrder());

    // Approach 2: Using Collections.reverseOrder() with
method reference
    List<Employee> sorted1 = employees.stream()

.sorted(Comparator.comparing(Employee::getName).reversed())
            .collect(Collectors.toList());

    // Approach 3: Using anonymous comparator for descending
order
    List<Employee> sorted2 = employees.stream()
            .sorted(new Comparator<Employee>() {
                @Override
                public int compare(Employee e1, Employee e2) {
                    return
e2.getName().compareTo(e1.getName()); // Reverse order
                }
            })
            .collect(Collectors.toList());

    // Approach 4: Using lambda expression
    List<Employee> sorted3 = employees.stream()
            .sorted((e1, e2) ->
e2.getName().compareTo(e1.getName()))
            .collect(Collectors.toList());
```

- *Multi-level sorting: Sort by City Descending, then by Name Ascending..*

```
    List<Employee> sortedEmployees = employees.stream()

.sorted(Comparator.comparing(Employee::getCity).reversed() //
City descending
                    .thenComparing(Employee::getName)) // Then
name ascending
            .collect(Collectors.toList());
```

- *Multi-level sorting: Sort by City Descending, then by Name Descending..*

```
    List<Employee> sortedEmployees = employees.stream()

.sorted(Comparator.comparing(Employee::getCity).reversed() //
City descending

.thenComparing(Employee::getName).reversed()) // Then name
descending
            .collect(Collectors.toList());
```

- *Traditional approach for implementing the comparator interface method.*

```
class SalaryComparator implements Comparator<Employee> {
    @Override
    public int compare(Employee e1, Employee e2) {
        return Double.compare(e1.getSalary(), e2.getSalary());
    }
}

Collections.sort(employees, new SalaryComparator());  // Sorts
in ascending order of salary
```

- *Select the top 3 Employees with the highest Salaries.*

```
System.out.println("\n--- Top 3 Highest Paid Employees ---");
List<Employee> top3Salaries = employeeList.stream()

.sorted(Comparator.comparingDouble(Employee::getSalary).revers
ed())
    .limit(3)
    .collect(Collectors.toList());
top3Salaries.forEach(System.out::println);
```

- *Select Employees starting from the 4th highest Salary onwards.*

```
System.out.println("\n--- Employees After Top 3 Paid ---");
List<Employee> afterTop3Salaries = employeeList.stream()

.sorted(Comparator.comparingDouble(Employee::getSalary).revers
ed())
    .skip(3) // Skip the first 3
    .collect(Collectors.toList());
afterTop3Salaries.forEach(System.out::println);
```

- *Select the 2nd & 3rd youngest Employees.*

```
System.out.println("\n--- 2nd and 3rd Youngest Employees ---
");
List<Employee> secondAndThirdYoungest = employeeList.stream()
    .sorted(Comparator.comparingInt(Employee::getAge))
    .skip(1) // Skip the youngest
    .limit(2) // Take the next two
    .collect(Collectors.toList());
secondAndThirdYoungest.forEach(System.out::println);
```

> *Output: Includes Emma (28) and Grace (28)*

Employee Statistics

- *Find the Employee with the Maximum Salary.*

```
Optional<Employee> highestPaidEmployee = employeeList.stream()

.collect(Collectors.maxBy(Comparator.comparingDouble(Employee:
:getSalary)));
highestPaidEmployee.ifPresent(System.out::println);
// Prints: Optional[Employee{id=2, name='Jane',
salary=85000.0}]

Optional<Double> maxSalary = employeeList.stream()
        .map(Employee::getSalary)
        .collect(Collectors.maxBy(Double::compareTo));
// Prints: Optional[85000.0]

/*
    Optional<T> max(Comparator<? super T> comparator)
    - Stream<T> is in the package: java.util.stream
*/
Optional<Employee> highestPaidEmployee = employeeList.stream()
        .max(Comparator.comparingDouble(Employee::getSalary));
// Prints: Optional[Employee{id=2, name='Jane',
salary=85000.0}]

Optional<Double> maxSalary = employeeList.stream()
        .map(Employee::getSalary)
        .max(Double::compareTo);
// Prints: Optional[85000.0]
```

> *Output: Emma's record*

- *Find the Employee with the Minimum Salary.*

```
System.out.println("\n--- Employee with Minimum Salary ---");
Optional<Employee> lowestPaidEmployee = employeeList.stream()

.collect(Collectors.minBy(Comparator.comparingDouble(Employee:
:getSalary)));
    // Alternatively:
.min(Comparator.comparingDouble(Employee::getSalary));
lowestPaidEmployee.ifPresent(System.out::println);
```

> *Output: Alice's record*

- *Calculate Summary Statistics for Employee Ages (Count, Sum, Min, Average, Max).*

```
import java.util.IntSummaryStatistics;

System.out.println("\n--- Summary Statistics for Employee Ages
---");
IntSummaryStatistics ageStats = employeeList.stream()
    .mapToInt(Employee::getAge) // Convert to IntStream
//  .mapToDouble(Employee::getSalary) // Convert to
DoubleStream - for the salaryStats
    .summaryStatistics(); // Calculate stats

System.out.println("Min: " + ageStats.getMin());
System.out.println("Max: " + ageStats.getMax());
System.out.println("Sum: " + ageStats.getSum());
System.out.println("Average: " + ageStats.getAverage());
System.out.println("Count: " + ageStats.getCount());
```

> *Output: Min: 25 Max: 40 Sum: 221 Average: 31.57... Count: 7*

- *Find the Highest Salary in each Department.*

```
import java.util.function.BinaryOperator;

System.out.println("\n--- Highest Salary per Department ---");
Map<String, Optional<Employee>> highestSalaryByDept =
employeeList.stream()
    .collect(Collectors.groupingBy(
        Employee::getDeptName,

Collectors.reducing(BinaryOperator.maxBy(Comparator.comparingD
ouble(Employee::getSalary)))
    ));

highestSalaryByDept.forEach((dept, empOpt) -> {
    System.out.print("Dept: " + dept + ", Highest Paid: ");
    empOpt.ifPresent(emp -> System.out.println(emp.getName() +
" (" + emp.getSalary() + ")"));
});
```

⚠ Stream Re-use Caveat

- *Streams cannot be reused after a terminal operation is called. Attempting to do so results in an `IllegalStateException`.*

```
Stream<String> namesStream =
employeeList.stream().map(Employee::getName);
System.out.println("\n--- Stream Re-use Attempt ---");
System.out.println("First consumption (forEach):");
namesStream.forEach(System.out::println); // Consumes the
stream

try {
    // Attempting to reuse the *same* stream instance
    long count = namesStream.count(); // This will fail
    System.out.println("Count (should not be reached): " +
count);
} catch (IllegalStateException e) {
    System.out.println("Error trying to reuse stream: " +
e.getMessage());
}
```

> *Output: Will print names, then throw `IllegalStateException: stream has already been operated upon or closed`*

- *To perform multiple operations, either recreate the stream or use a `Supplier`.*

```
import java.util.function.Supplier;

// Option 1: Recreate the stream
System.out.println("\n--- Recreating Stream ---");
employeeList.stream().map(Employee::getName).forEach(System.ou
t::println);
long countRecreated =
employeeList.stream().map(Employee::getName).count();
System.out.println("Count (recreated): " + countRecreated);

// Option 2: Use a Supplier
System.out.println("\n--- Using Supplier for Stream ---");
Supplier<Stream<Employee>> employeeStreamSupplier = () ->
employeeList.stream();
```

```
System.out.println("First consumption via supplier:");
employeeStreamSupplier.get().map(Employee::getName).forEach(Sy
stem.out::println); // Gets a new stream

System.out.println("Second consumption via supplier:");
long countSupplier = employeeStreamSupplier.get().count(); //
Gets another new stream
System.out.println("Count (supplier): " + countSupplier);
```

☑ Conclusion

> Java Streams offer a flexible and expressive paradigm for data processing. Mastering the distinction between **lazy intermediate operations** and **eager terminal operations**, along with understanding how to chain them effectively, unlocks the power of the Stream API for writing clean, declarative, and efficient Java code.

> Common problems, especially when working with collections of objects like `Employee`, can often be solved elegantly using combinations of stream operations like `filter`, `map`, `reduce`, `collect` (especially `groupingBy`), `sorted`, `distinct`, and summary statistics methods. Remember the single-consumption nature of streams and plan accordingly for multiple operations.