**Comparable vs. Comparator in Java**

In Java, both Comparable and Comparator are used to **define custom sorting logic** for objects.

| **Feature** | **Comparable** | **Comparator** |
| --- | --- | --- |
| **Package** | java.lang | java.util |
| **Purpose** | Defines natural sorting order of objects | Defines custom sorting order, useful for multiple sorting criteria |
| **Method** | compareTo(T o) | compare(T o1, T o2) |
| **Modifies class?** | Yes, class implements Comparable | No, separate class or lambda expression |
| **Sorts by** | Single field (natural ordering) | Multiple fields (custom ordering) |
| **Example** | Sorting Employee by id | Sorting Employee by name or salary |

**1️⃣ Comparable (Single Sorting)**

* If an object has a **default sorting order** (e.g., id, name), implement Comparable.
* This is useful when the **natural order is clear** and widely accepted (like sorting Strings alphabetically or Integers numerically).

**Example: Sorting Employees by ID**

import java.util.\*;

class Employee implements Comparable<Employee> {

private int id;

private String name;

public Employee(int id, String name) {

this.id = id;

this.name = name;

}

public int getId() {

return id;

}

public String getName() {

return name;

}

@Override

public int compareTo(Employee other) {

return Integer.compare(this.id, other.id); // Sorting by ID

}

@Override

public String toString() {

return "Employee{id=" + id + ", name='" + name + "'}";

}

}

public class ComparableExample {

public static void main(String[] args) {

List<Employee> employees = new ArrayList<>();

employees.add(new Employee(3, "Alice"));

employees.add(new Employee(1, "Bob"));

employees.add(new Employee(2, "Charlie"));

Collections.sort(employees); // Uses compareTo() method

System.out.println(employees);

}

}

**✅ Output:**

[Employee{id=1, name='Bob'}, Employee{id=2, name='Charlie'}, Employee{id=3, name='Alice'}]

✔ **Sorting by id (natural ordering).**  
✔ No need to pass a Comparator.

**2️⃣ Comparator (Multiple Sorting Criteria)**

* If you need **multiple sorting criteria**, use Comparator.
* This is useful when sorting by **different fields dynamically** (e.g., name, salary).

**Example: Sorting Employees by Name and ID**

import java.util.\*;

class EmployeeComparator {

// Sorting by Name

public static final Comparator<Employee> BY\_NAME = Comparator.comparing(Employee::getName);

// Sorting by ID

public static final Comparator<Employee> BY\_ID = Comparator.comparingInt(Employee::getId);

}

public class ComparatorExample {

public static void main(String[] args) {

List<Employee> employees = new ArrayList<>();

employees.add(new Employee(3, "Charlie"));

employees.add(new Employee(1, "Alice"));

employees.add(new Employee(2, "Bob"));

// Sorting by Name

employees.sort(EmployeeComparator.BY\_NAME);

System.out.println("Sorted by Name: " + employees);

// Sorting by ID

employees.sort(EmployeeComparator.BY\_ID);

System.out.println("Sorted by ID: " + employees);

}

}

**✅ Output:**

Sorted by Name: [Employee{id=1, name='Alice'}, Employee{id=2, name='Bob'}, Employee{id=3, name='Charlie'}]

Sorted by ID: [Employee{id=1, name='Alice'}, Employee{id=2, name='Bob'}, Employee{id=3, name='Charlie'}]

✔ **Sorting by name and id dynamically.**  
✔ No need to modify the Employee class.

**3️⃣ When to Use Which?**

| **Use Case** | **Choose** |
| --- | --- |
| When objects have a **natural order** (e.g., id, name) | Comparable |
| When sorting logic is **part of the class** | Comparable |
| When sorting by **multiple fields** (e.g., name, salary) | Comparator |
| When sorting logic should be **separate from the class** | Comparator |
| When using **lambda expressions** for sorting | Comparator |

**🔥 Conclusion**

✔ Comparable is best for **default (natural) ordering** inside a class.  
✔ Comparator is best when you **need flexible sorting logic** for multiple fields.  
✔ Use Comparator with **lambda expressions** for cleaner code.

**Comparable vs. Comparator in Java 8 (with Lambda Expressions)**

Java 8 introduced **lambda expressions** and **method references**, making it easier to implement Comparator without writing separate classes.

**1️⃣ Comparable in Java 8 (Natural Ordering)**

* Comparable still follows the same principle but benefits from **default methods** in interfaces.

**Example: Sorting Employees by ID**

import java.util.\*;

class Employee implements Comparable<Employee> {

private int id;

private String name;

public Employee(int id, String name) {

this.id = id;

this.name = name;

}

public int getId() {

return id;

}

public String getName() {

return name;

}

@Override

public int compareTo(Employee other) {

return Integer.compare(this.id, other.id); // Sorting by ID

}

@Override

public String toString() {

return "Employee{id=" + id + ", name='" + name + "'}";

}

}

public class ComparableJava8 {

public static void main(String[] args) {

List<Employee> employees = new ArrayList<>();

employees.add(new Employee(3, "Alice"));

employees.add(new Employee(1, "Bob"));

employees.add(new Employee(2, "Charlie"));

// Java 8 - Collections.sort() is still supported

Collections.sort(employees);

// Java 8 - Using List.sort()

employees.sort(Comparator.naturalOrder());

System.out.println(employees);

}

}

**✅ Output:**

[Employee{id=1, name='Bob'}, Employee{id=2, name='Charlie'}, Employee{id=3, name='Alice'}]

✔ Uses **natural order (id)**.  
✔ Java 8 introduced Comparator.naturalOrder() for lists.

**2️⃣ Comparator in Java 8 (Custom Sorting with Lambdas)**

* Instead of creating separate comparator classes, Java 8 lets us use **lambda expressions** and **method references**.

**Example: Sorting Employees by Name and ID**

import java.util.\*;

public class ComparatorJava8 {

public static void main(String[] args) {

List<Employee> employees = new ArrayList<>();

employees.add(new Employee(3, "Charlie"));

employees.add(new Employee(1, "Alice"));

employees.add(new Employee(2, "Bob"));

// ✅ Sorting by Name using Lambda

employees.sort((e1, e2) -> e1.getName().compareTo(e2.getName()));

System.out.println("Sorted by Name: " + employees);

// ✅ Sorting by Name using Method Reference

employees.sort(Comparator.comparing(Employee::getName));

System.out.println("Sorted by Name (Method Reference): " + employees);

// ✅ Sorting by ID using Comparator.comparingInt

employees.sort(Comparator.comparingInt(Employee::getId));

System.out.println("Sorted by ID: " + employees);

}

}

**✅ Output:**

Sorted by Name: [Employee{id=1, name='Alice'}, Employee{id=2, name='Bob'}, Employee{id=3, name='Charlie'}]

Sorted by Name (Method Reference): [Employee{id=1, name='Alice'}, Employee{id=2, name='Bob'}, Employee{id=3, name='Charlie'}]

Sorted by ID: [Employee{id=1, name='Alice'}, Employee{id=2, name='Bob'}, Employee{id=3, name='Charlie'}]

✔ **Lambda expressions** simplify custom sorting.  
✔ **Method references** (Comparator.comparing(Employee::getName)) make code cleaner.  
✔ Comparator.comparingInt() is optimized for primitive int fields.

**3️⃣ Sorting in Reverse Order (Java 8 Enhancements)**

* **Reversing Order** is now simpler using reversed().

**Example: Reverse Sorting**

employees.sort(Comparator.comparing(Employee::getName).reversed());

System.out.println("Sorted by Name (Descending): " + employees);

✔ No need for extra logic in compareTo() or compare().

**4️⃣ Sorting by Multiple Fields (Chaining in Java 8)**

* You can **chain multiple comparators** using .thenComparing().

**Example: Sort by Name, then by ID**

employees.sort(Comparator.comparing(Employee::getName).thenComparing(Employee::getId));

✔ If two employees have the same name, they are sorted by ID.

**🔥 Final Summary**

| **Feature** | **Java 7 Approach** | **Java 8 Improvement** |
| --- | --- | --- |
| **Sorting by Comparable** | Collections.sort(list) | list.sort(Comparator.naturalOrder()) |
| **Sorting by Comparator** | Implement Comparator class | Use **Lambdas / Method References** |
| **Reverse Sorting** | Implement compare() manually | .reversed() on Comparator |
| **Multiple Sorting Criteria** | Write custom logic | .thenComparing() |