**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**

package comparable;  
  
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.List;  
  
public class Employee implements Comparable<Employee>{  
 private int id;  
 private String name;  
  
 public Employee(int id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 @Override  
 public String toString() {  
 return id +" "+ name;  
 }  
 @Override  
 public int compareTo(Employee emp) {  
 return Integer.*compare*(this.id, emp.id); //Sort Based on id  
 }  
  
 public static void main(String[] args){  
 List<Employee> mylist = new ArrayList<>();  
 mylist.add(new Employee(3, "John"));  
 mylist.add(new Employee(2, "Jane"));  
 mylist.add(new Employee(4, "Bob"));  
 mylist.add(new Employee(1, "Alice"));  
  
 for(Employee emps:mylist){  
 System.*out*.println( emps);  
 }  
  
 Collections.*sort*(mylist);  
  
 System.*out*.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");  
  
 for(Employee emps:mylist){  
 System.*out*.println( emps);  
 }  
 }  
  
  
}

**✅ Output:**

3 John

2 Jane

4 Bob

1 Alice

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1 Alice

2 Jane

3 John

4 Bob

✔ **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:**

**package comparator;  
  
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.Comparator;  
import java.util.List;  
  
public class Student {  
  
 private String id;  
 private String name;  
 private String phone;  
  
 public Student(String id, String name, String phone) {  
 this.id = id;  
 this.name = name;  
 this.phone = phone;  
 }  
  
 public String getId() {  
 return id;  
 }  
  
 public void setId(String id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getPhone() {  
 return phone;  
 }  
  
 public void setPhone(String phone) {  
 this.phone = phone;  
 }  
 @Override  
 public String toString() {  
 return id +" "+ name +" "+ phone;  
 }  
  
  
 static class StudentComparator implements Comparator<Student> {  
 @Override  
 public int compare(Student s1, Student s2) {  
  
  
// int idCompare = s1.getId().compareTo(s2.getId()); // Compare by ID  
// if (idCompare != 0)  
// return idCompare;  
  
  
  
// int nameCompare = s1.getName().compareTo(s2.getName()); // Compare by Name  
// if (nameCompare != 0)  
// return nameCompare;  
  
 return s1.getPhone().compareTo(s2.getPhone());// Compare by Phone  
 }  
 }  
 public static void main(String[] args){  
 List<Student> students = new ArrayList<>();  
 students.add(new Student("3", "Kaushik", "8709252029"));  
 students.add(new Student("1", "Rahul", "7609246514"));  
 students.add(new Student("4", "Akash", "9076842310"));  
 students.add(new Student("2", "Abhishek", "7904238719")); // Same ID, different phone  
  
 // Sort using custom comparator  
 Collections.*sort*(students, new StudentComparator());  
  
 // Print sorted list  
 for (Student student : students) {  
 System.*out*.println(student);  
 }  
 }  
  
  
}**

**✅ Output:**

**1 Rahul 7609246514**

**2 Abhishek 7904238719**

**3 Kaushik 8709252029**

**4 Akash 9076842310**

**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**

package java\_8comparable;  
  
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.Comparator;  
import java.util.List;  
  
public class Employee implements Comparable<Employee> {  
 private int id;  
 private String name;  
  
 public Employee(int id, String name) {  
 this.id = id;  
 this.name = 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 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**

package java\_8comparator;  
  
class Product {  
 int id;  
 String name;  
 double price;  
  
 public Product(int id, String name, double price) {  
 this.id = id;  
 this.name = name;  
 this.price = price;  
 }  
  
 @Override  
 public String toString() {  
 return "Product{id=" + id + ", name='" + name + "', price=" + price + '}';  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public double getPrice() {  
 return price;  
 }  
  
 public void setPrice(double price) {  
 this.price = price;  
 }  
}

package java\_8comparator;  
  
import java.util.ArrayList;  
import java.util.Comparator;  
import java.util.List;  
  
public class ComparatorLambdaExample {  
 public static void main(String[] args) {  
 List<Product> products = new ArrayList<>();  
 products.add(new Product(1, "Laptop", 80000));  
 products.add(new Product(2, "Mobile", 50000));  
 products.add(new Product(3, "Tablet", 30000));  
  
 // Sorting by price (Ascending)  
 products.sort(Comparator.*comparing*(Product::getPrice));  
 System.*out*.println("Sorted by Price (Ascending): " + products);  
  
 // Sorting by name (Descending)  
 products.sort(Comparator.*comparing*(Product::getName).reversed());  
 System.*out*.println("Sorted by Name (Descending): " + products);  
 }  
}

**✅ Output:**

Sorted by Price (Ascending): [Product{id=3, name='Tablet', price=30000.0}, Product{id=2, name='Mobile', price=50000.0}, Product{id=1, name='Laptop', price=80000.0}]

Sorted by Name (Descending): [Product{id=3, name='Tablet', price=30000.0}, Product{id=2, name='Mobile', price=50000.0}, Product{id=1, name='Laptop', price=80000.0}]

✔ **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() |