**Comparable and Comparator**

**We often need to compare two values in our Java programs. Comparing primitive values like int, char, float is very easy and can be done with comparison operators like <, >, == etc.**

But comparing objects is a little different. For example, how would you compare two Employees? how would you compare two Students?

You need to explicitly define how the objects of user defined classes should be compared. For this purpose, Java provides two interfaces called Comparable and Comparator.

Once you define how the objects should be compared using any of these interfaces, you’ll be able to sort them using various library functions like Collections.sort or Arrays.sort.

## **Java Comparable interface intuition**

By default, a user defined class is not comparable. That is, its objects can’t be compared. To make an object comparable, the class must implement the Comparable interface.

The Comparable interface has a single method called compareTo() that you need to implement in order to define how an object compares with the supplied object -

public interface Comparable<T> {

public int compareTo(T o);

}

When you define the compareTo() method in your classes, you need to make sure that the return value of this method is -

* negative, if this object is less than the supplied object.
* zero, if this object is equal to the supplied object.
* positive, if this object is greater than the supplied object.

Many predefined Java classes like String, Date, LocalDate, LocalDateTime etc implement the Comparable interface to define the ordering of their instances.

Let’s now see an example to make things more clear.

## Java Comparable interface Example

The example below shows how to implement the Comparable interface in a user defined class and define the compareTo() method to make the objects of that class comparable.

import java.time.LocalDate;

import java.util.Objects;

class Employee implements Comparable<Employee> {

private int id;

private String name;

private double salary;

private LocalDate joiningDate;

public Employee(int id, String name, double salary, LocalDate joiningDate) {

this.id = id;

this.name = name;

this.salary = salary;

this.joiningDate = joiningDate;

}

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 getSalary() {

return salary;

}

public void setSalary(double salary) {

this.salary = salary;

}

public LocalDate getJoiningDate() {

return joiningDate;

}

public void setJoiningDate(LocalDate joiningDate) {

this.joiningDate = joiningDate;

}

// Compare Two Employees based on their ID

/\*\*

\* @param anotherEmployee - The Employee to be compared.

\* @return A negative integer, zero, or a positive integer as this employee

\* is less than, equal to, or greater than the supplied employee object.

\*/

@Override

public int compareTo(Employee anotherEmployee) {

return this.getId() - anotherEmployee.getId();

}

@Override

public String toString() {

return "Employee{" +

"id=" + id +

", name='" + name + '\'' +

", salary=" + salary +

", joiningDate=" + joiningDate +

'}';

}

}

In the above example, we’re comparing two employees by their IDs.

We’re just returning this.getId() - anotherEmployee.getId() from the compareTo() function, which will be

* negative if the ID of this employee is less then the ID of the supplied employee,
* zero if the ID of this employee is equal to the ID of the supplied employee, and
* positive if the ID of this employee is greater than the ID of the supplied employee.

It’s just a concise way of writing the following -

public int compareTo(Employee anotherEmployee) {

if(this.getId() < anotherEmployee.getId()) {

return -1;

} else if (this.getId() > anotherEmployee.getId()) {

return 1;

} else {

return 0;

}

}

Let’s now see how the Employee objects can be sorted automatically by Collections.sort method -

import java.time.LocalDate;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class ComparableExample {

public static void main(String[] args) {

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

employees.add(new Employee(1010, "Rajeev", 100000.00, LocalDate.of(2010,7, 10)));

employees.add(new Employee(1004, "Chris", 95000.50, LocalDate.of(2017, 3, 19)));

employees.add(new Employee(1015, "David", 134000.00, LocalDate.of(2017, 9, 28)));

System.out.println("Employees (Before Sorting) : " + employees);

// This will use the `compareTo()` method of the `Employee` class to compare two employees and sort them.

Collections.sort(employees);

System.out.println("\nEmployees (After Sorting) : " + employees);

}

}

## **Java Comparator interface intuition**

The Comparable interface that we looked at in the previous section defines a default ordering for the objects of a class. This default ordering is also called the natural ordering of the objects.

But what if you need to alter the default ordering just for a single requirement? For example, what if you want to sort the Employee objects in the previous example based on their names, not IDs?

You can’t change the implementation of the compareTo() function because it will affect the ordering everywhere, not just for your particular requirement.

Also, If you’re dealing with a predefined Java class or a class defined in a third party library, you won’t be able to change the default ordering. For example, The default ordering of String objects is to order them alphabetically. But what if you want to order them based on their length?

For such cases, Java provides a Comparator interface. You can define a Comparator and pass it to the sorting functions like Collections.sort or Arrays.sort to sort the objects based on the ordering defined by the Comparator.

The Comparator interface contains a method called compare() that you need to implement in order to define the ordering of the objects of a class -

public interface Comparator<T> {

int compare(T o1, T o2);

}

The implementation of the compare() method should return

* a negative integer, if the first argument is less than the second,
* zero, if the first argument is equal to the second, and
* a positive integer, if the first argument is greater than the second.

Let’s see an example to make things clear.

## Java Comparator interface Example

Let’s see how to sort a collection of Employee objects that we defined in the previous section based on different fields by defining different Comparators.

import java.time.LocalDate;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class ComparatorExample {

public static void main(String[] args) {

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

employees.add(new Employee(1010, "Rajeev", 100000.00, LocalDate.of(2010, 7, 10)));

employees.add(new Employee(1004, "Chris", 95000.50, LocalDate.of(2017, 3, 19)));

employees.add(new Employee(1015, "David", 134000.00, LocalDate.of(2017, 9, 28)));

employees.add(new Employee(1009, "Steve", 100000.00, LocalDate.of(2016, 5, 18)));

System.out.println("Employees : " + employees);

// Sort employees by Name

Comparator<Employee> employeeNameComparator = new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return e1.getName().compareTo(e2.getName());

}

};

/\*

The above Comparator can also be written using lambda expression like so =>

employeeNameComparator = (e1, e2) -> e1.getName().compareTo(e2.getName());

Which can be shortened even further using Java 8 Comparator default method

employeeNameComparator = Comparator.comparing(Employee::getName)

\*/

Collections.sort(employees, employeeNameComparator);

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

// Sort employees by Salary

Comparator<Employee> employeeSalaryComparator = new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

if(e1.getSalary() < e2.getSalary()) {

return -1;

} else if (e1.getSalary() > e2.getSalary()) {

return 1;

} else {

return 0;

}

}

};

Collections.sort(employees, employeeSalaryComparator);

System.out.println("\nEmployees (Sorted by Salary) : " + employees);

// Sort employees by JoiningDate

Comparator<Employee> employeeJoiningDateComparator = new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return e1.getJoiningDate().compareTo(e2.getJoiningDate());

}

};

Collections.sort(employees, employeeJoiningDateComparator);

System.out.println("\nEmployees (Sorted by JoiningDate) : " + employees);

}

}

# Output

Employees : [Employee{id=1010, name='Rajeev', salary=100000.0, joiningDate=2010-07-10}, Employee{id=1004, name='Chris', salary=95000.5, joiningDate=2017-03-19}, Employee{id=1015, name='David', salary=134000.0, joiningDate=2017-09-28}, Employee{id=1009, name='Steve', salary=100000.0, joiningDate=2016-05-18}]

Employees (Sorted by Name) : [Employee{id=1004, name='Chris', salary=95000.5, joiningDate=2017-03-19}, Employee{id=1015, name='David', salary=134000.0, joiningDate=2017-09-28}, Employee{id=1010, name='Rajeev', salary=100000.0, joiningDate=2010-07-10}, Employee{id=1009, name='Steve', salary=100000.0, joiningDate=2016-05-18}]

Employees (Sorted by Salary) : [Employee{id=1004, name='Chris', salary=95000.5, joiningDate=2017-03-19}, Employee{id=1010, name='Rajeev', salary=100000.0, joiningDate=2010-07-10}, Employee{id=1009, name='Steve', salary=100000.0, joiningDate=2016-05-18}, Employee{id=1015, name='David', salary=134000.0, joiningDate=2017-09-28}]

Employees (Sorted by JoiningDate) : [Employee{id=1010, name='Rajeev', salary=100000.0, joiningDate=2010-07-10}, Employee{id=1009, name='Steve', salary=100000.0, joiningDate=2016-05-18}, Employee{id=1004, name='Chris', salary=95000.5, joiningDate=2017-03-19}, Employee{id=1015, name='David', salary=134000.0, joiningDate=2017-09-28}]

As you saw in the above example, All the sorting methods also accept an instance of a Comparator interface. They use the ordering defined by the Comparator interface’s compare() function to sort the objects.

## Using Java 8 Comparator default methods

The Comparator interface contains various default factory methods for creating Comparator instances.

All the Comparators that we created in the previous section can be made more concise by using these factory methods.

Here is the same Comparator example that we saw in the previous section using Java 8 Comparator default methods -

import java.time.LocalDate;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class ComparatorExample {

public static void main(String[] args) {

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

employees.add(new Employee(1010, "Rajeev", 100000.00, LocalDate.of(2010, 7, 10)));

employees.add(new Employee(1004, "Chris", 95000.50, LocalDate.of(2017, 3, 19)));

employees.add(new Employee(1015, "David", 134000.00, LocalDate.of(2017, 9, 28)));

employees.add(new Employee(1009, "Steve", 100000.00, LocalDate.of(2016, 5, 18)));

System.out.println("Employees : " + employees);

// Sort employees by Name

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

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

// Sort employees by Salary

Collections.sort(employees, Comparator.comparingDouble(Employee::getSalary));

System.out.println("\nEmployees (Sorted by Salary) : " + employees);

// Sort employees by JoiningDate

Collections.sort(employees, Comparator.comparing(Employee::getJoiningDate));

System.out.println("\nEmployees (Sorted by JoiningDate) : " + employees);

// Sort employees by Name in descending order

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

System.out.println("\nEmployees (Sorted by Name in descending order) : " + employees);

// Chaining multiple Comparators

// Sort by Salary. If Salary is same then sort by Name

Collections.sort(employees, Comparator.comparingDouble(Employee::getSalary).thenComparing(Employee::getName));

System.out.println("\nEmployees (Sorted by Salary and Name) : " + employees);

}

}