Sorting is a very common procedure in our daily lives. We sort marks on the basic of their weight i.e. how high the number is. We sort students on the basic of their names in ascending or descending order. So sorting is pretty common.

Though some of these sorting, mainly the ones depending upon numbers and alphabets, are easy and straightforward. Some can be indirect like sorting of vases in our house on the basis of their material, their color or their height etc.

Similarly in Java, when we have to sort data in **Integer** or **String** its easy. For integers , we can make use of <,> or = etc operators to determine the order of inputs in sort and for Strings their alphabetical natural order would do.

String a=”Adarsh”;

String b=”Nishant”;

If(a.compareTo(b)>0)

{

S.O.P.(a+“ comes before ”+b+” in sorting order”);

}

SO you see its easy and straigt forward. But how?

That is because String class implements an interface called **Comparable.** This is a mandatory interface for any class you want to sort. Like we wanted the content inside the String class to sort so we used compareTo() and the reason we were able to do that was because it implements Comparable.

Now, Collection**s** utility class gives us methods like sort(List<> list) which will sort the input list.

So if the list contains string or Integer no problem they will be sorted just fine. But what is you have a class Student like below:

**package** com.collectinons;

**public** **class** Student{

**private** String name;

**private** **int** age;

//setters getters

}

And then you want the sorting of a list that contains this Student object.

public class Main {

public static void main(String[] args) {

List<Student> students=**new** ArrayList<>();

students.add(student1);

students.add(student2);

Student student1=**new** Student();

student1.setAge(20);

student1.setName("adarsh");

Student student2=**new** Student();

student2.setAge(1);

student2.setName("zoni");

Collections.*sort*(students);

}

}

How will Collections class know how to sort a list with objects which are defined by you, a developer. So it will throw an exception in this case.

For it to be able to sort it, the data inside the list must implements the ***Comparable*** interface and implement the compareTo(Object obj) method like:

**package** com.collectinons;

**public** **class** Student **implements** Comparable<Student>

{

**private** String name;

**private** **int** age;

//setter and getter

@Override

**public** **int** compareTo(Student arg0) {

**return** ((Integer)**this**.getAge()).compareTo((Integer)arg0.getAge());

}

}

Now inside the compareTo() we can give the criteria on the basis of which the comparision will take place.

Same happens when you have a Tree Data structure, be it a TreeSet or TreeMap or anything. We know that Tree data structures automatically sorts the objects inside it in natural order. This natural order is nothing but the order defined here in compareTo() method. So if your class is not implementing Comparable interface and you are putting objects of this class inside a Tree data structure, you will get an exception.

Now Comparabale is pretty good in defining a standard and natural order of sorting for your own class or for any user defined class. But lets say you defined the compareTo() method in Student class to sort according to the age here. And that is good because that is how it is required in most of the places in your use case.

But what if just in one place you are asked to sort the list with Student objects according to their name and not age? It will be a very bad practice to go and change the implementation of the compareTo() method just for this one scenario against the other many scenario of age.

So for these kind of situations **Comparator** anonymous class comes into picture. It is responsible for overriding the sorting order of data inside a list dynamically on the go without having to change the actual implementation inside the object Class.

Collections.*sort*(students,**new** Comparator<Student>() {

@Override

**public** **int** compare(Student o1, Student o2) {

**return** o1.getName().compareTo(o2.getName());

}

});

SO in above scenario, we have passed the students list and as a second parameter used comparator which overrides the compare() method and defines the algorithim for comparing two objects inside the passed list. This will override the algo written in the Student class itself.

https://www.youtube.com/watch?v=JSvVsOm4oX0

Comparable and comparator are two interfaces provided by Java for comparing objects and sorting them.

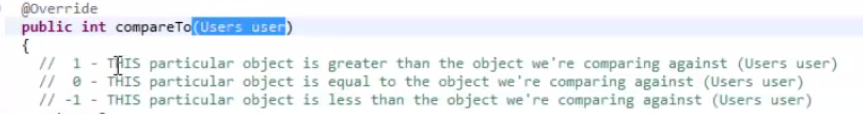
We use Collections.sort(). It has two variants:

* Collections.sort(List<T>)
* Collections.sort(List<T>, Comparator<?superT>)

# Collections.sort(List<T>):

Now the first one takes the list argument which contains the data to be sorted. T is the type of data tat the list hold and will be sorted. Now in this one, the type T must implement Comparable<Object> interface and implement compareTo(Object o) method. Now our implementation of the class to be compared i.e. T will go inside here.

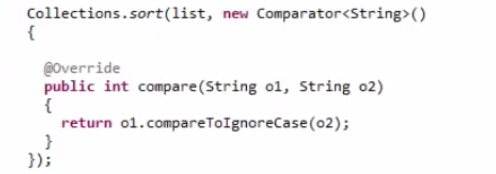
So this one associated with Comparable interface.



Comparable is more used for Objects made by programmer and when he wants to do comparing and sorting of those objects putting them inside a collection in the way defined by him.

# Collections.sort(List<T>,Comparator<?superT>):

For cases when we want to alter the implementation of compareTo of Java Defined objects like String we use comparator.



So here what we do is we, in the second parameter create an anonymous comparator class and override the compare method passing in the two objects to compare. And now define how will the two objects in consideration will be compared at any given point of time.

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| 1) Comparable provides **single sorting sequence**. In other words, we can sort the collection on the basis of single element such as id or name or price etc. | Comparator provides **multiple sorting sequence**. In other words, we can sort the collection on the basis of multiple elements such as id, name and price etc. |
| 2) Comparable **affects the original class** i.e. actual class is modified. | Comparator **doesn't affect the original class** i.e. actual class is not modified. |
| 3) Comparable provides **compareTo() method** to sort elements. | Comparator provides **compare() method** to sort elements. |
| 4) Comparable is found in **java.lang** package. | Comparator is found in **java.util** package. |
| 5) We can sort the list elements of Comparable type by **Collections.sort(List)** method. | We can sort the list elements of Comparator type by **Collections.sort(List,Comparator)** method. |