# Module 06 Team Activity - Comparators and Sorting

In this Team Activity, you will explore using Comparable interface to sort objects in Java, along with specialized sorts using streams. You will also explore how to use the `Collections.sort()` method to sort objects in a collection. We will also explore the use of Sorted Collections in java such as the `TreeSet` and `TreeMap` classes.

## Grading

Grades for team activities will be based on attendance and notes. You must attend, and as a team you need to generate notes that we can confirm your work. Ideally, you upload the notes as a PDF to the team meeting after you build them out.

> Tip
>
> Good notes become a study guide for you and your team! Make sure they include everything you need to help better understand the weekly material.

### ⭐ Working in Teams ⭐

When working in teams, remember do not let one person do all the work. Make sure to work together, and ask questions. It is also better if different people program, and you all take turns programming for various team assignments.

## Learning Objectives

This team activity is designed to help you understand the following concepts:

- Implement the `Comparable` interface to sort objects in a collection
- Sort data using the `Collections.sort()` method and sorted() method in streams
- Sort data using lambda expressions
- Use the `TreeSet` and `TreeMap` classes to keep data presorted in a collection
- Identify the benefits of a Strategy Pattern for sorting data

## Comparators and Sorting

In Java, there is the `Comparable` interface, that any object can implement. If that is implemented, then it is possible to sort and order objects as they are added to a collection. The `Comparable` interface has one method, `compareTo()` , that is used to compare two objects. The `compareTo()` method returns a negative integer, zero, or a positive integer if the object is less than, equal to, or greater than the specified object.

The question then becomes, what does this comparison mean.

Let's look at String (which implements Comparable). Given the following example, what do you think the output will be?

```
String s1 = "aloha";
String s2 = "world";


System.out.println(s1.compareTo(s2));
```

Now, let's try something a bit harder. What do you think the output will be? You just need to say greater than 0, less than 0, or zero!

```
String a = "a";
String b = "b";
String A = "A";
String B = "B";

System.out.println(a.compareTo(b));
System.out.println(a.compareTo(a));
System.out.println(b.compareTo(a));

System.out.println();
System.out.println(A.compareTo(B));
System.out.println(B.compareTo(A));

System.out.println();
System.out.println(a.compareTo(A));
System.out.println(A.compareTo(a));
```

Now try running the code. The example is in the solutions/StringComparableExamples.java file, or you can write your own and copy/paste it.

### 👉🏽 Discussion

What were some surprises? Why do you think 'a' is greater than 'A'?

> ⚠ Warning
>
> A common mistake in code is using comparable assuming the results are -1, 0, 1. As you can see from the example, less than 0, 0, greater than 0 is the correct answer. That is because java leaves it up to the implementation. In the case of strings, they convert it to the int value of the characters and subtract them, which can be negative, zero, or positive.

## Practicing on a Custom Class

Now, let's try implementing the `Comparable` interface on a custom class. Assume you are building a class to represent an airline flight. The class has the following attributes:

- `String airline` (United, Delta, etc.)
- `int flightNumber` (123, 456, etc.)
- `String airlineCode` (UA, DL, etc.)

The final flight ID is a combination of the airline code and the flight number. For example, United flight 123 would have a flight ID of UA123. You want to sort by flight ID, as that groups by airlines then by numbers.

```java
public class Flight implements Comparable<Flight> {
    private String airline;
    private int flightNumber;
    private String airlineCode;

    public Flight(String airline, int flightNumber, String airlineCode) {
        this.airline = airline;
        this.flightNumber = flightNumber;
        this.airlineCode = airlineCode;
    }

    public int getFlightNumber() {
        return flightNumber;
    }

    public String getFlightID() {
        return airlineCode + flightNumber;
    }

    @Override
    public int compareTo(Flight o) {
        throw new UnsupportedOperationException("Not implemented yet");
    }

    @Override
    public String toString() {
        return "Flight{" +
                "airline='" + airline + '\'' +
                ", flightNumber=" + flightNumber +
                ", airlineCode='" + airlineCode + '\'' +
                '}';
    }
}
```

👉🏽 Discuss the provided code. Does anything stand out? Notice the Comparable uses a generic, what is that? What do you think the `compareTo()` method should look like?

🔥 As a team, implement the `compareTo()` method to sort by flight ID. You can also find the template in Flight.java .

## Sorting Collections

To sort a collection of objects you can use a variety of methods:

- `Collections.sort(list)` - This is a static method in the Collections class that takes a list and sorts it in place.
- `list.sort(null)` - This is part of every list, if null is passed in for the parameter, it will sort the list in place using the compareTo() method.
  - You can also pass in a different type of Comparable to sort by a different method.

- `list.stream().sorted()` - This is a stream method that will sort the list and return a new list. It does not modify the original list!

Looking at the options above, let's focus on `list.sort(null)`. In the Flight class, you can sort a list of flights by calling `flights.sort(null)`, this could make your main method look like this:

```
Flight f1 = new Flight("Delta", 123, "DL");
Flight f2 = new Flight("American", 456, "AA");
Flight f3 = new Flight("United", 789, "UA");

System.out.println(f1.compareTo(f2));
System.out.println(f2.compareTo(f3));
System.out.println(f3.compareTo(f1));

List<Flight> flights = new ArrayList<>();
// can't use list.of() only, as it returns an immutable list, so we add one to new lis
flights.addAll(List.of(f1, f2, f3));


System.out.println(flights);

flights.sort(null);
System.out.println(flights)
```

If you haven't already, run the main method (adding those lines if you don't have them), and discuss the following:

- What does in place mean?
- Do you have the original order of the list?
- Pre and post sort, what is the value of flights[0]?

## Using Lambda Expressions

Sometimes you need to change the sort, and the `compareTo()` method is not enough. In Java 8, they introduced lambda expressions, which can be used to provide another sort option beyond what is already implemented in compareTo().

> Important
>
> Lambda expression are meant to be small and concise. They are not meant to be full methods, but rather a quick way to implement an interface with one method. If you find yourself writing something long, it is best to create a proper Comparable class. You can learn more about them here - though the examples provide would have made good lambda expressions.

If we wanted to sort by flight number only, we could look at the following for a lambda expression:

```
flights.sort((f1, f2) -> f1.getFlightNumber() - f2.getFlightNumber());
```

Breaking down the expression it is saying:

- flights.sort() - sort the list
- (f1, f2) - these are the two flights to compare (parameters to pass in)
- `->` - this is the lambda operator, it separates the parameters from the code
- f1.getFlightNumber() - f2.getFlightNumber() - this is the code to compare the two flights

🔥 As a team, implement a lambda expression to sort by airline code, and then see how the print out changes.

## Keeping Items Sorted

In Java, there are two classes that keep items sorted, `TreeSet` and `TreeMap` . Often, it is beneficial to keep items sorted as they are added, and these classes do that. As per the name, Set keeps a unique set of items sorted, and Map keeps a unique set of keys sorted for then accessing the values.

👉🏽 Discussion - Thinking back to the last module, was there any discernable order of HashSet and HashMap(keys) when you printed out the contents? This is because it is often a lot quicker to just store and retrieve items if order isn't important.

### TreeSet

The `TreeSet` class is a set that is sorted. It uses the `compareTo()` method to sort the items. If you add an item to a `TreeSet` , it will be sorted as it is added.

🔥 You also will want to implement .equals and .hashCode methods in the Flight class. This is because the TreeSet uses these methods to determine if an object is already in the set. There are some general rules to follow:

- If two objects are equal, they should have the same hash code, but not all objects need to have unique hash codes.
- If two objects are equal, they should also return 0 from the compareTo() method.

It is up to the programmer to ensure these rules are followed, and unfortunately, not following these rules can lead to some unusual errors.

**Assumption**: When implementing the `equals()` method, you can assume that flightID (combination of airline code and flight number) is unique, and you are free to use the .equals and .hashCode methods of the String class. - Don't recreate the wheel!

Now that we have the Flight class ready, let's see how to use a TreeSet:

```
Set<FlightSolution> flightSet = new TreeSet<>(flights);
System.out.println(flightSet);
```

🔥 Add the above code to your main function, making changes as needed.

What was printed? Did we have to call sort first? Why or why not?

👉🏽 Discussion - What is a major difference between a Set and List? How can this limitation cause issues if you are trying to keep items sorted?

> ⓘ Note
>
> There is often a question to why Java doesn't have a Sorted List. The answer has to do with the definition of a List. A List is an ordered collection, and the order is based on the order the items are added. If you change how it is being added, you are breaking the design contract of the List Interface. You can read more of this discussion at "Why is There No Sorted List in Java?"

Another advantage of the TreeSet is that accessing the values are very fast. Let's say you constantly need to access a flight by the flightID, the time to find it $O(log n)$, which is much faster than $O(n)$ for a list. Thus keeping a sorted set can be very beneficial, if you don't have a reason for duplicate entries. If you have duplicate entries, you are back to a list or some other structure.

## TreeMap

A TreeMap is a map that is sorted by the keys. It is similar to the HashMap, but the keys are kept in a specific order. As with the TreeSet, you need to make sure you implement the `compareTo` method in the key class, in addition to `.equals` and `.hashCode` .

```
TreeMap<Flight, Integer> flightSeats = new TreeMap<>();

// assuming F1, F2, F3 are already created
flightSeats.put(f1, 100);
flightSeats.put(f2, 200);
flightSeats.put(f3, 300);

System.out.println(flightSeats);
```

🔥 Add the above code to your main function, making changes as needed.

👉🏽 Discussion - What was printed? Did we have to call sort first? Why or why not?

Another common case is for the key to be a string of the unique identifier, and the value to be the object. This is a common pattern, and you will see it often. In the case of flight, it could be `TreeMap<String, Flight>` where String is the flightID.

## Sorting Strategy Pattern

A common question is that there are a lot of potential attributes to sort against, what is the best way to handle this? One way is to use the Strategy Pattern. The Strategy Pattern is a design pattern that allows you to define a family of algorithms, encapsulate each one, and make them interchangeable.

That is a lot of big words, but let's think about in terms of sorting. Your compareTo() method can be the 'default' case, and only often implemented if there is a uniqueID of some sort in the class (unique way to identify the object). It is often called the "natural order". If you want to sort by other attributes, you can create a new class that implements the `Comparator` interface, *for each* attribute you want to sort. This means you may have multiple classes that implement the `Comparator` interface, all focusing on the type of sorting strategy you want to use.

The solutions folder has an example of this using the Integer class. By default, the Integer class sorts based on the value of the integer. However, solutions/IntegerSortStrategy.java has a number of classes that implement the `Comparator` interface to sort a list of Integers in different ways. We then test that strategy in the solutions/StrategyTester.java file.

### 👉🏽 Discussion

What are some things you notice about the code as a group? Any questions come to mind? How can this Strategy Pattern be useful? Especially when it comes to sorting?

We will cover the Strategy Pattern in more detail in a future module, but we wanted to introduce it here as it is a common pattern used in Java.

## 🔥 Java Practice Problem

As part of **every** team activity, we will ask you to work on a Java Practice problem, and submit the code to the team files section (or as part of your notes). This is meant to give you practice similar to technical interviews, but also help build up your java skills. **Each team member needs to select a different problem!** But you can share/and should share answers and help each other. Remember, to learn a new language, the best thing you can do is practice! Here are some resources to find practice problems but you are not limited to them:

- CodeHS - Java Practice
- Coding Bat - Java
- Hacker Rank - Java(Basic)