

***\*\* As a reminder the activity itself was carried out through Vocareum. Vocareum is a cloud-based lab environment in which students submit programming assignments and completed programming tests related to this course, Data Structures.***

## Hands-On: Total Order

### Total Order

If a *total order* exists on a collection of values, it means that these values can be arranged in ascending (or descending) order. That is, any value in the collection can be compared to any other value in the collection on the basis of “less than or equal to”. Here are two links that provide more formal definitions of a total order

- [http://en.wikipedia.org/wiki/Total\\_order](http://en.wikipedia.org/wiki/Total_order)
- <http://mathworld.wolfram.com/TotallyOrderedSet.html>

If we want to perform operations such as sorting, finding the minimum or maximum, etc. on a collection of values, the values in that collection must have a total order defined on them or otherwise these operations don’t even make sense. Java provides two standard ways of defining a total order on a data class: the Comparable interface or the Comparator interface.

While Comparable and Comparator serve the same purpose (defining a total order on a data class), the context of their use is very different. The Comparable interface is implemented by a data class, and thus is used in the context of one instance of that data class comparing *itself* to another instance of that data class. For example, in the notes, class Book implemented the Comparable interface. So, if b1 and b2 are instances of Book, then we could ask b1 to compare itself to b2 based on the natural order of Book like so: b1.compareTo(b2).

In contrast, the Comparator interface is **not** implemented by the data class. Instead, a “third party” class is used to implement Comparator for a given data class. Again referring to the Book example from the notes, we had a class defined like so: public class CompareBooksByTitle implements Comparator<Book>. An instance of this class, say bcomp, can compare two Book objects on the basis of the particular total order defined by this class like so: bcomp.compare(b1, b1). Note that the data class (Book) is completely “unaware” of this other total order that CompareBooksByTitle defines. Also note that it is the ability to define multiple “third party” total orders that allowed us to write a single search method that can compare items according to totally different criteria from one call to the next.





The following sections will give you practice in implementing and using both the Comparable and Comparator interfaces.

### Comparable

The standard way to require that objects of a given class can be ordered is to have that class implement the Comparable interface. The implementation of Comparable defines the **natural**

**ordering** of a class, since it is the class itself that decides what the order is. Read about the Comparable interface here:

- <https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/Comparable.html>

1. Open [Student.java](#) in jGRASP and compile it.
2. Implement the compareTo method according to the commented description in the source code. The current (stub) version of compareTo is shown below.
3. **@Override**
4. **public int compareTo(Student s) {**
5. **return -99;**
6. **}**
7. Open [ClassRoll.java](#) in jGRASP and compile it.
8. Use ClassRoll.java to informally test your implementation of the natural order of the Student class. Note that you do not have to address the shuffling or Comparator aspect of the ClassRoll class yet. Just run the file as-is.
9. Use the provided jGRASP Canvas file ([ClassRoll.jgrasp\\_canvas.xml](#)) to watch the sort by natural order happen as you single-step in debug mode. Once you save the jGRASP Canvas XML file to the same directory as the source code, you can use the jGRASP Canvas like so:
  - Click on the Canvas file in the jGRASP Browse tab or click on the *Run in Canvas* button  in the jGRASP tool bar.
  - Once the Canvas window opens, resize it so that it best fits your display.
  - You can use any of the following controls to watch the program execute.
    - *Play*  - Starts the program running in auto-step mode.
    - *Step Over*  - Manually steps over each statement.
    - *Step In*  - Manually steps into each statement (method call).

## Comparator

Java provides the Comparator interface as the standard way of defining a total order on a class that is different from its natural order. Read about the Comparator interface here:

- <https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Comparator.html>

1. Open [ClassRoll.java](#) in jGRASP and compile it.
2. Implement the CompareStudentsBySection comparator according to the commented description in the source code. The current (stub) version is shown below.

3.     **private static class CompareStudentsBySection implements Comparator<Student>**  
      **{**
4.         */\*\* Compares s1 to s2 in with respect to the defined total order. \*/*
5.         **public int compare(Student s1, Student s2) {**
6.             **return -99;**
7.         **}**
8.     **}**
9.   Compile and run ClassRoll.java to informally test your solution.
10. Use the provided jGRASP Canvas file ([ClassRoll.jgrasp\\_canvas.xml](#)) to watch the sort by comparator happen as you single-step in debug mode.
11. Respond to the prompts at the points specified in the source code comments to shuffle the roll and sort it in descending order of section. Try out your solutions by running the ClassRoll program. You may want to read the API for Collections:
  - <https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Collections.html>
12. Use the provided jGRASP Canvas file ([ClassRoll.jgrasp\\_canvas.xml](#)) to watch the shuffling and reverse order sort happen as you single-step in debug mode.

### Submission

The submission page for this activity asks you to submit both Student.java and ClassRoll.java for a grade.