comparable # java # lab # tcss143

Lab 8 – Comparable Objects

School of Engineering and Technology, University of Washington Tacoma
TCSS 143 Computer Programming II, Spring 2026

Due Date

Thursday, May 21, 2026, end of lab session

Overview

In this lab you will practice making your own classes **sortable** by implementing the `Comparable<T>` interface. You will define what it means for one object of your class to come "before" or "after" another, and then use Java's built-in sorting tools to put a list of those objects in order.

Learning Objectives

By completing this lab, you will:

- Implement the `Comparable<T>` interface on a class you write.
- Define a **natural ordering** for objects of that class by writing `compareTo`.
- Sort a `List` of your objects using `Collections.sort` and verify the result.
- Recognize when ascending vs. descending natural order is appropriate.

Before You Begin

Make sure you have completed:

- Reviewed lecture notes and examples on the `Comparable<T>` interface.
- Reviewed the `compareTo` contract: returns a **negative**, **zero**, or **positive** `int`.

Java API Reference

`java.lang.Comparable<T>` (Java SE 17)

Setup

1. Open IntelliJ IDEA and create a new Java project (or open the project you have been using for lab work).
2. Inside the `src` folder, create a package named `lab8`.
3. Create the class files described in each exercise inside the `lab8` package.
4. Create a `Main` class with a `main` method to run and test your code.

Exercises

Exercise 1: A Sortable `Book` Class

Create a class `Book` with the following fields:

- `private String title`
- `private String author`
- `private int pageCount`

Add:

- A constructor that takes all three fields.
- Getter methods for each field.
- A `toString()` method that returns the book in the form: `"<title> by <author> (<pageCount> pages)"`.

Make `Book` implement `Comparable<Book>` so that the **natural ordering** of books is **alphabetical by title** (case-insensitive).

`compareTo` contract reminder:

Return value	Meaning
Negative `int`	`this` comes **before** the other book

Return value	Meaning
Zero	`this` and the other book are considered **equal in order**
Positive `int`	`this` comes **after** the other book

> **Hint**
>
> You don't have to invent the comparison logic for strings — `String` already implements `Comparable<String>`. Look at `String.compareToIgnoreCase`.

> **Capture Your Work**
>
> Once `Book` compiles, take a screenshot of the file in your IDE.

Exercise 2: Sort a List of Books

In your `Main` class, create a `List<Book>` containing **at least five** books with varied titles (mix the casing on purpose — e.g., `"the hobbit"`, `"Dune"`, `"animal farm"`).

Then:

1. Print the list **before** sorting (one book per line, using `toString`).
2. Sort the list using `Collections.sort(...)`.
3. Print the list **after** sorting.

Expected behavior:

After the call to `Collections.sort`, the books should appear in case-insensitive alphabetical order by title, regardless of how you typed them when creating the list.

> **Hint**
>
> `Collections.sort(list)` works because `Book` is `Comparable<Book>`. You do not have to pass a separate comparator — the natural ordering you defined in Exercise 1 is what `sort` uses.

Capture Your Work

Run `Main` and take a screenshot of the console output showing both the unsorted and sorted lists.

Exercise 3: A `Student` Class with Descending Natural Order

Create a class `Student` with:

- `private String name`
- `private double gpa`

Add a constructor, getters, and a `toString()` that returns: `"<name> (GPA: <gpa>)"`.

Make `Student` implement `Comparable<Student>` so that the natural ordering puts students with **higher GPAs first**. In other words, after sorting, the top student in the list should be the one with the highest GPA.

Examples:

Given:

```
List<Student> roster = new ArrayList<>();
roster.add(new Student("Avery", 3.20));
roster.add(new Student("Blake", 3.95));
roster.add(new Student("Casey", 2.85));
Collections.sort(roster);
```

After sorting, printing `roster` should produce:

```
Blake (GPA: 3.95)
Avery (GPA: 3.2)
Casey (GPA: 2.85)
```

Hint

The `compareTo` contract doesn't care what "before" means — only your method does. If you want larger GPAs to come first, you decide which `Student` returns the negative value when comparing.

Watch Out

`gpa` is a `double`. Returning `(int) (this.gpa - other.gpa)` will round `0.1` differences to `0` and break your ordering. Use `Double.compare(...)` instead — and think about which argument goes first to get descending order.

Capture Your Work

Run your test code and take a screenshot of the console output showing students sorted from highest GPA to lowest.

Submission

Submit the following to Canvas before the end of lab:

- Your `Book.java`, `Student.java`, and `Main.java` files (zipped, or pushed to your lab repo).
- The three screenshots requested above.

Lab is graded on completion. As long as each exercise compiles, runs, and produces the expected behavior, you receive full credit.

Reading Reference

Reference	Description
`java.lang.Comparable<T>`	Official Java API for the `Comparable` interface.
`String.compareToIgnoreCase`	Case-insensitive string comparison used in Exercise 1.
`Double.compare`	Safe comparison of `double` values used in Exercise 3.
`Collections.sort`	Sorts a `List` whose elements are `Comparable`.