Section 0.5: Comparable + Additional Resources

Section Problem Downloads

Download starter code:

Section 0.5_Review.zip

After downloading:

1. Unzip the file by double-clicking on it.
2. Open the folder within VS Code.
3. Attempt the problems!

Suppose we had written the following buggy implementation for `compareTo` within the `Student` class:

```
public class Student implements Comparable<Student> {
    private String name;
    private double gpa;

    public Student(String name, double gpa) {
        this.name = name;
        this.gpa = gpa;
    }

    @Override
    public int compareTo(Student other) {
        // Orders first by gpa (high to low), and then by name (lexicographic)
        double gpaCompare = other.gpa - this.gpa;
        if (gpaCompare == 0.0) {
            return this.name.compareTo(other.name);
        }
        return ((int)gpaCompare);
    }
}
```

And suppose we created the following instances of the `Student` class:

```
Student one = new Student("a", 3.9);
Student two = new Student("b", 3.3);
Student three = new Student("c", 2.9);
Student four = new Student("c", 2.9);
```

Which one of the following tests would reveal the bug present within our solution?

- `assertTrue(one.compareTo(two) < 0);`
- `assertTrue(one.compareTo(three) < 0);`
- `assertTrue(three.compareTo(four) == 0);`
- `assertTrue(three.compareTo(one) > 0);`

Comparable: Location [Simple Programming Problem]

Problem Description

Consider a class `Location` that stores information about places as a pair of latitude-longitude coordinates. Each location keeps track of its name (a string), its latitude (a real number), and its longitude (a real number).

Modify the class to be `Comparable` by adding an appropriate `compareTo` method. Locations should be ordered first by latitude with locations closer to the equator (closer to 0) considered less than locations farther from the equator. When the latitudes are equal, you should examine longitudes with locations closer to the prime meridian (closer to 0) considered less than locations farther from the prime meridian.

You may assume that your constructor is passed legal values for latitude and longitude. Use the `Math.abs` method to find the absolute value of a number.

Testing Your Code

Press `Run` to run the given `LocationClient.java` code's `main` method. Press `Mark` to run our JUnit tests. `LocationTest.java` is also available to be seen/edited - simply add test cases to the given code and hit 'Check' **(not Mark)** to see your code run on your new test suite!

⭐ Comparable: MovieRating [Advanced Programming Problem]

Problem Description

Consider the `MovieRating` class that keeps track of the ratings for a movie. Modify the class to be `Comparable` by adding an appropriate `compareTo` method.

In general, movies with lower `averageRating()` are considered "less" than other movies. Thus, movies with lower average ratings should appear at the beginning of a sorted list. Movies that have no ratings are considered "less" than all rated movies. If two movies both have not been rated, they are considered "equal". If both movies have been rated and have exactly the same average rating, break ties by considering the one with more `numRatings` to be "greater". If these are still the same, lastly break ties by comparing the `movieTitle()` with movie titles that come alphabetically first being considered "less"/sorted before.

Your method should not modify any movies's state. You may assume the parameter passed is not null.

Testing Your Code

Press `Run` to run the given `MovieRatingClient.java` code's `main` method. Press `Mark` to run our JUnit tests. `LocationTest.java` is also available to be seen/edited - simply add test cases to the given code and hit 'Check' **(not Mark)** to see your code run on your new test suite!

Comparable: Movie Ratings Solution Explanation

Solution:

```
30      public int compareTo(MovieRating other) {
31          int cmp = 0;
32          double thisAvgRating = this.averageRating();
33          double otherAvgRating = other.averageRating();
34          if (thisAvgRating != otherAvgRating) {
35              if (thisAvgRating < otherAvgRating) {
36                  cmp = -1;
37              } else {
38                  cmp = 1;
39              }
40          }
41          if (cmp == 0 || this.numRatings == 0 || other.numRatings == 0) {
42              cmp = this.numRatings - other.numRatings;
43          }
44          if (cmp == 0) {
45              cmp = this.title.compareTo(other.title);
46          }
47          return cmp;
48      }
49  }
```

Not shown above but class header includes "`implements Comparable<MovieRating>`"

From the spec, we are given the following ordering properties:

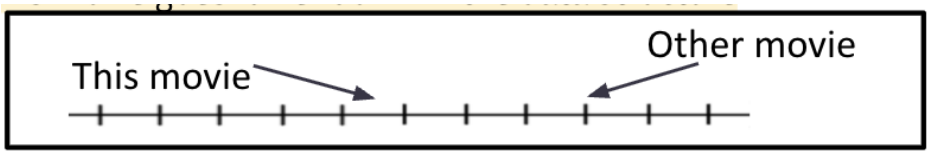
1. First, movies with a lower `averageRating()` are considered "less" than other movies. (Lower average ratings goes closer to the start of a sorted list)
2. Additionally, if there are no movie ratings, this is considered less than all rated movies.
3. If both movies aren't rated, they are "equal"
4. If both movies have the same average rating, then break ties with the movie that has a greater number of ratings.
5. If everything is still tied (both movies have the same average rating and number of ratings), break ties alphabetically by the movies' titles (the movie titles that are alphabetically first are less).

How these order properties translate to the coded solution:

Line 31 initializes an integer variable that lets us keep track of whether any orderings have been assigned to this `Movie` Rating object.

1. First, movies with lower `averageRating()` are considered "less" than other movies. (Lower average ratings go closer to the start of a sorted list) – Lines 32-40

If the two average ratings are not the same, then we check where our movie rating object (this Movie Rating) stands with respect to the other movie rating object we're comparing to. If our movie rating's average rating is less than the other movie rating, our movie rating object needs to go behind the other movie rating object. Thus, we assign `cmp` to -1.



Otherwise, if our movie rating's average rating is higher, our movie rating should go ahead of the other movie rating sowe assign `cmp` to +1.

If we "make it past" line 40 with our `cmp` variable still equal to 0, we know that the average ratings for both movies are the same, and we need to order the movies with these rules:

2. Additionally, if there are no movie ratings, this is considered less than all rated movies.
3. If both movies aren't rated, they are "equal"
4. If both movies have the same average rating, then break ties with the movie that has more number of ratings.

We tackle these 3 ordering properties with lines 41-43, using the number of ratings as a tiebreaker. If `cmp` is still 0 and either of the two movies has no ratings (`numRatings` = 0), we keep determining the ordering with these other ordering rules

.

Here we use a comparison trick that ONLY WORKS WITH INTEGERS. (line 42). Let's walk through how line 42 works with each of the ordering rules we established for this section of the method:

2. Additionally, if there are no movie ratings, this is considered less than all rated movies.
   - If this movie has no ratings, but the other movie has some ratings, then our cmp will be assigned a negative integer value because **0 - [positive integer for other Movie Ratings] = negative integer.**
   - If the other movie has no ratings and our movie does have some ratings, then similarly our cmp will be a positive integer.

3. If both movies aren't rated, they are "equal"
   - If both movies have no ratings, cmp will be 0

4. If both movies have the same average rating, then break ties with the movie that has more number of ratings.
   - This movie has more ratings than other movie, cmp is positive integer
   - Other movie has more ratings than this movie, cmp is negative integer.

   Notice: subtraction order really matters! If you switched line 42 such that cmp = other.numRatings - this.numRatings this would have completely flipped the ordering!

If we make it past line 43 and our `cmp` is STILL 0, that means the movies have equal ratings and we need to have our final tiebreaker with alphabetical title comparisons.

5. If everything is still tied (both movies have the same average rating and number of ratings), break ties alphabetically by the movies' titles (the movie titles that are alphabetically first are less). Lines 44 - 46

Here, we can use a built in `compareTo()` method within the String object! (The Double object also has a similar `compareTo()` method. You can check out more information on this here)

This built-in method compares the strings lexicographically* and gives a similar positive, negative, or 0 integer value based on where our String stands with respect to the other string (taken in as a parameter).

For example, if we had a String object "apple" and another String object "banana". (pretend the String reference names of these two objects are also apple and banana respectively :>)
If we did:

```
cmp = this.apple.compareTo(banana);
```

This would give us a negative result because apple < banana lexicographically.

Similarly, if we did:

```
cmp = this.banana.compareTo(apple);
```

This would give us a positive result because banana > apple lexicographically.

Notice: By switching the string object you're calling the `compareTo()` method from can completely switch the result! ORDERING MATTERS!!!!

Lastly, we return the cmp flag on line 47!

Yaay! Hope this helps and happy coding! 🎉🥳

*lexicographically: fancy way of saying alphabetically. You'll hear it a lot throughout the course, so it's sprinkled in to introduce it. Here's the dictionary definition from dictionary.com/browse/lexicographically : sorted in a way that uses an algorithm based on the alphabetical order used in dictionaries.

JUnit CheatSheet

Learning a new concept can be overwhelming so we've compiled a cheatsheet you can reference while you write your own JUnit tests!

Creating a JUnit Test Class and JUnit Test Case

To create a JUnit test class, make sure you import `org.junit.jupiter.api.*` and `static org.junit.jupiter.api.Assertions.*`. These will give you access to method annotations like `@Test` and `@BeforeEach` and assertion methods like `assertTrue()` and `assertFalse()`.

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class ExampleTestClass {
    @Test
    public void yourTestCase() {
        // Assertion methods called here
    }
}
```

JUnit Method Annotations

Method annotations are used by JUnit so that JUnit knows how to treat your methods. Before you write a method, you must attach a method annotation. This special syntax is then interpreted by JUnit to know how to execute your method.

`@Test`: Turns a public method into a JUnit test case.

```
@Test
public void test() {
    ...
}
```

`@Timeout(time)`: Times the test such that the test will fail after `time` milliseconds. Thus, the code must finish execution before `time`. Note that you still need the `@Test`.

```
// Test will fail after 1000 ms
@Test
@Timeout(1000)
public void test() {
    ...
}
```

`@BeforeEach`: The method will be executed before each `@Test`

```
private int num; // This is a field

// This method will execute before each @Test
@BeforeEach
public void setUp() {
    num = 0;
}

@Test
public void test() {
    assertSame(0, num);
    num++;
    assertSame(1, num);
}
```

JUnit Assertion Methods

Assertion methods are the building blocks of JUnit and how you will write testing code inside your test methods. When you use an assertion method, the result needs to match up with what the assertion method expects, otherwise, your test will fail. Below are the most common types of assertion methods that you will use:

`assertTrue(test)`: Fails if the `test` is `false`

```
@Test
public void test() {
    int x = 2;
    String s = "Hello World";
    assertTrue(true);
    assertTrue(x == 2);
    assertTrue(s.equalsIgnoreCase("Hello World"));
}
```

`assertFalse(test)`: Fails if the `test` is `true`

```
@Test
public void test() {
    int x = 2;
    String s = "Hello World";
    assertFalse(false);
    assertFalse(x != 2);
    assertFalse(s.contains("a"));
}
```

`assertEquals(expected, actual)`: Fails if the `expected` and `actual` are not equal

```
@Test
public void test() {
```

```
    String s1 = "Hello World";
    String s2 = "Hello World";
    String s3 = "Hello World";
    assertEquals(s1, s2);
    assertEquals(s2, s3);
    assertEquals(s3, s1);

    List<Integer> list1 = new ArrayList<>();
    List<Integer> list2 = new ArrayList<>();
    for (int i = 1; i <= 5; i++) {
        list1.add(i);
        list2.add(i);
    }
    assertEquals(list1, list2);
}
```

`assertSame(expected, actual)`: Fails if the `expected` and `actual` are not equal using reference semantics (==)

```
@Test
public void test() {
    int x = 2;
    assertSame(2, x);

    List<Integer> list1 = new ArrayList<>();
    List<Integer> list2 = list1;
    assertSame(list1, list2);
}
```

`assertNotSame(expected, actual)`: Fails if `expected` and `actual` are equal using reference semantics (==)

```
@Test
public void test() {
    int x = 2;
    assertNotSame(3, x);

    List<Integer> list1 = new ArrayList<>();
    List<Integer> list2 = new ArrayList<>();
    assertNotSame(list1, list2);
}
```

`assertNull(value)`: Fails if `value` is non-null

```
@Test
public void test() {
    Map<String, Integer> map = new HashMap<>();
    map.put("cse122", 1);
    assertNull(map.get("Hello World"));

    String s = null;
    assertNull(s);
```

```
}
```

`assertNotNull(value)`: Fails if `value` is null

```
@Test
public void test() {
    Map<String, Integer> map = new HashMap<>();
    map.put("cse122", 1);
    assertNotNull(map.get("cse122"));

    String s = "Hello World";
    assertNotNull(s);
}
```

`assertArrayEquals(Any[] expectedValues, Any[] actualValues)`: Fails if `expectedValues` and `actualValues` do not have the same elements, in the same order.

```
@Test
public void test() {
    int[] a = new int[] {1, 2, 3};
    int[] b = new int[] {1, 2, 3};
    assertArrayEquals(a, b);
}
```

`assertThrows(exception.class, () -> {code})`: Fails if `code` does not throw `exception`

```
@Test
public void test() {
    List<Integer> list = new ArrayList<>();
    assertThrows(IndexOutOfBoundsException.class, () -> {
        list.get(2); // List is currently: []
    });

    assertThrows(IndexOutOfBoundsException.class, () -> {
        list.add(1); // List is currently: [1]
        list.add(2); // List is currently: [1, 2]
        list.add(3); // List is currently: [1, 2, 3]
        list.remove(3); // Index 3
    });

}
```

Using JUnit to test Java's ArrayList Implementation:

Below is an example of a JUnit testing class that tests Java's ArrayList implementation:

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
```

```java
import java.util.*;

public class ArrayListTest {
    private static final int TIMEOUT = 2000;
    private List<String> list;

    @BeforeEach
    public void setUp() {
        list = new ArrayList<>();
    }

    @Test
    @Timeout(TIMEOUT)
    public void testAddingElements() {
        assertTrue(list.isEmpty());
        list.add("Hunter Schafer");
        list.add("Miya Natsuhara");
        list.add("CSE 122");

        assertEquals("Hunter Schafer", list.get(0));
        assertEquals("Miya Natsuhara", list.get(1));
        assertEquals("CSE 122", list.get(2));

        assertTrue(list.size() == 3);
    }

    @Test
    public void testContains() {
        assertTrue(list.isEmpty());
        list.add("CSE 122");

        assertTrue(list.contains("CSE 122"));
        assertFalse(list.contains("Hello World"));
    }

    @Test
    public void testNegativeIndexGet() {
        assertTrue(list.isEmpty());
        assertThrows(IndexOutOfBoundsException.class, () -> list.get(-1));
    }
}
```

⭐JUnit: Line

Suppose that you are provided with a pre-written class `Line` which implements the following behavior:

- `public void addPoint(Point other)`
  - Extends the current line to include the provided 'other' point. Each point is only counted once (duplicates are ignored).
- `public Point getStart()`
  - Returns the first Point in the line (the Point with the smallest x-value). Throws a `NoSuchElementException` if there are no points currently within the line.
- `public Point getEnd()`
  - Returns the last Point in the line (the Point with the largest x-value). Throws a `NoSuchElementException` if there are no points currently within the line.
- `public int getNumPoints()`
  - Returns the total number of points within this line segment.

Fill in the provided `Testing.java` following the instructions step-by-step to complete your testing suite.

> **NOTE:** This is an example of "clear-box" testing, where you know and can view the current solution directly. This can be beneficial as after implementing a solution, programmers will likely be more knowledgeable about potential edge cases that would be useful to account for in their tests!

OOP: Design Quiz

NOTE: Remember, while there is no "right" answer to how to design a class, there is a subset of more reasonable decisions. The following questions ask what data structure(s) are most appropriate to implement specific behavior. The answers are what the course staff think are best, but if you have another idea ask us why it might not be best!

Question 1

A CSE 123 student is interested in creating a class that can represent a phone book. The student wants to use the phone book to look up the phone numbers of their friends, given the friend's name. What underlying data structure(s) would be most appropriate for this class?

You should be able to explain your answer

- [ ] List
- [ ] Set
- [ ] Stack
- [ ] Queue
- [ ] Map
- [ ] Array
- [ ] 2D Array

Question 2

A CSE 123 student is interested in creating their own streaming service. The student wants to allow users to search for titles given keywords, and also keep track of which titles they've already watched. What underlying data structure(s) would be most appropriate for this class?

You should be able to explain your answer

- [ ] List
- [ ] Set
- [ ] Stack
- [ ] Queue
- [ ] Map
- [ ] Array
- [ ] 2D Array

Question 3

A CSE 123 student is interested in creating an online ticket-purchasing service. The student wants to allow vendors to process purchases in the order they arrived as well as keep track of users that have already made purchases to prevent scalping. What underlying data structure(s) would be most appropriate for this class?

You should be able to explain your answer

- [ ] List
- [ ] Set
- [ ] Stack
- [ ] Queue
- [ ] Map
- [ ] Array

- [ ] 2D Array

Question 4

A CSE 123 student is interested in creating a spreadsheet application that also allows users to keep a history of all edits such that previous changes can be reverted via ctrl-z. What underlying data structure(s) would be most appropriate for this class?

You should be able to explain your answer

- [ ] List
- [ ] Set
- [ ] Stack
- [ ] Queue
- [ ] Map
- [ ] Array
- [ ] 2D Array

OOP: ClockTime

Suppose that you are provided with a pre-written class `ClockTime` as shown below.

Write an instance method named `advance` that will be placed inside the `ClockTime` class to become a part of each `ClockTime` object's behavior. The `advance` method accepts a number of minutes as its parameter and moves the `ClockTime` object forward in time by that amount of minutes. The minutes passed could be any non-negative number, even a large number such as 500 or 1000000. If necessary, your object might wrap into the next hour or day, or it might wrap from the morning ("AM") to the evening ("PM") or vice versa. A `ClockTime` object doesn't care about what day it is; if you advance by 1 minute from 11:59 PM, it becomes 12:00 AM. For example, if the following object is declared in client code:

```
ClockTime time = new ClockTime(6, 27, "PM");
```

The following calls to your method would modify the object's state as indicated in the comments.

```
time.advance(1); // 6:28 PM
time.advance(30); // 6:58 PM
time.advance(5); // 7:03 PM
time.advance(60); // 8:03 PM
time.advance(128); // 10:11 PM
time.advance(180); // 1:11 AM
time.advance(1440); // 1:11 AM (1 day later)
time.advance(21075); // 4:26 PM (2 weeks later)
```

Assume that the state of the `ClockTime` object is valid at the start of the call and that the `amPm` field stores either "AM" or "PM".

This implementation of ClockTime with a hour, minute, and amPm field is only one of the many possible implementation! Try thinking about other possible selections of fields and how that would change the implementations of the other method!