Tutorial 7

Objectives

Practice using polymorphism and abstract things.

Attendance Quiz

Please log on to cuLearn using one of the computers in the tutorial room and complete the attendance quiz. You can only access the quiz if you log in using one of the computers in the lab. You cannot use a laptop for this. This is a time limited quiz. Be sure to do this as soon as you arrive.

At the end of the tutorial a TA will assign you a grade, call it G, which is 0, 1 or 2, depending on the progress you make during the tutorial. If you spend your time reading mail and chatting with your friends you will receive 0. If you have completed the attendance quiz on time then G will be your tutorial grade. If you have not completed the attendance quiz on time, then your tutorial grade will be max(0, G - 1/2). Each tutorial grade will therefore be one of 0, 0.5, 1.5 or 2.

In order to receive full marks for this tutorial, you must fully complete parts 1, 2 and 3.

---

0) Make a directory called **comp1406t7**. Download all the tutorial files to this directly. Run the **javadoc** program to generate the API for the classes.

1) Make a **MyBox** class that extends the **Box** class.

You will need to **override** the inherited **compareTo()** and **toString()** methods. When comparing a mybox (to a box), use the length of the inside attribute (shorter length means less than) and then use lexicographical (alphabetical) ordering for ties. The **toString()** method should disply the inside of a mybox.

A sample testing program is provided. It creates an array of boxes, prints them, sorts them and then prints them again. The testing program should output

```
before sorting ------------------------
[kitten, cat, dog, oh, cow, oxen]
after sorting -------------------------
[oh, cat, cow, dog, oxen, kitten]
```

Change the *strings* array to be sure your code works correctly with different inputs.

2) Complete the provided **Team** class by overriding the **compareTo** method (inherited from the **Comparable** interface) and overriding the **toString()** method (inherited from **Object**) as described below.

Each team will have zero or more *points*. The number of points that a team has is two times the number of wins a team has plus the number of draws. So, if a team has 3 wins and 1 draw then that team has 7 points.

Let A and B be two teams. The ordering of teams (when compared) is as follows:

1. A < B when A has less points than B.
2. A > B when A has more points than B.
3. When A and B have the same amount of points, then
    i. A < B when A has played more games than B
    ii. A > B when A has played less games than B
4. When A and B have the same amount of points and have played the same amount of games, then
    i. A < B when A has less wins than B
    ii. A > B when A has more wins than B
5. A == B when A and B have the same amount of points, played the same number of games, have the same number of wins and same number of draws.

When you print a team object (**toString()**) it should be of the form

```
"name[points]: w/l/d"
```

where w,l,d are the wins, losses and draws of the team. For example, the output might look like

```
Carleton[12]: 5/3/2
```

Use the **TestTeam** class to test your **Team** class.

3) Sometimes we need to sort (or compare) data using different rules.
Implementing **Comparable** only lets us have one rule though.
When we want multiple rules, we can define multiple **Comparator** objects which will each have a method

```
public int compare(T o1, T, o2)
```

that can be used. The **compare()** method returns a negative integer is o1o2.
It is very similar to the **compareTo()** method from the **Comparabale** interface. **Comparator** is another interface in Java.
Typically, a **Comparator** is created as an *anonymous class* in Java. That is, we define a class that implements the abstract compare method directly in our code where we need it.
For example, suppose we have a **Student** class:

```
public class Student{
  String lastName;
  String firstName;
  int    id;
}
```

We could sort a list of students as follows

```
Comparator<Student> cmp = new Comparator<Student>{
                           @Override
                           public int compare(Student s1, Student s2){
                             return s1.id - s2.id;
                           }
                          };
java.util.Arrays.sort(studentArray, cmp);
```

Here the **Arrays.sort()** method takes the array to be sorted and a comparator that specifies how to compare things in the array. The comparator itself is

created and defined just above it. We are creating an anonymous class that implements the Comparator interface.

Go back to the **TestTeam** class. In the main method, create a Comparator and re-sort the array of teams you have already sorted with this Comparator.
The new ordering is based on the following rules

Let A and B be two teams. The ordering of teams (when compared) is as follows:

1. A < B when A's name is shorter than B's
2. A > B when A's name is longer than B's
3. When A and B have names with the same length, then
    i. A < B when A has less wins than B
    ii. A > B when A has more wins than
    iii. A = B when A has the same number of wins as B