COP 3330 – Object Oriented Programming in Java
Comparable Interface

Java has one interface built in which is extremely useful: Comparable.

Here is a link to the Java doc:

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Comparable.html

The code for the interface looks like this:

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

This definition uses something called generics, which we haven't seen before. The idea behind the interface is that we would like for some classes to be "sortable". So, for example, if the user defined a class, it would be nice if the user could tell Java how to sort objects from that class. In other languages, it's common for there to be a compare function/method that takes in two objects (say obj1 and obj2) of the same type and returns a negative integer if obj1 comes before obj2, a positive integer if obj1 comes after obj2 and 0 if they are equal. This is precisely how the compareTo method is supposed to behave for any class that implements Comparable.

Of course, the issue is that unlike the Shape2D interface, we want our class to have a method compareTo that takes in an arbitrary type. (In the Shape2D interface, there were no formal parameters for any of the methods and the return types were not arbitrary, they were primitive types in Java.) The parameter we want the method to take in has to be of the same type (class) as the class we are defining.

This is where generics come in. The idea behind a generic is that if we ever want to specify an arbitrary class, we can call it class <T> and then for the scope of the definition we're giving, when we see T, we'll know that means the name of a class. This means that if we want to implement the Comparable interface, we must do something like this:

```
public class fraction implements Comparable<fraction> {

....

    public int compareTo(fraction other) {
        // Code to compare this to other.
    }
}
```

Thus, to use an interface with a generic, we must simply replace the T with the name of the class we're defining.

Let's go ahead and dive in and define the compareTo method in the fraction class. (Recall that we assume the denominator is positive.)

```
public int compareTo(fraction other) {
    return this.numerator*other.denominator -
           other.numerator*this.denominator;
}
```

Let's see why this works (assuming that our numerator and denominator are less than 40000).

Let this object equal n_1/d_1 and other equal n_2/d_2. For the time being, let's assume that

$$\frac{n_1}{d_1} < \frac{n_2}{d_2}$$

Recall that both denominators are greater than 0, so multiplying the equation by d_1d_2 is multiplying this equation by a positive integer, meaning that the inequality sign stays in the same direction:

$$n_1 d_2 < n_2 d_1$$

Now, subtract the expression on the right hand side from both sides to get:

$$n_1 d_2 - n_2 d_1 < 0$$

Thus, if this fraction is less than other, the corresponding expression on the left hand side above is negative, which is, by definition, what compareTo needs to return in this situation.

You can similarly show that our implementation of compareTo returns 0 if this and other are equal, and it returns a positive integer if this is greater than other.

To distinguish the class that implements Comparable from the original fraction class, we'll create a new class, fraction2 that implements Comparable<fraction2>, so technically our compareTo method reads:

```
public int compareTo(fraction2 other) {
    return this.numerator*other.denominator -
           other.numerator*this.denominator;
}
```

On the next page we include our application to test our newly designed fraction2 class that implements Comparable<fraction2>. Java has a sort function in Arrays:

```
public static <T extends Comparable<? super T>> void parallelSort(T[] a);
```

What all of this really means is that T is required to implement Comparable. But we can just call regular Arrays.sort instead of Arrays.parallelSort to get this to work.

```
import java.util.*;

public class SortFractions {

    final public static int NUMF = 10;

    public static void main(String[] args) {

        Random rndObj = new Random();

        fraction2[] list = new fraction2[NUMF];
        for (int i=0; i<NUMF; i++) {
            int n = rndObj.nextInt(21)-10;
            int d = rndObj.nextInt(10) + 1;
            list[i] = new fraction2(n, d);
        }

        for (int i=0; i<NUMF-1; i++)
            System.out.print(list[i]+", ");
        System.out.println(list[NUMF-1]);

        Arrays.sort(list);

        for (int i=0; i<NUMF-1; i++)
            System.out.print(list[i]+", ");
        System.out.println(list[NUMF-1]);
    }
}
```

Notice how EASY this is to use. Once you declare your own class which implements Comparable<T>, then you can just take any array of your object and sort it. Since you define the compareTo method in your class, you can sort your object any which way you see fit!!!

You literally just do

```
Arrays.sort(PUTARRAYHERE);
```

Now, let's look at a couple of examples of custom sorting using this feature via problems on a programming competition website, open.kattis.com.

Kattis Website and Problem: Sideways Sorting

The website open.kattis.com runs many programming competitions and hosts and archive of old programming competition problems for the public to work on. It's a wonderful resource for anyone who enjoys problem solving and programming.

A programming contest focuses on algorithmic problem solving and implementation ignoring many real-world issues inherent in software development. (As I like to say, it removes all the boring stuff...UI design, customer preferences, etc.)

Since UI is ignored, the programs one writes in programming competitions are not meant to be user-friendly. As such, a big difference between these programs and ones written in traditional introductory programming contests is that our solutions should NEVER prompt the user to enter information.

Instead, we are given strict instructions on the input we will be reading in for the problem we are attempting to solve. We must read in this input exactly. Then, we're given some task to complete with that input. Finally, to prove we've correctly solved that task, we must produce some output in a precise format to the input given to us. To judge correctness, several test cases are pre-made by judges (human judges) with correct solutions. In order for a contestant's solution to be judged correct, every test case must produce identical output (they check all non-white space characters) to the judge output.

When submitting a program to a programming competition website, here are the typical responses a submission can receive:

Correct
Wrong Answer
Run-Time Error
Time-Limit Exceeded
Compilation Error

Typically, your program is run on several test cases. If all are judged Correct, you receive the Correct verdict. If any test case isn't correct, then you receive that verdict instead. (Some website prioritize one of these errors over others, other websites just provide the response of the first incorrect test case. It's entirely possible that your code could produce a correct answer on one test case, a wrong answer on another, crash on a third test case, and take took long on a fourth test case!)

To practice utlizing the Comparable interface in Java, let's solve the following problem on Kattis:

https://open.kattis.com/problems/sidewayssorting

We must read our input into a 2d grid of char. (To make it easier to access each character, we'll do this instead of an array of String.) We'll find the method toCharArray() in the String class particularly useful in this regard. Similarly, when we need to create a String out of a character array, we can use the constructor for String that takes in a character array.

We'll create an object that stores two Strings: the word itself, and the word in lower case, and we'll define our compareTo method based on the lowercase string. The other key idea here is that we need to form our strings by reading down columns. We'll accomplish this by "flipping" the grid. We'll allocate a transposition of the grid space in advance, and when we read in each line, we'll fill in by columns instead of rows, so that the words we want to sort will appear on the rows and not the columns. When we output the results, we'll have to flip our results back to the original orientation.

First, let's take a look at our word class:

```
class word implements Comparable<word> {

    private String orig;
    private String lower;

    public word(String s) {
        orig = s;
        lower = s.toLowerCase();
    }

    public int compareTo(word other) {
        return lower.compareTo(other.lower);
    }

    public char[] getOrigArray() {
        return orig.toCharArray();
    }
}
```

There's not a lot here, just what we need. We take in a String into our constructor, but we store two items for our object, both the string itself and its lowercase version. (Strictly speaking, this isn't necessary because we could just call compareToIgnoreCase inside of our compareTo. I just thought this would be easier to understand and easier to view as actual custom sorting.) In our compareTo, we just return what the String compareTo returns between our instance variables storing the lowercase version of the strings.

Our main class implementation is trickier, where we have to deal with two transpositions. (Transposition is switching rows with columns.) In addition, the precise requirements for output are a bit annoying (only putting a blank line in between cases and processing cases with a sentinel value instead of knowing how many cases there are in advance.)

```
public class sidewayssorting {

    public static void main(String[] args) {

        Scanner stdin = new Scanner(System.in);
        int r = stdin.nextInt();
        int c = stdin.nextInt();

        while (r > 0) {

            char[][] input = new char[c][r];

            // Transpose input.
            for (int i=0; i<r; i++) {
                char[] tmp = stdin.next().toCharArray();
                for (int j=0; j<c; j++)
                    input[j][i] = tmp[j];
            }

            // copy the data into word objects.
            word[] list = new word[c];
            for (int i=0; i<c; i++)
                list[i] = new word(new String(input[i]));

            // Sort it.
            Arrays.sort(list);

            // Store the output here. Again, transpose back!
            char[][] output = new char[r][c];
            for (int i=0; i<c; i++) {
                char[] tmp = list[i].getOrigArray();
                for (int j=0; j<r; j++)
                    output[j][i] = tmp[j];
            }

            // This is what we want to output.
            for (int i=0; i<r; i++)
                System.out.println(new String(output[i]));

            r = stdin.nextInt();
            c = stdin.nextInt();

            // Annoying.
            if (r>0) System.out.println();

        } // end case loop
    }
}
```

Notice when we allocate the char array we make it [c][r], not the other way around, because we want to store the transposition of how the input is presented. By preallocating, all the spaces are available for us to write to. Then, when we read in the string, we store it in a char array and the copy the values into our grid into a column, not a row.

This allows us to easily create word objects by forming String objects to pass to the constructor of the class. Once we have the array of word objects, we can just sort it via Java's sort, since we've implemented Comparable<word>.

Now, we just have to transpose these results back into a [r][c] grid, which we do in a similar fashion. Finally, when printing the output, we make use of the String constructor that takes in a character array.

Sort of Sorting
While we're at it, let's do one more problem from Kattis, Sort of sorting:

https://open.kattis.com/problems/sortofsorting

Here we are again storing strings, but want to sort them by the first two letters. If those are tied, then we want to use the index into the original list to break the ties.

Since we're just sorting by regular lexicographical order of the first two letters, we can use the String class compareTo in our compareTo, but just on the appropriate substrings. Since all names are guaranteed to have at least two characters, our substring call will never crash the program.

Let's first take a look at our name class which will contain the ordering logic:

```
class name implements Comparable<name> {

    private String orig;
    private int id;

    public name(String s, int idx) {
        orig = s;
        id = idx;
    }

    // How the question defines this.
    public int compareTo(name other) {

        // First tiebreaker.
        int tmp = orig.substring(0,2).compareTo(other.orig.substring(0,2));
        if (tmp != 0) return tmp;

        // Final tie breaker.
        return id - other.id;
    }

    public String toString() {
        return orig;
    }
}
```

Here we just need to store the String and the index the word was in. In our compareTo, we first compare the two prefixes of the strings (length 2). If this breaks our tie, we return that answer. If not, we break the tie by looking at the difference of indexes. This is more succinct than an if

statement. Hopefully by now you see that we try to exploit the numerical properties of what we are looking at so that we can return the appropriate signed integer for our comparison without splitting an if into three cases.

Our main will look shockingly like the other main, but easier, since we don't have to do any transpositions or change any Strings to character arrays. (Basically I took the first solution and just removed stuff from it!) Here it is:

```
public class sortofsorting {

    public static void main(String[] args) {

        Scanner stdin = new Scanner(System.in);
        int n = stdin.nextInt();

        while (n > 0) {

            name[] list = new name[n];

            for (int i=0; i<n; i++)
                list[i] = new name(stdin.next(), i);

            Arrays.sort(list);

            for (int i=0; i<n; i++)
                System.out.println(list[i]);

            n = stdin.nextInt();

            if (n>0) System.out.println();
        }
    }
}
```

Even though these uses are fairly similar, hopefully it can be seen that this interface has a wide variety of uses, since sorting almost any sort of object becomes very useful in improving retrieval times for searching for specific objects of any type. Java makes is fairly easy to use via the Comparable interface!