**Sorting Students**

Hopefully you now have a decent understanding of some searching / sorting algorithms. What if you needed to sort something other than a primitive type (e.g. int)? If you tried to apply your sorting algorithms to a reference type (object), it wouldn't work – reference types can't be compared with the < and > operators!

The reason is that reference types, especially custom classes you create, have no 'natural order.' Integers and doubles do – numerical order. Because custom types were created by you, the programmer, you need to tell Java HOW to order custom types. To do this, you need to implement the java.lang.Comparable interface.

The Comparable interface, like all interfaces, is a contract – for your class to be-a Comparable (sortable) type, it must override the interface's methods. Luckily, there is only one method to override – int compareTo().

**Sorting a List of custom types**

As stated previously, Java doesn't have a 'natural order' for custom types. The 'how' is supplied by implementing the Comparable interface, making your class sortable in the way you want by overriding its compareTo() method. To do this:

1. Write a Student class that has the following:
   1. private String name
   2. private int studentID
   3. A constructor, getter methods, overridden toString and equals methods
2. This class should *implement* the Comparable interface (no import necessary, part of the java.lang package that is automatically imported). You should supply the *type parameter* when implementing the Comparable interface, as when declaring ArrayLists (e.g. ArrayList<Integer> list):

public class Student implements Comparable<Student>

1. Your Student class must override the compareTo() method to be-a Comparable (orderable) type. This method does the following:
   1. Returns a positive integer if this object should be considered "greater than" the parameter
   2. Returns a negative integer if this object should be considered "less than" the parameter
   3. Returns 0 if the two objects should be considered equal
2. Example, for our Student class:

@Override

public int compareTo(Student s) {

/\* something \*/ return 1;

/\* something \*/ return -1;

return 0:

}

The implementation of the compareTo() method can be as simple or complex as you need – you can compare multiple variables' values or just one.

1. (Riddle) I can live on a sill, I can be a tasty fruit. If I get injured, I'll need a good boot.
2. Complete the compareTo() method to compare Student objects based on their studentID only. View the code in the StudentTester class (supplied), then run its main() method; you should see the following output:

[100004: Sally, 100006: Bob, 100007: Sam, 100009: Joe]

How did that work?? How was the Collections.sort() method magically able to sort the Student objects in the list? This method (and any classes that do some type of sorting / ordering) uses a call to the compareTo() method when sorting (at the time of writing, using a variation on the merge sort algorithm called TimSort). In other words, the Collections.sort() method expects a List<Comparable> object - which your ArrayList of Students now is\*.

*\*A slightly more technical explanation - List is an interface that ArrayList implements, so an ArrayList* is-a *List. If Student implements Comparable, Student* is-a *Comparable. Therefore, an ArrayList<Student> object satisfies the* is-a *relationship with List<Comparable>. Collections.sort() will only work on a collection of Comparable objects – otherwise, there would be no way for Java to know how to order your type's objects.*

This will work, because your Student class (the type being sorted) has implemented the Comparable interface, thus satisfying the 'contract' that your class will provide the implementation for the compareTo() method. In other words, implementing the Comparable interface makes your class comparable (sortable)!

1. Comment out your current compareTo() method implementation. Write a new version that will now compare based on the (alphabetical) value of the name instance variable. How is this possible if you can't compare Strings like this: "hello" > "something"? Luckily, the String class *also* implements the Comparable interface! You can call String's overridden compareTo() method *inside* your method. Oh my!
2. Re-run the main() method in the StudentTester class, and your output should now look like this:

[100006: Bob, 100009: Joe, 100004: Sally, 100007: Sam]

1. (Riddle) Though I live beneath a roof, I never seem to dry. If you will only hold me, I swear I will not lie. What am I?
2. In the StudentTester class, add a loop that will read all the names in the **"names.csv"** input file, creating a Student object for each (Scanner handles .csv files just like .txt files). Assign each Student a unique, random 6-digit ID number and add it to an ArrayList (it may be useful to make this an instance variable, you'll use the list in the next exercise).

Copy / paste the sorting algorithm of your choice from the last project, and modify it to use the compareTo() method, rather than (only) the comparison operators (< and >) that you used for sorting integers. **Note:** if your Student class uses String's compareTo() method, remember that it returns positive, negative, or zero (NOT a particular value like -1 or 1). For example, "apple".compareTo("zebra") returns -25 (negative, since "apple" comes before "zebra").

Sort using your new method implementation, print the sorted List and bask in the glow of implemented interfaces, run-time polymorphism, and next-level algorithmic perfection.

1. Add a method int indexOf(Student s) to the StudentTester class. This method should use a *binary search* (iterative or recursive, up to you) to return the index of the supplied student (or -1). I hope you've overridden the method that Java uses to determine equivalency between two objects! (You may have to refer back to the powerpoints.) No calls to ArrayList's indexOf method!