ADSB Exercise 1

Hopefully by now you have understood the necessity and beauty of some of our sorting algorithms. In this exercise we will be demonstrating our knowledge of the three different sorts, bubble, insertion and selection sort. But before we do that, we are going to talk about the Comparable interface.

Part 1: Comparable

Your first job is to write me a Phone class, where we will be using the Comparable interface. Before you start writing the compareTo method, be sure to have the class, 5 instance variables (at least) and getters and setters for each. PLEASE DON'T FORGET TO COMMENT AS YOU GO!

Overriding the Object superclass' equals

Because Phone is an Object, like every other class, it has an inherited equals method. It looks a little like this:

```
public boolean equals(Object obj){
      return this == obj;
  }
```

What is this testing for? Identity or equality of two objects? In this case, it is Identity because of the use of ==, so we want to change that. **Write a .equals method that takes in an object, checks to make sure it is not null and IS and instance of Phone. Then make sure to check each instance variable one by one and return true or false depending on the results of the comparison.**

Comparable

This leads to another question, how to I compare objects in such a way that I will know which one should go first and which should go second? In comes the Comparable interface.

First, implement the comparable interface like so:

```
public class Phone implements Comparable<Phone>{/*Stuff*/}
```

We've used interfaces before so everything up to Comparable should looka little familiar. What is this <Phone> deal? This tells the JVM that when you implement the compareTo method (Comparable's sole required method), it will take in an Object of the Phone type and not just of the general Object type.

The interface Comparable authorizes an int-returning method compareTo, which takes a Phone as its parameter. Define and implement the compareTo method.

This method should return one of three possible values:

- **-1 (or any negative int) if this object is "less than" or "comes before" the parameter object**
- **0 if this object has the same value as the parameter object**
- **+1 (or any positive int) if this object is "greater than" or "comes after" the parameter object**

You should first confirm that other is not null, and if not, throw an IllegalArgumentException. (If we had not used the type parameter <Phone> with Comparable, compareTo would have taken an Object rather than a Phone as a parameter, and we would have needed to check that compareTo's parameter was the right type via instanceof)

Then, check whether this and the parameter are literally the same object in memory, in which case we should return 0. Then, go through each of the instance variables in an order that you decide. For each one compare the instance variable values of this and other; if they are different, return -1 or 1 as appropriate. If they are the same, move on to the next instance variable. If all of the instance variables are the same, we should return 0. Note that you'll be relying somewhat on String's compareTo method here!

In Phone's main method, write some code creating a few Phone objects and comparing them using your compareTo method. Does it behave as expected?

hashCode()

We have one last thing to do here, when we override the equals method, we must also override the hashCode() method. Every single object in Java has a hash code. Why do we need it? In order to store an object in a hashTable (we'll learn about that next semester), it has to have a unique hash code.

Here is a great definition from "Sheriff Frank" off of the good ol' interweb:

A hashcode is a number generated from any object. This is what allows objects to be stored/retrieved quickly in a Hashtable. [Mr. Paul's note: You'll treat these in Analysis of Algorithms]

Imagine the following simple example:

> On the table in front of you, you have nine boxes, each marked with a number 1 to 9. You also have a pile of wildly different objects to store in these boxes, but once they are in there you need to be able to find them as quickly as possible.

What you need is a way of instantly deciding which box you have put each object in. It works like an index; you decide to find the cabbage, so you look up which box the cabbage is in, then go straight to that box to get it.

Now imagine that you don't want to bother with the index. Instead, you want to be able to find out immediately from the object which box it lives in.

In the example, let's use a really simple way of doing this: the number of letters in the name of the object. So the cabbage goes in box 7, the pea goes in box 3, the rocket in box 6, the banjo in box 5 and so on. What about the rhinoceros, though? It has 10 characters, so we'll change our algorithm a little and "wrap around" so that 10-letter objects go in box 1, 11 letters in box 2 and so on. That should cover any object.

Sometimes a box will have more than one object in it, but if you are looking for a rocket, it's still much quicker to compare a peanut and a rocket than it is to check a whole pile of cabbages, peas , banjos and rhinoceroses.

That's a hash code. A way of getting a number from an object so it can be stored in a Hashtable. In Java a hash code can be any integer, and each object type is responsible for generating its own. Lookup the "hashCode" method of Object.

Here is the relationship we MUST maintain, if two objects are equal according to the equals() method they MUST have the same hashcode. Now for how, take a look at the following textbook excerpt for guidance:

1. Store some constant nonzero value, say, 17, in an `int` variable called `result`.
2. For each significant field `f` in your object (each field taken into account by the `equals` method, that is), do the following:
   a. Compute an `int` hash code `c` for the field:
      i. If the field is a `boolean`, compute `(f ? 1 : 0)`.
      ii. If the field is a `byte`, `char`, `short`, or `int`, compute `(int) f`.
      iii. If the field is a `long`, compute `(int) (f ^ (f >>> 32))`.
      iv. If the field is a `float`, compute `Float.floatToIntBits(f)`.
      v. If the field is a `double`, compute `Double.doubleToLongBits(f)`, and then hash the resulting `long` as in step 2.a.iii.
      vi. If the field is an object reference and this class's `equals` method compares the field by recursively invoking `equals`, recursively invoke `hashCode` on the field. If a more complex comparison is required, compute a "canonical representation" for this field and invoke `hashCode` on the canonical representation. If the value of the field is `null`, return `0` (or some other constant, but `0` is traditional).
      vii. If the field is an array, treat it as if each element were a separate field. That is, compute a hash code for each significant element by applying these rules recursively, and combine these values per step 2.b. If every element in an array field is significant, you can use one of the `Arrays.hashCode` methods added in release 1.5.
   b. Combine the hash code `c` computed in step 2.a into `result` as follows:

```
result = 31 * result + c;
```

3. Return `result`.
4. When you are finished writing the `hashCode` method, ask yourself whether equal instances have equal hash codes. Write unit tests to verify your intuition! If equal instances have unequal hash codes, figure out why and fix the problem.

Part 2: Sorting Videos

Now, based on what you know about sorting, work with (at most) one other person on three sorting videos. You will submit the videos to a form provided by me, you must make a video for bubble, insertion and selection sort with at LEAST 7 ITEMS. Take a look at the rubric below for guidance:

Points Awarded	Part 1	Part 2	Part 3	Total
Insertion Sort	Uses at least 7 items (+1 Point)	Correctly exemplifies the algorithm (+1 Point)	Correctly uploads to Youtube and is on time. (+1 Point)	3
Selection Sort	Uses at least 7 items (+1 Point)	Correctly exemplifies the algorithm (+1 Point)	Correctly uploads to Youtube and is on time. (+1 Point)	3
Bubble Sort	Uses at least 7 items (+1 Point)	Correctly exemplifies the algorithm (+1 Point)	Correctly uploads to Youtube and is on time. (+1 Point)	3

Part 3: 2D Array Exercises

All methods must be static

int min(int[] [] arr)

This method should return the smallest value in the 2D array.

sum(int[][] arr)

Write a method that sums the rows in the 2d array and places the sum in a new array with the same number of rows, but only one column.

[1,2,3] [6]

[3,4,5] → [12]

[5,6,7] [18]

isSequnce(int[][] arr)

Write a method that takes in a 2D array and checks columns to see if those numbers are in lest to greatest. Then, return a 1D array of Boolean values, where each indice should contain whether the column was in order (true) or not (false).

For example:

[1,5,3,7,6]

[2,4,4,8,7] -> [true, false, true, true, false]

[3,7,6,9,3]

isLatin(int [][] arr)

Write a method that checks to see whether arr is a Latin square. That means it must meet two criteria: It must be a square of some size n x n, and each row and column must hold the values 1, 2, ..., n with no repeats

Ex:

A	B	C
C	A	B
B	C	A

[Special danke to Mr. Paul for this method]