Lecture 20: Comparable

- Today we're going to talk about developing a class that represents an angle
    - An Angle could be used to keep track of a Latitude and Longitude
        - e.g. SeaTac Airport is at 47 deg 39 min North and 122 deg 30 min West
    - Each angle has both a number of degrees and a number of minutes
    - Start with a simple class with fields/constructor

      ```
      public class Angle {
          private int degrees;
          private int minutes;

          public Angle(int degrees, int minutes) {
              this.degrees = degrees;
              this.minutes = minutes;
          }
      }
      ```
- (Show the client program, run it)
    - The printed result doesn't give us much information - [Angle@42719c, Angle@30c221]
    - This is the default toString of all objects, but we want something better
    - So let's "override" the toString method
    - We'd really like to use the standard symbols for degrees and minutes (° and ') but we'll make do with "d" and "m"

      ```
      public String toString() {
          return degrees + "d " + minutes + "m";
      }
      ```
- (rerun the client program)
- Other functionality that you might want: add two angles together
    - What we want to do is something like this:

      ```
      Angle a1 = new Angle(23, 26);
      Angle a2 = new Angle(15, 48);
      Angle a3 = a1 + a2;
      ```
    - But we can't do that because the "+" is only for number addition and String concatenation
        - Some languages allow "operator overloading" - allowing a symbol to apply in more circumstances
        - But Java doesn't, so we have to use methods
    - We can do something like this:

      ```
      Angle a1 = new Angle(23, 26);
      Angle a2 = new Angle(15, 48);
      Angle a3 = a1.add(a2);
      ```
    - This is a Java convention
    - Implement the add method

      ```
      public Angle add(Angle other) {
          int d = degrees + other.degrees;
          int m = minutes + other.minutes;
          return new Angle(d, m);
      }
      ```

    - Remember, we can access the private fields of the other Angle (private means private to the class)
    - Modify the client code to add the third angle to the list, and run
- But now we see a problem - the added angle has 74 minutes, which isn't actually allowed
    - Minutes are between 0 and 60
    - What can we do?
    - We could change the constructor to "condense" everything, but that's not what I want to do
    - Instead, let's add a precondition to the constructor

      ```
      // pre: minutes <= 59 and minutes >= 0 and degrees >= 0
      ```

    - And throw an exception if the precondition is not satisfied

      ```
      // pre: minutes <= 59 and minutes >= 0 and degrees >= 0
      //      (throws IllegalArgumentException if not true)
      public Angle(int degrees, int minutes) {
          if (minutes < 0 || minutes > 59 || degrees < 0)
              throw new IllegalArgumentException();
      ```

    - But now we still need to handle the case in add() when the minutes exceed 59

      ```
      public Angle add(Angle other) {
          int d = degrees + other.degrees;
          int m = minutes + other.minutes;
          if (m >= 60) {
              m -= 60;
              d++;
          }
          return new Angle(d, m);
      }
      ```

    - We could also use mod and integer division
- Now I want to modify the Angle class so that we can put a collection of angles into sorted order
    - Add the following the client code

      ```
      int[][]data = {{30, 19}, {30, 12}, {30, 45}, {30, 8}, {30, 55}};
      for (int[] coords : data) {
          list.add(new Angle(coords[0], coords[1]));
      }
      System.out.println(list);
      Collections.sort(list);
      System.out.println(list);
      ```

    - This doesn't compile!
    - We haven't told Java yet how to put things in sorted order
        - We know that 45d15m is more than 30d30m, but how would Java figure that out?
    - In order to use built-in functionality like Collections.sort or Arrays.sort, we have to use the **Comparable<E>** interface
        - (show the Comparable documentation in the Java API)
        - This interface has exactly one method called compareTo

- Comparable<E> and compareTo
    - Many common classes that we've seen implement the Comparable interface
        - String, Integer
    - But some classes don't
        - Point - doesn't make sense to order a point (do you order by x or y? what makes one point less than another?)
    - The compareTo method returns
        - a negative integer if this object is "less than" the other
        - a positive integer if this object is "greater than" the other
        - 0 if this and the other object are "equal"
    - So implementing this interface means that we are certifying that this class can compare itself to another of the same type
- We start by making the Angle class implement the Comparable interface

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

- And then we write the compareTo method
    - First version

```
public int compareTo(Angle other) {
    if (degrees > other.degrees) {
        return 1;
    } else if (degrees < other.degrees) {
        return -1;
    } else {
        return 0;
    }
}
```

    - Second version

```
public int compareTo(Angle other) {
    return degrees - other.degrees;
}
```

    - But this isn't quite right, because what if the degrees are the same but the minutes are different?

```
public int compareTo(Angle other) {
    if (degrees == other.degrees)
        return minutes - other.minutes;
    else
        return degrees - other.degrees;
}
```

    - (run the code)
    - Tomorrow in section, more Comparable practice
    - On the final, you'll have to write a Comparable class - practice!

- Another application of Comparable: IntTree --> SearchTree<E>
  - I want to transform my IntTree into a tree capable of storing a binary search tree of any type of sortable data (e.g. String, Angle...)
  - Obviously this is better than writing a separate tree for each type - e.g. a StringTree, AngleTree...
  - Programming generics is tricky
    - I'm showing you how it's done, but I don't expect you to be able to do this on your own
- IntTreeNode --> SearchTreeNode<E>

```
public class SearchTreeNode<E> {
    public E data;
    public SearchTreeNode<E> left;
    public SearchTreeNode<E> right;

    public SearchTreeNode(E data) {
        this(data, null, null);
    }

    public SearchTreeNode(E data, SearchTreeNode<E> left,
                          SearchTreeNode<E> right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
}
```

- Then modify the IntTree - find/replace to change all IntTreeNode to SearchTreeNode<E>
- We also have to update the add() method
  - If we replace int with E and the switch out the node type, we're almost there
  - But we can no longer do this:

```
else if (value <= root.data)
```

  - So we have to change it to this:

```
else if (value.compareTo(root.data) <= 0)
```

- There's one more change: this won't compile
  - Problem is this: how does Java know that you can call compareTo on the value?
  - Well, we can assume that this is the case, and do a cast:

```
else if (((Comparable<E>) value).compareTo(root.data) <= 0)
```

  - If the client uses the SearchTree with an uncomparable value, it's their fault
  - This is what Sun does most of the time
  - Another approach: tell Java that the "E" type has a constraint, that it must implement Comparable

```
public class SearchTree<E extends Comparable<E>> {
```

  - Weird that it uses the extends keyword, but that's Java