`Comparable` interface (non-generic i.e. pre-Java 1.5)

Java has many pre-defined interfaces. One example that you will many times is `Comparable`. This interface has a single method `compareTo()` which is used to compare one object to another. Prior to Java 1.5 (prior to 2004), the `Comparable` interface was defined:

```
public interface Comparable{
   public int  compareTo(Object o);
}
```

Notice that the `compareTo()` method returns an integer. If you have a class `A` that implements `Comparable` and you have reference variables:

```
A  a1, a2;
```

then `a1.compareTo(a2)` should return:

- a negative integer, if the the object referenced by `a1` is "less than" the object referenced by `a2`,
- 0, if the object referenced by `a1` "equals" the object referenced by `a2` (in particular, `a1.compareTo(a1)` is 0).
- a positive integer, if the object referenced by `a1` is "greater than" the object referenced by `a2`.

Note: `compareTo()` should throw a NullPointerException if `a1` or `a2` is null.

Example: `Rectangle`

Let's define a Java class that implements this interface, namely `Rectangle`. Let `Rectangle` have two fields `height` and `width`, along with getters and setters and two other methods `getArea()` and `getPerimeter()`.

Note that there is only a single `compareTo()` method with the above signature in a class, and so we have to decide how to implement this method in `Rectangle`. We could compare rectangles by the values of their widths, or their heights, their areas or perimeters, etc. In the code below, we compare by area.

```
public class Rectangle implements Comparable{
   private width;
   private height;

   public Rectangle(double width, double height){
      this.width = width;
      this.height = height;
   }

   //   add getters and setters here
```

```
     public double getArea(){
         return width * height;
     }

     public int  compareTo(Object o) {
        if (this.getArea() > ( (Rectangle) o ).getArea())
           return 1;
        else if (this.getArea() < ( (Rectangle) o ).getArea())
           return -1;
        else return 0;
     }

     public double getPerimeter(){
         return  2*(width + height);
     }
   }

public class Test{
   public static void main(String() args){
      Rectangle  r1 = Rectangle(4.0,5.0);
      Rectangle  r2 = Rectangle(2.0,9.0);
      System.out.println("result: " + r1.compareTo(r2))

// would print "result: 1"  since 20.0 > 18.0
   }
}
```

generic `Comparable` interface (Java 1.5 and later)

The above is annoying since you need to downcast from `Object` to `Rectangle` in order to apply the interface method. Java 1.5 gives us a mechanism to avoid this casting, by defining *generic interfaces*. For example, rather than defining the interface `Comparable` so that the method `compareTo()` is expecting an `Object` argument, instead we let the type of the argument be *generic*. This is done as follows.

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

The `T` is arbitrary and stands for "type". It doesn't have to be a `T`, but by convention programmers use a single capital letter for the generic type. In other examples later we will use `<E>` which stands for "element" (of a list).

We can now re-write the `Rectangle` class using the generic type:

```
  public class Rectangle implements Comparable<Rectangle>{
            :
            :   //  same basic definitions as before
            :
     public int  compareTo(Rectangle r) {
        if (getArea() > r.getArea() )
           return 1;
        else if (getArea() < r.getArea() )
           return -1;
        else return 0;
     }
            :
            :
}
```

`Comparator` interface

Suppose we would like to be more flexible and have several different ways to compare `Rectangle`'s. We might compare them by area, or by perimeter, or by the minimum of their height and width, etc. The above scheme doesn't quite allow us to do this, since there is only one `compareTo` method in `Rectangle`. Java has another generic interface which we can use to solve this problem:

```
public interface Comparator<T>{
   compare(T  first,  T second);
}
```

Notice that the method here takes two arguments of type `T`, rather than one.

We next consider a method for comparing two `Rectangle` objects. This method is not defined in the `Rectangle` class, however. Rather, this method is defined in its own class and this class implements `Comparator`. This will seem very strange to you at first glance.

```
import java.util.Comparator;  // specifies package containing Comparator

public class CompareRectangleArea implements Comparator<Rectangle>{

   public int compare(Rectangle r1, Rectangle r2) {
     double diff = r1.getArea() - r2.getArea();
     if (diff < 0)
       return -1;
     else if (diff == 0)
       return 0;
     else return 1;
   }
}
```

Obviously you could write another class `ComparеRectanglePerimeter` which compares two rectangles by their perimeter.

In order for the method `compare()` to be used, we need to instantiate the class `CompareRectangleArea`, i.e. create an object. Notice that this object has no fields and only one method. (Strange but true.) How could this be used?

Suppose we were to modify the `Test` class on page 2 as follows

```
public class Test{

   public static void main(String() args){
      Rectangle  r1 = Rectangle(4.0, 5.0);
      Rectangle  r2 = Rectangle(1.5, 9.0);

      CompareRectangleArea cmpArea = new CompareRectangleArea();
      System.out.println("result: ", cmpArea.compare(r1,r2);

      //  would print "result: 1"  since 20 > 13.5

      CompareRectanglePerimeter   cmpPerimeter = new CompareRectanglePerimeter();
      System.out.println("result: ", cmpArea.compare(r1,r2);

      //  would print "result: -1"  since 18 < 21
   }
}
```

Collections.sort()

There is a public class `Collections` which contains many static methods. This is similar to the `Math` class which has static methods such as `sqrt`, `sine`, etc. These static methods can be called "out of the blue" when you need them.

One such method in the `Collections` class is `sort()`. It has two parameters: the first is an interface `List` and the second is a `Comparator`. For example, the Java classes `LinkedList` and `ArrayList` both implement the interface `List`.

Suppose we were to make an object which is a list of `Rectangle`'s. We can sort the objects in this list by calling:

```
  List<Rectangle>   listOfRectangles;
    :
    :    //  insert code here for creating the list
    :
  Collections.sort( listOfRectangles,   cmpArea);
```

You will need to use this in Assignment 2.