Homework 6
Heaps and Sorting

Goal: To code one of the primary n log n sorts and to code and use a binary max heap. You will also use the Comparator interface as opposed to Comparable.

Comparator Interface:
Up until now, we have always used the Comparable interface to store things in binary trees, or do other manipulations that require making comparisons between two things. While the Comparable interface is nice, it does have one primary limitation : you can only compare against one thing. What if we want to sort employees by their id number, or their address, or their email? This is where the Comparator interface comes in. We normally just implement a single method named compare that handles the comparison. Here is an example the would sort by employee numbers:

```
public class EmployeeNumberComparator implements Comparator<Employee> {
    public int compare (Employee e1, Employee e2 ) {
        return e1.getNumber() - e2.getNumber();
    }
}
```

Many algorithms in the java libraries allow you to pass in a comparator. For example, when you create a TreeSet, there is a constructor that accepts a Comparator. The tree will use that Comparator to enter things into the tree. In that case, the elements of the tree do not need to implement comparable. The built-in sorting algorithms in Arrays and Collections accept Comparators so we can search on different items.   We will be coding several different Comparator implementations later in the assignment.

General:
Imagine you are running a shop that provides print services. You have 5 printers and your goal is to get as many jobs completed as possible. There are many algorithms to try: Shortest Job first, Longest Job first, First Come First served, etc. Our goal will be to implement several data structures to support evaluating each of these algorithms.

Provided Files:
You are provided several files:

a. WorkOrder.java This class represents a single work order with fields that you will use to implement your comparators from. Your implemented data structures should all hold work orders. There is not code you need to add to this class.

b. OrderProcessing.java This class represents the shop with a number of machines (actually 5 for this assignment. You need to pass the constructor an integer which will be the number of randomly generated work orders to use for the simulation.   Call *runSimulation()* to start the simulation. The runSimulation method takes one parameter – the container to use for the simulation. You should not need to add code to this class unless you want to add additional debug output.

c. Container.java This is the interface definition that you will use for the various containers that you will implement. The methods are described in the interface. Note that the setComparator and arrange methods may not be needed by every container (for instance by queues). You are not allowed to modify this interface in any way.

d. OrderProcessingTest A simple JUnit class to test out parts of the simulation and print some output. Feel free to add any tests you like to this file. Note this is not actually using any asserts, so the green bar will actually not tell you anything except you ran to completion. You will need to check the console output and see if the information there is sorted correctly.

Implementation:
You will implement the following Comparators for the WorkOrder class:

1. NameComparator.java Compare based on the WorkOrder name.
2. HoursComparator.java Compare based on the WorkOrder hours.
3. PriorityComparator.java Compare based on the WorkOrder priority.

You will implement the following Containers:

1. Heap.java This class should implement a Max Heap. Using it in the simulation will be Longest Job first since next() will return the biggest item in the heap if you are using the HoursComparator. The arrange() method will run make_heap on any data that has already been entered.
2. SortedList.java This class should just add information to an array (note you will need to make this array dynamically expand if necessary when adding to the array). When you call arrange( ), the array should be sorted with quick sort. If the array or a partition has ten or fewer items, then use insertion sort. When choosing the pivot, use the median of three technique we talked about in class. When used with the HoursComparator, this should allow you to simulate shortest job first. When used with the PriorityComparator it should allow you to simulate highest priority job first.
3. Queue.java This class should add information to a standard queue. This will allow you to simulate handling jobs on a first-come, first-served basis. For this class, the setComparator and the arrange methods do nothing as they have no meaning for a queue. The backing store for the queue may be the LinkedList<> class in the Java Collections.

The standard way to run the simulation is:

1. Create the instance of OrderProcessing passing the number of workorders to the constructor.
2. Create the Container you want to use (Heap, Queue, or SortedList)
3. Set the comparator for the container — this is not required for Queue.
4. Call the runSimulation method passing in your container as the parameter. If you want more output, you may add other print statements, but comment them out before turning in the code.
5. You may NOT use any java implementations of the actual data structures: heap, queue, sorted list. You may use the List<> and LinkedList<> collections as backing stores for your queue container. The only method you may use from Collections is shuffle to randomize the data.

Turn-In:
Turn in the following files:
Heap.java
Queue.java
SortedList.java
HoursComparator.java
NameComparator.java
PriorityComparator.java

Grading:

Comparators	
Hours	10
Name	10
Priority	10
Containers	
Heap	25
SortedList	25
Queue	10
Code Quality	10