CSC-223
Lab 3
Work in a Group of 3

Geometry Fact Computer Representation: Linear Equivalence Class Structure

Given a set of items S, we refer to an *equivalence class* as the subset of S which includes all elements that are equivalent to each other. For example, let $S = [1, 10]$ (the inclusive integers from 1 to 10). We might define a rule stating that all even integers are 'equivalent', similarly for odd integers. Thus S could be partitioned into two *equivalence classes*, one containing all even integers and one containing all odd integers: $\{2, 4, 6, 8, 10\}$ and $\{1, 3, 5, 7, 9\}$, respectively.

Our goal for this assignment is to represent a set of equivalence classes in code: `EquivalenceClasses`. This requires that we implement some supporting class functionality: `LinkedEquivalenceClass` and `LinkedList`.

What You Need to Do: `LinkedList` Implementation

Even though the Java API provides a `LinkedList` class, the linked list is a fundamental data structure that everyone should implement (at least a few times) in their career. Implement a `LinkedList` class according to the following UML diagram; the class should be defined with a sentinel head node and a sentinel tail node and also define a private inner class `Node` similar to what was discussed in class.

```
                LinkedList<T>
-----------------------------------------------
#_head            : Node<T>
#_tail            : Node<T>
#_size            : int
-----------------------------------------------
+LinkedList()
+isEmpty()                  : boolean
+clear()                    : void
+size()                     : int
+addToFront(T element)      : void
+contains(T target)         : boolean
-previous(T target)         : Node
+remove(T target)           : boolean
-last()                     : Node
+addToBack(T element)       : void
+toString()                 : String
+reverse()                  : void
```

All methods should be implemented as linear-time algorithms or better. For practice, we recommending implementing methods using recursion (e.g., `contains`, `previous`, `toString`). The `reverse` method must be implemented as a linear time (and linear space due to use of the call stack) algorithm that reverses the list in place. *Hint: use recursion.*

For junit testing, it will benefit you to use the `toString` method for the `LinkedList`. This will facilitate easier testing of the internal values in the `LinkedList`. For example,

```
LinkedList<Integer> list = new LinkedList<Integer>();
list.addToFront(2);
assertEquals("2", list.toString());
```

What You Need to Do: `LinkedEquivalenceClass` Implementation

As described in our initial example above, an equivalence class is a means of storing 'equivalent' objects. However, in practice, for an equivalence class we often choose of the values to be a representative (a.k.a. canonical) element that represents all elements of the equivalence class. Using our previous example of even integers in the interval $[1, 10]$, we might have 2 be the representative element: $\{2 \mid 4, 6, 8, 10\}$ although the choice of 2 is arbitrary.

We will represent each equivalence class as having a canonical element along with all other elements in a linked list as our example depicts in Figure 1.

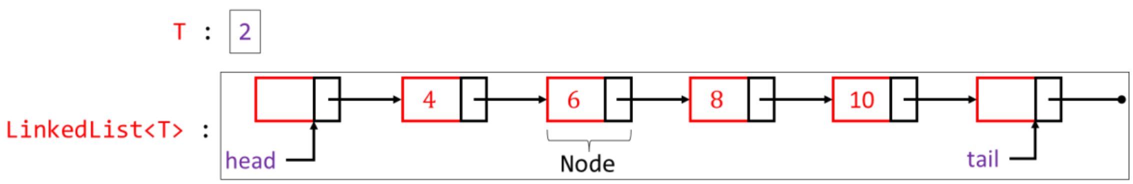

Figure 1: Representing an equivalence class containing $\{2 \mid 4, 6, 8, 10\}$ in which the `Comparator` determines all even (similarly, odd) integers are equivalent.

Your task is to implement a `LinkedequivalenceClass` class according to the following UML diagram.

LinkedEquivalenceClass<T>	
#_canonical	: T
#_comparator	: Comparator<T>
#_rest	: LinkedList<T>
+LinkedEquivalenceClass(Comparator<T>)	
+canonical()	: T
+isEmpty()	: boolean
+clear()	: void
+clearNonCanonical()	: void
+size()	: int
+add(T element)	: boolean
+contains(T target)	: boolean
+belongs(T target)	: boolean
+remove(T target)	: boolean
+removeCanonical()	: boolean
+demoteAndSetCanonical(T element)	: boolean
+toString()	: String

Comparators. As is evident in the UML diagram above, we are using a Comparator object to determine if elements *belong* in the same equivalence class. Recall that `Comparator<T>` is an interface that requires an implementing class to define a `compare(T o1, T o2)` method that will return 0 if `o1` is equivalent to `o2`. For example, $2 \equiv 4 \equiv 6 \equiv 8 \equiv 10$ thus `compare(4, 10) == 0`.

The `belongs` method in the UML diagram above will test whether an element is equivalent to the canonical element (and not the rest of the elements) while the `contains` method will initially check equivalence with the canonical element and subsequently check equality (with `equals`) against all elements in the equivalence class (including the canonical element). Although it may seem redundant, it must be repeated that equivalence among element (in the equivalence class setting using Comparator) is ***not*** the same as object equality (using an `equals` method).

For testing purposes, you might find it helpful to define an anonymous class object for whatever `Comparator` you wish to define. For example, the following code implements a simple partitioning of odd and even integer elements. *You must define more elaborate comparators for testing purposes.*

```
Comparator<Integer> c = new Comparator<Integer>()
                        {
                            // All even integers are 'equivalent'
                            // All odd integers are 'equivalent'
                            public int compare(Integer x, Integer y)
                            { return x % 2 == y % 2 ? 0 : 1; }
                        };
```

This code can easily be modified to partition integers into as many equivalence classes as desired.

What You Need to Do: `EquivalenceClasses` Implementation

Our goal is to store many equivalence classes. We will do so using an `ArrayList`, a naïve data structure in terms of runtime efficiency in this setting; see the UML class diagram below for implementation requirements.

`EquivalenceClasses<T>`	
`#_comparator`	`: Comparator<T>`
`#_rest`	`: List<LinkedEquivalenceClass<T>>`
`+EquivalenceClasses(Comparator<T>)`	
`+add(T element)`	`: boolean`
`+contains(T target)`	`: boolean`
`+size()`	`: int`
`+numClasses()`	`: int`
`#indexOfClass(T element)`	`: int`
`+toString()`	`: String`

Again, note that this class requires the user to define a `Comparator` object; this object will be the same object passed into each `LinkedEquivalenceClass` object.

Commenting

Comment well. See old lab instructions for details.

Submitting: Source Code

For this lab, you will demonstrate your source code to the instructor, in person. Be ready to demonstrate (1) successful execution of all junit tests and (2) the github repository which includes commented source code (see above) and a clear commit history with meaningful commit messages *from all group members*.