1 Filtered List

We want to make a FilteredList class that selects only certain elements of a List during iteration. To do so, we’re going to use the Predicate interface defined below. Note that it has a method, test that takes in an argument and returns True if we want to keep this argument or False otherwise.

```java
public interface Predicate<T> {
    boolean test(T x);
}
```

For example, if L is any kind of object that implements List<String> (that is, the standard java.util.List), then writing

```java
FilteredList<String> FL = new FilteredList<>(L, filter);
```

gives an iterable containing all items, x, in L for which filter.test(x) is True. Here, filter is of type Predicate. Fill in the FilteredList class below.

```java
import java.util.*;
public class FilteredList<T> {
    public FilteredList(List<T> L, Predicate<T> filter) {
    }
    @Override
    public Iterator<T> iterator() {
    }
}
```
Solution:

```java
import java.util.*;

class FilteredList<T> implements Iterable<T> {
    List<T> list;
    Predicate<T> pred;

    public FilteredList(List<T> L, Predicate<T> filter) {
        this.list = L;
        this.pred = filter;
    }

    public Iterator<T> iterator() {
        return new FilteredListIterator();
    }

    private class FilteredListIterator implements Iterator<T> {
        int index;

        public FilteredListIterator() {
            index = 0;
            moveIndex();
        }

        @Override
        public boolean hasNext() {
            return index < list.size();
        }

        @Override
        public T next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            T answer = list.get(index);
            index += 1;
            moveIndex();
            return answer;
        }

        private void moveIndex() {
            while (hasNext() && !pred.test(list.get(index))) {
                index += 1;
            }
        }
    }
}
```
Alternate Solution: Although this solution provides the right functionality, it is not as efficient as the first one. Imagine you only want the first couple items from the iterable. Is it worth processing the entire list in the constructor? It is not ideal in the case that our list is millions of elements long. The first solution is different in that we “lazily” evaluate the list, only progressing our index on every call to `next` and `hasNext`. However, this solution may be easier to digest.

```java
import java.util.*;

class FilteredList<T> implements Iterable<T> {
    List<T> list;
    Predicate<T> pred;

    public FilteredList(List<T> L, Predicate<T> filter) {
        this.list = L;
        this.pred = filter;
    }

    public Iterator<T> iterator() {
        return new FilteredListIterator();
    }

    private class FilteredListIterator implements Iterator<T> {
        LinkedList<T> items;

        public FilteredListIterator() {
            items = new LinkedList<>();
            for (T item: list) {
                if (pred.test(item)) {
                    items.add(item);
                }
            }
        }

        @Override
        public boolean hasNext() {
            return !items.isEmpty();
        }

        @Override
        public T next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            return items.removeFirst();
        }
    }
}
```
2 Iterator of Iterators

Implement an IteratorOfIterators which will accept as an argument a List of Iterator objects containing Integers. The first call to next() should return the first item from the first iterator in the list. The second call to next() should return the first item from the second iterator in the list. If the list contained n iterators, the n+1th time that we call next(), we would return the second item of the first iterator in the list.

Note that if an iterator is empty in this process, we continue to the next iterator. Then, once all the iterators are empty, hasNext should return false. For example, if we had 3 Iterators A, B, and C such that A contained the values [1, 3, 4, 5], B was empty, and C contained the values [2], calls to next() for our IteratorOfIterators would return [1, 2, 3, 4, 5].

```java
import java.util.*;

public class IteratorOfIterators {

    public IteratorOfIterators(List<Iterator<Integer>> a) {
    
    }

    @Override
    public boolean hasNext() {
    
    }

    @Override
    public Integer next() {
    
    }
}
```
Solution:

```java
public class IteratorOfIterators implements Iterator<Integer> {
    LinkedList<Iterator<Integer>> iterators;

    public IteratorOfIterators(List<Iterator<Integer>> a) {
        iterators = new LinkedList<>();
        for (Iterator<Integer> iterator : a) {
            if (iterator.hasNext()) {
                iterators.add(iterator);
            }
        }
    }

    @Override
    public boolean hasNext() {
        return !iterators.isEmpty();
    }

    @Override
    public Integer next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        Iterator<Integer> iterator = iterators.removeFirst();
        int ans = iterator.next();
        if (iterator.hasNext()) {
            iterators.addLast(iterator);
        }
        return ans;
    }
}
```
Alternate Solution: Although this solution provides the right functionality, it is not as efficient as the first one.

```java
public class IteratorOfIterators implements Iterator<Integer> {
    LinkedList<Integer> l;

    public IteratorOfIterators(List<Iterator<Integer>> a) {
        l = new LinkedList<>();
        while (!a.isEmpty()) {
            Iterator<Integer> curr = a.remove(0);
            if (curr.hasNext()) {
                l.add(curr.next());
                a.add(curr);
            }
        }
    }

    @Override
    public boolean hasNext() {
        return !l.isEmpty();
    }

    @Override
    public Integer next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        return l.removeFirst();
    }
}
```
3 DMS Comparator

Implement the Comparator DMSComparator, which compares Animal instances. An Animal instance is greater than another Animal instance if its dynamic type is more specific. See the examples to the right below.

In the second and third blanks in the compare method, you may only use the integer variables predefined (first, second, etc), relational/equality operators (==, >, etc), boolean operators (&& and ||), integers, and parentheses.

As a challenge, use equality operators (== or !=) and no relational operators (>,-, etc). There may be more than one solution.

```
public class DMSComparator implements ____________________ {

    @Override
    public int compare(Animal o1, Animal o2) {
        int first = o1.speak(new Animal());
        int second = o2.speak(new Animal());
        int third = o1.speak(new Dog());
        int fourth = o2.speak(new Dog());

        if (________________________________________________________) {
            return 0;
        }
       
        } else if (_________________________________________________) {
            return 1;
        } else {
            return -1;
        }
    }
}
```

Examples:

```
Animal animal = new Animal();
Animal dog = new Dog();
Animal poodle = new Poodle();

compare(animal, dog) // negative number
compare(dog, dog) // zero
compare(poodle, dog) // positive number
```
Solution:

```java
public class DMSComparator implements Comparator<Animal> {

    @Override
    public int compare(Animal o1, Animal o2) {
        int first = o1.speak(new Animal());
        int second = o2.speak(new Animal());
        int third = o1.speak(new Dog());
        int fourth = o2.speak(new Dog());

        if (first == second && third == fourth) {
            return 0;
        } else if (first > second || third > fourth) {
            return 1;
        } else {
            return -1;
        }
    }
}
```

Challenge Solution:

```java
public class DMSComparator implements Comparator<Animal> {

    @Override
    public int compare(Animal o1, Animal o2) {
        int first = o1.speak(new Animal());
        int second = o2.speak(new Animal());
        int third = o1.speak(new Dog());
        int fourth = o2.speak(new Dog());

        if (first == second && third == fourth) {
            return 0;
        } else if (third == 4 || (first == 3 && second == 2)) {
            return 1;
        } else {
            return -1;
        }
    }
}
```