CS 61B Spring 2021 — Iterators and Iterables

Exam Prep Discussion 5: February 16, 2021

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.

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

L is a list of strings.

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

```
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.

```
1  import java.util.*;
2  public class FilteredList<T> implements Iterable<T> {
3      List<T> L;
4      Predicate<T> filter;
5      public FilteredList (List<T> L, Predicate<T> filter) {
6          this.L = L;
7
8          this.filter = filter;
9      }
10     @Override
11     public Iterator<T> iterator() {
12         return new FilteredListIterator();
13     }
14
15
16
17
18
19
20
21
22
23
24
25
26 }
```

```
private class FilteredListIterator<T> {
    private int index;
    private FilteredListIterator() {
        index = 0;
        move();
    }
    private boolean hasNext() {
        return index < L.size();
    }
    private void move() {
        while (hasNext() && !filter.test(L.get(index))) {
            index++;
        }
    }
    private T next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        T ans = L.get(index);
        index++;
        move();
        return ans;
    }
}
```

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+1`th 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]`.

```
1  import java.util.*;
2  public class IteratorOfIterators implements Iterator<Integer> {
3        LinkedList<Iterator<Integer>> iterators;
4
5      public IteratorOfIterators(List<Iterator<Integer>> a) {
6              iterators = new LinkedList<>;
7              for( Iterator<Integer> x : a ) {
8                    if ( x.hasNext() ){
9                       iterators.add(x);
10                   }
11
12
13     }          }
14
15     @Override
16     public boolean hasNext() {
17
18           return !iterators.isEmpty();
19
20
21     }
22
23           if(!hasNext()) {
24                 throw new NoSuchElementException();
25     @Override
26     public Integer next() {
27
28           Iterator<Integer> iterator = iterators.removeFirst();
29           Integer ans = iterator.next();
30
31     }     if( iterator.hasNext() ){
32 }             iterators.addLast(iterator);
              }
           return ans;
```

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.

do not use relational operators

```
class Animal {
   int speak(Dog a) { return 1; }
   int speak(Animal a) { return 2; }
}
class Dog extends Animal {
   int speak(Animal a) { return 3; }
}
class Poodle extends Dog {
   int speak(Dog a) { return 4; }
}
```

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
```

```
1  public class DMSComparator implements ____Comparator<Animal>____ {
2
3      @Override
4      public int compare(Animal o1, Animal o2) {
5          int first = o1.speak(new Animal());
6          int second = o2.speak(new Animal());
7          int third = o1.speak(new Dog());
8          int fourth = o2.speak(new Dog());
9
10         if (____first == second && third == fourth____) {
11             return 0;   // the same dynamic type
12
13         } else if (____third == 4 || (first == 3 && second == 2)____) {
14             return 1;
15         } else {
16             return -1;
17         }
18     }
19 }
```

dynamic type

parameter (static type)

invoking type (o1, o2)	Animal	Dog
Animal	2	1
Dog	3	1
Poodle	3	4

	Animal	Dog	Poodle
Animal	=	<	<
Dog	>	=	<
poodle	>	>	=