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);
}
```

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> ________________________________________ {
3
4
5      public FilteredList (List<T> L, Predicate<T> filter) {
6
7
8
9      }
10     @Override
11     public Iterator<T> iterator() {
12
13     }
14
15
16
17
18
19
20
21
22
23
24
25
26 }
```

Solution:

```
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.

```
1  import java.util.*;
2
3  class FilteredList<T> implements Iterable<T> {
4      List<T> list;
5      Predicate<T> pred;
6
7      public FilteredList(List<T> L, Predicate<T> filter) {
8          this.list = L;
9          this.pred = filter;
10     }
11
12     public Iterator<T> iterator() {
13         return new FilteredListIterator();
14     }
15
16     private class FilteredListIterator implements Iterator<T> {
17         LinkedList<T> items;
18
19         public FilteredListIterator() {
20             items = new LinkedList<>();
21             for (T item: list) {
22                 if (pred.test(item)) {
23                     items.add(item);
24                 }
25             }
26         }
27
28         @Override
29         public boolean hasNext() {
30             return !items.isEmpty();
31         }
32
33         @Override
34         public T next() {
35             if (!hasNext()) {
36                 throw new NoSuchElementException();
37             }
38             return items.removeFirst();
39         }
40     }
```

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  _____________________________ {
3
4
5      public IteratorOfIterators(List<Iterator<Integer>> a) {
6
7
8
9
10
11
12
13     }
14
15     @Override
16     public boolean hasNext() {
17
18
19
20
21     }
22
23
24
25     @Override
26     public Integer next() {
27
28
29
30
31     }
32 }
```

Solution:

```
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.

```
1  public class IteratorOfIterators implements Iterator<Integer> {
2      LinkedList<Integer> l;
3
4      public IteratorOfIterators(List<Iterator<Integer>> a) {
5          l = new LinkedList<>();
6          while (!a.isEmpty()) {
7              Iterator<Integer> curr = a.remove(0);
8              if (curr.hasNext()) {
9                  l.add(curr.next());
10                 a.add(curr);
11             }
12         }
13     }
14
15     @Override
16     public boolean hasNext() {
17         return !l.isEmpty();
18     }
19
20     @Override
21     public Integer next() {
22         if(!hasNext()) {
23             throw new NoSuchElementException();
24         }
25         return l.removeFirst();
26     }
27 }
```

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.

```
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 ____________________________ {
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 (_____________________________________________________) {
11             return 0;
12
13         } else if (_____________________________________________) {
14             return 1;
15         } else {
16             return -1;
17         }
18     }
19 }
```

Solution:

```
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:

```
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;
        }
    }
}
```