CS 61B
Fall 2025

Comparators, Iterators, and Iterables

Exam-Level Discussion 04: September 22, 2025

1 MetaComparison

Given `IntList x`, an `IntList y`, and a `Comparator<Integer> c`, the `IntListMetaComparator` performs a comparison between `x` and `y`.

Specifically, the `IntListMetaComparator` performs a pairwise comparison of all the items in `x` and `y`. If the lists are of different lengths, the extra items in the longer list are ignored. Let α be the number of items in `x` that are less than the corresponding item in `y` according to `c`. Let β be the number of items in `x` that are greater than the corresponding item in `y` according to `c`. If $\alpha > \beta$, then `x` is considered less than `y`. If $\alpha = \beta$, then `x` is considered equal to `y`. If $\alpha < \beta$, then `x` is considered greater than `y`. For example:

```
Comparator<Integer> c = new FiveCountComparator(); //compares # of fives
IntList x = [ 55, 70,  90, 115, 5];                //e.g. 55 has 2 fives
IntList y = [150, 35, 215,  25];
IntListMetaComparator ilmc = new IntListMetaComparator(c);
ilmc.compare(x, y); // returns negative number
```

For the example above, according to the `FiveCountComparator`, we have that $55 > 150$, $70 < 35$, $90 < 215$, and $115 = 25$. This yields $\alpha = 2$ and $\beta = 1$, and thus `ilmc.compare` will return a negative number. Fill in the code below:

```
public class IntListMetaComparator implements Comparator<IntList> {
    ------------------------------------

    public IntListMetaComparator(Comparator<Integer> givenC) {
        ------------------------------------
    }

    /* Returns negative number if more items in x are less,
       Returns positive number if more items in x are greater.
       If one list is longer than the other, extra items are ignored. */
    public int compare(IntList x, IntList y) {
        if ((____________________) || (_____________________)) {
            ------------------------------------------------
        }
        ------------------------------------------------------

        if (_________________________) {

            return ____________________________;

        } else if (_________________________) {

            return ___________________________________;
        } else {

            return ___________________________________;
        }
    }
}
```

2 Inheritance Syntax

Suppose we have the classes below:

```
public class ComparatorTester {
    public static void main(String[] args) {
        String[] strings = new String[] {"horse", "cat", "dogs"};
        System.out.println(Maximizer.max(strings, new LengthComparator()));
    }
}

public class LengthComparator implements Comparator<String> {
    @Override
    public int compare(String a, String b) {
        return a.length() - b.length();
    }
}

public class Maximizer {
    /**
     * Returns the maximum element in items, according to the given Comparator.
     */
    // public static <T> T max(T[] items, Comparator<T> c) {
    public static <String> String max(String[] items, Comparator<String> c) {
        ...
        int cmp = c.compare(items[i], items[maxDex]);
        ...
    }
}
```

(a) Suppose we omit the `compare` method from `LengthComparator`. Which of the following will fail to compile?

- ◯ `ComparatorTester.java`
- ◯ `LengthComparator.java`
- ◯ `Maximizer.java`
- ◯ `Comparator.java`

(b) Suppose we omit `implements Comparator<String>` in `LengthComparator`. Which file will fail to compile?

- ☐ `ComparatorTester.java`
- ☐ `LengthComparator.java`
- ☐ `Maximizer.java`
- ☐ `Comparator.java`

(c) Suppose we removed `@Override`. What are the implications?

3 Iterator of Iterators

Implement an **`IteratorOfIterators`** which takes in a **`List`** of **`Iterator`**s of **`Integers`** as an argument. The first call to **`next()`** should return the first item from the first iterator in the list. The second call 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]`**.

```
public class IteratorOfIterators ____________________________________________ {
    private List<Iterator<Integer>> iterators;
    private int curr;
    public IteratorOfIterators(List<Iterator<Integer>> a) {
        iterators = new LinkedList<>();
        for (_____________________________________________) {
            if (_____________________________________________) {
                ____________________________________________;
            }
        }
        curr = 0;
    }
    @Override
    public boolean hasNext() {
        return _____________________________________________;
    }
    @Override
    public Integer next() {
        if (!hasNext()) { throw new NoSuchElementException(); }
        Iterator<Integer> currIterator = ____________________________________________;
        int result = _____________________________________________;
        if (_____________________________________________) {
            ____________________________________________;
            if (iterators.isEmpty()) {
                ____________________________________________;
            }
        } else {
            curr = _____________________________________________;
        }
        return result;
    }
}
```