CS 61B Spring 2025 — Inheritance, Comparators, Generic Functions

Exam-Level 03: February 10, 2025

1 Default

Suppose we have a **MyStack** interface that we want to implement. We want to add two default methods to the interface: **insesrtAtBottom** and **flip**. Fill in these methods in the code below.

```
public interface MyStack<E> {
    void push(E element); // adds an element to the top of the stack
    E pop();              // removes and returns the top element of the stack
    boolean isEmpty();    // returns true if the stack is empty
    int size();           // returns the number of elements in the stack

    // inserts the item at the bottom of the stack using push, pop, isEmpty, and size
    private void insertAtBottom(E item) {

        version1
        Mystack<E> tem=new Mystack<>();
        while(!isEmpty){
          tem.push(pop());
        }
        push(item);
        while(!tem.isEmpty){
          push(tem.pop());
        }
    }

    // flips the stack upside down (hint: use insertAtBottom)
    default void flip() {

        if(isEmpty){
          return;
        }
        E tem=pop();
        flip();
        insertAtBottom(tem);



    }
}
```

it's wrong because a interface cannot be instanciated

```
version2(recursion)
if(isEmpty){
  push item;
  return;
}
E tem=pop();
inserAtBottom(item);
push(tem);
```

2 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> {
    _____int alpha=0, beta=0;_____________ private comparator<interger>  givenc;

    public IntListMetaComparator(Comparator<Integer> givenC) {
        ____________return givenc.compare(int a,int b);_____ this.givenc=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 ((_____x==null_____) || (_____y==null_____)) {
            _____return 0;_____________________________
        }
        _____compare(x.rest,y.rest);_____int compvalue=givenc(x.first, y.first);

        if (_givenc(x.first,y.first)>0_____) {    compvalue>0

            return __beta++_____return compare(x.rest, y.rest) + 1;

        } else if (_givenc(x.first,y.first)<0_____) { compvalue < 0

            return __alpha++_____return compare(x.rest, y.rest) - 1;
        } else {

            return __compare(x.rest, y.rest)_____;
        }
    }
}
```

3 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) {
        ...
        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?

works fine

(d) Suppose we changed where the type parameter appears so that the code in **`Maximizer`** looks like:

```
public class Maximizer<T> {
    public T max(T[] items, Comparator<T> c) {
        ...
```

What would change about the way we use **`Maximizer`**?

Maximizer<>()

have to instanciate first,eg.Maximizer<String> m=new Maximizer<>; m.max(,);

(e) Suppose we changed the method signature for **`max`** to read **`public static String max(String[] items, Comparator<String> c)`**. Would the code shown still work?

yes