CS 61B
Spring 2025

Inheritance, Comparators, Generic Functions

Discussion 03: February 10, 2025

1 It's a Bird! It's a Plane! It's a CatBus!

(a) On a research expedition studying air traffic, we discovered a new species: the Flying Interfacing **CatBus**, which acts like a vehicle and has the ability to make noise (safety is important!).

Given the **Vehicle** and **Noisemaker** interfaces, fill out the **CatBus** class so that **CatBus**es can rev their engines and make noise at other **CatBus**es with a **CatBus**-specific sound.

```
interface Vehicle {
    public void revEngine();
}

interface Noisemaker {
    public void makeNoise();
}

public class CatBus ___implements___ ___Vehicle___, ___Noisemaker___ {
    @Override
    ___public void revEngine();___ { /* CatBus revs engine, code not shown */ }

    @Override
    ___public void makeNoise();___ { /* CatBus makes noise, code not shown */ }

    /** Allows CatBus to make noise at other CatBuses. */
    public void conversation(CatBus target) {
        makeNoise();
        target.makeNoise();
    }
}
```

(b) It's a lovely morning in the skies and we've encountered a horrible **Goose**, which also **implements Noisemaker** (it has a knife in its beak!). Modify the **conversation** method signature so that **CatBus**es can **makeNoise** at both **CatBus** and **Goose** objects while only having one argument, **target**.

```
public void conversation(Noisemaker target) {
 makeNoise();
   target.makeNoise();
 }
```

2 Default

(a) Suppose we have a **MyQueue** interface that we want to implement. We want to add two default methods to the interface: **clear**, **remove** and **max**. Fill in these methods in the code below.

```
public interface MyQueue<E> {
    void enqueue(E element); // adds an element to the end of the queue
    E dequeue();             // removes and returns the front element of the queue
    boolean isEmpty();       // returns true if the queue is empty
    int size();              // returns the number of elements in the queue

    // removes all items from the queue
    default void clear() {
        while(!isEmpty()){
          dequeue();
        }

    }

    // removes all items equal to item from the queue
    // the remaining items should be in the same order as they were before
    default void remove(E item) {
            Myqueue queue=new Myqueue;
            while(!isEmpty){
              E temp=dequeue();
              if(temp!=item){
                queue.enqueue(temp);
              }
            }
            while(!queue.isEmpty){
               enqueue(queue.dequeue);
    }       }
```

```
default void remove(E item) {
 int removed = 0;
 int currSize = size();
 while (removed < currSize) {
 E currItem = dequeue();
 if (!currItem.equals(item)) {
 enqueue(currItem);
}
      removed++;
 }
 }
```

```
    // returns the maximum element in the queue according to the comparator
    // the items in the queue should be in the same order as they were before
    // assume the queue is not empty
    default E max(Comparator<E> c) {

            E max=dequeue();
            int num=1,size=size();
            while(num<size){
               （不会）
            }

    }
}
```

```
default E max(Comparator<E> c) {
 int removed = 0;
 int currSize = size();
 E currMax = null;
 while (removed < currSize) {
 E currItem = dequeue();
 if (currMax == null) {
         currMax = currItem;
 } else if (c.compare(currItem, currMax) > 0) {
         currMax = currItem;
 }
 enqueue(currItem);
      removed++;
 }
 return currMax;
 }
```

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`

because it is claiming to be a Comparator, but it is missing a compare method

If a class implements an interface, it must override all the methods declared in that interface.

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

○ [✓ `ComparatorTester.java`]

○ [✓ `LengthComparator.java`]

○ `Maximizer.java`

○ `Comparator.java`

ComparatorTester, because we are trying to provide a LengthComparator (which isn’t a Comparator) to the method max, which expects a Comparator.

LengthComparator, because compare is no longer overriding anything, thus causing the @Override to trigger a compiler error.

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

it's 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**?

new maximizer.max(,)

We'd have to instantiate a Maximizer object to use it, e.g.
Maximizer<String> m = new Maximizer<>();
This isn't as nice

(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?

it will not work, but it will not if it's T[] instead of String[].