CSM 61B
Spring 2020

Iterators & Exceptions
Mentoring 5: September 21-25, 2020

1 Iterator Interface

In Java, an **iterator** is an object which allows us to traverse a data structure in linear fashion. Every iterator has two methods: `hasNext` and `next`.

```
interface IntIterator {
    boolean hasNext();
    int next();
}
```

1.1 Consider the following code that demonstrates the `IntArrayIterator`.

```
int[] arr = {1, 2, 3, 4, 5, 6};
IntIterator iter = new IntArrayIterator(arr);
if (iter.hasNext()) {
    System.out.println(iter.next());     // 1
}
if (iter.hasNext()) {
    System.out.println(iter.next() + 3); // 5
}
while (iter.hasNext()) {
    System.out.println(iter.next());     // 3 4 5 6
}
```

Define an `IntArrayIterator` class that works as described above.

```
public class IntArrayIterator __________________ IntIterator {



    public IntArrayIterator(int[] arr) {




    }

    public boolean hasNext() {







    }

    public int next() {







    }
}
```

1.2 Define an `IntListIterator` class that adheres to the `IntIterator` interface.

1.3 Define a method, `printAll`, that prints every element in an `IntIterator` **regardless of how the iterator is implemented.**

2 Insects

2.1 What would Java display for the following?

```
class Insect {
    public void stay() {
        System.out.println("Staying...");
    }
    public void speak() {
        System.out.println("I am an insect");
    }
}
class Ant extends Insect {
    @Override
    public void speak() {
        System.out.println("I am an ant");
    }
    public void attack() {
        System.out.println("Ant attacked");
    }
}
class Bee extends Insect {
    @Override
    public void speak() {
        System.out.println("I am a bee");
    }
    public void move() {
        System.out.println("Bee moved");
    }
}
Insect i = new Insect();
i.speak();
Ant a = new Ant();
a.speak();
Bee b = new Bee();
b.speak();
i = new Ant();
i.speak();
i.attack();
((Ant) i).attack();
b = new Insect();
b.speak();
b.move();
```

3 Multiples

3.1 Define a procedure, `multiples`, that returns an `SLList` containing the elements at indices `k`, `k + j`, `k + 2*j`, and so forth to the end of the list.

```
public class SLList {
    private IntNode sentinel;
    private static class IntNode {
        public int value;
        public IntNode next;
        public IntNode(int value, IntNode next) {
            this.value = value;
            this.next = next;
        }
    }
    public SLList() {
        this.sentinel = new IntNode(-1, null);
    }
    public SLList multiples(int k, int j) {









}
```

4 Generic

4.1 A normal generic linked list contains objects of only one type. But we can imagine a generic linked list where entries alternate between two types.

```
public class AltList<X,Y> {
    private X item;
    private AltList<Y,X> next;
    AltList(X item, AltList<Y,X> next) {
        this.item = item;
        this.next = next;
    }
}

AltList<Integer, String> list =
    new AltList<Integer, String>(5,
        new AltList<String, Integer>("cat",
            new AltList<Integer, String>(10,
                new AltList<String, Integer>("dog", null))));
```

This list represents `[5, cat, 10, dog]`. In this list, assuming indexing begins at 0, all even-index items are `Integers` and all odd-index items are `Strings`.

Write an instance method called `pairsSwapped()` for the `AltList` class that returns a copy of the original list, but with adjacent pairs swapped. Each item should only be swapped once. This method should be non-destructive: it should not modify the original `AltList` instance. Assume that the list has an even, non-zero length.

For example, calling `pairsSwapped()` on the list `[5, cat, 10, dog]` should yield the list `[cat, 5, dog, 10]`.

```
public class AltList<X,Y> {
    public pairsSwapped() {




    }
}
```