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 implements IntIterator {
    private int index;
    private int[] array;
    public IntArrayIterator(int[] arr) {
        array = arr;
        index = 0;
    }
    public boolean hasNext() {
        return index < array.length;
    }
    public int next() {
        int value = array[index];
        index += 1;
        return value;
    }
}
```

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

```
public class IntListIterator implements IntIterator {
    private IntList node;
    public IntListIterator(IntList list) {
        node = list;
    }
    public boolean hasNext() {
        return node != null;
    }
    public int next() {
        int value = node.first;
        node = node.rest;
        return value;
    }
}
```

Meta: This can be gone through and explained faster since the students just implemented `IntArrayIterator`.

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

```
public static void printAll(IntIterator iter) {
    while (iter.hasNext()) {
```

```
        System.out.println(iter.next());
    }
}
```

Meta: In general, for all parts of this question, give students time to work on the problem as well as prompting them to think about what they need in order to start them off. (For example, what do you need to do to implement the `IntIterator` interface? What do you need to know for `hasNext` to work?)

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();                  "I am an Insect"
Ant a = new Ant();
a.speak();                  "I am an Ant"
Bee b = new Bee();
b.speak();                  "I am a Bee"
i = new Ant();
i.speak();                  "I am an Ant"
i.attack();                 Compile Time Error
((Ant) i).attack();           "Ant attacked"
b = new Insect();           Compile Time Error
b.speak();                  Not possible
```

```
b.move();                    Not possible
```

In the first six lines, the static and dynamic types of i, a, and b are the same, so when they speak they just call the method in the dynamic class. Once i gets assigned to a new Ant(), it now has static type Insect and dynamic type Ant. During compile time for i.speak(), Java look for the `speak()` method in `Insect`. Then during runtime, it'll execute the `speak` method in `Ant` class. During compile time for i.attack(), Java will look for the `attack()` method in `Insect`. Since Java cannot find such method in `Insect`, this will result in a compile time error. By casting i to an `Ant`, Java will look at the `Ant` class during compile time and find the `attack()` method. Since b is static type `Bee`, it cannot be assigned to a new `Insect` because `Bee` is a subclass of `Insect`. The rest of the lines depend on the $b = newInsect()$ line, so they won't be possible.

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) {
        IntNode node = this.sentinel.next;
        for (int i = 0; i < k && node != null; i++) {
            node = node.next;
        }
        SLList multiples = new SLList();
        IntNode target = multiples.sentinel;
        while (node != null) {
            target.next = new IntNode(node.value, null);
            target = target.next;
            for (int i = 0; i < j && node != null; i++) {
                node = node.next;
            }
        }
        return multiples;
    }
}
```

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 AltList<Y,X> pairsSwapped() {
        AltList<Y,X> ret = new AltList<Y,X>(next.item, new AltList<X,Y>(item, null));
        if (next.next != null) {
            ret.next.next = next.next.pairsSwapped();
        }
        return ret;
    }
}
```