CSM 61B Spring 2018

Abstract Classes & Iterators

Mentoring 4: February 14, 2018

1 An Appealing Appetizer

1.1
```
public interface Consumable {
    public void consume();
}
public abstract class Food implements Consumable {
    String name;
    public abstract void prepare();
    public void play() {
        System.out.println("Mom says, 'Don't play with your food.'");
    }
}
public class Snack extends Food {
    public void prepare() {
        System.out.println("Taking " + name + " out of wrapper");
    }
    public void consume() {
        System.out.println("Snacking on " + name);
    }
}
```

(a) Compare and contrast interfaces and abstract classes.

(b) Do we need the `play` method in `Snack`?

(c) Does this compile? `Consumable chips = new Snack();`

2 Generics

2.1 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() {









    }
}
```

3 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();
}
```

3.1 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.

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

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