CSM 61B
Spring 2018

Abstract Classes & Iterators

Mentoring 4: February 14, 2018

Solutions

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.

- Java classes cannot extend multiple superclasses (unlike Python) but classes can implement multiple interfaces.
- Interfaces are implicitly public.
- Interfaces can't have fields declared as instance variables; any fields that are declared are implicitly **`static`** and **`final`**.
- Interfaces use the **`default`** keyword to declare *concrete* implementations while abstract classes use the **`abstract`** keyword to declare *abstract* implementations.
- Interfaces define the way we interact with an implementing object or functions of an object. Conversely, abstract classes define an "is-a" relationship and tell us more about the object's fundamental identity.

Meta: Warn students to be careful searching about this topic as CS 61B uses Java 8 but most content online covers Java 7, which behaves differently.

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

No, we do not need the `play` method because it's already defined in the abstract class. Java will lookup the parent class's method if it cannot find it in the child class.

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

Yes, the code compiles since `Snack` inherits from the `Food` class which implements the `Consumable` interface.

Meta: Use this question as a mini-lecture for interfaces to build up to the next question. This question should not take a lot of time.

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 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;
    }
}
```

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.

```
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;
    }
}
```

3.2 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`.

3.3 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?)