CSM 61B Spring 2017

Abstract Classes & Interfaces

Week 5: February 13, 2017

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 uses the `default` keyword to declare concrete implementations while abstract classes use the `abstract` keyword to declare abstract implementations.
- Interfaces define 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 existence.

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

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

2 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 Iterator<E> {
    boolean hasNext();
    E next();
}
```

2.1 Consider the following code that demonstrates the `ArrayIterator`.

```
Integer[] arr = {1, 2, 3, 4, 5, 6};
Iterator<Integer> iter = new ArrayIterator<>(arr);

System.out.println(iter.hasNext());  // true
System.out.println(iter.next());     // 1

System.out.println(iter.next() + 3); // 5

while (iter.hasNext()) {
    System.out.println(iter.next()); // 3 4 5 6
}
```

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

```
public class ArrayIterator<E> implements Iterator<E> {
    private int index;
    private E[] array;
    public ArrayIterator(E[] arr) {
        array = arr;
        index = 0;
    }
    public boolean hasNext() {
        return index < array.length;
    }
    public E next() {
        E value = array[index];
        index += 1;
        return value;
    }
}
```

2.2 Define an `IntListIterator` class that adheres to the `Iterator` interface.

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

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

```
public static void printAll(Iterator<E> iter) {
    while (iter.hasNext()) {
        System.out.println(iter.next());
    }
}
```

3 Duplicate Iterator *Extra for Experts*

3.1 Fill out `DuplicateIterator` so that it works as used in the example main method. When given two sorted input iterators, the `DuplicateIterator` returns the elements that are within both iterators. Assume that `findNextElement` is correctly implemented.

```
public class DuplicateIterator<T extends Comparable<? super T>> implements Iterator<T> {
    private Iterator<T> iter1, iter2;
    private T nextElement = null;

    public DuplicateIterator(Iterator<T> iter1, Iterator<T> iter2) {
        this.iter1 = iter1;
        this.iter2 = iter2;
        findNextElement();
    }

    public boolean hasNext() {
        return nextElement != null;
    }

    public T next() {
        T next = nextElement;
        findNextElement();
        return next;
    }

    /** Sets the nextElement instance variable to the next duplicate element
      * (or null if there is no remaining duplicate element). */
    private void findNextElement() { ... }

    public static void main(String[] args) {
        Iterator<Integer> iter1 = Arrays.asList(1, 2, 4, 5, 6, 9).iterator();
        Iterator<Integer> iter2 = Arrays.asList(1, 2, 3, 5, 7, 10).iterator();
        DuplicateIterator<Integer> di = new DuplicateIterator<>(iter1, iter2);
        di.forEachRemaining(o -> System.out.print(o.toString() + " ")); // 1 2 5
    }
}
```