CSC 20 Chapter 10: Interfaces, Generics, ArrayList

Next week's Lab 2 is focussed on writing programs that use Java's `ArrayList` class. This class exhibits two Java features that enhance abstraction: Interfaces and Generic types.

Interfaces

An interface is a list of public methods without any attached implementation.

```
public interface Shape {
    public double getArea();
    public double getPerimeter();
}
```

Once you have written an interface, you can write a class that implements the interface.

```
public class Square implements Shape {
    private double side;
    public double getArea() {
        return side*side;
    }
    public double getPerimeter() {
        return 4*side;
    }
}

public class Circle implements Shape {
    private double r;
    public double getArea() {
        return 0.5*Math.PI*r*r;
    }
    public double getPerimeter() {
        return 4*r;
    }
    public void foo() {
    }
}
```

This is a promise that the class `Circle` will contain all the methods listed in the interface `Shape`. It can contain more things -- fields and more methods -- but it *must* contain the promised methods.

A variable can be declared of an interface type and any object whose class implements the interface can be assigned to the variable. This is an example of the "is-a" relationship. If "A is-a B" that a variable of type A can be assigned to a variable of type B.

```
Circle c;
Shape s;
s = c;   // No error because a Circle "is a" Shape
c = s;   // Error!! A Shape is not a Circle
```

This is most commonly seen when a variable is of an interface type and is initialized using a class that implements the type.

```
Shape s = new Circle();    // No error because a Circle "is a" Shape
```

Because `s` is of type `Shape`, only the methods defined in `Shape` are accessible.

```
Shape s = new Circle();    // No error because a Circle "is a" Shape
s.getArea();               // Okay: s is of type Shape, and Shape has getArea.
s.foo();                   // Not okay: s is of type Shape, and Shape doesn't have getArea.
```

We will see this with `ArrayList`.

```
ArrayList list1 = new ArrayList();
List list2 = new ArrayList();
```

Using type `List` instead of `ArrayList` is preferred it's programming more abstractly: that `list2` is a list is what's important; that it is implemented as an `ArrayList` is less important.

List interface

[Show javadoc for List. Google "java 8 list"]

It's an abstraction for a list of items. You can add, remove, search, etc. But, what's the type of objects you can have on the list?

Rather than have a different list type for for each type of object contained in the list, it's defined "generically"

```
List<String>   // List of Strings
List<Integer>  // List of Integers (not ints, Lists can only hold reference types)
List<int[]>    // List of arrays
```

Those are all type, to create an actual list object we need to use `new` and create an object that implements the interface.

```
List<String> list = new ArrayList<String>();  // Both ArrayList and List are defined
generically
```

Here's a complete program that reads lines of text from the keyboard and outputs them in sorted order.

```
import java.util.*;

public class LineSorter {

    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        Scanner in = new Scanner(System.in);
        String tempString = in.nextLine();
        while (tempString.length() > 0) {
            list.add(tempString);
            tempString = in.nextLine();
        }
        Collections.sort(list);
        for (int i=0; i<list.size(); i++) {
            System.out.println(list.get(i));
        }
    }

}
```

Some notes:
~ Only references are held in Lists. No copying happens for add or get.
~ Java auto-converts between int/Integer, char/Character, double/Double, etc.
~ Lists auto-expand as needed.
~ Raw arrays are more efficient than Lists, but extra efficiency is rarely needed.

Comparable interface

`Collections.sort()` only works if the items on the list can be compared to one another. There is an interface that signals to Java that objects are comparable.

We know how to sort integers using the Java library.

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

If a class implements the Comparable interface, it is promising to implement the `compareTo` method. Arrays.sort and Collections.sort call `compareTo` to compare two objects for ordering them.

If `this.compareTo(other)` is invoked, it should return
~~ a negative value if `this` should be considered less than `other`.
~~ zero if `this` should be considered the same as `other`.
~~ a positive value if `this` should be considered greater than `other`.

For example, lets say you wanted to compare `Point` objects. Then the class must be implemented like.

```
public class Point implements Comparable<Point> {
    private double x,y;

    public int compareTo(Point other) {
        double dist = Math.sqrt(x * x + y * y);
        double otherDist = Math.sqrt(other.x * other.x + other.y * other.y);
        if (dist < otherDist)      return -1;
        else if (dist > otherDist) return 1;
        else                       return 0;
    }
}
```

Notice the `T` in `Comparable<T>`. Since `this` and `other` are usually the same type, `T` is usually the same type as the class.

ArrayList<E> implementation

To practice abstraction and learn how a generically typed class is implemented, let's write our own version of `List<E>` and `ArrayList<E>`.

```
public interface List<E> {
    public void add(E elem);           // Insert elem at the end of the list
    public void add(int idx, E elem);  // Insert elem at index idx
    public void clear();               // Remove all elements from list
    public E get(int idx);             // Return element at index idx
    public int indexOf(Object o);      // Return first index of o, or -1 if not in List
    public E remove(int idx);          // Remove and return element index idx
    public int size();
}
```

Write implementation in class, (should include javadocs but doesn't) and unit tests. Here's about a far as we'll get.

```
import java.util.*;

public class MyArrayList<E> {

    public static final int DEFAULT_SIZE = 8;

    private E[] list;
    private int size;  // Number of valid elements

    @SuppressWarnings("unchecked")
    public MyArrayList() {
        size = 0;
        list = (E[]) new Object[DEFAULT_SIZE];  // Hack to get array of E.
    }

    public void add(E elem) {
        if (size == list.length) {
            list = (E[]) Arrays.copyOf(list,2*size);
        }
        list[size] = elem;
        size += 1;
    }

    public E get(int idx) {
        return list[idx];
    }

    public int size() {
        return this.size;
    }

    public static void main(String[] args) {
        MyArrayList<Integer> test = new MyArrayList<Integer>();
        System.out.println(test.size());
        test.add(123);
        System.out.println(test.get(0));
        System.out.println(test.size());
        for (int i=0; i<10; i++) {
            test.add(i);
        }
        for (int i=0; i<11; i++) {
            System.out.println(test.get(i));
        }
        }


}
```