Lab9: Interfaces

CSSSKL162: Programming Methodology Skills,

Summary

This lab is designed to introduce you to some frequently encountered Interfaces in Java, and then to get familiar with writing your own interfaces. When a class implements the Comparable interface, we are able to sort objects of this set; if a class implements Cloneable, then we can call clone() to make copies of such an object. We'll start with an example using Students and the {Comparable, Cloneable} interfaces, then move on to a brief introduction to multithreading (also using Interfaces).

Implementing the Comparable Interface for Sorting

We start by building a Student class that tracks their name and GPA, and then turn to implementing the Comparable Interface for this class. The idea is that we should be able to compare two students by their GPA and then sort a collection of students in order relative to their GPAs.

(1) Build a Student Class with only two data items: a String name and a double GPA.
(2) Modify the class definition so that it "implements Comparable"
(3) When designing the compareTo() method, use the GPA to determine if student1 > student2.
    a. Consider returning the difference between the two students as the magnitude of difference (either positive or negative)
(4) Build main to test your Students, and in this main build a list of 10 Students with different GPAs.
(5) Download and run the InterfaceDriver to test comparing students (comparableDemo in code)

Implementing the Cloneable Interface

There are two examples in this section for the Cloneable interface. We'll do the first one together, which is making students cloneable.

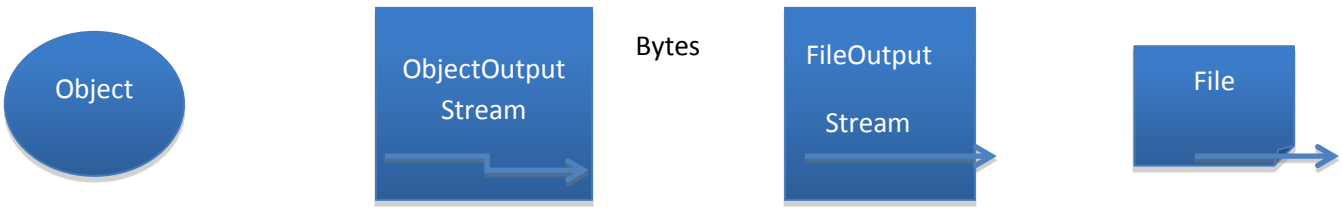
(1) Add "implements Cloneable" to your Student class definition
(2) Define the clone method using "@Override" , make it public and return a new student object
    a. Build a copy constructor and then the clone is just one line of code
        i. "return new Student(this);"
(3) Run the corresponding driver code to test Student clones.

In this next example, we'll build two classes that implement the Cloneable Interface. The QuizTracker class contains an ArrayList of QuizScore objects, and it's up to us to build these two classes so that we can make deep copies of QuizTrackers (and share no internal aliases). The key idea here is that to make deep copies of compound objects (or lists of objects), we need to copy (using clone) every object in every list, and even make copies of the list objects (ArrayLists here) themselves.

(1) Start by building a small QuizScore Class that contains only one data item: an int that corresponds to the score received on the quiz.
(2) Build a second class, QuizTracker, that contains an ArrayList of QuizScore objects (ArrayList<QuizScore>) as its only data item
    a. This is just like an IntList covered previously, but instead of ints we'll be storing QuizScores and instead of arrays we can use an ArrayList here.
(3) Add getters and setters to QuizScore for the score, and provide an add(QuizScore) method for class QuizTracker (note: once done with (4), return to this step to be sure you add a clone of the QuizScore handed as input to this function to avoid privacy leaks)
(4) Implement the Cloneable Interface for the QuizScore class
(5) Implement the Cloneable Interface for the QuizTracker class, as well
    a. This should ***not*** use ArrayList.clone(), as this is a *shallow copy.*
    b. Instead, to create a copy of a QuizTracker, you must first build a new ArrayList for the copy, and
        i. For each QuizScore object stored in our list, call clone() on each and add the clone() to the newly created ArrayList in (b)
    c. In this way, if we make a copy of the container (ArrayList), and all the contents in the container (QuizScores), we've succeeded in producing a *deep copy* for the QuizTracker class.

The Serializable Interface

To Serialize objects, we must first indicate that this is a permissible operation for our class. Certain classes should never be "frozen" to disk, such as real-time bank transactions or pacemaker operations. Java asks that you specifically indicate it's ok to write your object to disk using this ***Tagging*** interface. Such an interface has no methods, but serves to identify sets of objects. Writing an object to disk requires a FileOutputStream() to write bytes to a file ***decorated*** by an ObjectOutputStream(), which will translate Objects to bytes for the FOS. This chain of operations looks like the diagram below.



To create an Object writer, use the code:

```
ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream("data.obj"));
```

And to correspondingly read those objects, consider the code:

```
ObjectInputStream is = new ObjectInputStream(new FileInputStream("data.obj"));
```

1. Go to your Student.java file and add "implements Serializable"
    a. Note that no additional coding is required
2. Run the corresponding driver code from your set of interface drivers.
3. What was the output?
4. What did we accomplish?

The Runnable Interface

In this section, we'll build a class that can be run along other threads in a multithreaded (or multi-core) architecture. In Java, there exists a Thread class that we can extend and override to provide Thread specific activities, but even easier than this (and preferred) is to implement the Runnable Interface. The Runnable Interface has only one method (run()), and any object that is Runn-***able*** can be handed to a Thread Constructor and is queued for concurrent execution.

(1) Build a class that "implements Runnable"
(2) Make a constructor for this class that takes a string, and save this string to print out later (we'll use this to determine which thread is executing based on the string)
(3) Make the class do something interesting inside Runnable, such as print out the string passed to the constructor and print some calculation to the console
(4) If your class is named Foo, build two threads in the form: "Thread t1 = new Thread(new Foo()));"
(5) To start the execution of the t1 thread, invoke "t1.start();"
(6) To start the execution of the t2 thread, invoke "t2.start();"
(7) For extra guidance here, or to compare your code with another solution, see Multi.java for a working example.

Implementing Interfaces Using ActionListener

Implementing interfaces is one technique Java uses to allow for multiple inheritance. In this section, we'll be working with a Window Class (called a JFrame) that has only one GUI component added to it: a

JButton. When you download and run this code, pressing the button currently does nothing. In this section, your job is to modify the Application Class so that it will handle the button's events. This class will need to implement the ActionListener interface, and then we can *attach or register* this event handler with the button we're interested in.

(1) Extend the Application Class so that it "implements ActionListener"
(2) Define the ActionListener method "public void actionPerformed(ActionEvent e)" for the Application Class
    a. Make this function do something noticeable, like pop up a JOptionPane message dialog, so you know when your class is actually being called to handle events.
(3) In the code, look for and uncomment the line of code that looks like: "myButton.addActionListener(this);"

Implementing the MouseListener Interface

In this section, we'll build a JFrame that can respond to mouse events such as a mouse click, a mouse rollover, etc.

1. Build a new class called MyWindow that extends JFrame
2. Use this class definition to get started:

```
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import javax.swing.JFrame;


public class MyWindow extends JFrame implements MouseListener
{

   public MyWindow() {
         setSize(400,400);
         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         setVisible(true);

         //addMouseListener(this);
   }

   //todo: add MouseListener methods (see outline below)
```

3. Add the following method stubs (to be filled out later)

Method Summary	
void	**mouseClicked**(MouseEvent e) Invoked when the mouse button has been clicked (pressed and released) on a component.
void	**mouseEntered**(MouseEvent e) Invoked when the mouse enters a component.
void	**mouseExited**(MouseEvent e) Invoked when the mouse exits a component.
void	**mousePressed**(MouseEvent e) Invoked when a mouse button has been pressed on a component.
void	**mouseReleased**(MouseEvent e) Invoked when a mouse button has been released on a component.

4. In your JFrame constructor, add the line "addMouseListener(this);"
5. In mouseClicked, add a System.out.println() so when the user clicks on the JFrame something is outputted to the console.
6. Declare an instance variable: "ArrayList<Student> myShapes = new ArrayList<Student>();"
7. When the user clicks on the JFrame, add a new student to the student ArrayList
    a. Grab the mouse x,y using the event object handed to your mouseClicked function and use it as the gpa.