Week 11 Lab 8-2 – Interfaces

Write implementing classes for a Game interface and create and use an abstract class, including demonstrating polymorphism. For extra credit, create a **Coin** class that implements Comparable.

A. Create two classes that implement a Game interface:

- There is a Game interface defined in the Week 11 Chapter 8 Source Code folder, and also a PlayGames class that uses objects from classes that implement that interface. The Game interface declares a single abstract method *play* that returns an *int*, the score from playing the game, where bigger is better.
- 2. PlayGames creates and stores Game objects in a Game[] and uses a *popRandom* method to randomly choose one of the games to play next by running that object's *play* method.
- 3. There is an example of one such game, AdditionGame, that implements the Game interface. PlayGames creates an AdditionGame object and stores it in the Game[] array.
- 4. You must write at least 2 more classes that implement Game, store objects from those classes in the Game[] array in PlayGames, and run PlayGames to test that they work. You will test them manually by interacting with PlayGames, not by writing specific tests in a *main* method.

B. Create and use an abstract class:

Modify the Animal class to be abstract and change its *greet* method to be abstract:

- 1. Changing **Animal** on in-class slide 5 to be abstract should only take two changes. The *greet* method should no longer have a method body.
- 2. Also, write the **Friendly** interface shown in the slides and change Animal to implement it.
- 3. Create a simple **Dog** class that looks like the one shown in the inheritance slides and have it extend Animal. Its *greet* method must @Override the abstract *greet* method in Animal and fully define it. Note that Dog can not reuse Animal's *greet*, because there is no longer a *greet* definition in Animal.
- 4. Write a *main* method either in Dog or in a separate class that creates a Dog object, assigns it to a **Dog** reference variable, and runs that object's *greet* method:

Dog d = new Dog("fido"); d.greet();

5. Also in *main* assign that Dog object or another new one to an **Animal** reference variable and run its *greet* method:

Animal a = d; a.greet();

6. Finally in *main* assign a Dog object to a **Friendly** reference variable and run its *greet* method: :

Friendly f = d; f.greet();

All of these should work properly, using Dog's *greet*. You have just demonstrated polymorphism and dynamic or late binding!

C. Extra Credit: Create a class that implements Comparable:

For <u>extra credit</u>, create a **Coin** class with instance variables *private String name* (like "nickel"), *private double value* (like .05), and *private double weight* (like $5 \rightarrow$ grams) and a 3-parameter constructor to set all of them, and have that class <u>implement Comparable</u>:

- 1. The class has the single constructor mentioned above plus *setters* for all three instance variables; you can use those *setters* in the constructor if you want.
- 2. Also write a *toString* method whose *String* contains all three instance variables.
- 3. To implement Comparable, define a *public int compareTo(Object other)* method that works as follows:
 - It uses *instanceof* to check if *Object other* is a Coin; if not, it returns <u>1</u>.
 - It uses downcasting to assign *other* to a new Coin reference variable.
 - It returns *String*'s *compareTo* result <u>if not 0</u>; if 0, return a result comparing the *value* of the two Coins; if they have the same *value*, compare their *weight*.
- 4. Write a *main* method that creates a small Coin[] (say, length 5) and fill it with Coin objects that are not ordered, then run *Arrays.sort* on the array and finally print the Coin objects from the array to see if they have been sorted properly.

This exercise is worth 7 <u>extra credit</u> points.

Show me or our TA your program source code and how it works.