Problem Set 5 Solution

Inner Classes, Abstract Classes, Polymorphism

---

1. Inner Classes
    1. Write a class named `Outer` that contains an inner class named Inner. Add a method to `Outer` that returns an object of type `Inner`. `Outer` has a private `String` field (initialized by the constructor), and `Inner` has a `toString()` that displays this field. In `main()`, create and initialize a reference to an `Inner` and display it.

    2. Write a class named `Outer2` that contains an inner class named `Inner`, and the `Outer2` class itself has a method that returns an instance of the inner class. In a separate class named `InnerApp`, make an instance of the inner class without creating an object of the outer class.

    SOLUTION

    1. Outer

    2. Outer2, InnerApp

---

2. Abstract Classes.

    For each of the following, tell whether the code will compile. If not, explain why.

    1. ```
       public abstract class X { }
       ```

       ANSWER

       Yes.

    2. ```
       public class X {
          public abstract void stuff();
       }
       ```

       ANSWER

       No. Since the class has an abstract method, the header should have the `abstract` keyword.

    3. ```
       public abstract class X {
          public abstract void stuff() {
             System.out.println("abstract");
          }
       }
       ```

       ANSWER

       No. An abstract method cannot have a body.

    4. ```
       public abstract class X {
          public void stuff() {
             System.out.println("go figure");
          }
       }
       ```

       ANSWER

       Yes.

    5. ```
       public abstract class X {
          public abstract void stuff();
       }
       public class Y extends X { }
       ```

       ANSWER
       ```

No. `If Y` is not abstract, it must override the inherited method `stuff` with an implemented body. (Otherwise, `Y` should be declared as `abstract`.)

6.
```
public interface I {
   void stuff();
}
public abstract class X {
   public abstract void stuff();
}

public class Y extends X implements I {
   public void stuff() { }
}
```

ANSWER

Yes.

7.
```
public abstract class X {
   private int i,j;
   public void stuff1() { }
   public void stuff2() { }
}
```

ANSWER

Yes.

8.
```
public abstract class C {
   public void write() {
      System.out.println("C");
   }
   public static void main(String[] args) {
      C c = new C().write();
   }
}
```

ANSWER

No. `C` is an abstract class, cannot be instantiated.

9.
```
public abstract class C {
   public abstract void write();
}
public class D extends C {
   public void write() {
      System.out.println("D");
   }
   public static void main(String[] args) {
      C c = new D();
      c.write();
   }
}
```

ANSWER

Yes.

---

3. There is an application that defines a `Person` class and a `Student` class. The `Student` class is defined as a subclass of `Person`. Every person has a home address, while every student has a school address as well.

   Consider printing addresses of all people in the application, assuming there is a single array list that stores all `Person` and `Student` objects. How would the address that is printed for students depend on the way the `Student`

class address methods are designed/implemented? What alternatives in design can you think of, and what are the pros and cons of these alternatives in printing the addresses?

SOLUTION

Every `Person` is expected to have one default home address, but `Student` can have another address for school.

There are two ways to think about this.

- One way is to think purely from the polymorphism point of view.

  Suppose you were to set up a mixed collection of `Person` and `Student` objects like this:

  ```
  List<Person> persons = new ArrayList<Person>;
  ...  // populate persons with Person and Student objects
  ```

  Then, if you were to run a loop through this collection to print address, you would statically type the stepping reference as `Person`, say like this:

  ```
  for (Person p: persons) {
     System.out.println(p.getAddress());
  }
  ```

  This means, you want to look at every entity, including a `Student`, as a `Person`, which then implies that all addresses printed should be home addresses. If you take this point of view, then the `Student` class should *not* override the inherited `getAddress` method from `Person`, and should have a new `getSchoolAddress` method.

- The other way to think about this is from the point of view of class design independent of how applications might use objects at run time. In this case, the method `getAddress` for a `Student` would override the inherited-from-`Person` implementation to return the school address instead. And a new `getHomeAddress` method would be coded to return the home address.

---

4. This problem gives an example where polymorphism is useful. Consider the class hierarchy given below :

```
public abstract class Shape implements Comparable<Shape> {

    public void print() {
        System.out.println("Shape");
    }

    public abstract double getArea();

    public static final Shape biggest(Shape[] s)  {
        /** TO BE COMPLETED BY YOU **/
    }

    ... // OTHER METHODS/FIELDS YOU MAY NEED TO ADD TO ANSWER THE QUESTION
}

public class Circle extends Shape {
    double radius;

    public Circle(double r) {
        radius = r;
    }

    public void print() {
        System.out.println("Circle");
    }

    public double getArea() {
        return Math.PI*radius*radius;
    }
```

```
}

public class Rectangle extends Shape {

    double height;
    double length;

    public Rectangle(double l,double h) {
        length = l;
        height = h;
    }

    public void print() {
        System.out.println("Rectangle");
    }

    public double getArea() {
        return length*height;
    }
}

public class App {

    public static void main(String[] args) {
        Shape[] s = new Shape[3];

        s[0] = new Circle(7);
        s[1] = new Rectangle(5,10);
        s[2] = new Circle(4);

        System.out.println("The biggest area of all shapes is : "+Shape.biggest(s));
        return;
    }
}
```

Complete the method

```
public static Shape biggest(Shape[] s)
```

in the `Shape` class. This method should return the shape with the largest area. Note that `Shape` implements the `Comparable` interface. Different `Shape`s should be compared using their area. Now if we extend the Shape hierarchy to include more shapes, say rhombus, then will your method run without any problems?

SOLUTION

```
public static final Shape biggest(Shape[] s) {
    if (s.length == 0) { return null; }
    Shape biggestShape = s[0];
    for (int i = 1; i < s.length; i++) {
        if (biggestShape.compareTo(s[i]) < 0) {
            biggestShape = s[i];
        }
    }
    return biggestShape;
}

public int compareTo(Shape s) {
    double areaDifference = getArea() - s.getArea();
    if (areaDifference == 0) {
        return 0;
    }
    return areaDifference < 0 ? -1 : 1;
}
```