Lab5 part II: Inheritance and the Shape Hierarchy*

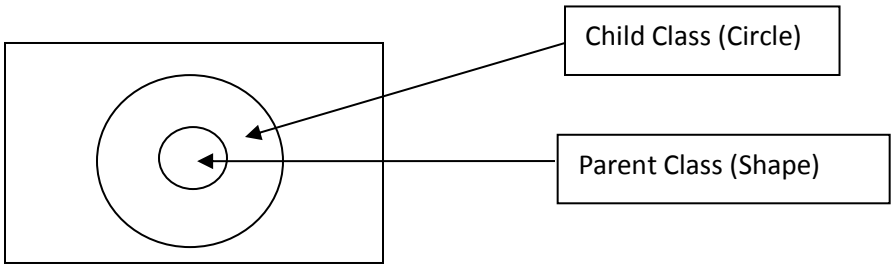
CSSSKL162: Programming Methodology Skills

Summary

Build two classes that *inherit* from an existing Shape superclass. These subclasses should all have the phrase "*extends Shape"* in their class signatures, as in: "public class MyNewShape extends Shape {". Your classes can and should be tested in isolation before combining your classes with the final drivers, so consider building a main in each subclass that tests out just that class. When you are confident in your subclasses, you can use this driver to see your shapes render to a window (a JPanel).

Introduction

This lab continues exploring the idea of inheritance, which is a critical concept in object-oriented programming. Inheritance (as defined in the software realm) borrows from its Mendelian genetics: just as you can inherit your mother's eyes or your father's sense of humor, one child class can inherit characteristics and behaviors of a parent class. More specifically to Java classes, when one child class inherits from a parent class (or a subclass inherits from a superclass), this child class gets a copy of all the methods and data from the parent class – you can envision this as a copy-and-paste operation, from the parent to the child. Once the child class *extends* the parent class, any methods or data items found in the parent class are now a part of the child class, and the child class is free to define extra features. In Venn diagram terms, the parent class is strictly a subset of the child class, and would look like:



What this picture highlights is the relationship between parent classes and child classes – everything the parent class has, the child also has. More to the point: any public method defined in the parent will also be defined in the child class, and this acts as a type of contract or guarantee. Indeed, it is a class invariant that the interface for a child class will contain every (public) method defined in the parent class interface (except for constructors). If we can promise that a Circle class has every method found in the Shape class, then anywhere in software a Shape is called for, a Circle could be substituted (this concept is called ***substitutability*** and is related to ***polymorphism***, but more on that later). If we have a function that expects a Ball object, and we have a subclass object VolleyBall, then we can pass the VolleyBall to the method with no problems, because a VolleyBall *is a* Ball, just like a Circle *is a* Shape. Inheritance then defines this type of "is a" relationship, which is a one-way relation between two classes. Note that the relationship isn't like the bi-conditional operator in that, all VolleyBalls are definitely Balls, but not all

Balls are guaranteed to be VolleyBalls (some could be BaseBalls or BasketBalls, for example). When describing inheritance relationships, we'll use an arrow to indicate this "one-way" characteristic. When building classes that are interrelated via inheritance, inheritance hierarchies naturally arise; these are simply tree structures that display the "is a" inheritance relationships between the classes in your software.

Before we talk about the methods your Class must provide (or more specifically, *override*), we should get to know our Parent class Shape. The methods and data for this class are outlined below, and the code is available for download via the website (and you'll need it in the same directory as your child classes).

Data Members

- int x; //all shapes have an x,y coordinate pair in Java2D
  - Should this be public? Private?
- int y; //and so we'll have the Parent class manage this data
  - What access modifier should we use here?
- (optional) Color myColor;
  - Colors are immutable, and so will have no getters/setters and also will not suffer from privacy leaks.
  - Check with your lab instructor as to whether you will implement this.

Method Members

- Shape(int x, int y) //a constructor used to initialize the data members
- double getArea(); //used to calculate the area of this shape
  - For a Circle, use Math.PI * r * r, and for a Square, side * side.
- void paint(Graphics g); //the paint method is called on each shape to draw itself
  - See the Spray subclass for a sample implementation of the paint function
    - This was draw() in previous Shape classes.
- getX() //accessor
- getY() //accessor
- setX(int) //mutator
- setY(int) //mutator

As you can see from above, the class is quite small. Now take a look in the code to see how many of the Shape functions are actually empty! Shape is really too generic of a class to offer an implementation for

draw() or getArea(), as these are custom to specific subclasses of shape – in fact, it's our job to provide versions of these methods in our subclass that actually do draw shapes or calculate areas. We could easily have made Shape into an abstract class or even an interface, but for this demonstration we'll stick with a basic, mutable, nonstatic class. Your class will start with the line "public Circle extends Shape", and this will only compile if Shape.java is also in the same working directory (or project) when compiling. As a Circle, you will need to add more data and methods that are custom to being a Circle; these are specifics that not just any shape would have, such as a radius data item and a getRadius() accessor function.

Circle Extends Shape

The first thing we need to do is download the Shape superclass, the Spray subclass, and the driver **PolyDemo.java**. This should compile and run with no modifications, but will only display a set of "spray" shapes on the screen (which are ovals that randomly change their width and height). You will create two subclasses of Shape (just like Spray) in the following steps.

(1) Download all the files and put them into one project.
(2) Run the PolyDemo.java and observe the output.
(3) Build a new class called "Circle" using the inheritance keyword "extends"
    a. "public class Circle extends Shape"
(4) Override the getArea() method
    a. This method should return a double corresponding to the area of your shape.
(5) Override the draw() method
    a. This method will draw the shape onto the Graphics context g (or g2D).
        i. Look at Spray for an example of how to do this, or try:
            1. g.draw3DRect(x,y,width,height, raised)
            2. g.drawOval(x,y,width,height)
(6) Next, define members that are custom to Circles, such as:
    a. private double radius;
    b. double getRadius();
    c. void setRadius(double);
(7) Modifying the PolyDemo function getRandShape() so that it can create and return objects of your new Circle class.
    a. Do this by replacing one of the switch cases that state "retVal = new Spray()" with "retVal= new Circle()".
(8) Run the PolyDemo.java and observe your new Circle subclass also being rendered to the screen.

By following the three steps above, you should be able to iteratively develop each Shape subclass and test it first in isolation (make a small main inside that class to test it out), and then in the larger system that will use your subclass to actually draw stuff to the screen (**PolyDemo.java**).

YourClass extends Shape

In this section, it's up to you to define a second, custom Shape. If you're uncertain, consider implementing a **Square**, which is similar in spirit to the **Circle** above, but the radius becomes a sideLength and your paint() and area() calculations change. If you feel comfortable with making Shapes, consider a more advanced shape, such as a PokeBall, Sierpinski Triangle, Rocket Ship, or anything your can imagine and render in 2D. Cubes in (pseudo) 3D look nice as well. To build your custom class:

(1) Build a new class using the inheritance keyword "extends"
    a. "public class YourClass extends Shape"
(2) Override the getArea() method
    a. This method should return a double corresponding to the area of your shape.
(3) Override the draw() method
    a. This method will draw the shape onto the Graphics context g (or g2D).
        i. Look at Spray for an example of how to do this, or try:
            1. g.draw3DRect(x,y,width,height, raised)
            2. g.drawOval(x,y,width,height)
(4) Next, define members that are custom to your shape, such as:
    a. A base and height for a triangle or rectangle.
    b. A sidecount for a generic polygon.
(5) Modifying the PolyDemo function getRandShape() so that it can create and return objects of your new class.
    a. Do this by replacing one of the switch cases that state "retVal = new Spray()" with "retVal= new YourClass()".
(6) Run the **PolyDemo.java** and observe your new subclass also being rendered to the screen.

A Short Primer on Java2D

In this lab, you are to develop two shape subclasses, and you can draw them however you want using Java2D. Java uses a **Graphics2D** object to render things into drawable areas in Java's windows (or JFrames). We won't need to go much further than a Google search to see all the functions you can call on a **Graphics2D** object, but some of them are: {drawRect, drawRoundRect, drawFillRect, drawOval, drawString} You could make a Shape subclass that represents a letter, and to draw that letter, you'd provide a line of code in your draw() method like "g.drawString(…)" to draw the individual character. If making ovals, use the drawOval function, and the same strategy for rectangles. While the Savitch text is a bit light on the subject of Graphics, it does cover some data on drawing ovals and arcs on page 1033, and also shows you how to specify colors on page 1045. DrawString is at 1051, and again, there are tons of examples online for Java2D graphics code, but the main idea is: you are given a graphics object, and you make calls on that object to produce graphics you can view on the screen. If your editor has auto-

complete enabled, you may simply type "g.", then the list of graphics functions will be highlighted for you, *and there's a ton*, so don't get overwhelmed. If you like, you can just leave the draw function very minimal until you have a grasp on how your class inherits from Shape, and how it fits into the driver code.

Late Binding & Polymorphism

Two concepts that are critical to Object-Oriented programming are **late binding (or dynamic dispatch)** of method calls and **polymorphism,** which allows for many (poly) forms (morph) of the target function to exist – Java's runtime environment will select the correct draw() function at runtime (this is late binding) from a set of draw() functions each defined in Shape, Square, Circle, Triangle, etc. A language that supports the redefinition of a function by a subclass then might produce multiple versions of one function, such as draw(), toString() or equals(). Having a draw() method in Square and a draw() method in Circle could be ambiguous, as there are now many forms of the draw() method that each exist in different subclasses. In the case of overridden functions (i.e, a function with "many forms"), Java doesn't determine ahead of time which draw() function to invoke; this is known as **static binding**, and is used when the specific function can be determined offline or at compile-time. Static binding isn't possible in all cases if a language supports polymorphism; sometimes the object can only be known at runtime, which requires a mechanism such as late binding that uses RTTI to determine the exact type of the object. Consider the code from **PolyDemo.java** that returns a random shape (called getRandShape() in PolyDemo.java).

```
Shape a = getRandShape();  //a could be a Square, Circle, Triangle, or some other Shape subclass

a.draw();   //this draw() method could refer to Square's draw(), Circle's draw() – many draws exist!
```

If the getRandShape() function can indeed return any Shape, then how would we know ahead of the function call which draw() function to invoke? If the object is a Square, we want to call the Square's draw() function, and if the object type is really a Circle, then we don't want to invoke any draw() method but the one found in the Circle class. How can we accomplish this type determination without an oracle? Java can consider an unknown object and reflect upon its characteristics to determine an objects type, at any point in a program's execution. Java does this by storing additional information per object at runtime, also known as RTTI (runtime type information). Now that we've discussed late binding and polymorphism, lets consider some questions next. Answer these in comments inside your Employee subclasses.

- What methods are polymorphic in the Employee Hierarchy?
- How could we build a method like getRandShape() above but for use with Employees?

- If we built a getRandomEmployee() method that returns various Employee subclass objects; write a few lines of code that would demonstrate **late binding**

---

* This lab is originally written by Rob Nash; modifications by A.Retik