Lab 3 - Inheritance
Due (via Bitbucket and hard copy) Wednesday, 16 September 2015
50 points

Lab Goals

- Practice implementing some classes that inherit from each other
- Get your code to work with an changable test file
- Think about how to solve a variety of design challenges

Assignment Details

In a recent lecture, we showed how we can set up our Rock-Paper-Scissors program to use inheritance, with the Human and Computer classes both inheriting from a shared Player class. In this way, we keep the shared behavior between the two players in one location, so that it can be modified for both players with only a single edit. As we build a deeper hierarchy of classes, inheritance becomes even more important.

There are several classic programming examples used to show inheritance, including an Animal hierarchy, a Vehicle hierarchy, and a Clock hierarchy. In this lab, we will be implementing a fourth classic example, the Shape hierarchy.

The hierarchy of classes that you will create is shown at the top of the next page in Figure 1. Detailed descriptions for each of the classes follow, and a final UML class diagram for each class is shown in Figure 2.

Shape Class (10 points)

To begin, we need to create a Shape class. For each of the shapes that we create, we will want to have the ability to calculate their perimeter and area; however, each function will have a different way to calculate perimeter and area, so we can’t put the functions that calculate here. But we can create float variables for perimeter and area, as well as accessors and mutators to get and set those variables. You may initialize perimeter and area to 0 here. Therefore, the Shape parent class should consist of two private variables and four public functions.

Also included in this section is the ability for your classes to work with a main function called ShapeTester that I provided in labs/lab3/src directory. Please look carefully at this code, as it is commented to show the order of the parameters that I expect for the constructors. I recommend starting this lab slowly, implementing one branch of the hierarchy at a time, testing with the main
Figure 1: A summary view of the classes involved in this lab and the inheritance relationships between them.

function by commenting and uncommenting relevant code lines in the main function. A printout of the output for ShapeTester is included after all of the shape descriptions.

Shape Children (15 points)

Figure 1 shows that we need to create three child classes that will inherit from Shape: Round, Line, and Quad. Details for each of these implementations follow.

Round Class

The Round class is an intermediate class in the hierarchy, located above the Oval and Circle classes but below the Shape class. We still calculate the area and perimeter of these shapes differently, so that behavior doesn’t belong at this level. We can add one property here though. Both child shapes are round (hence the name of the class), so both will make use of \( \pi \) in calculating the perimeter and area. In this class, we can declare a private float for pi equal to 3.1415926535 (you can initialize it here), and an accessor to get that value. Therefore, the Round class should consist of one private variable and one public function.

Line Class

The Line class is at the bottom of its hierarchy chain, so it will actually implement the two functions calculateArea() and calculatePerimeter(). We know that the area of a line is 0, so this one is trivial. We will think of a line as a rectangle with a height of 0, so the perimeter of the line is equal to double its length, or \( 2 \times \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \). We will also need to create a Line constructor.
to get those $x_1, y_1, x_2,$ and $y_2$ variables in that order, as well as instance variables to store them (because these will belong to the Line class, they will not need accessors and mutators). Therefore, the Line class should consist of four private variables, a constructor, and two public functions.

Quad Class

The Quad class is another intermediate class located between the Shape class and its actual shapes. Much like the Round class, we will calculate the area and perimeter differently for each of its children, so that behavior doesn’t belong at this level. We can add variables for width and height at this level though, as well as the accessors and mutators to get and set them. Therefore, the Quad class should consist of two private variables and four public functions.

Shape Grandchildren and Great-Grandchildren (15 points)

Figure 1 shows that the Round and Quad classes have children of their own, which will actually implement the calculateArea() and calculatePerimeter() functions. Details for each of these implementations follow.

Oval Class

The Oval class will be similar to what we needed to do for the Line class, as well as all subsequent classes in this section. We will need two variables for our oval, representing half of the major axis and minor axis of the oval (think of them as the radius of the oval at both its largest and smallest). We can call these variables major and minor. We will need a constructor to set these variables whenever a new Oval is created. Finally, we will need to implement the calculateArea() and calculatePerimeter() functions. The area of an oval is $\pi \times \text{major} \times \text{minor}$, and the perimeter of an oval is approximated by $2 \times \pi \times \sqrt{\frac{\text{major}^2 + \text{minor}^2}{2}}$. The calculateArea() and calculatePerimeter() functions should perform these calculations, and then set the values into the area and perimeter variables at the Shape class level. Therefore, the Oval class should consist of two private variables, a constructor, and two public functions.

Circle Class

The Circle class is similar to the Oval class, in that we will be calculating the perimeter and area again. Instead of two variables, we only need one: radius, as well as a constructor to set it. Finally, the formula to calculate the perimeter of a circle is $2 \times \pi \times \text{radius}$, and the formula to calculate the area is $\pi \times \text{radius}^2$. The Circle class should consist of one private variable, a constructor, and two public functions.

Rect Class

The Rect class is a special case because it is a parent to the Square class, but also has its own area and perimeter. We already have the width and height variables in the Quad class, so the Rect class
Due Wed., 16 Sept. 2015

only needs a constructor to set them and the functions to calculate perimeter \((2 \times (\text{width} + \text{height}))\) and area \((\text{width} \times \text{height})\). The \texttt{Rect} class will have a constructor and two public functions.

**Trapzd Class**

The \texttt{Trapzd} class will get its height and base width from the \texttt{Quad} class as well, but it requires a second width (for the top) to calculate the area, and two side lengths to calculate the perimeter. Therefore, we need to create three additional private variables at this level, in addition to the constructor and the area and perimeter functions. The area of a trapezoid is \(\text{height} \times \frac{(\text{width} + \text{otherBase})}{2}\), and the perimeter is \(\text{width} + \text{otherBase} + \text{side1} + \text{side2}\). The \texttt{Trapzd} class will have three private variables, a constructor, and two public functions.

**Square Class**

Last but not least is the \texttt{Square} class, which inherits from the \texttt{Rect} class. A square is just a simple case of a rectangle where the height and width are equal. Therefore, you can set up your constructor to use either \texttt{height} or \texttt{width} for your area and perimeter calculations. Assuming that we use \texttt{width}, the formula for the area of a square is \(\text{width}^2\), and the formula for the perimeter is \(4 \times \text{width}\). The \texttt{Square} class will have a constructor and two public functions.

Note the because the \texttt{Square} class inherits from the \texttt{Rect} class, and the \texttt{Rect} constructor requires two parameters, the first line in our \texttt{Square} constructor needs to convert from one to two parameters, something like \texttt{super(w, 0)} or \texttt{super(w, w)}.

![Figure 2: A detailed view of the classes involved in this lab and the inheritance relationships between them.](image)
Sample Output with ShapeTester

TESTING LINE SHAPE
---------------
Line area = 0.0
Line perimeter = 16.1245154965971

TESTING OVAL SHAPE
---------------
Oval area = 226.194671052
Oval perimeter = 59.607529593072904

TESTING CIRCLE SHAPE
---------------
Circle area = 30.974846926448603
Circle perimeter = 19.729201863980002

TESTING RECT SHAPE
---------------
Rect area = 42.0
Rect perimeter = 34.0

TESTING TRAPZD SHAPE
---------------
Trapzd area = 16.0
Trapzd perimeter = 11.0

TESTING SQUARE SHAPE
---------------
Square area = 9.8596
Square perimeter = 12.56

Additional Questions (10 points)

Please also take the time to answer the following questions:

1. Explain, in your own words, the object-oriented design principles of Abstraction, Encapsulation, and Modularity. How do these relate to the design of this shapes hierarchy?

2. Let’s say that later on, we want to add a Triangle class. Where would you place this shape in the hierarchy? Defend your answer.

3. Let’s say that later on, we want to add several sub-Triangle classes: IsoscelesTriangle, RightTriangle, EquilateralTriangle, and ScaleneTriangle. How would you design this hierarchy? Defend your answer.
Submission Details

For this assignment, please submit the following, both to your cs112f2015-<your user name> repository and as a paper copy:

1. Commented source code from all nine classes: Shape, Round, Line, Quad, Oval, Circle, Rect, Trapzd, and Square.

2. Proof that your implementation works with my ShapeTester class, or an explanation for why/which parts you think are broken.

3. A document containing responses to the “Additional Questions” prompts.


Before you turn in this assignment, you also must ensure that the course instructor has read access to your BitBucket repository that is named according to the convention cs112f2015-<your user name>. Please note that each student in the class is responsible for completing and submitting their own version of this assignment. However, you also will be assigned to work to a team that is tasked with ensuring that all of its members are able to complete each step of the assignment. Team members should make themselves available to each other to answer questions and resolve any problems that develop during the laboratory session. While it is acceptable for members of a team to have high-level conversations, you should not share source code or full command lines with your team members. To ensure that you can communicate effectively, members of each team should sit next to each other in the room. Please see the instructor if you have questions about this policy.