**Summary**

Build 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 JPanel. Note that this is also one of the drivers I will use when grading your work, so you should be sure to verify that these classes compile and execute before submitting.

The deliverables description are near the end of this assignment description.

**Files to Download**

* [PolyDemo.javaPreview the document](https://canvas.uw.edu/courses/1331881/files/58017731/download?wrap=1)
* [Shape.javaPreview the document](https://canvas.uw.edu/courses/1331881/files/58017736/download?wrap=1)
* [Spray.javaPreview the document](https://canvas.uw.edu/courses/1331881/files/58017732/download?wrap=1)
* [Square.javaPreview the document](https://canvas.uw.edu/courses/1331881/files/58017740/download?wrap=1) (this is given as an example)

**Introduction to Inheritance, Shape, and Shape's Children**

In the previous assignments, we were practicing new and reviewing concepts. This assignment is the first in which we’ll explore 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 behaviours 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 biconditional 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).

**Shape Class: Data Members**

* int x: x-coordinate. All shapes have an x,y coordinate pair in Java2D, and so we have the parent class manage the data. Should this be public? Private? What access modifier should we use here and for y?
* int y: y-coordinate.
* Color myColor (optional):
  + 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.

**Shape Class: Method Members**

* Shape(int x, int y): A constructor used to initialize the data members
* double getArea(): Calculate the area of this shape. For a Circle, use Math.PI \* r \* r, and for a Square, side \* side.
* void draw(Graphics g): This method is called on each shape to draw itself
  + See the Spray subclass for a sample implementation of the draw function
  + In some other Shape classes, this method is called paint. The only difference is the name.
* getX(): Accessor for x
* getY(): Accessor for y
* setX(int value): Mutator for x
* setY(int value): Mutator for y

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, non-static 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”, i.e., “public class Circle extends Shape”
4. Override the getArea() method: This method should return a double corresponding to the area of your shape.  If your shape is too complex such the area is too difficult to calculate or is not well-defined, you don't have to override this method.
5. Override the draw() method: This method will draw the shape onto the Graphics context g (or g2D). Look at Spray for an example of how to do this, or try:
   * g.draw3DRect(x,y,width,height, raised)
   * g.drawOval(x,y,width,height)
6. Next, define members that are custom to Circles, such as:
   * private double radius;
   * double getRadius();
   * void setRadius(double);
7. Modifying the PolyDemo function getRandShape() so that it can create and return objects of your new Circle class. 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 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).

**A Short Primer on Java2D**

This is the part of the assignment that has the least constraints on it. You are to develop a shape subclass, and you can draw it 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 text is a bit light on the subject of Graphics, it does cover some data on drawing ovals and arcs and also shows you how to specify colors (see the index). DrawString is also included, 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. Note that we will walk over an example Square class in class before the assignment is due, and I’ll demonstrate some of the things you can do with a graphics object. In eclipse, if you 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.

**Notes and Hints**

* **Make sure you comment all methods and the class with javadoc comments.**  You cannot receive full-credit for the rubric's commenting criterion if you do not have a **javadoc comment for every method,** no matter how small.  If you have files based on code someone else has written (e.g., is based on a skeleton), you have to javadoc comment the methods someone else wrote also.
* Your code should not depend on absolute path specifications or any other environment-specific specifications.  The program should **not** ask for user input from the console.
* Because the lab is graded on effort, I understand that your lab shapes may not be as well-developed coming out of the lab, but for this homework, you need to bring them up to homework standards (most prominently, you need to comment them well).  **Please note in comments which shapes are from the lab and which are new.**

**What to Submit**

* Good news:  Sine wave of life indicates strong overlap between your lab and your assignment this week! You will submit 2 shapes you made in lab and another 2 more shapes for your homework, and they should be awesome: Try a pokeball or Death Star. (Thus, you will submit a total of 4 shapes for this homework.)  You ***cannot*** submit duplicate shapes amongst these 4 shapes.
* Of these 4 shapes, you cannot submit a Square or Spray (since the examples are given above).
* Submit your PolyDemo too.

Thus, you will submit a total of 5 files.  (You do not have to turn in the Shape class since you won't make any changes to it.)  For the 5 files you turn in, you have to provide class and method Javadoc comments for all classes and methods, including for classes/methods you did not write (PolyDemo).

Please **submit your files inside a zip archive called ShapeInheritance.zip.**  If you are not sure how to create a zip archive, do a Google search on "create zip archive windows" (swap out your computer's operating system, as needed).  Make sure you give yourself enough time before the due time to learn about and create the archive!

**About This Document**

Original by Rob Nash. Minor edits by Johnny Lin, Spring 2017-September 2019.

**Rubric**

Shapes Inheritance Rubric (1) (1)

| Shapes Inheritance Rubric (1) (1) | | |
| --- | --- | --- |
| **Criteria** | **Ratings** | **Pts** |
| This criterion is linked to a Learning Outcome Shape Subclass1 |  | 8.0 pts |
| This criterion is linked to a Learning Outcome Shape Subclass2 |  | 8.0 pts |
| This criterion is linked to a Learning Outcome PolyDemo Modified Class |  | 2.0 pts |
| This criterion is linked to a Learning Outcome Comments in Code |  | 8.0 pts |
| This criterion is linked to a Learning Outcome Demonstrates 4 shapes (Lab + HW) |  | 4.0 pts |
| This criterion is linked to a Learning Outcome Code Executes |  | 4.0 pts |
| Total Points: 34.0 | | | |

**Submission**

Submitted!

Oct 31 at 11:50pm

[Submission Details](https://canvas.uw.edu/courses/1331881/assignments/4923159/submissions/3729452)

[Download ShapeInheritance.zip](https://canvas.uw.edu/courses/1331881/assignments/4923159/submissions/3729452?download=59464125)

Grade: 34 (34 pts possible)

Graded Anonymously: no

[View Rubric Evaluation](https://canvas.uw.edu/courses/1331881/assignments/4923159/submissions/3729452#rubric)

**Comments:**

Nice work!