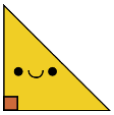


Getting into Shape

Inheritance and Polymorphism – Interfaces and Abstract Classes
Multiple Constructors, Downcasting, Overriding Methods, throw

Name _____ Date _____ Period _____

I. Many of this project's steps are spelled out very specifically. Some steps are hinted at. However, some steps need to be figured out by you. For example, the `equal` method might not be specified but can be deduced by you.

II. Create a new project called `GeometricShapeYourName` – use your name in place of `YourName`; e.g., `GeometricShapeKing`.

III. Remember to:
   a. Read Barron's regarding inheritance, polymorphism, interfaces, and abstract classes.
   b. Write the Javadoc documentation as you go along. You will save yourself work in the long run. See section VI. d. below.
   c. Compile as you go along.

IV. Create an interface called `Nearable`. (This is a silly name.) – *This is a class activity.*
   a. Create a method called `isNearlyEqual` that accepts two `Object` parameters and returns a boolean.
      i. Have the method return `true` if two objects are nearly equal.
   b. Create a method called `compareToNearly` that accepts two `Object` parameters and returns an integer.
      i. Have the method return an `int`, like `compareTo` does.
      ii. Use Javadoc to communicate what the return values mean.

V. Create an interface called `AnotherComparable`. (I know this is another silly name.)
   a. Create a method called `compareAnotherWay`. This works much like `compareTo`.
      i. Have the method return an `int`, like `compareTo` does.
      ii. Use Javadoc to communicate what the return values mean.

VI. Create an abstract class called `GeometricShape`.
   a. The class implements `Comparable`, `AnotherComparable` and `Nearable`.
   b. Create a final instance variable:

```
public final static double EPSILON = 1.0e-5;
```

   c. Create one instance variable to hold the assigned label of the shape, for example, a label such as "ABC" might be a composite label of the vertices of a triangle and the label that is assigned the triangle. You may think of a label as a name associated with a shape.
   d. Note: Do an excellent job of documenting as you write code. This discipline saves you time in the long run. You might find that you copy and paste methods. You want to copy good document that just takes a few moments to adjust for a new class.
   e. Write a constructor with one parameter specifying the label to be given the shape.
   f. Write an assessor (getter) and a mutator (setter) for the label instance variable.
   g. Write methods:

i. `calculateArea`
   1. What is the return type?
   2. Is it concrete or abstract?

ii. `calculatePerimeter`
   1. What is the return type?
   2. Is it concrete or abstract?

iii. `compareTo`
   1. The comparison is dependent upon the geometric shape; thus, this is an abstract method.
   2. Add the annotation `@Override`.

iv. `compareAnotherWay` – *This is a class activity.*
   1. Write a comparison of the two areas, using `isNearlyEqual`.
   2. Add the annotation `@Override`.

v. `isNearlyEqual` – *This is a class activity.*
   1. Write a comparison for two `Double` objects, taking into account `EPSILON`.
   2. Add the annotation `@Override`.

vi. `compareToNearly` – *This is a class activity.*
   1. Add the annotation `@Override`.

vii. `isPolygon`

viii. `toString` – *This is a class activity.*
   1. Add the annotation `@Override`.

```
/**
 * Formats the geometric object's class name, the object's
 * label, its area and perimeter.
 *
 * @return a formatted line about the geometric object
 */
@Override
public String toString( )
{
    String className = this.getClass().getName();
    if (className.equals("IsoscelesRightTriangle"))
        className = "IsoRtTri";
    else if (className.equals("EquilateralTriangle"))
        className = "EquilTri";
    return String.format("%s\t %s\n\t\t\tarea = %8.5f\tperimeter = %8.5f ",
                         className,
                         getLabel( ),
                         calculateArea( ),
                         calculatePerimeter( ) );
}
```

VII. Create a concrete class called `Circle`.
   a. What does this class extend?
   b. What does it need to implement? Think.

c. What instance variable does it need?
d. Create constructors.
    i. Create one constructor that accepts two parameters.
    ii. Create a unit circle constructor with a label parameter that utilizes the keyword `this` to invoke the constructor with two parameters.
e. Write methods:
    i. Note: not all methods are listed. You need to determine and write the appropriate methods so that this concrete class compiles.
    ii. `setRadius` – *This is a class activity.*
        1. Throws an `IllegalArgumentException` with the message "Radius needs to be positive".
        2. Remember to document the throw.
    iii. `calculatePerimeter`
        1. Use `Math.PI`.
        2. Add the annotation `@Override`.
    iv. `calculateArea`
        1. Use `Math.PI`.
        2. Add the annotation `@Override`.
    v. `isPolygon`
        1. For the sake of this project, a polygon is defined as a geometric shape with finite number of straight lines.
        2. Add the annotation `@Override`.
    vi. `compareTo` – *This is a class activity.*
        1. Compare two radii.
        2. Make use of `compareToNearly`.
        3. Return -999 if `Object` parameter is not an instance of a `Circle`.
        4. Add the annotation `@Override`.
    vii. `equals`
        1. Make use of `Circle`'s `compareTo`.
        2. Add the annotation `@Override`.
    viii. `toString` – *This is a class activity.*
        1. Make use of `super.toString`.
        2. Add the annotation `@Override`.

```
@Override
public String toString ()
{
    return super.toString( ) +
           String.format("\n\t\t\tradius = %10.5f",this.getRadius());
}
```


f. Test your code using `CircleTester`. The expected output is found in the document 4411 SK CircleTester Output, which is part of the downloaded files for this project – including this one.

VIII. Create an abstract class called `Polygon`.
- a. Create an instance variable that stores the number of sides of the `Polygon` object.
- b. Write an appropriate constructor that takes two parameters.
- c. Write these methods, deciding case-by-case whether abstract or concrete is appropriate.
  - i. `isPolygon`
    1. Add the annotation `@Override`.
  - ii. `isRegular`
  - iii. `getNumSides`

IX. Create a concrete class called `Triangle`.
- a. What does it extend?
- b. Create three instance variables: `sideA`, `sideB`, and `sideC`.
- c. Write a constructor that takes a label and 3 side lengths.
  - i. Use `orderSides`. Look down a few lines for instructions.
- d. Write methods:
  - i. Note: not all methods are listed. You need to determine and write the appropriate methods so that this concrete class compiles.
  - ii. setters that must use `orderSides`.
  - iii. `orderSides` – postcondition: `sideA` >= `sideB` >= `sideC`. *This is a class activity.*
    1. Throw an `IllegalArgumentException` with the message "Invalid side lengths".
    2. Remember to document the throw.
  - iv. `calculatePerimeter`
    1. Add the annotation `@Override`.
  - v. `calculateArea`
    1. Use Heron's formula.
    2. Use `calculatePerimeter`.
    3. Add the annotation `@Override`.
  - vi. `equals`
    1. Return `true` if the 3 side lengths are approximately equal to the 3 side lengths of another triangle. Return `false` if they are not or if the `Object` obj is not a `Triangle`.
    2. Add the annotation `@Override`.
  - vii. `compareTo` – *This is a class activity.*
    1. Use `equals` to return 0.
    2. Use `compareToNearly`.
    3. Return a positive number under the following algorithm.
       a. If `this.sideA` > obj's `sideA`.

b. If `this.sideA` is nearly equal to obj's `sideA` and `this.sideB` > obj's `sideB`.
c. If `this.sideA` is nearly equal to obj's `sideA`, `this.sideB` is nearly equal to obj's `sideB`, and `this.sideC` > obj's `sideC`.

4. Return a negative number using the following algorithm.
   a. If `this.sideA` < obj's `sideA`.
   b. If `this.sideA` is nearly equal to obj's `sideA` and `this.sideB` < obj's `sideB`.
   c. If `this.sideA` is nearly equal to obj's `sideA`, `this.sideB` is nearly equal to obj's `sideB`, and `this.sideC` < obj's `sideC`.
5. Return -999 if the `Object` obj is not a `Triangle`.
6. Add the annotation `@Override`.

viii. `isRegular`
1. Make use of `isNearlyEqual`.
2. Add the annotation `@Override`.

ix. `toString` – *This is a class activity.*
1. Make use of `super.toString`.
2. Add the annotation `@Override`.

```
@Override
public String toString ( )
{
    String str = "\n\t\t\ta = %8.5f\tb = %8.5f\tc = %8.5f" +
                 "\n\t\t\tside count = %d\t\tregular: %b";
    return super.toString( ) +
           String.format( str,
                   getSideA( ),
                   getSideB( ),
                   getSideC( ),
                   getNumSides( ),
                   isRegular( ));
}
```

e. Write your own `TriangleTester`, using `CircleTester` as a model.

X. Create a concrete class called `IsoscelesRightTriangle`.
a. What is needed here? Think.
b. It has no instance variables.
c. Constructor takes a label and the length of a leg.

d. Override `setSideA`, `setSideB`, and `setSideC`. They must use `orderSides`. Why? Add the correct annotation.

XI. Create a concrete class called `EquilateralTriangle`.

a. This is a class for equilateral triangles.
b. What is needed here?
c. It has no instance variables.
d. Create one constructor for `EquilateralTriangle` with two parameters.
e. Override `setSideA`, `setSideB`, and `setSideC`. They must use `orderSides`. Why? Add the correct annotation.

XII. Create a concrete class called `Rectangle`.

a. This is analogous to `Triangle`. What needs to be done?
b. It needs two instance variables: `length` and `width`.
c. The constructor accepts three parameters in this order: a label, a length, and a width. Do **not** create an `orderSides`. Lengths are not necessarily larger than width.
d. Write methods:
    i. Note: not all methods are listed. You need to determine and write the appropriate methods so that this concrete class compiles.
    ii. Write `setLength`.
        1. Use `Circle`'s `setRadius` as a model.
        2. What is a reasonable value for the length?
        3. Throws an `IllegalArgumentException` with the message "Length is not legal for a rectangle".
        4. Remember to document the throw.
    iii. Write `setWidth`.
        1. Use `Circle`'s `setRadius` as a model.
        2. Throws an `IllegalArgumentException` with the message "Width is not legal for a rectangle".
        3. Remember to document the throw.
    iv. Write `compareTo`.
        1. Use `Triangle`'s `compareTo` as a model.
        2. Compare two lengths before comparing the two widths. This is much like `compareTo` in `Triangle` but using lengths before widths, instead of `sideA`s before `sideB`s.
        3. Make use of `isNearlyEqual`.
        4. Return -999 if the `Object` obj is not a `Rectangle`.
    v. `toString` – *This is a class activity.*
        1. Make use of `super.toString`.
        2. Add the annotation `@Override`.


```
@Override
public String toString ( )
{
    String str = "\n\t\t\tlength = %8.5f\twidth = %8.5f" +
                 "\n\t\t\tside count = %d\t\tregular: %b";
    return super.toString( ) +
           String.format( str,
                          getLength( ),
                          getWidth( ),
                          getNumSides( ),
                          isRegular( ));
}
```

XIII. Create a concrete class called `Square`.

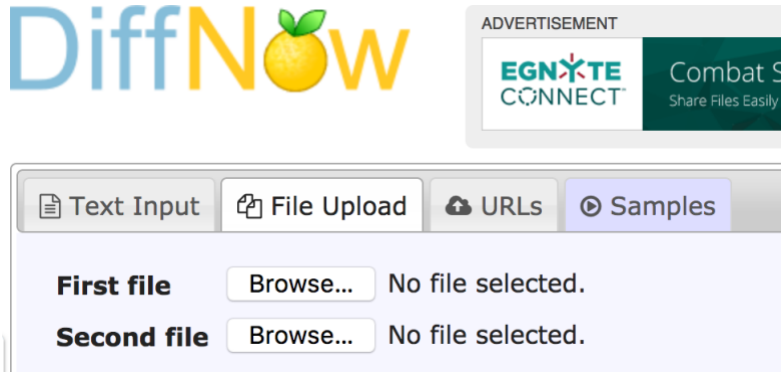
a. What is this analogous to?
b. It has no instance variables.
c. Create two constructors for `Square`:
    i. A constructor with two parameters.
    ii. A unit square constructor with a label parameter that utilizes the keyword `this` to invoke the constructor with two parameters.
d. Write `setLength` and `setWidth`. Be careful to maintain the square-ness of the object.

XIV. Verify that Checkstyle has no errors. All overridden methods must be explicated documented for this assignment.

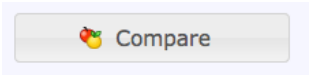
XV. Test and validation of project.

a. Find your BlueJ GeometricShape folder. Know its pathname. Do not skip this step. Get help if you need it.
b. Make sure that the following two files are this project's folder: `GeometricShapeFileTester` and `MasterOut.txt`.
c. Test with `GeometricShapeFileTester`. Run its `main`. When it runs, it does not output to the terminal window. Instead, it outputs the results in **out.txt**.
d. Look at the contents of out.txt. For example, are the names of the different shapes `null`? If so, you have some more debugging to do.

e. Compare the two output files. There are easy ways to do this. DiffNow is nice but there is a limit of how often you can run it. The other way is to use www.QuickDiff.com. The tools are similar. The following describes how to use DiffNow. Go to www.diffnow.com



- Click on **File Upload** tab
- At the **First file**, browse to **MasterOut.txt**
- At the **Second file**, browse **out.txt**
- Click on



The two files are compared. The differences are highlighted. You are done when there are no differences. Make corrections as needed.

- Rerun the test until the output is correct.

Image credits:

Gold rectangle smiley face (clip art). (no date). Emoticon Line in Shape. Kissclipart. Retrieved from https://www.kissclipart.com/image-of-rectangle-clipart-clip-art-g8hnbw/.

Happy Faces #1214827(clip art). (no date). Clipart Library. http://clipart-library.com/green-smiley-face.html.

Right triangle (clip art). (no date). JIMMYESP. Retrieved from https://jimmyesl.com/shape-names-vocab/.

Triangle smiley (clip art). (no date). Openclipart. Retrieved from https://publicdomainvectors.org/en/free-clipart/Triangle-smiley/64098.html.

