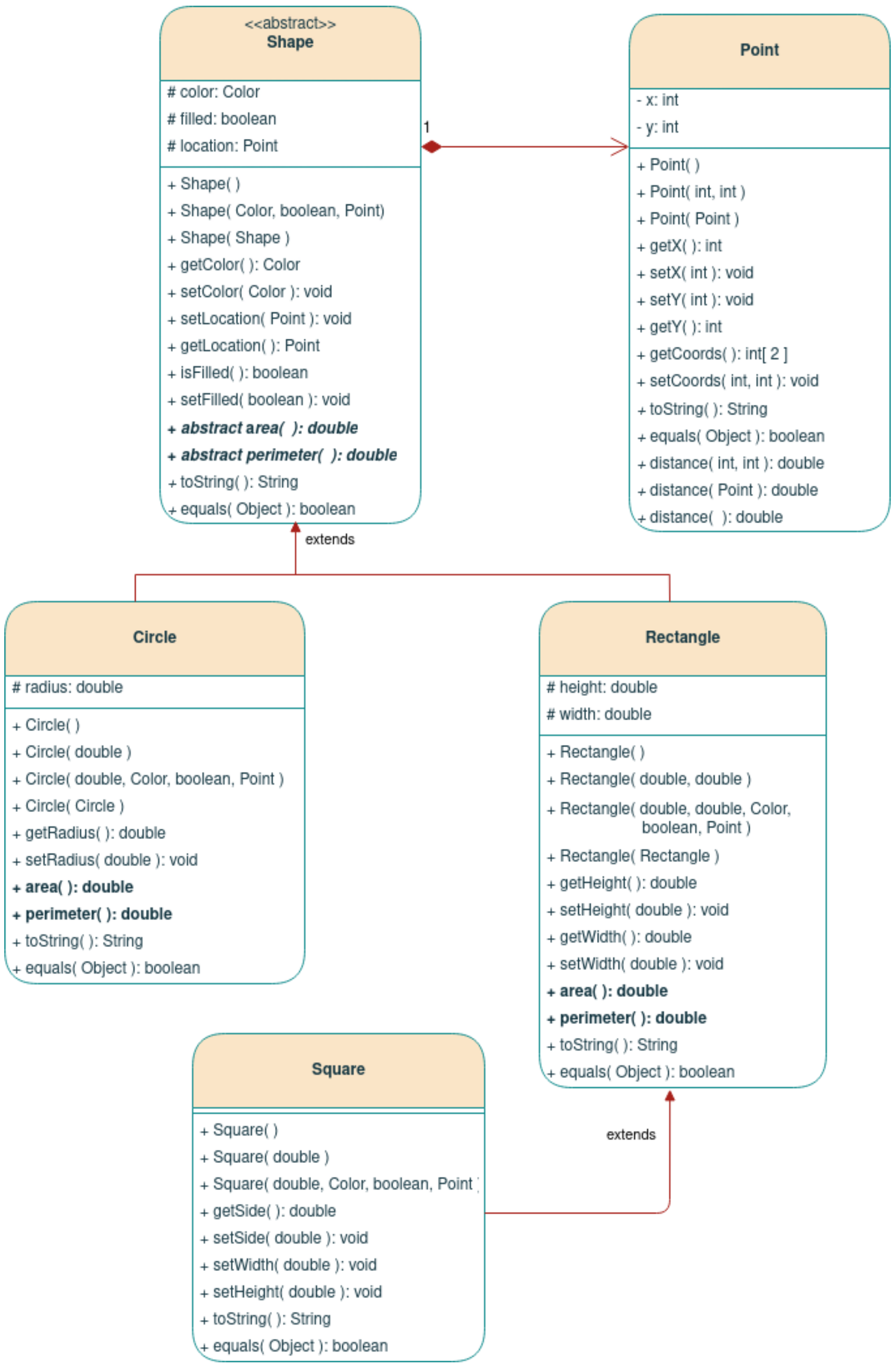
CSCI165 Computer Science II
Lab Assignment
Abstract Classes, Inheritance, Composition and Polymorphism

You should read the document ***Inheritance_Abstract_Classes.pdf*** before you begin.

Abstract Superclass Shape and Its Concrete Subclasses



TASK ONE: Define the abstract superclass `Shape` and its subclasses `Circle`, `Rectangle` and `Square`, as shown in the class diagram above.

There are two new concepts in this lab: ***abstraction*** and ***protected access***.

Abstraction: `Shape` is an `abstract` class containing 2 `abstract` methods: `getArea()` and `getPerimeter()`. Remember that an abstract method is one that does not provide a method body (implementation). The design goal here is to force a sub class to provide its implementation by overriding the abstract method. This mechanism allows us to ensure that all classes in this inheritance hierarchy (below Shape) will possess this behavior, guaranteeing that we will be able to invoke these methods on concrete subclasses without having to downcast. This allows us the ability to plug new objects into existing software with minimal effort, as long as there is a ***guarantee*** that any new object introduced into the hierarchy will possess these behaviors. Abstract classes and methods is this guarantee. The subclass **must** implement the abstract methods or the code will not compile.

Protected Access: in the Shape class, all instance variables have `protected` access, i.e., they are directly accessible via subclasses and classes in the same package. Protected access allows us to loosen the privacy protection for subclasses. This is a design choice.

Method overriding: To ensure that the compiler performs checks on our overridden methods, mark them all with the annotation `@Override`.

In this exercise, define `Shape` as an `abstract` class, which contains:

- Three `protected` instance variables `color`(`java.awt.Color`), `filled`(`boolean`) and `location(Point)`. The `protected` variables can be directly accessed by its subclasses and classes in the same package but not accessed in any other way. They are denoted with a `'#'` sign in the class diagram.
- Use the `Point` class from last week's lab
- Getter and setter for all the instance variables, and `toString(), equals().`
- Two `abstract` methods `getArea()` and `getPerimeter()` (shown in italics in the class diagram). These methods will not have method bodies because we are not implementing the behavior at this level (there is not enough information), we are simply declaring them so that they can be invoked through a reference variable of type `Shape`.
- `equals()` should include ***super*** calls and appropriate type, identity, origin and null checks.

The subclasses `Circle` and `Rectangle` will ***override*** the `abstract` methods `getArea()` and `getPerimeter()` and provide the proper ***implementation*** since there will be appropriate details. `Circle` and `Rectangle` will then be considered `concrete` classes

They also *override* the `toString()` and `equals()` but understand that the overriding of these methods is not ***required*** because there is inherited implementation from the super class. We override these methods by choice, not by mandate. Abstract classes issue an overriding mandate

The subclass `Square` does not add new data into the hierarchy but it will need to add some rules validation. A `Square` ***is a*** `Rectangle` with the additional constraint of the length and width being the same. How do we accomplish this?

We override the `setLength()` and `setWidth()` methods and add the constraints there. Then when dealing with a Square instance you will only be dealing with the overridden versions of those methods.

Even if you up-casted the Square to a Rectangle, the ***polymorphism mechanism*** would ensure that the Square versions of these methods would be called. Very nice. Understand that this method overriding is not mandatory, as there is inherited implementation from the super class. This is a design choice. We ***do not*** need to override the methods `getArea()` and `getPerimeter()` because they were implemented in `Rectangle` and inherited and the implementation can be shared.

TASK TWO: Write a Driver to test the following statements involving polymorphism and explain the outputs in comments. I am interested in you being able to identify the correct version of the various method invocations. ***Some statements may trigger compilation errors.***

Explain the errors, if any.

Note: Trying to copy/paste from a PDF could cause more headaches than it's worth. Don't attribute a text encoding issue to code errors.

```
Point p = new Point(1, 2);
Shape s1 = new Circle(5.5, "red", false, p);
System.out.println(s1);                 // which versions?
System.out.println(s1.getArea());
System.out.println(s1.getPerimeter());
System.out.println(s1.getColor());
System.out.println(s1.isFilled());
System.out.println(s1.getRadius());

Circle c1 = (Circle)s1;                // which versions?
System.out.println(c1);
System.out.println(c1.getArea());
System.out.println(c1.getPerimeter());
System.out.println(c1.getColor());
System.out.println(c1.isFilled());
System.out.println(c1.getRadius());

Shape s3 = new Rectangle(1.0, 2.0, "red", false, p);
System.out.println(s3);                // which versions?
System.out.println(s3.getArea());
System.out.println(s3.getPerimeter());
System.out.println(s3.getColor());
System.out.println(s3.getLength());

Rectangle r1 = (Rectangle)s3;          // which versions?
System.out.println(r1);
System.out.println(r1.getArea());
System.out.println(r1.getColor());
System.out.println(r1.getLength());

Shape s4 = new Square(6.6);            // which versions?
System.out.println(s4);
System.out.println(s4.getArea());
System.out.println(s4.getColor());
System.out.println(s4.getSide());

Rectangle r2 = (Rectangle)s4;          // which versions?
System.out.println(r2);
System.out.println(r2.getArea());
System.out.println(r2.getColor());
System.out.println(r2.getSide());
System.out.println(r2.getLength());

Shape s2 = new Shape();                // which versions?
Square sq1 = (Square)r2;
System.out.println(sq1);
System.out.println(sq1.getArea());
System.out.println(sq1.getColor());
System.out.println(sq1.getSide());
System.out.println(sq1.getLength());
```

TASK THREE: In the Driver class, define the method

`public static Shape findLargest( ArrayList< Shape > shapes)`

This method should find and return the Shape with the largest area without any typecasting. From main build an `ArrayList` of shapes and fill it with a collection of `Circle`, `Rectangle` and `Square` instances. Pass the collection to the `findLargest` method and print the results via a call to `toString()`. **Write a Unit Test to ensure this works**

In comments, describe how this method is able to accomplish this task.

In the Driver class, define the method

`public static double totalArea( ArrayList< Shape > shapes)`

This method should compute the total area of all Shape objects combined. Perform the same demonstration as described above. **Write a Unit Test to ensure this works**

TASK FOUR: In similar fashion to the classes above, create the class `Triangle` as a subclass of `Shape`. Appropriately override `toString()` and `equals()`.

Include the **@Override** annotation for all overridden methods.

Now add some Triangle instances to the `ArrayList` in the Driver and call `findLargest()` again. Notice how you can easily define and introduce a brand new Object into an existing code base. The definition of `findLargest()` and `totalArea()` do not need to change and can adapt to new objects being introduced and is able to run code ***that did not even exist*** just a few minutes earlier. This is the true utility of abstraction, polymorphism and inheritance.

In comments, describe how this method is able to accomplish this without modification.

TASK FIVE: Add one final class, `SemiCircle` as a sub class of Circle. Understand that you are not ***required*** to override `getArea()` and `getPerimeter()` for this class due to the inherited versions, but you do need to override those methods to ensure proper functionality.

Plug some `SemiCircles` into the ArrayList and let polymorphism do it's magic. The most interesting aspect of this exercise is that the `findLargest()` and `totalArea()` methods does not need to change, and can easily adapt to new objects being introduced.

TASK SIX: Have each of the classes in the hierarchy implement a custom interface called `Resizable` that contains a single method definition **`void resize( int percent )`** Implement the method by defining how to resize each of your objects**.**

In the Driver class create the method
`public static void resize( ArrayList< Resizable > resizables, int percent )`

Iterate through the list and resize each of the objects by the percent argument. Display the toString for each of the shapes before and after the resize.

Rubric on following page

Requirement	Points
Correctness: Classes meet all requirements and functions as described in the instructions. Classes pass all unit tests based on requirements.	10
Inheritance and Composition: Each concept and all of its unique characteristics implemented correctly	5
Abstraction and Polymorphism: Method overriding is correct and polymorphism is demonstrated without type casting.	5
Comments: All lab questions on thoughts/explanations are properly addressed	5
Javadoc: methods are documented using appropriate Javadoc style comments	2
Unit Tests: Tests for findLargest and totalArea are thorough	3

SUBMISSION: Push all relevant files to your repo.