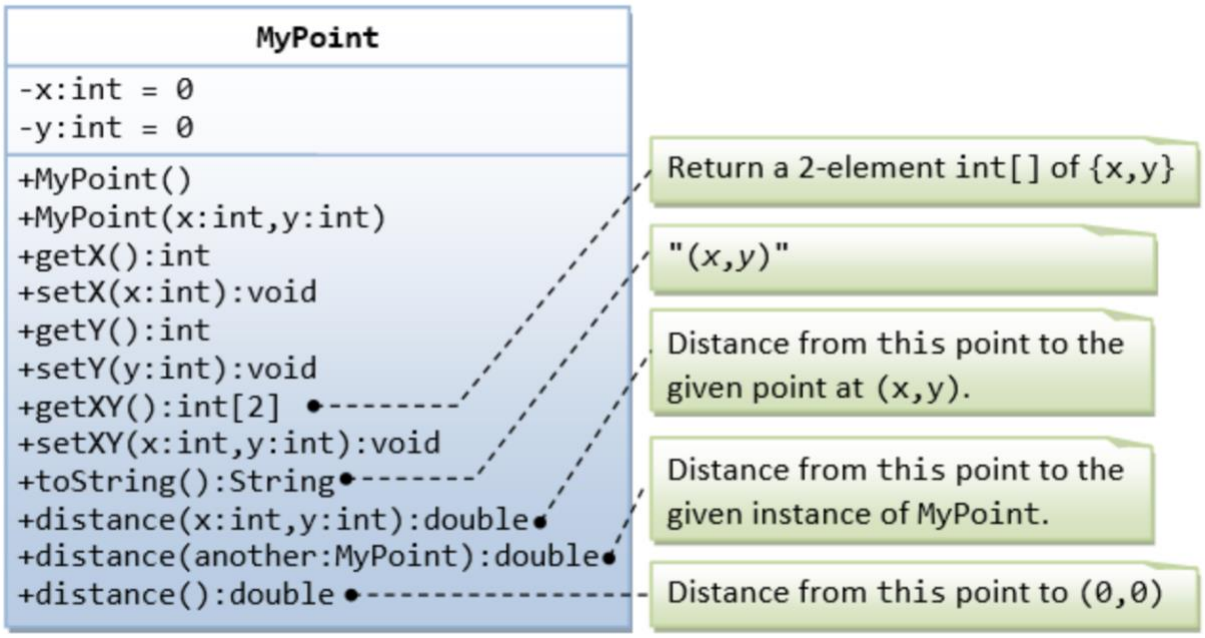
CSCI165 Computer Science II
Lab Assignment
Inheritance and Composition and a glimpse into Polymorphism



Define a class called `MyPoint`, which models a 2D point with `x` and `y` coordinates. It contains:

- Two instance variables `x` (`int`) and `y` (`int`).
- A no-argument constructor that constructs a point at the default location of `(0, 0)`.
- An overloaded constructor that constructs a point with the given `x` and `y` coordinates.
- Getter and setter for the instance variables `x` and `y`.
- A method `setXY()` to set both `x` and `y`.
- A method `getXY()` which returns the x and y in a 2-element `int` array.
- A `toString()` method that returns a string description of the instance in the format "`(x, y)`".
- A method called `distance(int x, int y)` that returns the distance from *this* point to another point at the given `(x, y)` coordinates, e.g.,

```
MyPoint p1 = new MyPoint(3, 4);
System.out.println(p1.distance(5, 6));
```

- An overloaded `distance(MyPoint another)` that returns the distance from *this* point to the given `MyPoint` instance (called `another`), e.g.,

```
MyPoint p1 = new MyPoint(3, 4);
MyPoint p2 = new MyPoint(5, 6);
System.out.println(p1.distance(p2));
```

- Another overloaded `distance()` method that returns the distance from `this` point to the origin `(0,0)`, e.g.,

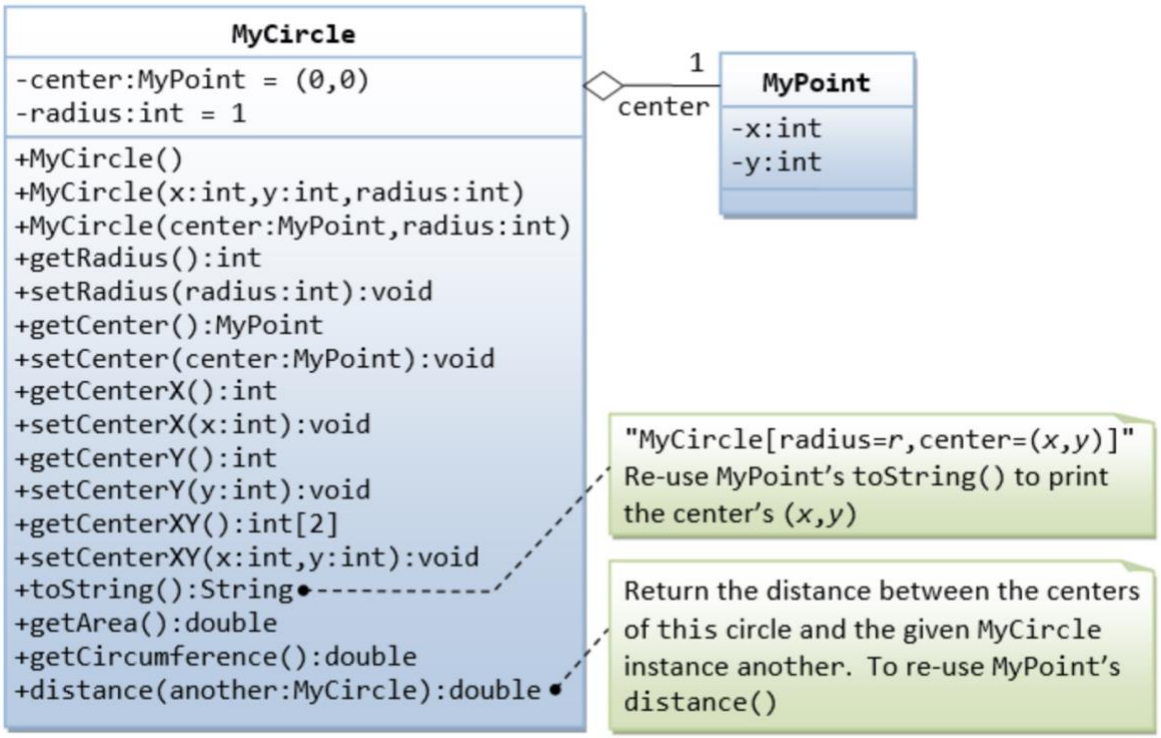
```
MyPoint p1 = new MyPoint(3, 4);
System.out.println(p1.distance());
```

- Although not shown in the UML diagram, add an equals method with the @Override annotation. Check the readings for an example. ASK QUESTIONS!! I expect you to understand the details here.

Driver:

Create a Driver class that allocates `10` points in an array of `MyPoint`, initialized to `(1, 1)`, `(2, 2)`, ... `(10, 10)`. You must use a loop for this. Loop through the array calling toString on each instance.

Composition Exercise:



Define a class called `MyCircle`, which models a circle with a `center (x,y)` and a `radius`. The `MyCircle` class uses an instance of `MyPoint` class (created in the previous exercise) as its `center`.

The class contains:

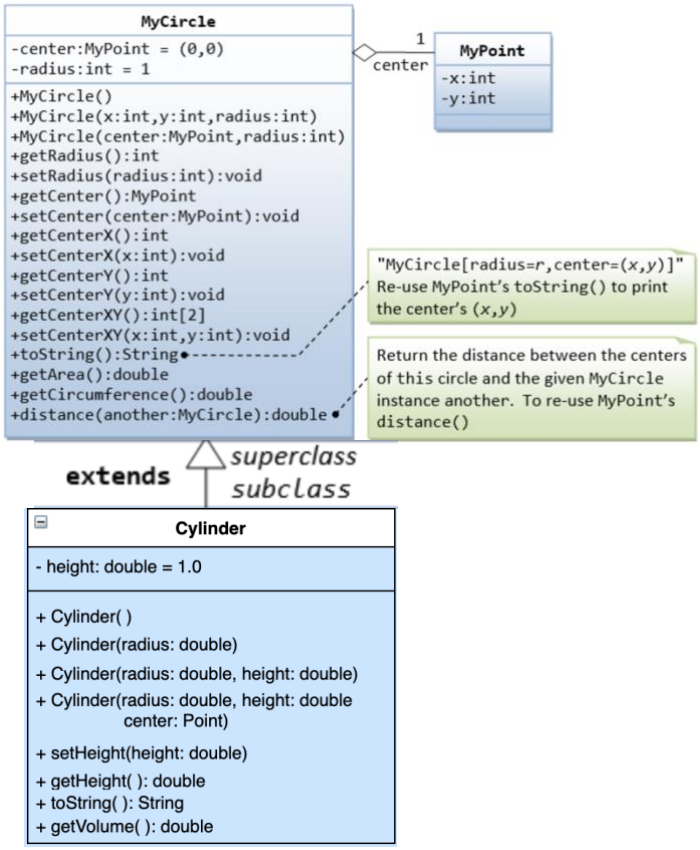
- Two `private` instance variables: `center` (an instance of `MyPoint`) and `radius` (`int`).
- A constructor that constructs a circle with the given center's (`x`, `y`) and `radius`.

- An overloaded constructor that constructs a `MyCircle` given a `MyPoint` instance as `center`, and `radius`.
- A *no-argument* constructor that constructs a circle with center at `(0,0)` and radius of `1`.
- Various getters and setters: check the UML diagram. PROTECT PRIVACY . . . you may need to add copy constructors
- A `toString()` method that returns a string description of this instance in the format "`MyCircle[radius=r,center=(x,y)]`". Reuse the `toString()` of `MyPoint`.
- `getArea()` and `getCircumference()` methods that return the area and circumference of `this` circle. Use the @Override annotation
- A `distance(MyCircle another)` method that returns the distance of the centers from `this` instance and the given `MyCircle` instance. You should use `MyPoint`'s `distance()` method to compute this distance.
- Although not shown on the UML diagram, add an equals method with the @Override annotation.

Driver:

Using the same Driver class, allocate 10 circles in an array of type `MyCircle`. Use the previously created points as the centers.

Inheritance and Composition Exercise:



In this exercise, a subclass called `Cylinder` is derived from the superclass `MyCircle` as shown in the class diagram. Study how the subclass `Cylinder` invokes the superclass' constructors (via `super()` and `super(radius)`) and inherits the variables and methods from the superclass `Circle`. Method inheritance is illustrated in the `getVolume()` definition shown below. You cannot directly reference the height property because it is private, but you can get it via a method call.

```
public class Cylinder extends MyCircle {

    private double height;

    // Constructor with default color, radius and height
    public Cylinder() {
        super(); // call superclass no-arg constructor
        height = 1.0;
    }

    // Constructor with default radius, color but given height
    public Cylinder(double height) {
        super();
        this.height = height;
    }

    // Constructor with default color, but given radius, height
    public Cylinder(double radius, double height) {
        super(radius); // call superclass overloaded constructor
        this.height = height;
    }

    // A public method for retrieving the height
    public double getHeight() {
        return height;
    }

    // A public method for computing the volume of cylinder
    // use superclass method getArea() to get the base area
    public double getVolume() {
        return getArea() * height;
    }
}
```

Add the following constructor

- One that accepts a MyPoint, radius and height. Call the appropriate super class constructor

Equals method

- Add an **equals( )** method. This equals method must have the @Override annotation

Driver:

Using the same Driver class

- Instantiate a few Cylinder objects, demonstrating the various constructors.
- Demonstrate that you can call various inherited MyCircle methods through the Cylinder instances.

Polymorphism Foreshadowing:

- Create a new array of type MyCircle of Size 10
- Because of the inheritance relationship ***Cylinder is a MyCircle*** you can place Cylinder objects into a collection of type MyCircle. Illustrate this by choosing 5 MyCircle instances from the previously defined array and also add 5 Cylinder objects.
- **Polymorphic behavior:** Loop through the array and call toString on each object. Analyze the output and notice that the appropriate toString was called for each object, even though the array is of type MyCircle. This is polymorphic behavior.
- Try to call a method that is just defined in Cylinder and watch the compiler complain. Why is this? Do you see any requirements for polymorphic behavior? This is subtle.
- Let's take this one step further. Define an array of type Object of size 9. Notice the inheritance relationships
    - A MyPoint is an Object
    - A MyCircle is an Object
    - A Cylinder is a MyCircle, therefore by extension, it is also an Object
- Place 3 instances of each of those classes into the Object array. Loop through the array calling toString. Analyze the output and notice that the appropriate toString is being called for each instance.
- Try to call a method that is unique to the subclasses. Why does the compiler complain?
- **Experiment:** Add instances of ***ANY*** class you have defined into this Object array.
    - Customer
    - Address
    - Product
    - Date
    - String
    - WHATEVER
- Loop through the array and call toString on each instance. How does Java know which method to call? How is this even happening?

Submission: Push whatever files you have. I will need everything to run your code. With no modifications