Reviewing OO Concepts

Users want to draw circles onto the display canvas.

```java
public class Circle {
    // more code here
}
```
OO Programming is about visualizing a class, modeling the class and then coding the class.

- Programming is and will always be a mental activity.
- UML modeling gives shape to your mental model.
  - *To make your mental model more concrete*
  - *To validate your mental model with stakeholders*
  - *To share with other developers*
- The UML model acts as a guide during development.
The object-oriented paradigm is based on several basic concepts.

- These include:
  - Object identity
  - Abstraction
  - Encapsulation
  - Information hiding
  - Associations
  - Inheritance
  - Polymorphism

Imagine a drawing application in which the user can place shapes on a canvas. Let's start with a circle.
All OO programming starts with classes and objects.

- A class is a template for run-time objects.
- Use UML class notation to model your mental model of a circle.
- Java classes implement these models.

public class Circle {
    // more code here
}
One class may have many unique objects.

```java
public void make_multiple_objects() {
    Circle c1 = new Circle();
    Circle c2 = new Circle();
    Circle c3 = new Circle();
    if (c1 != c2) {
        // Two distinct objects have different identities.
    }
}
```
A large part of object-oriented design is about assigning responsibilities to classes.

- Considering a circle, the user will need to:
  - Select a circle by clicking on it.
  - Move a circle by dragging it to a new position.
  - Scale the circle by dragging the edge.

- Of course the set of behaviors is totally dependent upon the domain of the specific application. For example a CAD app also provides:
  - Show circumference and area of a circle
  - Align circles and with other shapes to a grid
  - Calculate unions, intersections, and exclusions between circles and other shapes

- We'll talk about design more fully later but for now let’s focus on OO concepts and UML.
Objects perform behaviors defined by their class.

- Look to the verbs to identify behaviors.
- As an artist I also need to:
  - **Select** a circle by clicking on it.
  - **Move** a circle by dragging it to a new position.
  - **Scale** the circle by dragging the edge.
- This starting point forms a sketch of a Java class.

```java
import java.awt.*;

public class Circle {
    void draw() { /* TBD */ }
    boolean hasPoint() { /* TBD */ }
    void move() { /* TBD */ }
    void scale() { /* TBD */ }
}
```
Objects use attributes defined in the class while performing behaviors.

- Include the known attributes of an object in the class definition.
- Identify the data types for each attribute.
  - *Might be "primitives" like int and String*
  - *Or it might be other domain types, like Position*
- Keep the attributes hidden using `private`

```java
public class Circle {
    private Position center;
    private int radius;
    // more code here
}
```
Design the class interface to provide the behaviors that the client needs.

- Getters and setters are not benign!
  - *Provide them only when absolutely necessary*

- Provide semantically interesting methods
  - *Don't use setCenter(), rather the circle movesTo() a position*

- Be particularly careful about exposing the class' data structures like maps, sets, lists, etc.
  - *Don't provide getters and setters for these*
OK, let's go back to our developer. She now needs to design a Rectangle class.

Users want to draw rectangles onto the display canvas. And select, move and scale them.

Do you notice any duplication with Circle?

### Rectangle
- `topLeftCorner : Position`
- `width : int`
- `height : int`

+ `move(p:Position) : void`
+ `scale(f:float) : void`
+ `draw()`
+ `hasPoint(p:Position) : boolean`

### Circle
- `center : Position`
- `radius : int`

+ `draw()`
+ `hasPoint(p:Position) : boolean`
+ `move(p:Position) : void`
+ `scale(f:float) : void`
There's a principle in software development: *Don't repeat yourself.*

- Both *Circle* and *Rectangle* have a position.
- They have `move` methods and other methods with identical signatures.

- What should you do to not repeat yourself?
Pull shared attributes and behaviors into a super class.

The drawing app now deals with two kinds of shapes: circles and rectangles.

Shape

- position : Position
- move(p:Position) : void
- scale(f:float) : void
- draw()
- hasPoint(p:Position) : boolean

Circle

- radius : int
- scale(f:float) : void
- draw()
- hasPoint(p:Position) : boolean

Rectangle

- width : int
- height : int
- scale(f:float) : void
- draw()
- hasPoint(p:Position) : boolean
Should the super class be abstract?

- Specifically for the drawing app, can you add a "shape" (ie, a generic shape) to the canvas?
  - If yes, then it can not be abstract.
  - If no, then restrict the ability to instantiate the Shape class by making it abstract.
public abstract class Shape {
    protected Position position;

    protected Shape(final Position position) {
        this.position = position;
    }
    public void move(Position position) {
        this.position = position;
    }
    public abstract void draw();
    // more code not shown
}
Here's the code for the `Circle` subclass.

```java
public class Circle extends Shape {
    private int radius;

    public Circle(final Position center, final int radius) {
        super(center);
        this.radius = radius;
    }

    public void draw() { /* TBD */ }

    public void scale(float factor) {
        this.radius = (int) (radius * factor);
    }

    public boolean hasPoint(Position p) {
        return p.distanceTo(position) <= radius;
    }
}
```

Use the `extends` keyword to allow the `Circle` class to inherit the attributes and methods of the super class: `Shape`.

Use the `super` keyword to invoke the `Shape` constructor.

You can use protected members of the `Shape` class.
Our developer has been busy and has created the following Java/Swing application architecture.

Arrows indicate direction for navigation. DrawingUI has a reference (can reach) the DrawingCanvas.

Lines indicate association. Numbers indicate multiplicities. Each Shape is on only one DrawingCanvas. DrawingCanvas has zero or more Shape objects.

The role name indicates the name of the attribute shapes is the name of the Shape attribute in DrawingCanvas.
This simple system exhibits additional object-oriented programming concepts.

The Circle and Rectangle classes each define different implementations of the `draw()` method because you draw circles and rectangles differently.

The Graphics object to draw on is passed into the `draw()` methods in `DrawingCanvas` and the `Shape` implementations.
public class DrawingCanvas {
    private Set<Shape> shapes = new HashSet<>();

    public void addShape(final Shape s) {
        shapes.add(s);
    }

    public void draw(Graphics g) {
        // Draw each shape
        for (Shape s : shapes) {
            s.draw(g);
        }
    }
}

The DrawingCanvas class draws a set of shapes.

s is defined as a Shape object on which the draw() method is called. How does the Circle.draw() method get called for circles, and the Rectangle.draw() method for rectangles?
The lecture reviewed OO concepts and used defensive programming practices.

**OO Concepts Reviewed**
- Object identity
- Encapsulation
- Information hiding
- Inheritance
- Abstraction
- Associations
- Polymorphism

**Defensive programming**
- Private/protected attributes and methods
- Final attributes and parameters
- Minimized use of getters and setters
- Hide internal data structures