Reviewing OO Concepts

Users want to draw circles onto the display canvas.

```
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.
We'll use a drawing application as our example application domain.

- Imagine a drawing application in which the user can place shapes on a canvas. Let's start with a circle.

Users want to draw circles onto the display canvas.
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.

Users want to draw *circles* onto the display canvas.

```java
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.
    }
}
```
Objects perform behaviors defined by their class.

- Look to the verbs to identify behaviors.

```java
public class Circle {
    void draw() {
        // TBD
    }
}
```
OO design is all about assigning responsibilities to classes

- In a drawing app the user will need to:
  - *Select a shape by clicking on it.*
  - *Move a shape by dragging it to a new position.*
  - *Scale the shape by dragging a corner.*

- Of course the set of behaviors is totally dependent upon the domain of the specific application. For example a CAD app also provides:
  - *Show measurements (perimeter and area) of a shape*
  - *Align shapes to a grid*
  - *Calculate shape unions and intersections and exclusions*

- We'll talk about design later but for now let's focus on OO concepts and UML.
So the design for our drawing app will be...

- As an artist I also need to:
  - *Select a shape to interact with.*
  - *Move a shape by dragging it to a new position.*
  - *Scale the shape by dragging a corner.*

- In some cases we know return values of methods, shown here:
This design starting point can even form a sketch of a Java class.

- We still don't have some details, such as parameters to these methods.
- But we can sketch out a skeleton class:

```java
public class Circle {
    void draw() { /* TBD */ }
    boolean hasPoint() { /* TBD */ }
    void move() { /* TBD */ }
    void scale() { /* TBD */ }
}
```
In order to do the work of behaviors, objects will use attributes defined by the class.

- Include the known attributes of an object into the class definition.

A circle *has a center* position and a *radius*.

<table>
<thead>
<tr>
<th>Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>center</td>
</tr>
<tr>
<td>radius</td>
</tr>
<tr>
<td>draw</td>
</tr>
<tr>
<td>hasPoint</td>
</tr>
<tr>
<td>move</td>
</tr>
<tr>
<td>scale</td>
</tr>
</tbody>
</table>
Attributes have data types.

- Identify the data types for each attribute.
  - *Might be "primitives" like int and String*
  - *Or it might be other domain types, like Position*

Well, I did say the center is a **position**. The radius must be a number.

```java
public class Circle {
    Position center;
    int radius;
    // more code here
}
```
But make sure that you hide your attributes within the class.

- The way to hide a class's attributes is to make them private.

```java
public class Circle {
    private Position center;
    private int radius;
    // more code here
}
```

- And then provide methods to inspect or mutate the object.
  
  - *Only expose data if necessary; provide methods to do what the client needs rather than expose data*
  
  - *Only provide mutator methods as required, and use the domain language*
Getters and Setters are not benign.

- Don't do this:

```java
public class Circle {
    private Position center;
    private int radius;

    public Position getCenter() {
        return center;
    }
    public void setCenter(Position c) {
        this.center = c;
    }

    public int getRadius() {
        return radius;
    }
    public void setRadius(int r) {
        this.radius = r;
    }
}
```
Design methods that are semantically interesting.

- Don't set the center, rather the circle *moves*.

```java
public class Circle {

    private Position center;
    private int radius;

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

    public void move(Position p) {
        this.center = p;
    }

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

    public boolean hasPoint(Position p) {
        return p.distanceTo(center) <= radius;
    }
}
```
You should also hide your class's data structures such as lists, sets, maps and other collections.

- Don't create getters/setters to the collection:

```java
public class DisplayCanvas {
    private Set<Circle> circles = new HashSet<>();
    public Set<Circle> getCircles() {
        return circles;
    }
    public void setCircles(Set<Circle> circles) {
        this.circles = circles;
    }
}
```

- Protect your data structures:

```java
public class DisplayCanvas {
    private Set<Circle> circles = new HashSet<>();
    public Iterable<Circle> getCircles() {
        return circles;
    }
    public void addCircle(Circle circle) {
        this.circles.add(circle);
    }
}
```
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.

### Rectangle

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

+ `move(p:Position) : void`
+ `scale(f:float) : void`
+ `draw()`
+ `hasPoint(p:Position):boolean`
The Rectangle implementation looks like this.

```java
public class Rectangle {
    private Position topLeftCorner;
    private int width;
    private int height;

    public Rectangle(
        final Position topLeftCorner,
        final int width,
        final int height)
    {
        this.topLeftCorner = topLeftCorner;
        this.width = width;
        this.height = height;
    }

    public void move(Position toPosition) {
        this.topLeftCorner = toPosition;
    }

    public void scale(float factor) {
        width = (int) factor * width;
        height = (int) factor * height;
    }

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

    public boolean hasPoint(Position p) {
        /* TBD */
    }
}
```

Do you notice any duplication with Circle?
There's a principle in software development: *Don't repeat yourself.*

- Both *Circle* and *Rectangle* have a position.
- They have identical *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
```
Here's the code for the Shape super class.

```java
public class Shape {

    protected Position position;

    public Shape(final Position position) {
        this.position = position;
    }

    public void move(Position position) {
        this.position = position;
    }

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

    // 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.
Should the super class be abstract?

- Specifically for the drawing app, can you add a "shape" (ie, a generic shape) to the canvas?
  - *If so, then the current implementation is fine.*
  - *If not, then restrict the ability to instantiate the Shape class.*
Use italics on labels for abstract "things".

```java
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
}
```

- Make the class abstract.
- Make all constructors protected.
- Make some methods abstract.
Our developer has been busy and has created the following Java/Swing application architecture.

Looking at the attributes of the DrawingUI and DrawingCanvas classes it appears that there are relationships in the objects. Is there a UML notation for representing these types of relationships?
UML uses a line to connect classes that have associations.

Add a role name if the instance variable is known.
We can specify directionality of the associations.
We can also specify multiplicity of the associations.
Now let's consider how this is coded...
The UI code that delegates drawing to the canvas object passes in the Graphics context.

```java
public class DrawingUI extends JComponent {
    private final DrawingCanvas myDrawing;

    // more code here

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        // Draw the canvas
        myDrawing.draw(g);
    }
}
```
The DrawingCanvas class draws a set of shapes.

```java
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);
        }
    }
}
```
And now each shape's specific draw methods.

- The **Circle** draw method:

  ```java
  public void draw(Graphics g) {
    final int diameter = 2 * radius;
    final Position pos = getPosition();
    g.drawOval(pos.getX() - radius, pos().getY() - radius, diameter, diameter);
  }
  ```

- The **Rectangle** draw method:

  ```java
  public void draw(Graphics g) {
    final Position pos = getPosition();
    g.drawRect(pos().getX(), pos().getY(), width, height);
  }
  ```
Did you notice the polymorphism in the drawing code?

- Take a look again at the `DrawingCanvas` code:

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

How does the compiler know which shape draw method is invoked? (Circle or Rectangle?)
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