CA#1 Information – Part 1

Note: This piece of CA will involve creating and manipulating shapes in a Swing application. I've uploaded some template code and associated video to the VLE which should give you a basic framework to get started with. Also, in the video I talk a bit about the requirements of the CA – this may be useful to refer to.

Please note: some of the Java graphics libraries have their own shape classes, e.g. java.awt.Rectangle. We will not be using these classes – we will be providing our own custom classes.

Shapes

You will create your own Shapes inheritance hierarchy.

Initially, it should provide the following classes:

Shape - this will be an abstract superclass

Rectangle

Square

Circle

Note: All Shape Classes should override `toString()` to allow for console-based validation/debugging.

The Shape Class instance fields (not exhaustive)

```
Color color; //The Color class is in java.awt
boolean filled;
int xCenter, yCenter;
```

It is important to note that all our Shapes will be defined via the center of the of the object. This will require some mapping when it comes to drawing the shapes using, for example, java.awt.Graphics.drawRect().

The reason that I say "not exhaustive" is that other fields may be required to achieve the desired functionality. Of course, this is for you to figure out as you design and implement your solution. This caveat applies to all classes.

Your Shape Class will provide (at least) the following abstract method:

```
public abstract void drawShape (Graphics g);
```

I'll explain more about this later in the document.

The Rectangle Class instance fields

```
private int width;
private int height;
```

Note: I'm not defining anything to do with the Square class in this document. That will be left as an exercise.

The Circle Class instance fields

```
private int radius;
```

Bounding Box

All shapes will have an associated bounding box[1].

Initially, we can assume a relatively simple solution, because we have not (yet) introduced any requirements for things like rotation of shapes.

Figure 1 below illustrates a bounding box for a circle (for rectangles and squares, the bounding box will map exactly to the shape).

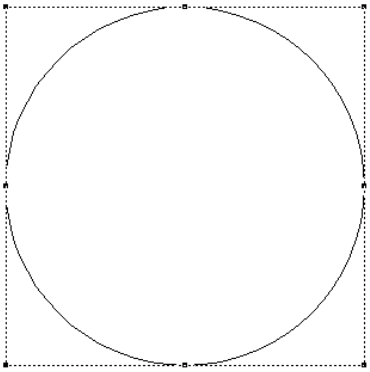

Figure 1: Bounding Box for a circle

And why might we require bounding boxes?

For Shape selection. If I click on the shape window of my application, I can determine if I have "selected" any shapes by comparing the mouse click's windowed coordinates with the coordinates of my bounding box.

While this isn't entirely accurate - there are obviously parts of the bounding box that are not part of the circle – it allows us to have a uniform mechanism for intersection checking. The alternative is that you would need different algorithms for different shapes[2].

Bounding Box Class Instance Fields

We can fully define a bounding box using two diagonally opposite points on the box.

```
private Point bottomLeft;
private Point topRight;
```

For convenience, we'll create our own Point class to store x,y coordinates.

Point Class Instance Fields

```
private int x;
private int y;
```

[1] **Bounding boxes** are *imaginary* **boxes** that are around objects that are being checked for things like intersection (part of our requirements) or collision (for games, physics, etc – not part of our requirements).

[2] I may add that alternative mechanism as an additional requirement later.

Shapes Manager

Rather than draw shapes directly (within paintComponent) as the template code does, we will use a ShapesManager class to store a list of shapes. That way we can, for example, iterate through our list checking if a mouse-click intersects with any of the shapes' bounding boxes.

The ShapesManager Class instance fields

```
private ArrayList<Shape> shapes;
```

The ShapesManager Class methods

```
public void drawShapes(Graphics graphicsContext)
```

This method will simply iterate over the list and call each shape's `drawShape()` method – remember that we defined `drawShape()` as an abstract method in the Shape class.

At this stage you might be asking yourself something along the lines of, "what exactly is the story with `public abstract void drawShape (Graphics g);` *?"*
Or perhaps, "How is the `java.awt.Graphics` *object supplied? Who supplies it?"*

Well, hopefully the information below will help to clarify:

Each component (in our case, a JPanel) can override a method with the following signature:

```
@Override
protected void paintComponent(Graphics g)[3]
```

Basically, this means that they (the panel/component) can include drawing commands to draw text, shapes, etc. directly within the component, `g.drawOval(100,100, 40,40);` Effectively, this implies that any method wishing to perform drawing will need access to that graphics object.

For our purposes, though, instead of doing our shape drawing directly within the `paintComponent()` method, we can delegate it to the ShapesManager:

```
shapesManager.drawShapes(g);//Now our classes can access the graphics context
```

So, this effectively means that….

[3] This method is called internally by the graphics sub-system whenever it deems appropriate, e.g. if you resize a window.

However, we can also force a call of this whenever we deem it appropriate, if we wish, by calling `repaint()`

The Class containing paintComponent() needs a reference to the ShapeManager object

In which class do we override paintComponent()?

Well, that's our CustomPanel class. And it exists as an object of our CustomWindow class (this is Composition, of course). So we need the proper constructors to pass a reference to the ShapeManager object down to the panel. The following snippet of code (which would be in our main() method) will hopefully help you to piece together what needs to be done:

```
CustomWindow aWindow = new CustomWindow(shapesManager);
```

At this stage, you should have all the requisite information needed to create your basic hierarchy, manage and maintain your shapes via a manager object, and display them on a Panel.

As you will have seen from the video clip, the next thing I want you to do is to toggle whether a shape in drawn in filled or outline mode every time I click on an area of the panel containing a shape's bounding box.

To do this you will need to figure out two more things: how to receive mouse clicks-in your code, and how to determine if that mouse-click's position intersects with any of the shapes.

Receiving Mouse-Clicks with addMouseListener()

This will be a very short treatment of the subject to give you the basics of what you need to know. If you'd like to investigate further then one site of interest might be:
https://docs.oracle.com/javase/tutorial/uiswing/events/mouselistener.html

Components (our Panel in this case) can register to receive mouse events by doing the following:

1. Implementing the mouseListener interface (or extending the abstract MouseAdapter class). See https://docs.oracle.com/javase/7/docs/api/java/awt/event/MouseListener.html
2. Calling the component's addMouseListener() method to register with the UI system to receive the events.

Since this isn't a GUI module, I'll give you some code that achieves those basic requirements (and I'll talk about it a little in class):

```
public class CustomPanel extends JPanel {

    public CustomPanel()
    {
```

```
        addMouseListener(new MouseAdapter() {
            /**
             * {@inheritDoc}
             *
             * @param e
             */
            @Override
            public void mousePressed(MouseEvent e) {
                super.mousePressed(e);

                //In this example I'm simply printing out the information stored in
                //the MouseEvent object
                System.out.println(e.toString());

            }
        });

    }
}
```

Determining if the mouse's x,y, position falls within the boundaries of a Bounding Box

Given that our bounding boxes are axis-aligned, it shouldn't be too difficult to figure out a basic algorithm. We can talk about it in class.

What do we do with the shapes that match/intersect the mouse position?

You should simply invert/toggle that shape's `isFilled` Boolean. The next time the panel is redrawn you should see the change (providing, of course, that your code already checks the isFilled value and draws accordingly).

Note: to force a redraw, you should call the panel's `repaint()` method.