

PROGRAMMING ASSIGNMENT 4:

Read Savitch: Chapter 7 and class notes

Programming: You will have 6 files - all need to be located in the dir. named **PA4**:

PA4.java	ShapeP4.java
PointP4.java	CircleP4.java
RectangleP4.java	TriangleP4.java

DUE: Tuesday, May 14, 2002 at 11:00pm

Turnin: **turninPA4**

You will copy the 5 classes you wrote in PA3 to your PA4 directory, make some modifications to them to support drawing shapes, and write a PA4 class (an applet) to implement a simple graphics manager to draw the shapes in a cartesian plane.

Be sure to compile with **javac *.java** to ensure all your files are compiled each time. There is a sample PA4 and PA4.html in the Assignments directory.

ShapeP4

Add the following abstract method declaration to your converted ShapeP5:

```
public abstract void draw( Graphics g, int xOffset, int yOffset );
```

draw() will be polymorphically invoked from class PA4 from within **paint()**. Graphics g is the reference to the Graphics context on which to draw (the same reference that is passed as a parameter to **paint()**). **xOffset** and **yOffset** are the relative offsets from the upper left corner of the applet where coordinate (0,0) is located in the cartesian plane. These are essentially one half of the current width and one half of the current height, respectively, of the applet.

PointP4

No new members. Just convert PointP3 to PointP4.

CircleP4

The **CircleP4()** no-arg constructor of this class will be rewritten to ask the user for the circle's center point's x and y coordinates and the circle's radius via separate **JOptionPane.showInputDialog()** queries. If the user presses the Cancel button on any input, skip this and the rest of the input and default the skipped input to 0. Keep asking for input if the input string is empty (zero length). You can assume only valid input will be entered.

To implement the **draw()** method: make a copy of the center point, translate this copy relative to the current offsets: **xOffset** in the X direction and (**yOffset** - 2*this point's y) in the Y direction. This translates the center point of the circle to be drawn to the coordinates relative to the center of the applet which is (0,0) in the cartesian plane. Now we need to calculate the upper left corner of the bounding box that defines the circle. This is done by translating the same center point by -radius in the X direction and -radius in the Y direction.

Now we are ready to draw the circle. Use **drawOval(int x, int y, int width, int height)** from class Graphics. The x and y offsets are just the values of x and y of the translated point, and width and height (of the bounding box) are both **radius*2**. Use **Math.round()** to round the doubles to longs and then cast to ints.

RectangleP4

The **RectangleP4()** no-arg constructor of this class will be rewritten to ask the user for the rectangle's upper left corner point's x and y coordinates and the rectangle's length and width via separate **JOptionPane.showInputDialog()** queries. If the user presses the Cancel button on any input, skip this and the rest of the input and default the skipped input to 0. Keep asking for input if the input string is empty (zero length). You can assume only valid input will be entered.

To implement the **draw()** method: make a copy of the upper left corner of the rectangle, translate this copy relative to the current offsets: **xOffset** in the X direction and (**yOffset** - 2*this point's y) in the Y direction. This translates the upper left corner of the rectangle to be drawn to the coordinates relative to the center of the applet which is (0,0) in the cartesian plane.

Now we are ready to draw the rectangle. Use **drawRect(int x, int y, int width, int height)** from class Graphics. The x and y offsets and the length and width are just the values of x, y, length, and width of the translated point. Use **Math.round()** to round the doubles to longs and then cast to ints.

TriangleP4

The **TriangleP4()** no-arg constructor of this class will be rewritten to ask the user for the triangle's x and y coordinates for the three points via separate **JOptionPane.showInputDialog()** queries. If the user presses the Cancel button on any input, skip this and the rest of the input and default the skipped input to 0. Keep asking for input if the input string is empty (zero length). You can assume only valid input will be entered.

To implement the **draw()** method: make a copy of the three points, translate these copies relative to the current offsets: **xOffset** in the X direction and (**yOffset** - 2*this point's y) in the Y direction. This translates each point of the triangle to be drawn to the coordinates relative to the center of the applet which is (0,0) in the cartesian plane.

Create two arrays of three ints each for the x and y coordinates of the points to draw the triangle. Use **drawPolygon(int[] xPoints, int[] yPoints, int nPoints)** from class Graphics. The x and y coordinates of the first point of the triangle should be in **xPoints[0]** and **yPoints[0]**, respectively. The x and y coordinates of the second point of the triangle should be in **xPoints[1]** and **yPoints[1]**, respectively. Likewise for the third point.

PA4

```
public class PA4 extends JApplet implements ListSelectionListener
{
    public void init() { ... }
    public void paint( Graphics g ) { ... }
    public void valueChanged( ListSelectionEvent e ) { ... }
}
```

class PA4 is an applet that will drive a simple graphics manager for the shapes above. To select which shape to draw, PA4 will use a **JList** with the names of the three shapes to draw. To identify which shape is selected in the **JList**, we will use a **ListSelectionListener** and implement the **valueChanged()** method. Each time a shape is selected, put a reference to the newly created shape into an array of ShapeP4 objects. For purposes of this assignment, make the size of this array five elements. Remember: all instance variables must be **private**.

init()

The **JList** object will be aligned to the top right of the applet. Use FlowLayout with right alignment:

```
Container c = getContentPane();
c.setLayout( new FlowLayout( FlowLayout.RIGHT ) );
```

Pass to the JList constructor an array of Strings with the names of the shapes as the array elements: **Circle**, **Rectangle**, and **Triangle**. Add *this* applet as the listener (event handler) for list selection events from the list of shape names. Set the selection mode to single selection.

```
list = new JList( shapeNames );
list.addListSelectionListener( this );
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

paint()

paint() will be called any time the applet needs to be drawn/redrawn. To redraw the lightweight GUI components (in this case just the JList object) on the applet after the applet has been cleared, call **paint()** in the superclass (JApplet). **getWidth()** and **getHeight()** is used to get the current width and height of the applet. Half of these is the offset in the x and y direction where we can find the (0,0) coordinate of the cartesian plane on which all the shapes will be drawn. Remember, applets use the upper left corner of the applet as location (0,0) and use only positive x and y offsets from that (0,0) location. We will be using a cartesian plane in which (0,0) is the center point of the applet no matter how we change the dimension of the applet.

Set the drawing color to black and draw the x and y axes as shown in the examples and the PA4 sample program. Place all +/- bounds at an inset of 10 pixels. A useful method **g.getFontMetrics().stringWidth(str)** should be used to determine the number of pixels wide a given string is in the current font. You should use this (and the inset) to determine where to draw the positive x offset/bounds of the X axis.

Once the lines and values of the axes are drawn, change the drawing color to blue. Cycle through the array of ShapeP4 objects for the number of shapes in the array, and polymorphically invoke the **draw()** method similar to:

```
shapes[i].draw( g, xOffset, yOffset );
```

valueChanged()

valueChanged() is going to be called any time a change occurs on the list. Many list selection events can be generated from a single user action such as a mouse click. The **getValueIsAdjusting()** method returns true if the user is still manipulating the selection. This particular program is interested only in the final result of the user's action, so the **valueChanged()** method updates the shapes[] array only if **e.getValueIsAdjusting()** returns false and **list.isEmpty()** is false.

Within a **try** block, if the number of shapes in the array of shapes is less than the length of the array, do the following: Using

```
shapeString = (String) list.getSelectedValue();
```

get the String value of the selected item in the list. Then use

```
ShapeP4 shape = (ShapeP4) Class.forName( shapeString + "P4" ).newInstance();
shapes[numOfShapes++] = shape;
```

to dynamically create an object from the selected JList item and assign this shape to the next shapes array element. **newInstance()** will invoke the no-arg constructor for the given class. More details of this in class. Clear the selected item with **list.clearSelection()** so it can capture the event of it being selected again as a changed value. Then **repaint()** to clear the applet and redisplay the coordinates and shapes.

If the number of shapes in the array of shapes is not less than the length of the array, still clear the selection (**list.clearSelection()**) and display the message **"Shapes container is full -- cannot create new Shape!"** to the status bar of the applet (using **showStatus()**).

Class.forName() and **newInstance()** can throw checked exceptions. We will discuss Exceptions in more detail soon. For this program, you can use a generic catch block to catch all Exceptions and display to the terminal the exception. The try-catch block described above will look something like this:

```
try
{
    // If the number of shapes in the array of shapes is less than the
    //   the array's length.

    // Get the the value (String) of the selected list item.

    // Dynamically instantiate an instance of an object of this type with "P4"
    //   appended for our shapes types in this PA and assign this reference to
    //   the array of shapes incrementing the number of shapes now in the array.

    // Clear the selection so we can select that same list item again.

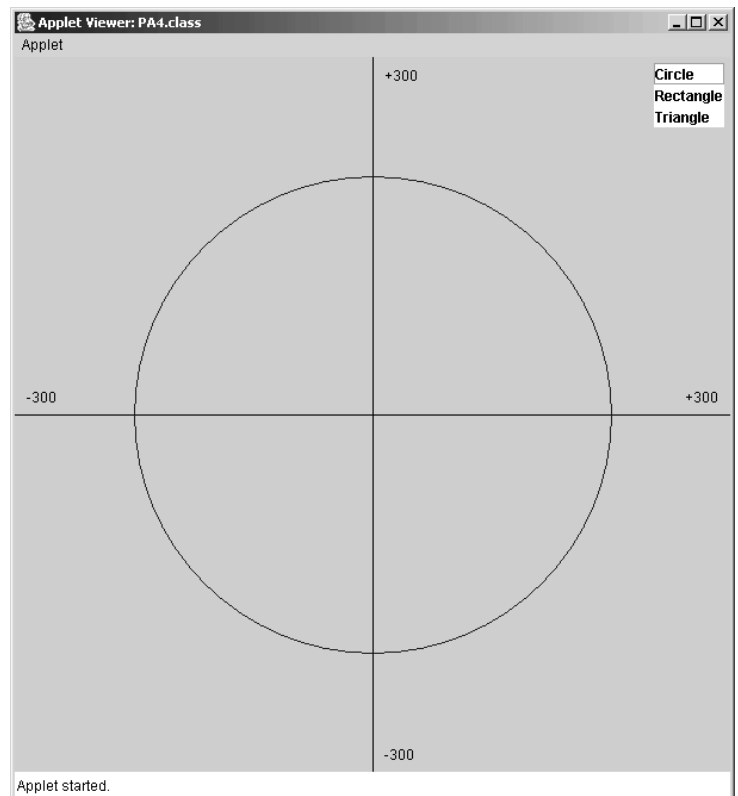
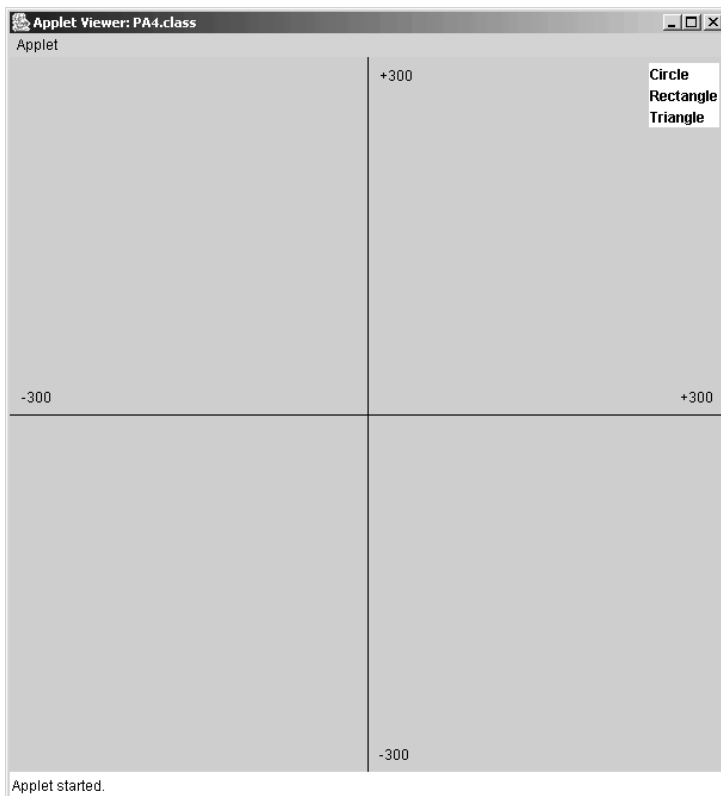
    // Call repaint() to update the graphics display.

    // Else if there isn't any room in the array of shapes for this selected shape

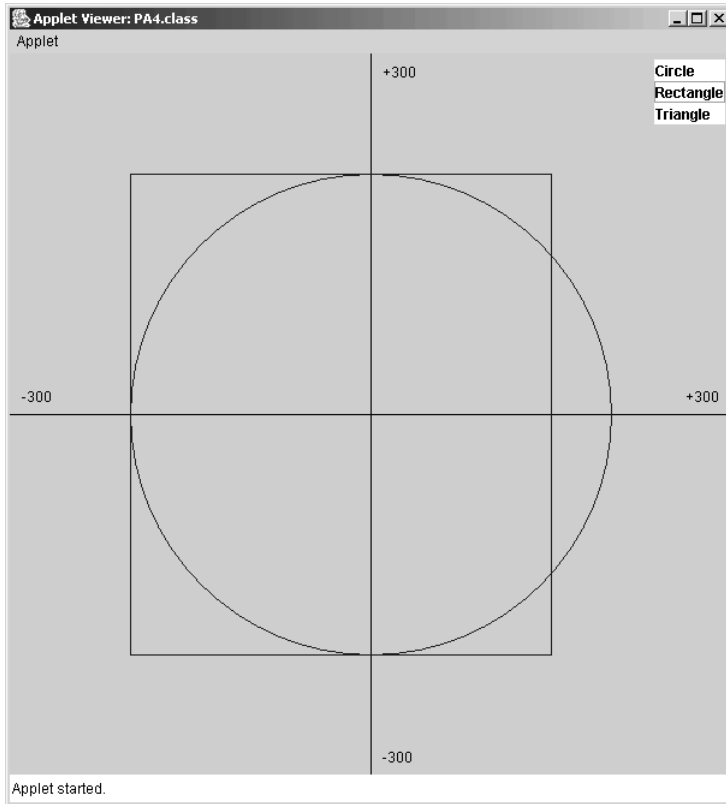
    // Clear the selection

    // Display the shapes container full message to the status bar of the applet
    //   and the terminal.
}
catch( Exception e )
{
    System.out.println( e );
}
```

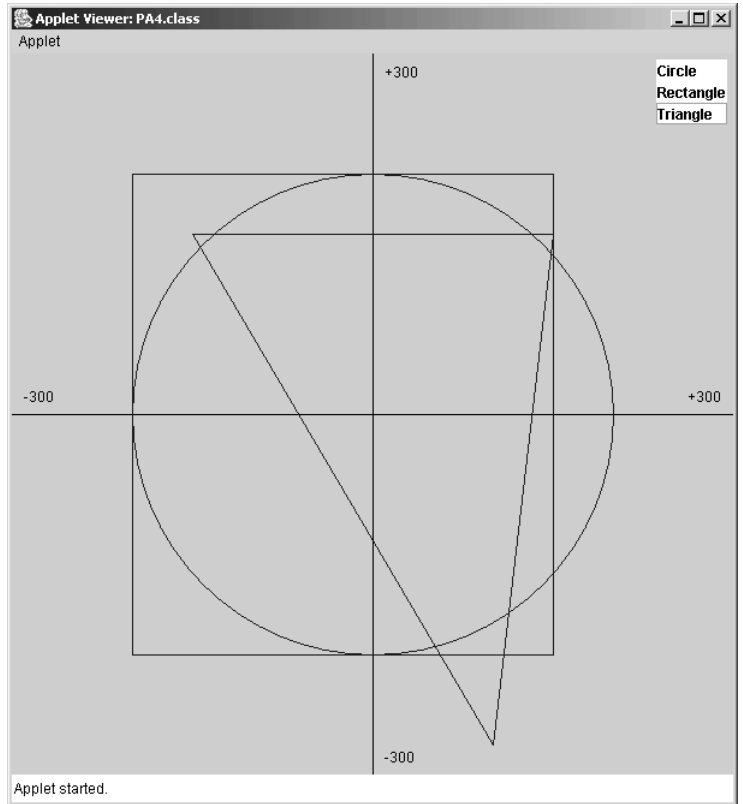
Examples: **appletviewer PA4.html**



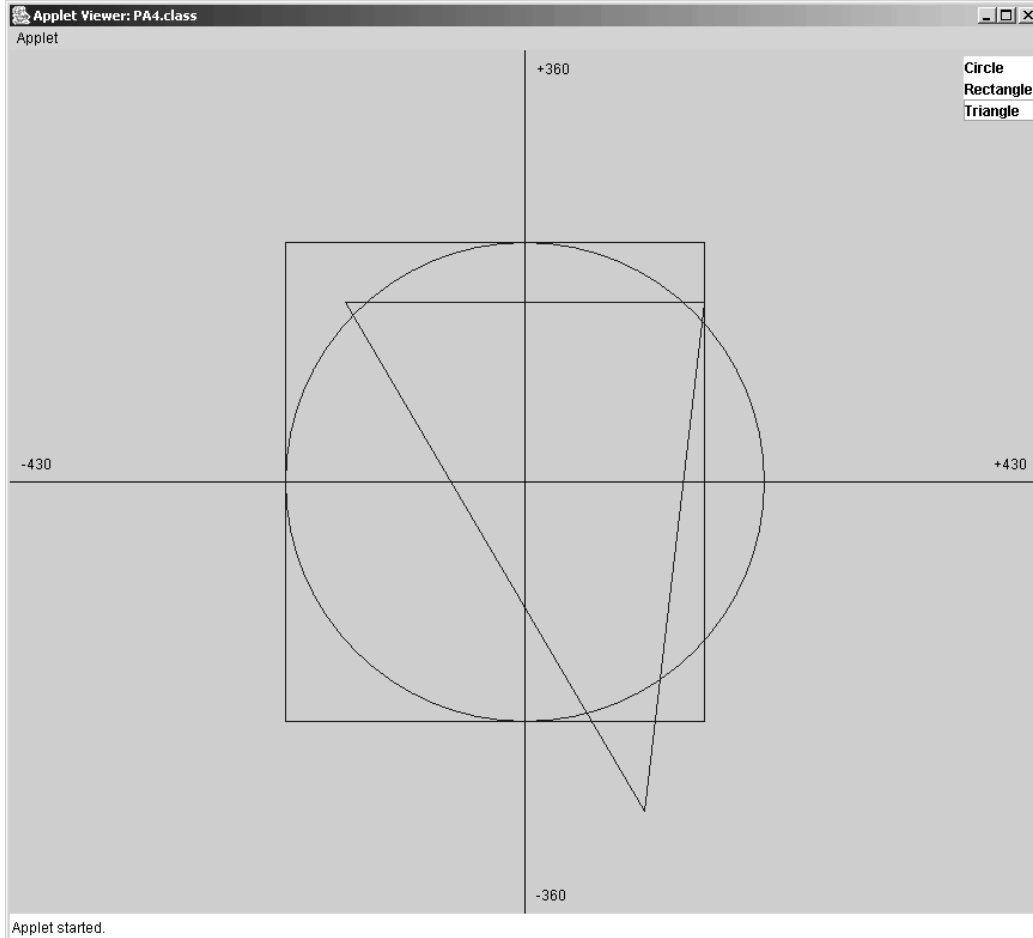
Circle at (0,0) radius = 200



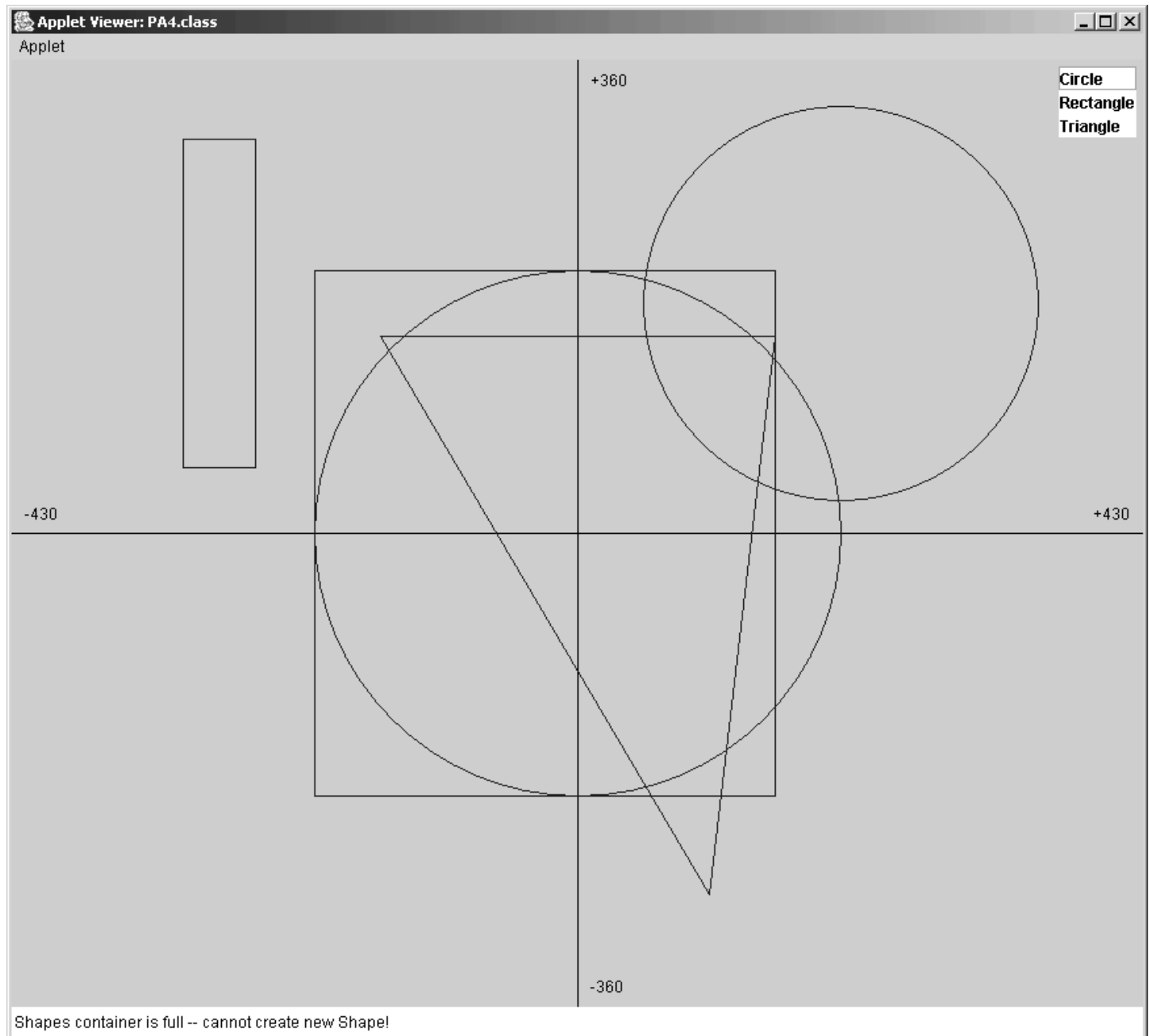
Rectangle at (-200,200)
width = 350; height = 400



Triangle at (-150,150);
(150,150); (100,-275)



Change the dimension of
the applet window



Add another Circle at (200,175) with radius = 150
Add another Rectangle at (-300,300) length = 55; width = 250
Then try to add another Circle. Note message in status bar.

Applet menu
Reload to start over.

Be sure to test 1) having no input in an input dialog box and 2) hitting the Cancel button to indicate zero default values from that input and the rest of the inputs for that shape.