

1. Here is a simple event driven program. What is missing from it? That is, what needs to be added to this program so that it compiles, runs, and noticeably responds to events? (Note: Do not delete any code from the program. Only add code to it.)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class WhatIsMissing
{
    public WhatIsMissing() {
        final JFrame jf = new JFrame("WhatIsMissing");
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setLayout(new FlowLayout());
        final JButton jb = new JButton("Hello");
        jb.addActionListener(this);
        jf.add(jb);
        jf.setLocationRelativeTo(null);
        jf.setSize(400, 400);
    }

    @Override public void mouseClicked(MouseEvent e) {
        System.out.println(e);
    }

    @Override public void componentMoved(ComponentEvent e) {
        System.out.println(e);
    }

    @Override public void componentResized(ComponentEvent e) {
        System.out.println(e);
    }
}
```

2. Describe the GUI created by the following program (for example, draw a sketch of the GUI) and then describe what happens when you click on each component in the GUI.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Review_Problem_2 {
    public Review_Problem_2() {
        final JFrame jf = new JFrame("Problem 2");
        jf.setLayout(new FlowLayout());
        final JButton jb1 = new JButton("Apple");
        final JCheckBox cb1 = new JCheckBox("Pear");
        final JButton jb2 = new JButton("Grape");
        jf.add(jb1); jf.add(cb1); jf.add(jb2);
        ActionListener actionHandler1 = new ActionHandler1();
        ActionListener actionHandler2 = new ActionHandler2();
        jb1.addActionListener(actionHandler1);
        jb1.addActionListener(actionHandler2);
        cb1.addActionListener(actionHandler1);
        jb2.addActionListener(actionHandler2);
        jf.setSize(400, 400);
        jf.setVisible(true);
    }
    class ActionHandler1 implements ActionListener {
        @Override public void actionPerformed(ActionEvent e) {
            System.out.println("Elephant");
        }
    }
    class ActionHandler2 implements ActionListener {
        @Override public void actionPerformed(ActionEvent e) {
            System.out.println("Bear");
        }
    }
    public static void main(String[] args) {
        new Review_Problem_2();
    }
}
```

3. Explain how this program works. You should be able to explain the purpose of every line of code.

```
/* 1*/ import java.awt.*;
/* 2*/ import java.awt.event.*;
/* 3*/ import javax.swing.*;
/* 4*/ public class Review_Problem_3 implements ActionListener {
/* 5*/     private boolean checked = false;
/* 6*/     public Review_Problem_3() {
/* 7*/         final JFrame jf = new JFrame("Problem 3");
/* 8*/         jf.setLayout(new FlowLayout());
/* 9*/         final JButton jb1 = new JButton("Hello");
/*10*/        final JButton jb2 = new JButton("GoodBy");
/*11*/        final JCheckBox cb1 = new JCheckBox("Confuse Them");
/*12*/        jf.add(jb1); jf.add(jb2); jf.add(cb1);
/*13*/        cb1.addActionListener(this);
/*14*/        jb1.addActionListener(new ActionListener());
/*15*/        jb2.addActionListener(new ActionListener(){
/*16*/            @Override public void actionPerformed(ActionEvent e) {
/*17*/                System.out.println(!checked ? "Goodbye" : "Hello?");
/*18*/            });
/*19*/        jf.setSize(400, 400);
/*20*/        jf.setVisible(true);
/*21*/    }
/*22*/    @Override public void actionPerformed(ActionEvent e) {
/*23*/        checked = ! checked;
/*24*/    }
/*25*/    class ActionHandler implements ActionListener {
/*26*/        @Override public void actionPerformed(ActionEvent e) {
/*27*/            System.out.println(!checked ? "Hello" : "Goodbye?");
/*28*/        }
/*29*/    }
/*30*/    public static void main(String[] args) {
/*31*/        new Review_Problem_3();
/*32*/    }
/*33*/ }
```

4. Draw a sketch of the GUI created by this program. Outline the regions in the GUI and just indicate with a word the background color.

```
import java.awt.*;
import javax.swing.*;
public class Review_Problem_4 {
    public Review_Problem_4() {
        final JFrame jf = new JFrame("Problem 4 GUI");
        final JPanel jp1 = new JPanel();
        final JPanel jp2 = new JPanel();
        final JPanel jp3 = new JPanel();
        final JPanel jp4 = new JPanel();
        jf.getContentPane().setBackground(Color.magenta);
        jp1.setBackground(Color.cyan);    jp2.setBackground(Color.green);
        jp3.setBackground(Color.yellow); jp4.setBackground(Color.pink);
        final JButton jb1 = new JButton("One");
        final JButton jb2 = new JButton("Two");
        final JButton jb3 = new JButton("Five");
        final JLabel jl1 = new JLabel("Plum");
        final JLabel jl2 = new JLabel("Lemons");
        final JCheckBox cb1 = new JCheckBox("Six");
        jp1.add(jb1);  jp1.add(jl1);  jp1.add(jb2);  jp1.add(jl2);
        jp2.add(jb3);  jp2.add(cb1);
        jp3.add(new JLabel("Three"));
        jp4.add(new JButton("Four"));
        jf.add(jp1, BorderLayout.SOUTH);
        jf.add(jp2, BorderLayout.NORTH);
        jf.add(jp3, BorderLayout.EAST);
        jf.add(jp4, BorderLayout.WEST);
        jf.add(new JLabel("An impressive GUI."), BorderLayout.CENTER);
        jf.setSize(400, 400);
        jf.setVisible(true);
    }
    public static void main(String[] args) {
        new Review_Problem_4();
    }
}
```

5. The following block of code creates a `Framebuffer` object with two `Viewport` objects, places a `Model` of a disk (with radius 1) at the center of the image-plane, then it sets up a view rectangle for each `Viewport` and renders the view rectangle into its `Viewport`. Draw a sketch of the contents of the `Framebuffer` after this code executes. In your sketch, label each of the two `Viewports`, `vp1` and `vp2`. See the last page of this document about the view rectangle and viewport APIs.

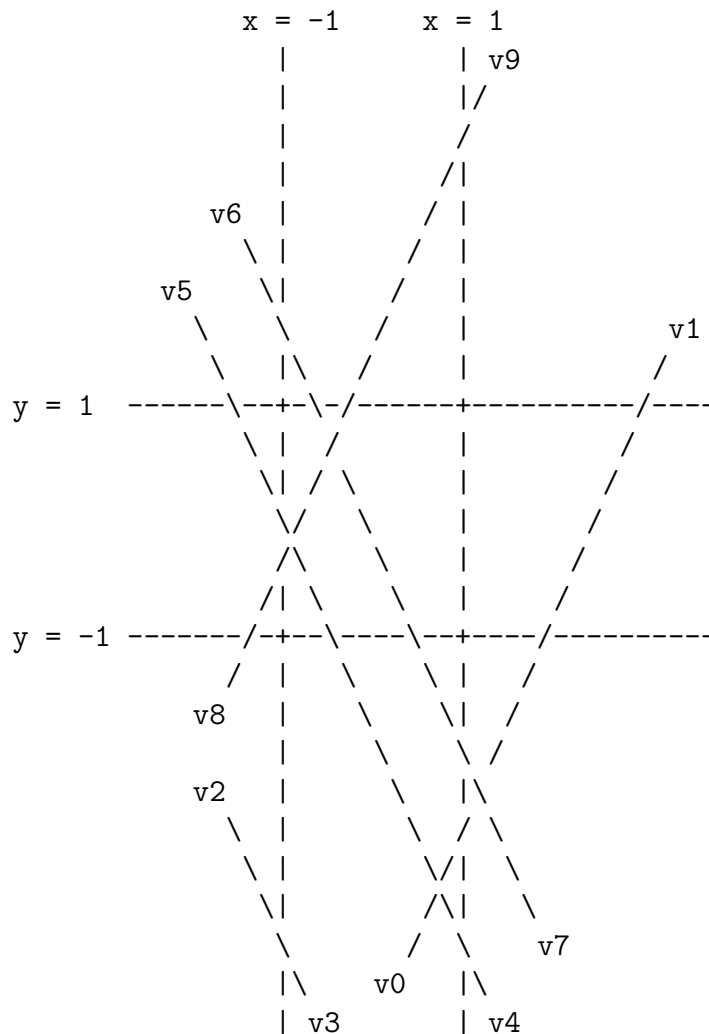
```
Framebuffer fb = new FrameBuffer(400, 200, Color.gray);
Framebuffer.Viewport vp1 = fb.new Viewport(0, 0, 200, 100, Color.white);
Framebuffer.Viewport vp2 = fb.new Viewport(200, 100, 200, 100, Color.white);
vp1.clearVP();
vp2.clearVP();
Scene scene = new Scene("Review_Problem_5");
Model disk = new Disk(1.0, 1, 64);
ModelShading.setColor(disk, Color.black);
scene.addPosition( new Position(disk) );
scene = scene.changeCamera(Camera.projOrtho(-1.0, 1.0, 0.0, 1.0);
Pipeline.render(scene, vp1);
scene = scene.changeCamera(Camera.projOrtho(-1.0, 1.0, -1.0, 0.0);
Pipeline.render(scene, vp2);
```

6. The following block of code creates a `Framebuffer`, places a `Model` of a disk (with radius 1) at the center of the image-plane, then it sets up the view rectangle and the `Viewport`, and finally it renders the view rectangle into the viewport. In each part below, give a sketch of the view rectangle and the viewport. See the last page of this document about the view rectangle and viewport APIs.

```
final FrameBuffer fb = new FrameBuffer(500, 500, Color.gray);
Scene scene = new Scene("Review_Problem_6");
final Model disk = new Disk(1.0, 1, 64);
ModelShading.setColor(disk, Color.black);
scene.addPosition( new Position(disk) );
/**** Declarations of: left, right, bottom, top, x, y, w, h ****/
// Set up the view rectangle (in the image-plane).
scene = scene.changeCamera(Camera.projOrtho(left, right, bottom, top));
// Set up the Viewport (in the FrameBuffer).
Framebuffer.Viewport vp = fb.new Viewport(x, y, w, h, Color.white);
vp.clearVP();
Pipeline.render(scene, vp);
```

- (a) `// Parameterize the view rectangle.
double left = -0.5, right = 1.0;
double bottom = -0.5, top = 1.0;
// Parameterize the viewport.
int x = 50, y = 50;
int w = 400, h = 400;`
- (b) `// Parameterize the view rectangle.
double left = -1.0, right = 1.0;
double bottom = 0.0, top = 1.0;
// Parameterize the viewport.
int x = 0, y = 250;
int w = 500, h = 250;`
- (c) `// Parameterize the view rectangle.
double left = 0.0, right = 1.0;
double bottom = 0.0, top = 1.0;
// Parameterize the viewport.
int x = 250, y = 0;
int w = 250, h = 500;`
- (d) `// Parameterize the view rectangle.
double left = -1.0, right = 3.0;
double bottom = -1.0, top = 3.0;
// Parameterize the viewport.
int x = 0, y = 0;
int w = 400, h = 400;`

7. If the recursive clipping algorithm clips line segments using this order, $y = 1$, $y = -1$, $x = 1$, $x = -1$, for the clipping boundaries, then for each line segment shown below, determine how many times the line segment will be clipped before it is either accepted or rejected.
- Line segment from v0 to v1.
 - Line segment from v2 to v3.
 - Line segment from v4 to v5.
 - Line segment from v6 to v7.
 - Line segment from v8 to v9.



8. Redo the previous problem but with the following order for the clipping boundaries: $y = 1$, $x = 1$, $y = -1$, $x = -1$.

9. (a) Compute the 16 numbers that make up the 4×4 homogeneous matrix in the `Position` object after executing the following line of code.

```
position.transform( Matrix.translate(3.0, 0.0, 0.0)
                    .times( Matrix.scale(3.0, 1.0, 1.0) ) );
```

- (b) Compute the 16 numbers that make up the 4×4 homogeneous matrix in the `Position` object after executing the following line of code.

```
position.transform( Matrix.scale(3.0, 1.0, 1.0)
                    .times( Matrix.translate(3.0, 0.0, 0.0) ) );
```

10. Assume that `p` is a reference to a `Position` object and that `m1`, `m2`, `m3` are references to `Matrix` objects. In what ways are the following two lines of code similar, and in what ways do they differ?

```
p.transform( m1.times(m2).times(m3) );
p.transform( m1.times(m2.times(m3)) );
```

11. Define a `Matrix` object `m` and a `Position` object `p`. What does this line of code do?

```
p.transform(p.getMatrix().times(m));
```

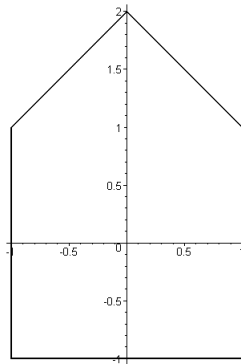
How does it differ from this line of code?

```
p.transform(m.times(p.getMatrix()));
```

12. Assume that `p` is a reference to a `Position` object. What is wrong with the following line of code? Explain in detail what the line of code does. How might you fix it?

```
p.getMatrix().times(Matrix.translate(3,0,0).times(Matrix.scale(2,2,2)));
```

13. (a) Suppose we have defined a subclass `DrawModel` of `Model` that represents the following image in the xy -plane. Draw a diagram of the scene graph that the following code constructs. Draw a sketch of the scene that the code would produce in the camera's image-plane when the scene is rendered. For each of the three model images, give the coordinates of the model's origin point.



```
Model drawModel = new DrawModel();
Position p1 = new Position(drawModel);
Position p2 = new Position(drawModel);
Position p3 = new Position(drawModel);
p2.transform( Matrix.translate(1.0, 3.0, 0.0).times(
    Matrix.rotateZ(45.0)) );
p3.transform( Matrix.translate(1.0, 0.0, 0.0).times(
    Matrix.translate(1.0, 0.0, 0.0)) );
p1.transform( Matrix.translate(-2.0, -2.0, 0.0).times(
    Matrix.rotateZ(-90.0)) );
Scene scene = new Scene();
scene.addPosition(p1, p2, p3);
```

- (b) With the addition of the following code, what is drawn in the framebuffer?

```
ModelShading.setColor(drawModel, Color.black);
scene = scene.changeCamera(Camera.projOrtho(-2.0, 2.0, -2.0, 3.0));
Framebuffer fb = new FrameBuffer(500, 600, Color.gray);
Framebuffer.Viewport vp = fb.new Viewport(50,50,400,500,Color.white);
vp.clearVP();
Pipeline.render(scene, vp);
```

14. Suppose you have available to you a subclass `DrawUnitCircle` of `Model` that draws in the xy -plane a circle of radius one centered at the origin. Suppose that `theta1` and `theta2` are each defined as a (different) non-zero double. What would the following code draw in the camera's image plane? What does its scene graph look like?

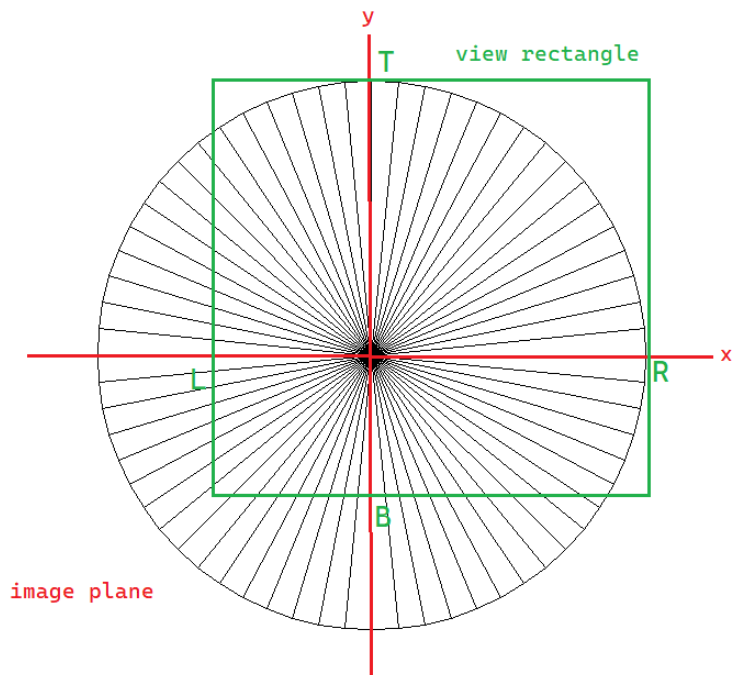
```
Model circle = new DrawUnitCircle();
Position p1 = new Position(circle);
Position p2 = new Position(circle);
Position p3 = new Position(circle);
p1.transform( Matrix.scale(2.0, 2.0, 2.0) );
p2.transform( Matrix.rotateZ(theta1).times(
    Matrix.translate(3.0, 0.0, 0.0)) );
p3.transform( Matrix.rotateZ(theta2).times(
    Matrix.translate(1.5, 0.0, 0.0)).times(
    Matrix.scale(0.5, 0.5, 1.0)) );
Scene scene = new Scene();
scene.addPosition(p1, p2, p3);
```

15. Define `Matrix` objects `m1`, `m2`, `m3`, `m4` and a `Position` object `p`. Write a formula for the final `Matrix` in `p`.

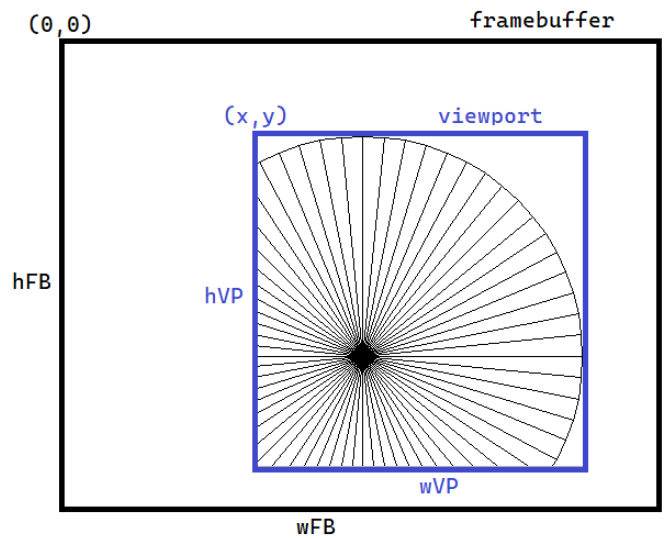
```
p.transform(m1);
p.transform(m2);
p.transform( m3.times(p.getMatrix()) );
p.transform( p.getMatrix().times(p.getMatrix()) );
p.transform( m4.times( p.getMatrix() ) );
```

16. Define `Matrix` objects `m1`, `m2`, `m3` and a `Position` object `p`. Simplify this code as much as possible.

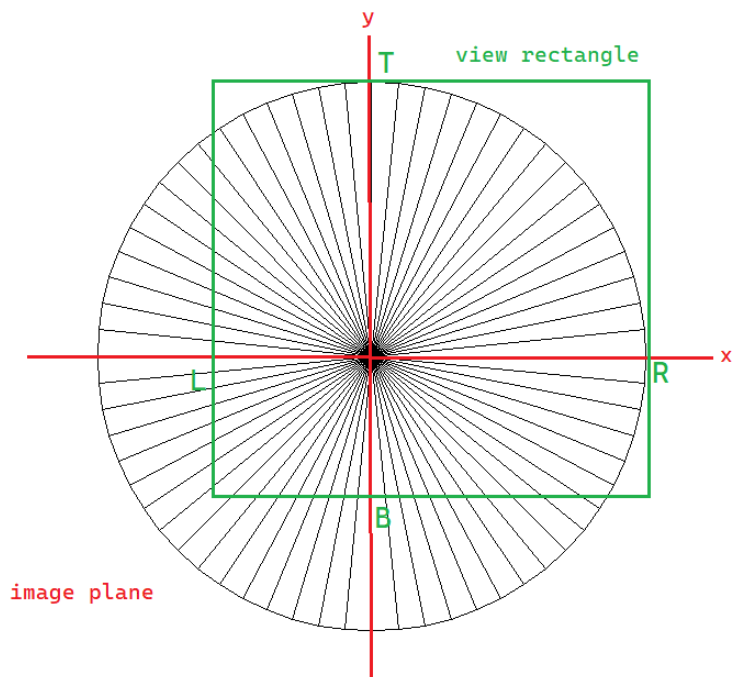
```
p.transform(Matrix.identity());
p.transform(p.getMatrix().times(m1));
p.transform(m2.times(p.getMatrix()));
p.transform(p.getMatrix().times(m3));
```



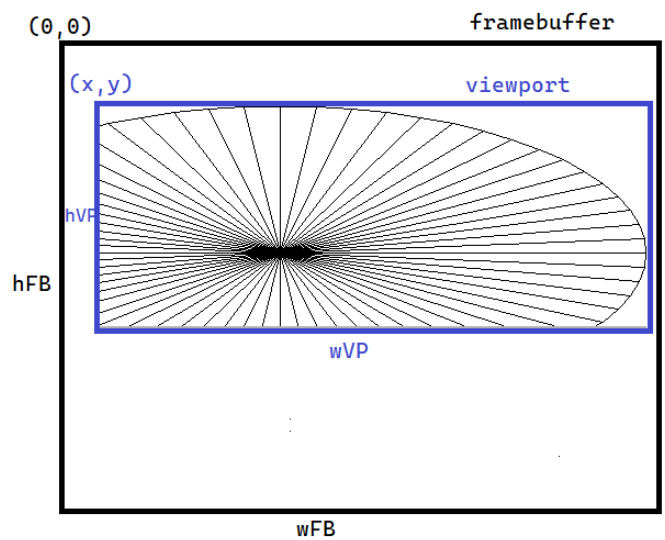
```
Camera.projOrtho(left, right, bottom, top)
or
Camera.projPerspective(left, right, bottom, top)
```



```
Framebuffer fb = new FrameBuffer(wFB, hFB)
Framebuffer.Viewport vp = fb.new Viewport(x, y, wVP, hVP)
```



```
Camera.projOrtho(left, right, bottom, top)
or
Camera.projPerspective(left, right, bottom, top)
```



```
Framebuffer fb = new FrameBuffer(wFB, hFB)
Framebuffer.Viewport vp = fb.new Viewport(x, y, wVP, hVP)
```