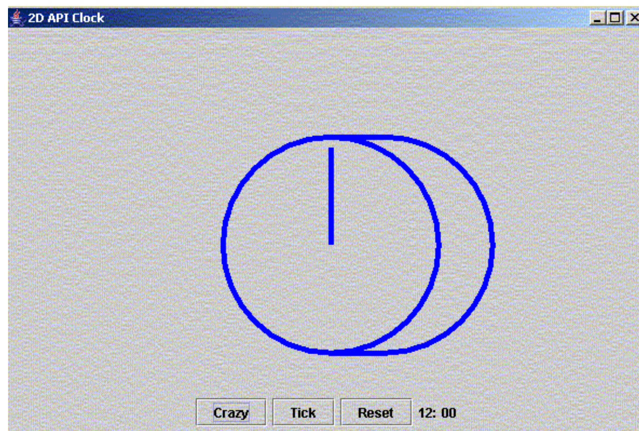
1.00 Lecture 21

2D API
2D Transformations

Reading for next time: Numerical Recipes p. 32-36
Just read the text; don't worry about reading the C code

Clock, revisited

- **We'll use the model-view-controller version of the clock and draw with the 2D API (application programming interface):**



•Download ClockController, ClockModel, ClockView

Clock View with 2D API

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

public class ClockView extends JPanel {
   private ClockModel model;
   private static final double CD= 200;           // Clock diameter
   private static final double X= 100;            // Dist from upper lh corner
   private static final double Y= 50;             // Dist from upper lh corner
   private static final double XC= X + CD/2;        // Clock center x
   private static final double YC= Y + CD/2;        // Clock center y
   private static final double HR= 0.3F*CD;         // Size of hour hand
   private static final double MI= 0.45F*CD;        // Size of minute hand

   public ClockView(ClockModel cm) {
        model = cm;
   }
// Continued
```

Clock View with 2D API, p.2

```
   public void paintComponent(Graphics g) {
     super.paintComponent(g);
     Graphics2D g2 = (Graphics2D) g;              // Cast g to g2 context
     double minutes= model.getMinutes();
     double hourAngle = 2*Math.PI * (minutes - 3 * 60) / (12 * 60);
     double minuteAngle = 2*Math.PI * (minutes - 15) / 60;

     Ellipse2D.Double e = new Ellipse2D.Double(X, Y, CD, CD);
     Line2D hr= new Line2D.Double(XC, YC, XC+(HR*Math.cos(hourAngle)),
        YC+ (HR * Math.sin(hourAngle)) );
     Line2D mi= new Line2D.Double(XC, YC, XC+
        (MI* Math.cos(minuteAngle)), YC+ (MI * Math.sin(minuteAngle)) );

     g2.setPaint(Color.BLUE);
     BasicStroke bs= new BasicStroke(5.0F,
                 BasicStroke.CAP_BUTT, BasicStroke.JOIN_BEVEL);
     g2.setStroke(bs);
     g2.draw(e);
     g2.draw(hr);
     g2.draw(mi);
   }
}
```

Exercise

- **Add the two lines and arc in `paintComponent()` to create the picture shown in the first slide**
  - `Line2D.Double(double x0, double y0, double x1, double y1)`
    - **Draws a line from (x0, y0) to (x1, y1)**
    - **Make your line length = clock diameter / 4**
  - `Arc2D.Double(double x, double y, double w, double h, double start, double extent, int type)`
    - **Draws an arc with upper left hand corner (x,y), width w and height h. These first 4 arguments are the same as the ellipse or circle arguments**
    - **Start is the start angle, in degrees**
    - **Extent is the angle of the arc, in degrees**
    - `Type is a style; use Arc2D.OPEN`
- **Optional: Draw the hour and minute hands in different colors and different line widths.**

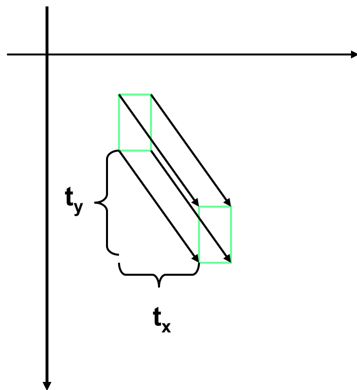
Affine Transformations

- **The 2D API provides strong support for *affine transformations*.**
  - **Affine means linear (of the form y= ax +b)**
- **An affine transformation maps 2D coordinates so that the straightness and parallelism of lines are preserved.**
- **All affine 2D transformations can be represented by a 3x3 floating point matrix.**
- **There are a number of “primitive” affine transformations that can be combined: scaling, rotation, and translation.**

Transformations in the 2D API

- **Transformations are represented by instances of the `AffineTransform` class in the `java.awt.geom` package.**
- **You can create a new AffineTransform object with its no argument constructor.**
  - `AffineTransform at = new AffineTransform();`
- **You can invoke the following methods (and others) on an AffineTransform object:**
  - `at.scale(double sx, double sy)`
  - `at.translate(double tx, double ty)`
  - `at.rotate(double theta)`
  - `at.rotate(double theta, double x, double y)`
- **These methods build a *stack* of basic transforms: last in, first applied**

Translation



$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ 1 \end{bmatrix}$$

Translation Example

To display a `RectanglePanel` in a `JFrame`:

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

public class RectangleTest {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Rectangle transform");

   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(500,500);
        Container contentPane= frame.getContentPane();
        RectanglePanel panel = new RectanglePanel();
        contentPane.add(panel);
        frame.setVisible(true);
    }
}
```

Translation Example

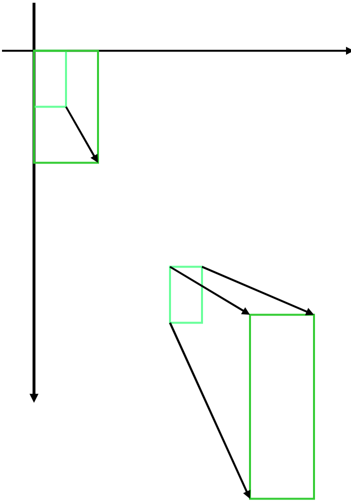
```
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;     // For 2D classes

public class RectanglePanel extends JPanel {
    public void paintComponent(Graphics g) {
       super.paintComponent(g);
       Graphics2D g2= (Graphics2D) g;
       Rectangle2D rect= new Rectangle2D.Double(0,0,50,100);
       g2.setPaint(Color.BLUE);

       AffineTransform baseXf = new AffineTransform();
       // Shift to the right 50 pixels, down 50 pixels
       baseXf.translate(50,50);
       g2.transform(baseXf);

       g2.draw(rect);
    }
}       // Download and run RectangleTest, RectanglePanel
```

Scaling



$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} s_x * x \\ s_y * y \\ 1 \end{bmatrix}$$

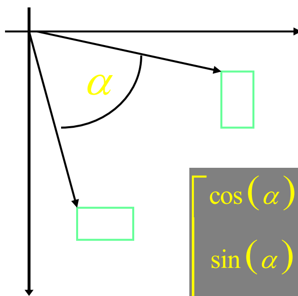
Scaling Exercise

- Modify RectangleTest, RectanglePanel:
- First, write code to scale `rect` at the origin using `RectanglePanel` as a basis.
  - Follow the same steps you saw in the translation exercise.
  - Instead of `translate`, invoke the `scale` method.
  - `scale` takes two doubles as arguments: the first for scaling x, the second for y.
- Next, modify `rect` so that it is not at the origin. How does `scale` act on shapes that aren't at the origin?
  - Modify the first two arguments, which are the (x,y) of the upper left-hand corner of the rectangle

Scaling Notes

- Basic scaling operations take place with respect to the origin. If the shape is at the origin, it grows. If it is anywhere else, it grows and moves.
- s_x, scaling along the x dimension, does not have to equal s_y, scaling along the y.
- For instance, to flip a figure vertically about the x-axis, scale by $s_x=1$, $s_y=-1$.

Rotation



$$\begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x\cos(\alpha) - y\sin(\alpha) \\ x\sin(\alpha) + y\cos(\alpha) \\ 1 \end{bmatrix}$$

Rotation Exercise

- Modify RectangleTest, RectanglePanel again:
- Write code to rotate `rect` using `RectanglePanel` as a basis.
- Follow the same steps as you did in the scaling exercise.
  - Invoke `baseXf.rotate()` with a single argument: the angle, in radians, to rotate the rectangle.
  - This method will appear to both rotate and move the rectangle with respect to the origin.
  - You might find `Math.PI` or `Math.toRadians(double degrees)` useful.
- To avoid rotating `rect` completely out of view, rotate by only a small amount (10 or 20 degrees).
- How does rotating `rect` change when `rect` is at the origin? When it isn't?

Composing Transformations

- Suppose we want to scale point (x, y) by 2 and then rotate by 90 degrees.

$$\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \left(\begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \right)$$

rotate scale

Composing Transformations, 2

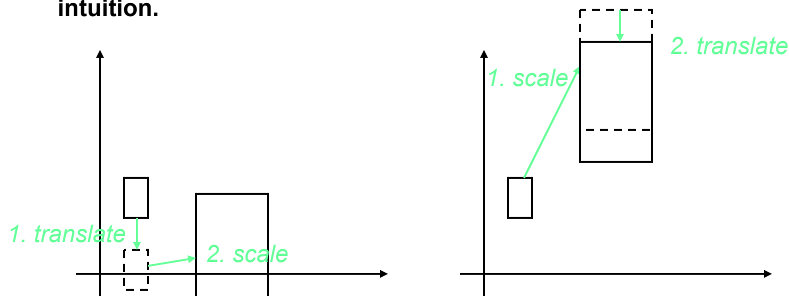
Because matrix multiplication is associative, we can rewrite this as

$$\left(\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}\right)\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -2 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Composing Transformations, 3

- Because matrix multiplication does not regularly commute, the order of transformations matters. This squares with our geometric intuition.

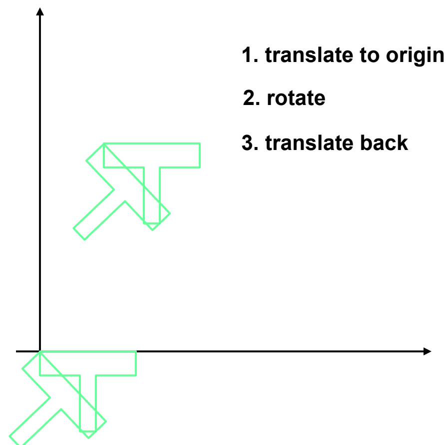


- If we invert the matrix, we reverse the transformation.

Transformations and the Origin

- When we transform a shape, we transform each of the defining points of the shape, and then redraw it.
- If we scale or rotate a shape that is not anchored at the origin, it will translate as well.
- If we just want to scale or rotate, then we should translate back to the origin, scale or rotate, and then translate back.

Transformations and the Origin, 2



Transformations in the 2D API

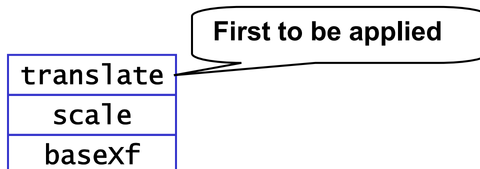
- Transformations are represented by instances of the `AffineTransform` class in the `java.awt.geom` package.
- Build a compound transform by
  1. Creating a new instance of `AffineTransform`
  2. Calling methods to build a *stack* of basic transforms: last in, first applied:
     - `translate(double tx, double ty)`
     - `scale(double sx, double sy)`
     - `rotate(double theta)`
     - `rotate(double theta, double x, double y)`
       rotates about (x,y)

Transformation Example

```
baseXf = new AffineTransform();
baseXf.scale( scale, -scale );
baseXf.translate( -x, -y );
```

If we now apply baseXF it will translate first, then scale.

Remember in Java that transforms are built up like a stack, last in, first applied.



(TransformTest and TransformPanel show an example)

Converting Swing GUIs to Applets

- Create an HTML page with appropriate code to load the applet (covered in lecture 37)
- Declare an applet class name, extends JApplet
- Eliminate your main() method:
- Remove calls to
  - setSize(); done in HTML file
  - setDefaultCloseOperation(); applet ends when browser closes
  - setTitle(); no titles allowed
  - setVisible(); done by browser
- Don't construct a JFrame (eliminate its constructor)
  - Applets use the browser window instead
- Move any remaining code from main() or the JFrame constructor to the init() method of the applet
  - Often no statements will remain in the old main()
  - Often your JFrame constructor can move 'as is'

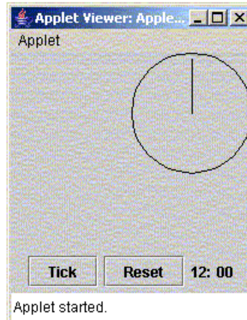
Clock Applet

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

public class AppletTest extends JApplet{
   public void init() {
        Container contentPane= getContentPane();
        ClockPanel clock= new ClockPanel();
        contentPane.add(clock);
   }
}

// ClockPanel class from events lecture works "as is".
// No need for ClockFrame class:
//    Eliminated statements in list from main(), constructor
//    Moved remaining statements into init()
```

Clock Applet Exercise



Download ClockPanel
Write AppletTest (compare to old main, constructor)
Run -> Run As Applet in Eclipse

Test Your Swing Knowledge

- Can a class be its own event handler?
  - Yes
  - No
- What interface must a class implement to listen to events?
  - ____________________
- If so, what method(s) does this interface have?
  - ____________________
- How would you draw a JButton with rounded edges?
  - ____________________

Test Your Swing Knowledge, p. 2

- Why don't we draw JComponents directly on a JFrame?
  - ______________________________
- On what object do we draw JComponents?
  - ____________________
- Should you put System.exit(0) at the end of main() methods that create Swing objects?
  - Yes No Why? _______________
- What method does repaint() call?
  - ____________________
- Why not call that method directly?
  - ____________________

Test Your Swing Knowledge, p. 3

- An anonymous inner class constructor can have arguments
  - True False
- You can refer to anonymous inner class objects by using the 'inner' keyword
  - True False
- An anonymous inner class can only have the methods that implement its Listener interface
  - True False
- An anonymous inner class has the name 'this'
  - True False
- An anonymous inner class has access to its enclosing class' data and methods
  - True False