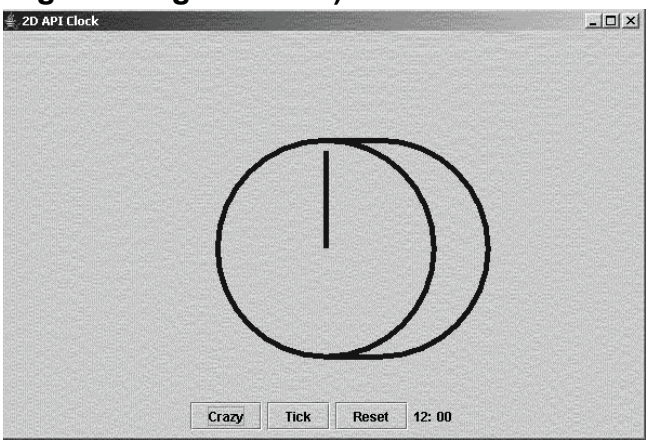
1.00 Lecture 21

Drawing complex objects:
2D API
2D Transformations

Reading for next time: None

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= 0.3*CD;     // Size of hour hand
   private static final double MI= 0.45*CD;    // Size of minute hand

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

Clock View with 2D API, p.2

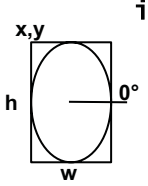
```
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;         // Cast g to g2 context
    double minutes= model.getMinutes();
    double hourAngle = 2*Math.PI * (minutes - 3 * 60) / (12 * 60);
    double minuteAngle = 2*Math.PI * (minutes - 15)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

    Ellipse2D 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 1

- **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)`
    - **Creates 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)`



    - **Creates an arc with upper left hand corner (x,y), width w and height h. These first 4 arguments are the same as an ellipse, and allow space for a 360 degree arc**
    - **Start is the start angle, in degrees. (Go counterclockwise)**
    - **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.**

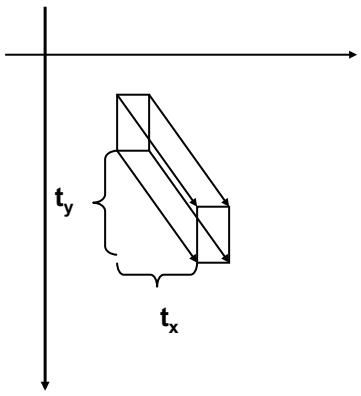
Affine Transformations

- **The 2D API provides *affine transformations*.**
  - **Affine means linear (of the form y= ax +b)**
  - **These transform from one coordinate system to another while retaining straightness and parallelism of lines**
- **All affine 2D transformations can be represented by a 3x3 matrix: scaling, rotation, translation, shearing, …**
  - **These "primitive" affine transformations can be also combined**
- **We usually create a small number of graphic objects (ellipses, rectangles, etc.) and keep transforming them to create complex drawings or animations**
  - **We actually transform the coordinate system, not the objects**
  - **Thus, all objects on the JPanel appear to be transformed each time a transform is applied, but we only draw the ones we want**
- **A caution: If your drawing is off the JPanel, Java will not warn you. It's easy to transform objects off the JPanel.**

Transformations in the 2D API

- Transformations are represented by instances of the `AffineTransform` class in `java.awt.geom`
- Create a new AffineTransform object with its no-argument (default) constructor
  - `AffineTransform at = new AffineTransform();`
- Invoke the following methods (and others):
  - `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 RectangleFrame extends JFrame {
   public RectangleFrame() {
        Container contentPane= getContentPane();
        RectanglePanel panel = new RectanglePanel();
        contentPane.add(panel, BorderLayout.CENTER);
   }
     public static void main(String args[]) {
        RectangleFrame frame = new RectangleFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(500,500);
        frame.setVisible(true);
}  }
```

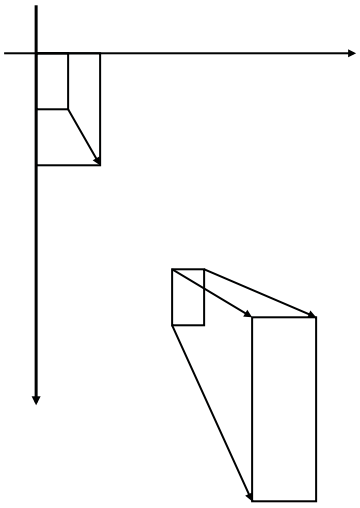
Translation Example

```
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.RED);
       g2.draw(rect);                // Original position

       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 RectangleFrame, RectanglePanel
```

Scaling



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

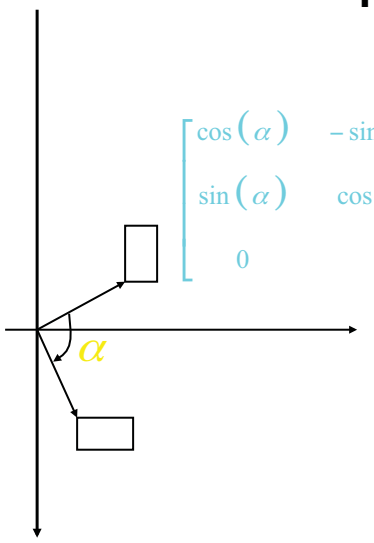
Scaling Notes

- 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 $s_x=1$, $s_y=-1$.
- There is also a shear() transform—see javadoc.

Exercise 2: Scaling

- **Modify RectangleFrame, 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**

Rotation



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

Exercise 3: Rotation

- Modify RectangleFrame, 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.
  - 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?
  - Use the 3 argument version of rotate() to experiment: `at.rotate(double theta, double x, double y)`

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

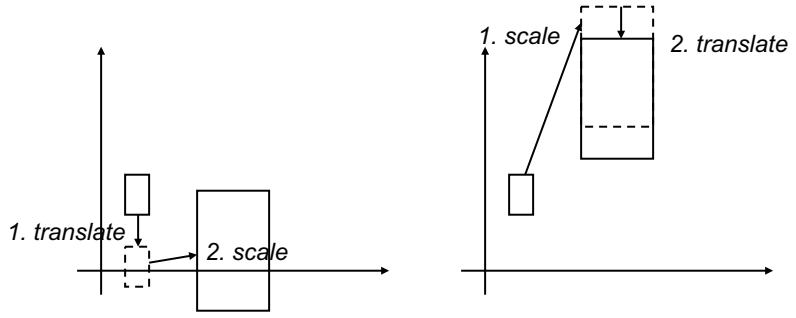
Because matrix multiplication is associative, we can rewrite this as

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

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

Composing Transformations, 3

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

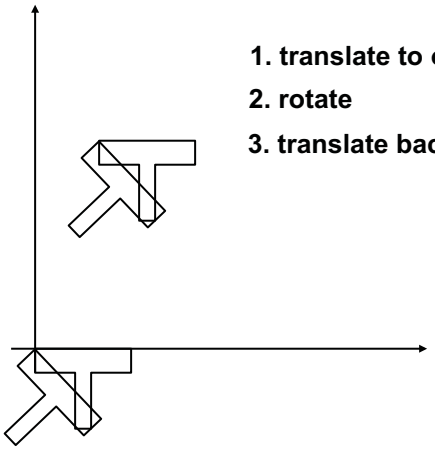


- **If we invert the matrix, we reverse the transformation.**

Transformations and the Origin

- 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.
  - at.rotate(double theta, double x, double y) does this for rotation
  - You must do it yourself for scale()

Transformations and the Origin, 2



1. translate to origin
2. rotate
3. translate back

Compound Transformations

Build a compound transform by

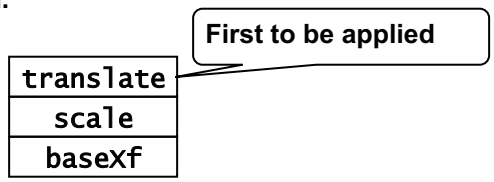
1. **Creating a new instance of** `AffineTransform`
2. **Calling methods to build a *stack* of basic transforms: last in, first applied:**
   - `translate(double tx, double ty)`
   - `scale(double sx, double sy)`
   - `rotate(double theta)`
   - `rotate(double theta, double x, double y)` **rotates about (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.



Exercise 4

- **Modify RectanglePanel**
  - **Initially, rectangle is 50 by 100, at origin**
  - **Apply the following transforms:**
    - **Translate rectangle 50 pixels east, 200 pixels south**
    - **Scale by factor of 1.5, but leave upper left corner of rectangle in same position**
    - **Rotate by 30 degrees clockwise (rotate around the upper left corner)**
  - **Draw the original rectangle in red**
  - **Draw the transformed rectangle in blue**
  - **Remember to apply transforms in reverse order. The exercise is a bit sneaky.**
  - **Remember to translate back to the origin to scale an object without moving it**

1.00 / 1.001 / 1.002 Introduction to Computers and Engineering Problem Solving
Spring 2012