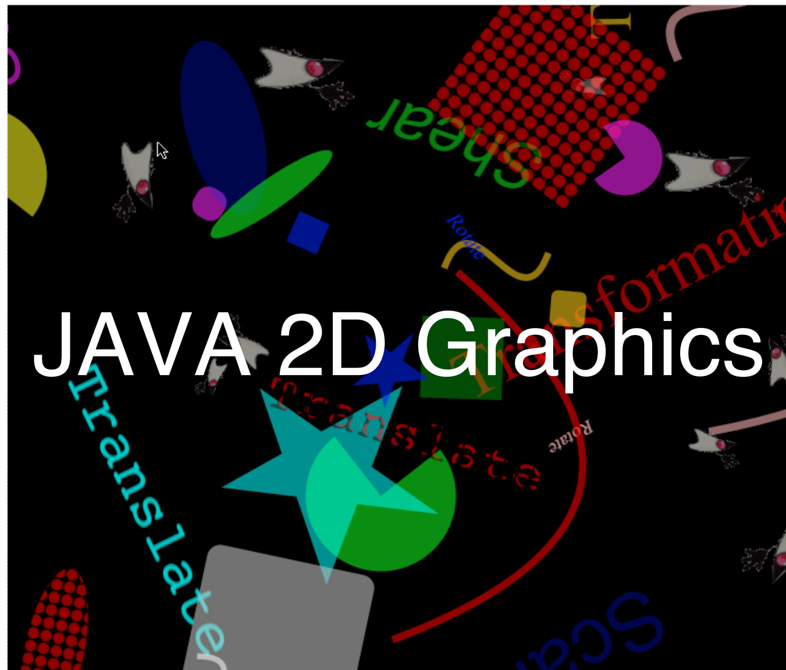
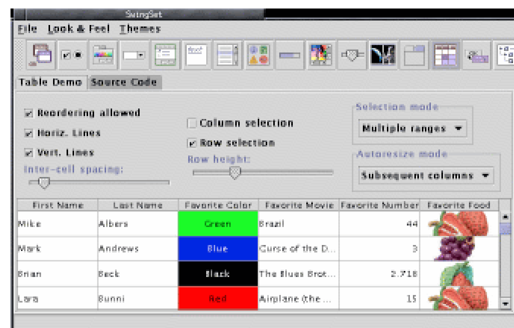
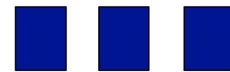


AWT and JFC/Swing

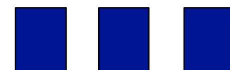
- AWT and JFC/Swing are A Good Thing (TM)
  - Forces consistency in application look and feel
  - Learning curve is more manageable
  - Applications have a more "professional" look
  - Rapid application development



- Canned goods always limit you to what the manufacturer believes you will need
- AWT did a reasonable job at portable graphics
- JFC/Swing improved this with a better look and feel as well as a more components
- Is there anything else we might want to do?

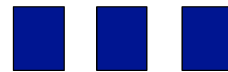


- Highly interactive user interfaces
  - Games (old-style Atari/Nintendo type)
  - HUD overlay graphics
- Drawing programs and bitmap editors
  - AutoCAD, Photoshop
- Demonstration/presentation software
  - Powerpoint
  - Interactive product tours
- We want to draw on the screen!

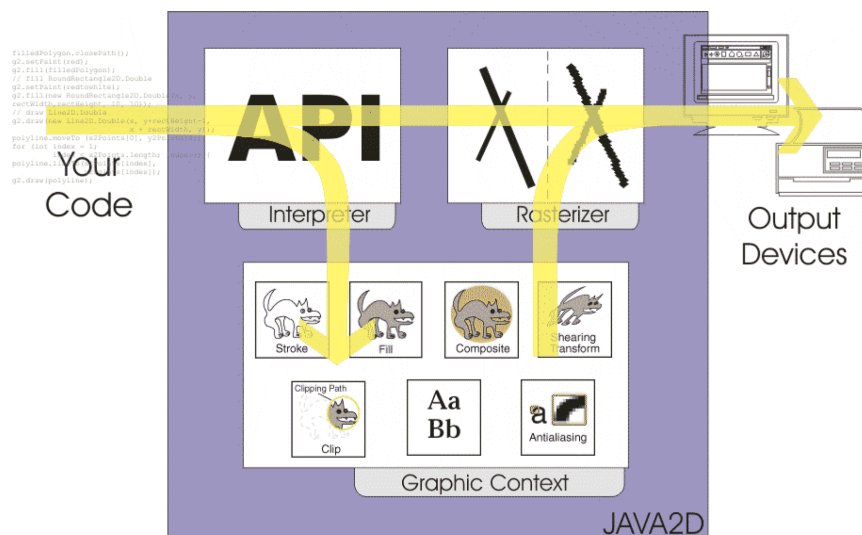


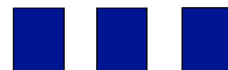
JAVA2D is NOT for...

- Run-of-the-mill “forms & actions” GUIs
  - JFC/Swing
- Streaming video / video capture
  - JAVA Media Framework
- 3D scenes/rendering (FPS type games)
  - JAVA3D
- High performance bitmap manipulation
  - JAVA Advanced Imaging API



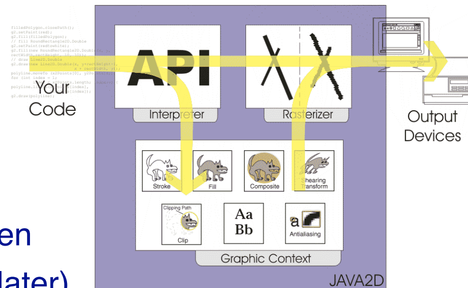
JAVA2D Rendering Model

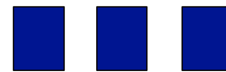




API and Destinations

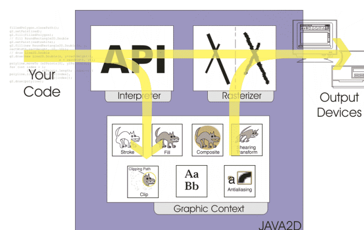
- API allows drawing of…
  - Text strings
  - Geometry
  - Bitmap (raster) images
- Destinations include…
  - Window (viewport) on screen
  - Full screen (J2SE 1.4 and later)
  - Image buffer
  - Printer (physical printing device)

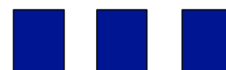




Graphic Context

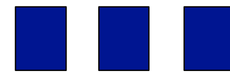
- Seven Aspects
  - Font
  - Composite (intersections)
  - Transform (displacement, scale…)
  - Stroke (pen size)
  - Paint (fill color / pattern)
  - Hints (performance / quality tradeoffs
  - Clip (drawing extents)





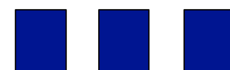
Drawing to a Window (Viewport)

- The predominent UI paradigm is WIMP
  - Windows - Icons -Menus - Pointer
- In order to conform to this:
  - We need to be given a "window" to draw into
  - Randomly scribbling on the screen would be inconsistent
- Of course, we'll use JFC/Swing for this
- You need to import javax.swing.* and java.awt.*



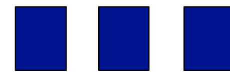
All roads lead to JComponent

- Every Swing object inherits from JComponent
- JComponent has a few methods that can be overridden in order to draw special things
  - public void paint(Graphics g)
  - public void paintComponent(Graphics g)
  - public void repaint()



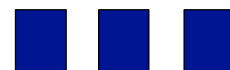
Which JComponent

- So if we want custom drawing, we take any JComponent and extend it...
  - JPanel is a good choice
- Why don't we simply extend JFrame?
  - What if the user wants to use your custom drawing component in a larger screen?
  - What if you want to take your component and put it into an applet for a browser instead of a full JFrame?



public void paint(Graphics g)

- This method is called whenever the system needs to draw the component
- If this were AWT, we would override this method and put our drawing routines here
- Generally, we don't override this in Swing

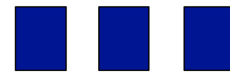


paintComponent(Graphics g)

- In Swing, we override this method to do our custom work:
- public void paintComponent(Graphics g){

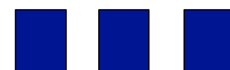
```
public void paintComponent(Graphics g){
    super.paintComponent(g);
    customDrawingMethod(g);
}
```

- We then create our custom rendering method somewhere else in the class or in a separate class



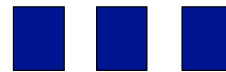
A Rendering Loop Example

```
public class SimpleRenderer {
  public static void render(java.awt.Graphics2D g) {
    g.setStroke(new java.awt.BasicStroke(15.0f));
    g.drawLine(100,100,540,380);
    g.setStroke(new java.awt.BasicStroke(5.0f));
    g.drawLine(320,150,320,400);
    g.drawLine(400,150,400,400);
    g.drawLine(480,150,480,400);
  }
}
```



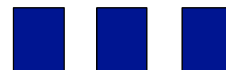
Rendering to a Window

```
public class SimpleJ2D extends javax.swing.JPanel { public void
    paintComponent(java.awt.Graphics g) {
        java.awt.Graphics2D g2 =(java.awt.Graphics2D)g;
        SimpleRenderer.render(g2);
    }
    public static void main(String[] args) {
        SimpleJ2D SJ2D = new SimpleJ2D();
        javax.swing.JFrame f = new javax.swing.JFrame();
        f.getContentPane().add(SJ2D);
        f.setSize(640,480);
        f.show();
    }
}
```



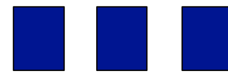
Your Rendering Infrastructure

- Why a separate class?
  - Output device independence!
  - Same rendering loop can output to viewport (window), full screen, printer and to another app
- Suggestion:
  - Create interface Renderer (single method render)
  - Define rendering logic in an implementation of the interface Renderer
  - Output device classes use implementations of Renderer



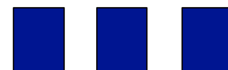
Rendering to a Printer

```
public class PrinterRendering implements Printable {
    public int print(Graphics g, PageFormat f, int pageIndex) {
        Renderer.paint(g);
    }
    public static void main(String[] args) {
        PrinterJob printerJob = PrinterJob.getPrinterJob();
        PageFormat pageFormat = printerJob.defaultPage();
        if (printerJob.printDialog() {
            printerJob.print();
        } else {
            System.out.println("Print job cancelled");
        }
    }
}
```



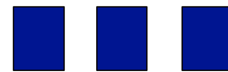
Rendering to an Image

```
BufferedImage bi = new BufferedImage(xsize, ysize, type);
Graphics2D g = bi.createGraphics();
// notice that createGraphics returns a g2d object directly, no cast!
Renderer.render(g);
// save the image
File file = new File("images","test.jpg");
FileOutputStream out = new FileOutputStream(file);
JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(out);
JPEGEncodeParam param = encoder.getDefaultJPEGEncodeParam(bi);
param.setQuality(1.0f, false);
encoder.setJPEGEncodeParam(param);
encoder.encode(bi);
```



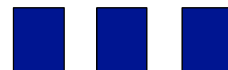
myRenderer(Graphics g)

- First we need to setup our environment
- We will almost always need to know the size of the screen that we'll be drawing on...
  - It's good practice to make your drawing routines scalable with the size of the component
- ```
  Graphics2D g2 = (Graphics2D)g;
  Dimension dim = getSize();
  double w = dim.getWidth();
  double h = dim.getHeight();
  ```



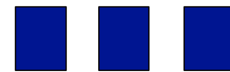
Coordinate Systems

- User Space
  - The programmer operates in this space
  - Origin is on the upper left
  - Positive X goes “right”
  - Positive Y goes “down”
- Device Space
  - Where the actually drawing takes place
  - Device dependent
  - JAVA2D performs the mapping


  ```

Reconciling Coordinate Spaces

- A Single AffirmTransform relates the two spaces
  - Default mapping for viewports: 72 units per inch
  - Most monitors are 72 DPI
  - Approximately the same size as a “point” in typography
  - Default AT scales up for most printers



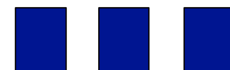
Affine Transformations

- Homogenous coordinates:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} m00 & m01 & m02 \\ m10 & m11 & m12 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

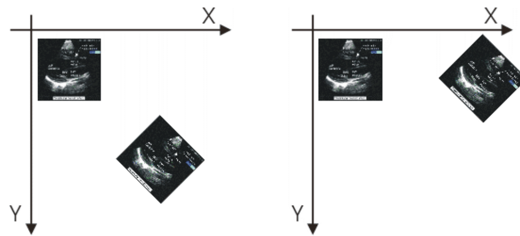
P’ = M P

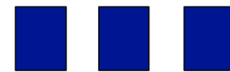
- Translation: m02 = tx, m12 = ty
- Scale: m00 = sx, m11 = sy
- Shear: m01 = shx, m10 = shy
- Rotation: m00 = cos(theta), m01 = -sin(theta),
  m10 = sin(theta), m11 = cos(theta)



Affine Transforms as Matrix Mults

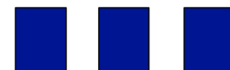
- Compose Affine Transforms together:
  - P'' = M1 P' = M1 M0 P
  - Mt = M1 M0
  - P'' = Mt P
- The order of the transform matters!



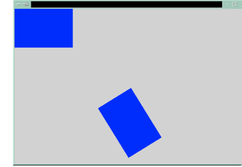


Transforming Objects

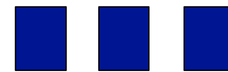
- There is a class called AffineTransform
- You can call things like translate(x,y), rotate(degrees) and skew(x,y) on any instance of AffineTransform
- You can then use "setTransform" on the g2 object reference to apply your new transform



Affine Transformation Example



```
public class XformPanel extends JPanel {
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    customDrawingMethod(g);
  }
  public void customDrawingMethod(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    double w = (getSize()).getWidth();
    double h = (getSize()).getHeight();
    g2.setPaint(Color.blue);
    g2.fill(new Rectangle2D.Double(0.0, 0.0, w/4, h/4));
    AffineTransform AT = new AffineTransform();
    AT.translate(w/2,h/2);
    AT.rotate(45);
    g2.setTransform(AT);
    g2.fill(new Rectangle2D.Double(0.0, 0.0, w/4, h/4));
  }
}
```



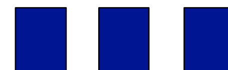
Setting the Stroke

Simple Example:

```
BasicStroke BS = new BasicStroke();
BasicStroke ThickBS = new BasicStroke(15.0f);
g.setStroke(ThickBS);
```

Grammar:

```
BasicStroke(
    float WIDTH,
    BasicStroke.[ CAP_BUTT | CAP_SQUARE | CAP_ROUND ],
    BasicStroke.[ JOIN_BEVEL | JOIN_MITER | JOIN_ROUND,
    float[] { ONLENGTH, OFFLENGTH, ... }
)
```



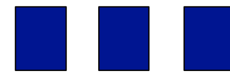
Setting the Fill

Simple Example:

```
Color TiAgMetallic = new Color(200, 230, 230);
g.setColor(TiAgMetallic);
Color SemiTransparentBlue = new Color(0, 0, 255, 0.5f);
g.setColor(SemiTransparentBlue);
```

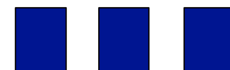
Other kinds of fills:

GradientPaint()
TexturePaint()



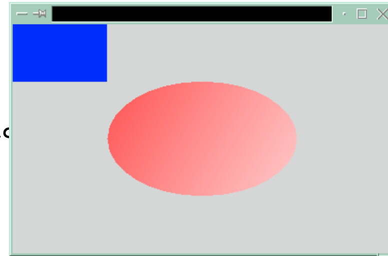
Filled Objects

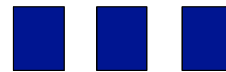
- Use the setPaint method to change a fill color:
  - g2.setPaint(java.awt.Color.red);
- Use the fill method instead of draw to draw filled into objects:
  - g2.fill(new Rectangle2D.Double(x, y, rectWidth, rectHeight));
- You can even draw a gradient:
  - GradientPaint redtowhite = new GradientPaint(0,0,Color.red, 100,100, Color.white);
  - g2.setPaint(redtowhite);



Filled Objects Example

```
public class FilledObjectPanel extends JPanel {
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    customDrawingMethod(g);
  }
  public void customDrawingMethod(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    double w = (getSize()).getWidth();
    double h = (getSize()).getHeight();
    g2.setPaint(Color.blue);
    g2.fill(new Rectangle2D.Double(0.0, 0.0, w/4, h/4));
    Point2D.Double origin = new Point2D.Double(0.0, 0.0);
    Point2D.Double end = new Point2D.Double(w, h);
    GradientPaint rtow = new
                  GradientPaint(origin,Color.red,end,Color.white);
    g2.setPaint(rtow);
    g2.fill(new Ellipse2D.Double(w/4, h/4, w/2, h/2));
  }
}
```





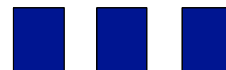
Setting the Font

Get the list of all available fonts:

```
GraphicsEnvironment env =
   GraphicsEnvironment.getLocalGraphicsEnvironment();
Font[] fonts = env.getAllFonts();
for(int i = 0; i lt fonts.length; i++) {
   System.out.println((fonts[i]).getName());
}
```

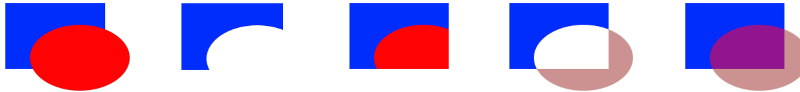
Using a particular font:

```
Font font = new Font(“Futura”, Font.PLAIN, 32);
g.setFont(font);
```



Compositing Elements

What happens when you overlay one element on top of another element? Complete opacity? Blending of colors?

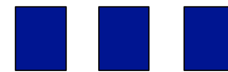


Code:

```
AlphaComposite MyComposite
  = AlphaComposite.getInstance(AlphaComposite.RULE, 0.5f);
g2.setComposite(MyComposite);
```

Demo:

http://java.sun.com/docs/books/tutorial/2d/display/Composite.html

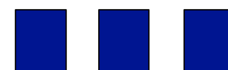


Clipping Region

•Define region where things can drawn successfully
•Convenient method for making it rectangular
•Demo of a moving clip:
http://java.sun.com/docs/books/tutorial/2d/display/ClipImage.html
•Some examples:

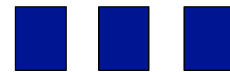
```
g.clipRect(25,25,50,50);

Shape clipShape = ...
g.setClip(clipShape);
```



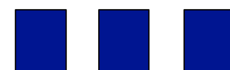
Rendering Hints

- Keys and Values
  - Alpha interpolation - quality, or speed
  - Antialiasing - on or off
  - Color rendering - quality or speed
  - Dithering - disable or enable
  - Fractional metrics - on or off
  - Interpolation - nearest-neighbor, bilinear, or bicubic
  - Rendering - quality, or speed
  - Text antialiasing - on, or off
- There is no guarantee these do anything!



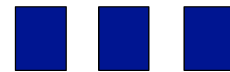
Drawing a Simple Line

- The "draw()" method takes as arguments elements that can be instantiated without a name
- g2.draw(new Line2D.Double(x,y+w-1,x+h,y));
- You could also instantiate and keep a list of the objects that you've drawn for later use
- import java.awt.geom.* for this primitive



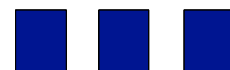
Draw a Thick Ellipse

- The setStroke can take options like widestroke or dashed to change the line type or width
- g2.setStroke(new BasicStroke(width));
- There are primitives you can placed into a draw call for all sorts of shapes
- g2.draw(new Ellipse2D.Double(x, y, w, h));



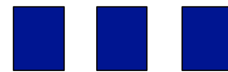
General Paths

- Use the GeneralPath drawing object to create lines of arbitrary configuration
- You have to make sure that you can describe your path in terms of a quadratic or cubic equation
- Really complicated paths can be sequences of cubic paths



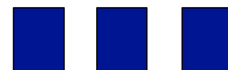
General Path Example

```
GeneralPath() path = new GeneralPath();
path.moveTo(x, y);
path.lineTo(x + deltaX, y - deltaY);
path.lineTo(x + 2*deltaX, y);
path.curveTo(0.0f,-10.0f,-20.0f,60.0f,60.0f,100.0f);
path.curveTo(140.0f,60.0f,120.0f,-10.0f,60.0f,20.0f);
path.closePath();
g2.draw(path);
```



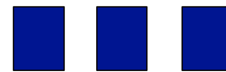
Using Images

- java.awt.Image abstract the notion of bitmapped pictures (GIF, JPEG, etc…)
- Call Toolkit.getDefaultToolkit().getImage(url) to retrieve the image
- Call g2.drawImage(myImg, x, y, this) to draw
- Lots of built in image processing functionality
- Look at JAI (Java Advanced Imaging) API for additional functionality



More Complete Example

```
String myImageURL =
    "http://www.someserver.com/~me/mypic.gif";
URL myURL = new URL(myImageURL);
Image img =
    Toolkit.getDefaultTookit().getImage(myURL);
try {
    MediaTracker tracker = new MediaTracker(this);
    tracker.addImage(img, 0);
    tracker.waitForAll();
} catch ( Exception e ) {}
g.drawImage(image, xpos, ypos, null);
```



Drawing Text

Simple text is very easy:

```
g.setFont("Whatever Font");
g.drawString("Hello World" xpos, ypos);
```

Wrapped text is **very** hard:

```
public static void drawString(Graphics2D g, String theText, float initX, float initY, float maxX) {

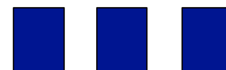
    java.text.AttributedString ASmsg = new java.text.AttributedString(theText);
    java.text.AttributedCharacterIterator paragraph = ASmsg.getIterator();
    int paragraphStart = paragraph.getBeginIndex();
    int paragraphEnd = paragraph.getEndIndex();

    java.awt.font.LineBreakMeasurer lineMeasurer =
            new java.awt.font.LineBreakMeasurer(paragraph, new java.awt.font.FontRenderContext(null, false, false));

    float formatWidth = maxX - initX;
    float drawPosY = initY;
    lineMeasurer.setPosition(paragraphStart);

    while (lineMeasurer.getPosition() lt paragraphEnd) {
        java.awt.font.TextLayout layout = lineMeasurer.nextLayout(formatWidth);
        drawPosY += layout.getAscent();
        float drawPosX;
        if (layout.isLeftToRight()) {
            drawPosX = initX;
        } else {
            drawPosX = formatWidth - layout.getAdvance();
        }

        // Draw the TextLayout at (drawPosX, drawPosY).
        layout.draw(g, drawPosX, drawPosY);
    }
}
```



Tic Tac Toe Panel

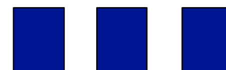
```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class TicTacToePanel extends JPanel {
  int xo [][] = { { 0, 1, 0 }, { 1, 2, 2 }, { 0, 1, 0 } };
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    customDrawingMethod(g);
  }
  public void customDrawingMethod(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    double w = (double) (getSize()).getWidth();
    double h = (double) (getSize()).getHeight();
    drawPlayingBoard(g2,w/3.0,h/3.0);
    drawXO(g2,w/3.0,h/3.0);
  }
// CONTINUED ON NEXT SLIDE
```

Tic Tac Toe Panel Continued

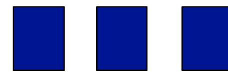
```
public void drawPlayingBoard(Graphics2D g2, double w, double h) {
  g2.setStroke(new BasicStroke(2));
  g2.draw(new Line2D.Double(w, 0, w, 3*h));
  g2.draw(new Line2D.Double(2.0*w, 0, 2.0*w, h*3));
  g2.draw(new Line2D.Double(0, h, w*3, h));
  g2.draw(new Line2D.Double(0, 2.0*h, w*3, 2.0*h));
}

public void drawXO(Graphics2D g2, double w, double h) {
  g2.setStroke(new BasicStroke(5));
  for (int p = 0; p < 3; p++) {
    for (int q = 0; q < 3; q++) {
       if (xo[p][q] == 1) {
          g2.draw(new Ellipse2D.Double(p*w, q*h, w, h));
       } else if (xo[p][q] == 2) {
          g2.draw(new Line2D.Double(p*w,q*h,(p+1)*w,(q+1)*h));
          g2.draw(new Line2D.Double(p*w,(q+1)*h,(p+1)*w,q*h));
       }
    }
  }
```



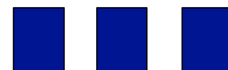
Tic Tac Toe Frame

```
import javax.swing.*;
import java.awt.*;
public class TicTacToeFrame extends JFrame {
  public TicTacToeFrame() {
     TicTacToePanel TTTP = new TicTacToePanel();
     getContentPane().add(TTTP,
                              BorderLayout.CENTER);
     setSize(new java.awt.Dimension(600,400));
  }
  public static void main(String[] args) {
     (new TicTacToeFrame()).show();
  }
}
```



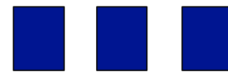
Animation

- repaint() is used to schedule the component to be repainted... but sometimes that is not fast enough
- A simple hack is to override repaint()
  - public void repaint() {
      paint()
    }
- This causes heavy CPU usage but gets the results that you would expect because it repaints as fast as possible



Animation

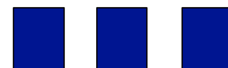
- A better idea than forcing repaint() to call paint()
  - Create a helper thread
  - Use the helper thread to change variable related to what will be drawn on the screen
  - Have the helper thread call repaint periodically
- You may also want to have event driven painting
  - Create a UnicaseRemoteObject helper
  - Have the server notify the panel to redraw
  - Use RMI to retrieve global animation variables from the server in the repaint method



Animation Example

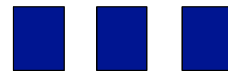
```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;

public class AnimPanel extends JPanel implements ActionListener {
  private Timer timer;
  public AnimPanel() {
    timer = new Timer(500, this); // half second
    timer.setInitialDelay(0);
    timer.setCoalesce(true);
    timer.start();
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    customDrawingMethod(g);
  }
  public void actionPerformed(ActionEvent e) {
    this.repaint();
  }
```



Animation Example

```
public void customDrawingMethod(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    double w = (getSize()).getWidth();
    double h = (getSize()).getHeight();
    double r1 = Math.random();
    double r2 = Math.random();
    g2.setPaint(Color.blue);
    g2.fill(new Rectangle2D.Double(r1*w-w/16, r2*h-h/16, w/8, h/8));
  }
  public static void main(String[] args) {
    JFrame Frame = new JFrame();
    (Frame.getContentPane()).add(new AnimPanel());
    Frame.setSize(new java.awt.Dimension(600,400));
    Frame.show();
  }
}
```



Summary

- JAVA2D provides a way to “draw” on a window
- Use this to:
  - Create custom (non-toolkit-based) user interfaces
  - Build simple animations (2D games)
  - Visualize two dimensional data (charts & graphs)
  - Load and display bitmaps in a window