

## Introduction to Computation and Problem Solving

### **Class 17:** ***Lab: The Graphics 2D API***

Prof. Steven R. Lerman  
and  
Dr. V. Judson Harward

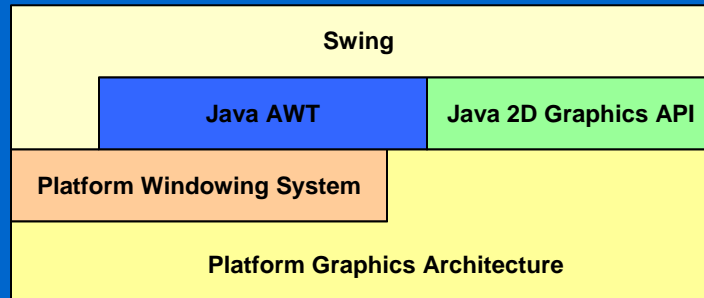
1

### **The Origins of the Java Graphics 2D API**

- The original Java GUI toolkit, the AWT, was a quick and dirty solution. It used native peer components to draw all widgets.
- Swing draws all components except the top level containers using Java methods rather than relying on platform specific widgets.
- To do this, Swing required improved graphics.
- The Java 2D API was born as enabling technology for Swing.
- There is now a Java 3D API also.
- See tutorial at <http://java.sun.com/docs/books/tutorial/2d/index.html>

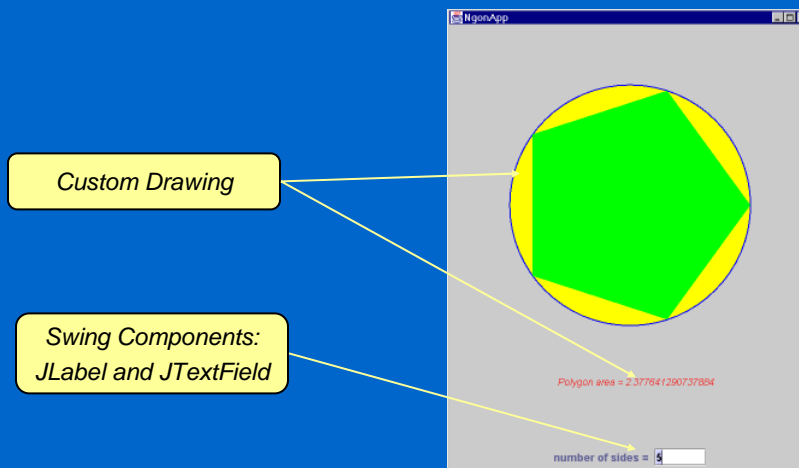
2

## Java Graphics Architecture



3

## NgonApp



4

## So, How Do I Draw in Java

- You might think that if you want to do your own drawing in Java, you do it in the main thread just as you create a GUI out of components in the main thread.
- In fact, you have to create components on which you draw like all your other GUI components in the main thread.
- But the drawing has to be done in the event thread.
- And to understand why, we need to start by considering when drawing happens.

5

## When Does a GUI Get Drawn?

- On initial display (but not necessarily when the program gets started)
- When the display gets “damaged”. For example when it gets hidden and then re-exposed.
- When the content changes, and Swing or the code written by the programmer calls for a refresh (`repaint( )`).

– Remember the `tick()` method from the previous lab?

```
public void tick() {  
    minutes++; // increment # of minutes  
    repaint(); // refresh clock display  
}
```

6

## How does a GUI Get Drawn?

- Swing schedules the drawing. It may combine multiple redraw requests that occur in rapid succession.
- Swing calls the following three methods in order (on the event thread):  
`paintComponent()`  
`paintBorder()`  
`paintChildren()`
- The last recursively paints a container's children.

7

## How to Do Custom Drawing

- Standard Swing components like `JLabel` and `JComboBox` use `paintComponent()` to draw themselves.
- If you want to do custom drawing, extend a container class, usually `JPanel`, and override `paintComponent()`.
- Use calls from the 2D API in `paintComponent()` to draw what you want on the `JPanel` background.
- Override `getPreferredSize()` or call `setPreferredSize()` to size `JPanel` to your drawing.

8

## The Graphics Class

- `paintComponent()` is called with argument `Graphics g`, that serves as a *drawing toolkit* initialized to the component's defaults.
- In more recent versions of the JDK, the argument is really a `Graphics2D` object, which extends `Graphics` for the 2D API. So cast it. `Graphics` was the original AWT class.
- Using the 2D API usually starts off like this:

```
public void paintComponent( Graphics g ) {  
    super.paintComponent( g );  
    Graphics2D g2 = (Graphics2D) g;  
    //drawing commands go here  
}
```

9

## Where Does the Graphics Argument Come From?

- The `Graphics` argument to the `paintComponent()` method is a snapshot of your component's default graphics values like font and drawing color at the moment the paint methods are called.
- It is only a copy of these values. Each time the paint methods are called, you get a new version of the `Graphics` object.
- No changes you make to a `Graphics` instance in one call to `paintComponent()` are remembered the next time you enter the method. And no changes, like `setFont()` are propagated back to the component itself.

10

## Basic 2D API Operations

You can use the `Graphics2D` argument to

1. draw outline figures using the method  
`public void draw( Shape s )`
2. draw filled figures using the method  
`public void fill( Shape s )`

You can use the `Graphics` or `Graphics2D` argument to

3. draw an image using one of the methods:  
`public void drawImage( . . . )`
4. draw a text string using the methods:  
`public void drawString( . . . )`

11

## Graphics 2D Rendering Context

Much of the power of the 2D API comes from the user's ability to set attributes of the `Graphics2D` object known collectively as the rendering context:

- `public void setStroke(Stroke s)`
- `public void setPaint(Paint p)`
- `public void setFont(Font f)`
- `public void setComposite(Composite c)`
- `public void setTransform(Transform t)`
- `public void setRenderingHints(Map m)`

12

## Custom Drawing Template

```
import java.awt.*; // for Graphics2D, Paint, Shape, ...
import java.awt.geom.*; // for concrete Shape classes
import javax.swing.*; // for JPanel, etc
public class MyPanel extends JPanel {
    . . .
    public void paintComponent( Graphics g ) {
        super.paintComponent( g );
        Graphics2D g2 = (Graphics2D) g;
        . . .
        g2.setPaint/Stroke/Font/etc(...);
        Shape s = new Shape2D.Float/Double( ... );
        g2.draw/fill( s );
        . . .
    }
}
```

13

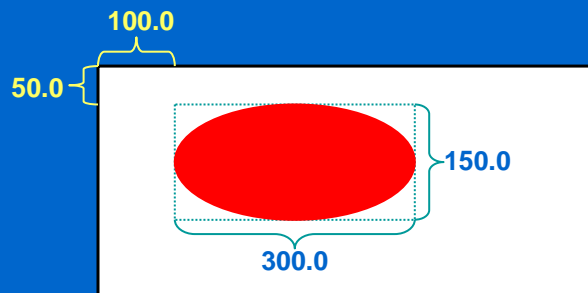
## 2D Shapes

- Shape is an interface defined in `java.awt`, but the classes that implement Shape are all defined in `java.awt.geom`.
- Shapes all come in two versions, one with high precision coordinates and one with low, e.g.:  
`Ellipse2D.Double` // high precision  
`Ellipse2D.Float` // low precision
- Each shape has different constructor arguments, doubles or floats depending on whether they are high precision or low.

14

## Creating an Ellipse

```
Shape s = new Ellipse2D.Double(  
    100.0, // x coord of bounding box  
    50.0, // y coord of bounding box  
    300.0, // width  
    150.0 // height );
```



15

## Graphics2D Coordinate System

- The `Graphics2D` object uses a variety of *world coordinates* (as opposed to *device coordinates*) that Java calls *user space*.
- `Graphics2D` shapes and operations are defined in floating point (`float` or `double`) but the `y` coordinate increases downwards.
- Some `Graphics2D` calls only take `floats`.
- The floating point coordinates are designed to make your graphics independent of the output device.
- The 2D API is designed for printing as well as output to screens at different resolutions.

16

## The Graphics2D Default Transformation

- The 2D rendering pipeline applies a geometric *transformation* to map user space to device space.
- The default transformation maps 1.0 user space units to ~1/72 of an inch, which happens to be the typical pixel size on a screen or a printer's point size on a printer.
- So unless you do something special, 2D drawing defaults to pixel coordinates.

17

## Ellipse.java, 1

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

public class Ellipse extends JPanel {
    private Shape ellipse;

    public static void main( String[] args ) {
        JFrame frame = new JFrame( "Ellipse" );
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add( new Ellipse(), "Center" );
        frame.setSize( 650, 300 );
        frame.setVisible( true );
    }
}
```

18

## Ellipse.java, 2

```
public Ellipse() {
    ellipse = new Ellipse2D.Double( 100.0, 50.0,
                                    300.0, 150.0 );
    setBackground( Color.white );
    setOpaque( true );
}

public void paintComponent( Graphics g ) {
    super.paintComponent( g );
    Graphics2D g2 = (Graphics2D) g;
    g2.setPaint( Color.red );
    g2.fill( ellipse );
}
}
```

19

## Strokes and Paint

- **Stroke** is an interface, and **BasicStroke** is the only supplied class that implements it.
- You can think of **Stroke** as a pen and **Paint** as the corresponding ink.
- A **BasicStroke** can have width and a dashed pattern as well as options that define how a **Stroke** ends and treats joints.
- **Color** implements the **Paint** interface so a Java **Color** is the simplest sort of **Paint**.
- **Paints** can use a pattern or texture (for example, plaid).

20

## Exercise

1. Use the `Graphics2D draw()` method instead of `fill()` to get an outline instead of a filled ellipse.
2. Change the `stroke` to be 10 units thick. What are the units?
3. Make the ellipse a circle.
4. Change the circle to be a square (lookup `Rectangle2D`).

21

## GeneralPaths

- How can you define your own `Shape`?
- Use a `GeneralPath`. You define the outline by adding path components that can be `Shapes`, `Lines`, or curves.

```
void append( Shape s, boolean connect );
void lineTo( float x, float y );
void moveTo( float x, float y );
void quadTo( float x1, float y1,
             float x2, float y2 );
void closePath();
```

22

## NullSymbol.java, 1

```
import javax.swing.*;
import java.awt.geom.*;
import java.awt.*;
public class NullSymbol
  extends JPanel {
    private GeneralPath  slash0;
    private Stroke       brush;

    public NullSymbol() {
        setPreferredSize( new Dimension( 600, 400 ) );
        setBackground( Color.white );
        setOpaque( true );
        brush = new BasicStroke( 10, BasicStroke.CAP_ROUND,
                                BasicStroke.JOIN_ROUND );
        slash0 = new GeneralPath();
        Shape e = new Ellipse2D.Double( 220,100,160,200 );
        slash0.append( e, false );
        slash0.moveTo( 350, 100 );
        slash0.lineTo( 250, 300 );
    }
}
23
```

## NullSymbol.java, 2

```
public void paintComponent( Graphics g ) {
    super.paintComponent( g );
    Graphics2D g2 = (Graphics2D) g;

    g2.setStroke( brush );
    g2.setPaint( Color.blue );
    /*
    g2.setRenderingHint( RenderingHints.KEY_ANTIALIASING,
                          RenderingHints.VALUE_ANTIALIAS_ON
    );
    **/
    g2.draw( slash0 );
}
}
```

24

## Drawing Text

- Up until now we have presented text using JLabels.
- You can draw text directly using the Graphics2D methods:

```
public void drawString( String s,
                       int x, int y );
public void drawString( String s,
                       float x, float y );
```

- The coordinates define the left end of the baseline on which the text appears.

25

## Signature.java, 1

```
import javax.swing.*;
import java.awt.Font;
import java.awt.*;
import java.awt.geom.*;
public class Signature
    extends JPanel {
    private String name = "Judson Harward";
    private Font signFont;
    private Stroke underStroke;
    private Line2D underline;

    public Signature() {
        setPreferredSize( new Dimension( 600, 400 ) );
        setBackground( Color.white );
        setOpaque( true );
        underStroke = new BasicStroke( 1 );
        underline = new Line2D.Float( 100F,300F,500F,300F );
        signFont = new Font( "Serif", Font.ITALIC, 24 );
    }
}
```

26

## Signature.java, 2

```
public void paintComponent( Graphics g ) {
    super.paintComponent( g );
    Graphics2D g2 = (Graphics2D) g;

    g2.setStroke( underStroke);
    g2.setPaint( Color.lightGray );
    g2.draw( underline );
    g2.setFont( signFont );
    g2.setPaint( Color.blue );
    g2.drawString( name, 110F, 300F );
}
}
```

27

## Exercise: Building NgonApp, 1

- You are going to build an application that illustrates how the area of an inscribed polygon approaches that of a circle as the number of sides increases.
- Unpack the files `NgonApp.java` and `NgonView.java` from the `JavaFiles.zip` in the lecture directory on the class web site and save them into a new directory. Create a new project in Eclipse for the directory that you just saved the files into.
- Compile and run.

28

## Building NgonApp, 2

NgonApp should look like this before you change anything:



29

## Building NgonApp, 3

- Let's examine the source code.
- `NgonApp.java` contains the `main()` method and the class that lays out the GUI. You will not have to change it. Note that it creates an instance of `NgonView` and puts it in the center of the content pane.
- It uses a `JTextField` to ask for the number of polygon sides. When you type a return into a `JTextField` it will issue an action event. Check how that `ActionEvent` is handled. Remember that it must be an error to specify a polygon with less than 3 sides. We will handle that error in `NgonView`.

30

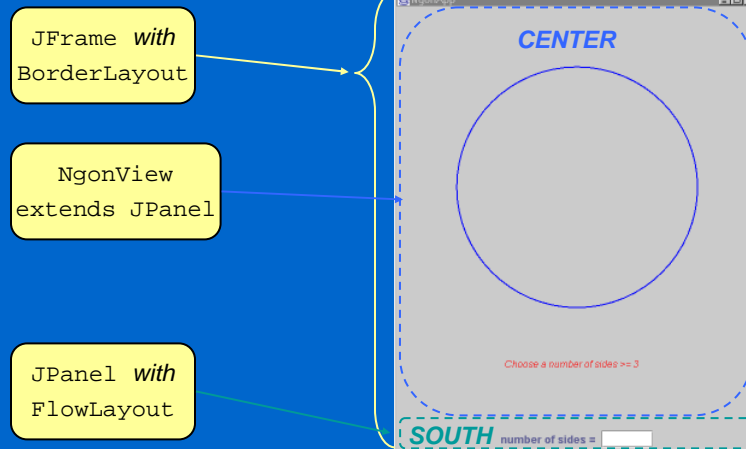
## Initial NgonView

`NgonView` is the class that does all the custom drawing. The initial version has just the `paintComponent()` code to draw the background and certain helper methods.

- `void setSides(int n)`: installs a new number of polygon sides
- `double getRegPolyArea( int n )`: calculates the area of a regular polygon with  $n$  sides inscribed in the unit circle
- Dimension `getPreferredSize()`: returns a fixed size
- `float transX( float x )`,  
`float transY( float y )`: we'll come back to these in a minute

31

## NgonApp, 1<sup>st</sup> Version After Adding Code



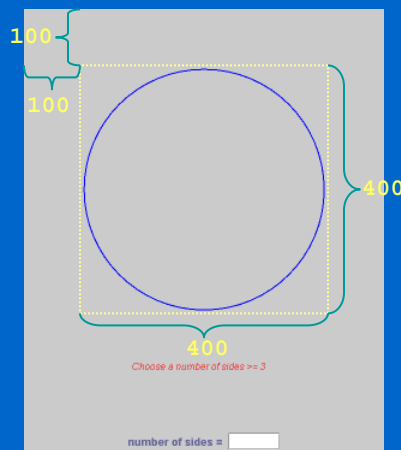
32

## NgonApp, 1st Modification

- Modify `NgonView.java` so that it will either display an error message or display the polygon area every time the number of sides changes. You will need to create an appropriate font, but then should only have to modify `paintComponent()` to draw the appropriate message at `(textX, textY)`. Compile and test.
- Now modify `paintComponent()` again to draw a blue circle with a stroke two pixels wide as shown on the next slide. Don't use `transX/transY()`.
- Compile and test.
- Now try filling the circle in yellow. Do you want to draw the outline first and then fill, or vice versa? Why?

33

## NgonView, Mod 1 Positioning the Circle



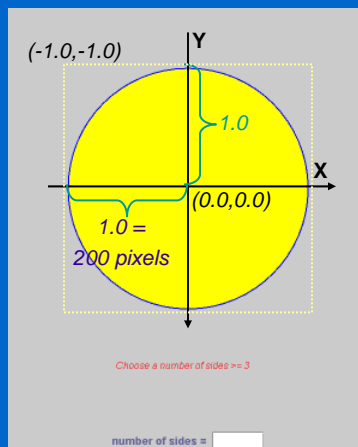
34

## NgonApp, Modification 2

- In order to draw a regular polygon inscribed in a unit circle, it is going to be much easier to think in terms of a coordinate system with its origin at the center of the circle as in the following slide. Note that the scale is set so that the circle is a unit circle. `transX()` and `transY()` translate points in this coordinate system to the pixel coordinates of the window. Test it. The center of the unit circle is  $(0,0)$ . What is `transX(0)`? `transY(0)`?
- See if you can recreate the filled circle on NgonView by using `transX/Y()` to define the upper left corner of the bounding box and `SCALE` to define the width and height.
- Compile and test.

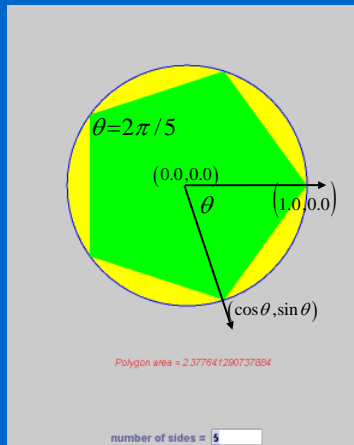
35

## NgonView, Mod 2 Using Transformed Coordinates



36

## NgonView, Mod 3 Inscribing the Polygon, 1



37

## NgonView, Inscribing the Polygon, 2

- Now use a `GeneralPath` (and `transX` and `transY( )` methods) to create the filled inscribed polygon. Start the path at  $(1,0)$ . Calculate the central angle between vertices. Use a loop to generate the first  $n-1$  sides, and `closePath()` to generate the last side. Remember that because  $y$  increases downwards the first vertex will be below the  $X$ -axis, not above as in regular coordinate geometry. Add the appropriate method call to `paintComponent( )` to fill the polygon.
- Compile and test.

38