Object Oriented Programming

1. Event handling in Java
2. Introduction to Java Graphics

Reminder: What is a callback?

- *Callback* is a scheme used in event-driven programs where the program registers a subroutine (a "callback handler") to handle a certain event.
- The program does not call the handler directly but when the event occurs, the run-time system calls the handler, usually passing it arguments to describe the event.

Reminder: Inner Classes

- Inner classes can be created within a method or even an arbitrary scope.
- When to use inner classes:
  - When implementing an interface of some kind so that you can create and return a reference.
  - When solving a complicated problem and you want to create a class to aid in your solution, but you don’t want it publicly available.
- Being class members, inner classes can be made *private* or *protected*, which is not possible with normal (non-inner classes)

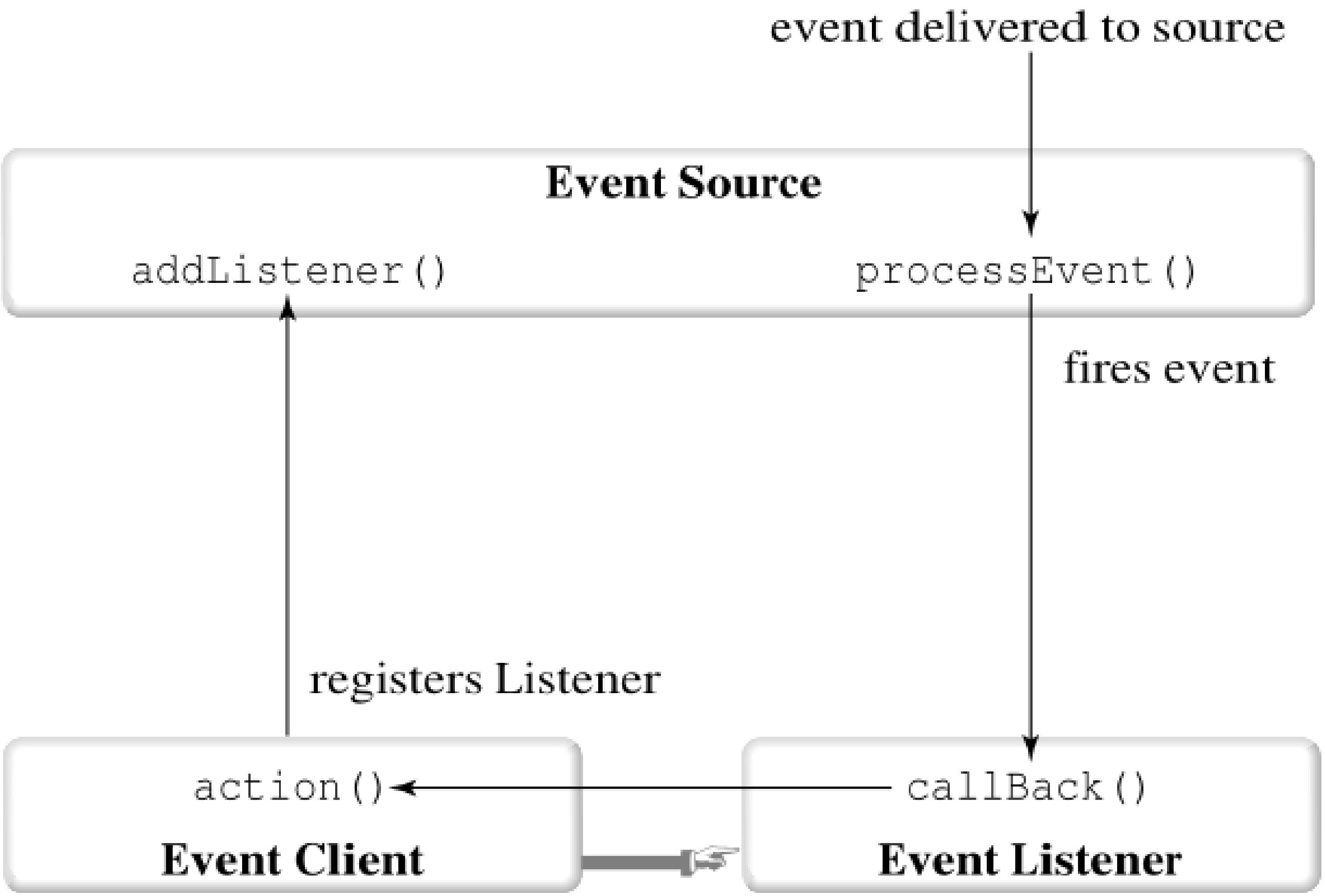
Events, Event Sources, and Event Listeners

- All user actions belong to an abstract set of things called *events*.
- An event *describes*, in sufficient detail, a particular user *action*.
- The Java run time *notifies* the program when an interesting event occurs.
- Programs that handle user interaction in this fashion are said to be *event driven*.
- User interface *events* include key presses, mouse moves, button clicks, and so on
- A program can indicate that it only cares about certain *specific* events

Events, Event Sources, and Event Listeners

- Event *listener*:
  - Notified when event happens
  - Belongs to a class that is provided by the application programmer
  - Its methods describe the *actions* to be taken when an event occurs
  - A program *indicates which events* it needs to receive by installing *event listener objects*
- Event *source*:
  - Event sources *report* on events
  - When an event occurs, the event source *notifies all event listeners*

Event-handling Model



Events, Event Sources, and Event Listeners

- Example: Use `JButton` components for buttons; attach an `ActionListener` to each button
- `ActionListener` interface:

```
public interface ActionListener {
   void actionPerformed(ActionEvent event);
}
```

- Need to supply a class whose `actionPerformed` method contains instructions to be executed when button is clicked
- `event` parameter contains details about the event, such as the time at which it occurred
- Construct an object of the listener and add it to the button:

```
ActionListener listener = new ClickListener();
button.addActionListener(listener);
```

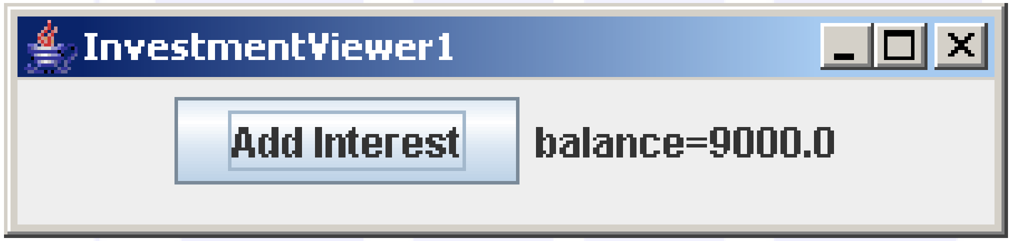
An example (+BlueJ Demo)

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
/**
   An action listener that prints a message.
*/
public class ClickListener implements
            ActionListener
{
   public void
      actionPerformed(ActionEvent event)
   {
     System.out.println("You clicked me.");
   }
}
/*-----------------------------------------*/
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
/**
   This program demonstrates how to
   install an action listener.
*/
```

```
public class ButtonTester {
   private static final int
      FRAME_WIDTH = 100;
   private static final int
     FRAME_HEIGHT = 60;
   public static void main(String[] args) {
      JFrame frame = new JFrame();
      JButton button = new JButton("Click
here!");
      frame.add(button);
      ActionListener listener = new
ClickListener();
      button.addActionListener(listener);
      frame.setSize(FRAME_WIDTH,
                    FRAME_HEIGHT);
       frame.setDefaultCloseOperation(
        JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

Building Applications With Buttons

- Example: investment viewer program; whenever button is clicked, interest is added, and new balance is displayed



balance=9000.0

- Construct an object of the `JButton` class:

```
JButton button = new JButton("Add Interest");
```

- We need a user interface component that displays a message:

```
JLabel label=new JLabel("balance="+account.getBalance();
```

Building Applications With Buttons

- Use a `JPanel` container to group multiple user interface components together:

```
JPanel panel = new JPanel();
panel.add(button);
panel.add(label);
frame.add(panel);
```

- Listener class adds interest and displays the new balance:

```
class AddInterestListener implements ActionListener {
   public void actionPerformed(ActionEvent event) {
      double interest = account.getBalance() *
         INTEREST_RATE / 100;
      account.deposit(interest);
      label.setText("balance=" + account.getBalance());
   }
}
```

- Add `AddInterestListener` as inner class so it can have access to surrounding **`final`** variables (`account` and `label`). (BlueJ demo: InvestmentViewer1).

Processing Text Input

- Use **`JTextField`** components to provide space for user input

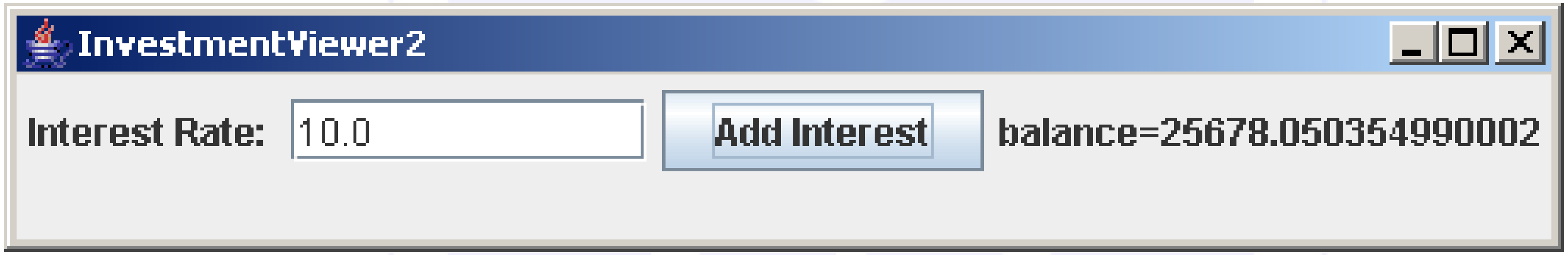
```
final int FIELD_WIDTH = 10; // In characters
final JTextField rateField = new JTextField(FIELD_WIDTH);
```

- Place a **`JLabel`** next to each text field

```
JLabel rateLabel = new JLabel("Interest Rate: ");
```

- Supply a button that the user can press to indicate that the input is ready for processing

Processing Text Input



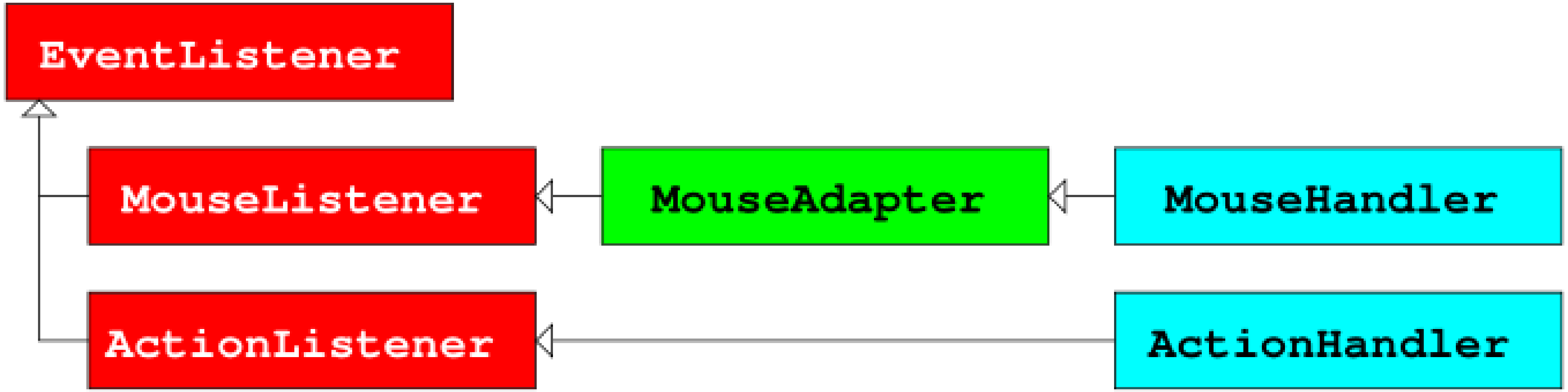
- The button's **actionPerformed** method reads the user input from the text fields (use **getText**)

```
class AddInterestListener implements ActionListener {
  public void actionPerformed(ActionEvent event) {
    double rate=Double.parseDouble(rateField.getText());
    . . .
  }
}
```

BlueJDemo: InvestmentViewer2

Writing Event Handlers

- A possible way



Interfaces

Abstract classes

Concrete classes

Mouse Events

- Use a mouse listener to capture mouse events
- Implement the **MouseListener** interface:

```
public interface MouseListener {
   void mousePressed(MouseEvent event);
   // Called when a mouse button has been pressed on a component
   void mouseReleased(MouseEvent event);
   // Called when a mouse button has been released on a component
   void mouseClicked(MouseEvent event);
   // Called when the mouse has been clicked on a component
   void mouseEntered(MouseEvent event);
   // Called when the mouse enters a component
   void mouseExited(MouseEvent event);
   // Called when the mouse exits a component
}
```

Mouse Events

- `mousePressed, mouseReleased`: called when a mouse button is pressed or released
- `mouseClicked`: if button is pressed and released in quick succession, and mouse hasn't moved
- `mouseEntered, mouseExited`: mouse has entered or exited the component's area
- Add a mouse listener to a component by calling the `addMouseListener` method:

```
public class MyMouseListener implements MouseListener
{
   // Implements five methods
}
MouseListener listener = new MyMouseListener();
component.addMouseListener(listener);
```

Mouse Events

- Sample program: when user clicks move the rectangle component
- Call `repaint` when you modify the shapes that `paintComponent` draws:

```
box.setLocation(x, y);
repaint();
```

- Mouse listener: if a mouse button is pressed, listener moves the rectangle to the mouse location
- BlueJ demo: RectangleMover.java

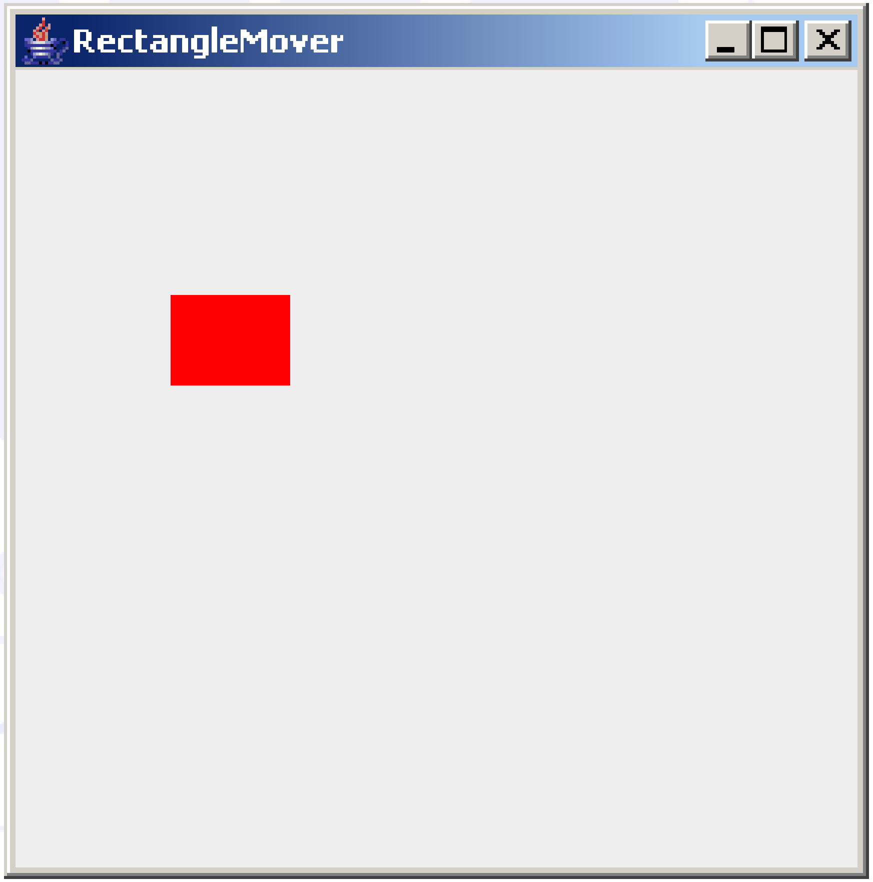
Mouse Events

```
class MousePressListener implements MouseListener
{
   public void mousePressed(MouseEvent event) {
      int x = event.getX();
      int y = event.getY();
      component.moveTo(x, y);
   }
   // Do-nothing methods
   public void mouseReleased(MouseEvent event) {}
   public void mouseClicked(MouseEvent event) {}
   public void mouseEntered(MouseEvent event) {}
   public void mouseExited(MouseEvent event) {}
}
```

- **All five methods of the interface must be implemented; unused methods can be empty**

Mouse Events Example

- BlueJDemo: RectangleMover



Java Graphics Systems

- The Java SDK contains two different graphics systems
  - The Abstract Windowing Toolkit (AWT), which was the original Java graphics system
  - The Swing package, which is a newer, more flexible graphics system
- We'll discuss only Swing graphics

Components and Containers

- The two principal types of graphics objects are **Containers** and **Components**
- A **Component** is visual object containing text or graphics
- A **Container** is a graphical object that can hold *components* or other *containers*
  - The principal container is a **Frame**. It is a part of the computer screen surrounded by *borders* and *title bars*.

Displaying Java Graphics

- To display Java graphics:
  1. Create the component or components to display
  2. Create a frame to hold the component(s), and place the component(s) into the frame(s).
  3. Create a “listener” object to detect and respond to mouse clicks, and assign the listener to the frame.
- Now we'll use components of class `JPanel`, and containers of class `JFrame`

Displaying Java Graphics

Required packages

Create "Listener"

Create component

Create frame

Add listener and component to frame

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class TestJPanel {
   public static void main(String s[]) {
      // Create a Window Listener to handle "close" events
      MyWindowListener l = new MyWindowListener();
      // Create a blank yellow JPanel to use as canvas
      JPanel c = new JPanel();
      c.setBackground( Color.yellow );
      // Create a frame and place the canvas in the center
      // of the frame.
      JFrame f = new JFrame("Test JPanel ...");
      f.addWindowListener(l);
      f.add(c, BorderLayout.CENTER);
      f.pack();
      f.setSize(400,400);
      f.setVisible(true);
   }
}
```

(DisplayGraphicsEx1)

Listeners

- A “listener” class listens for mouse clicks or keyboard input on a container or component, and responds when it occurs
  - We will use a “Window” listener to detect mouse clicks and to shut down the program

```
import java.awt.event.*;
public class MyWindowListener extends WindowAdapter {

  // This method implements a simple listener that detects
  // the "window closing event" and stops the program.
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  };
}
```

Trap mouse clicks in the “Close Window” box, and exit when one occurs

Displaying Graphics on a Component

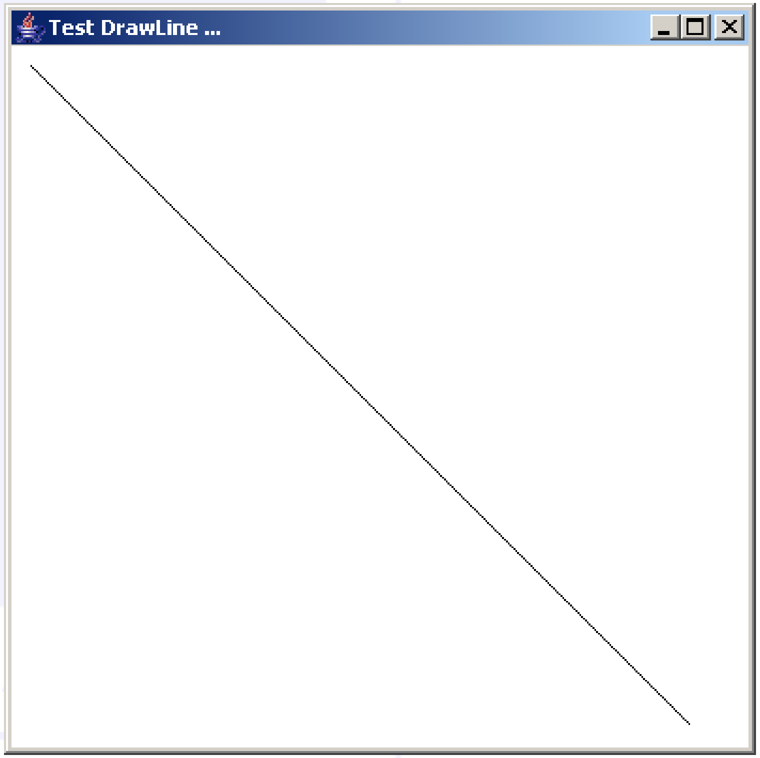
- The `paintComponent` method is used to draw graphics on a component.
  - The call is:

    ```
    paintComponent( Graphics g )
    ```

  - The `Graphics` object must be immediately cast to a `java.awt.Graphics2D` object before it can be used with Swing graphics
  - Once this is done, all of the classes in `java.awt.geom` can be used to draw graphics on the component

Example: Drawing a Line

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
public class DrawLine extends JPanel {
   public void paintComponent ( Graphics g ) {
      // Cast the graphics object to Graphics2D
      Graphics2D g2 = (Graphics2D) g;
      // Set background color
      Dimension size = getSize();
      g2.setColor( Color.white );
      g2.fill(new Rectangle2D.Double(0,0,
              size.width,size.height));
      // Draw line
      g.setColor( Color.black );
      Line2D line = new Line2D.Double (10., 10., 360.,
360.);
      g2.draw(line);
    }
...
   main method here
...
```



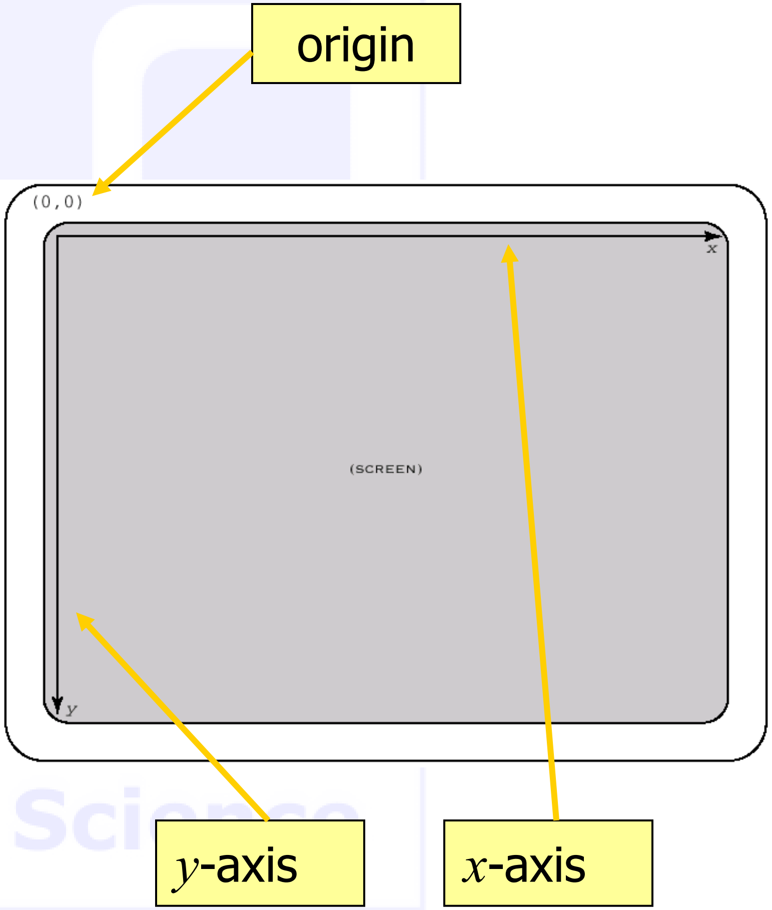
(DisplayGraphicsEx2)

Create Line2D object

Draw line represented by object

The Graphics Coordinate System

- Java uses a graphics coordinate system with the origin (0,0) in the *upper left-hand corner*
  - *x* axis is positive to the right
  - *y* axis is positive down
- By default, the units of measure are *pixels*
  - There are 72 pixels / inch
- Unit of measure can be changed



The `Line2D` Classes

- There are two concrete classes for creating lines: `Line2D.Float` and `Line2D.Double`. The only difference between them is the units of the calling parameters.
- Constructors:

```
Line2D.Double( double x1, double y1,
               double x2, double y2 )
Line2D.Float( float x1, float y1,
               float x2, float y2 )
```

- These classes create a line from (x_1,y_1) to (x_2,y_2)

Controlling Object Color

- The color of a graphics object is controlled by the `Graphics2D` method `setColor`.
- The color may be any object of class `java.awt.Color`, including the following pre-defined values:

```
Color.black
Color.blue
Color.cyan
Color.darkGray
Color.green
Color.lightGray
Color.magenta
Color.orange
Color.pink
Color.red
Color.white
Color.yellow
```

Controlling Line Width and Style

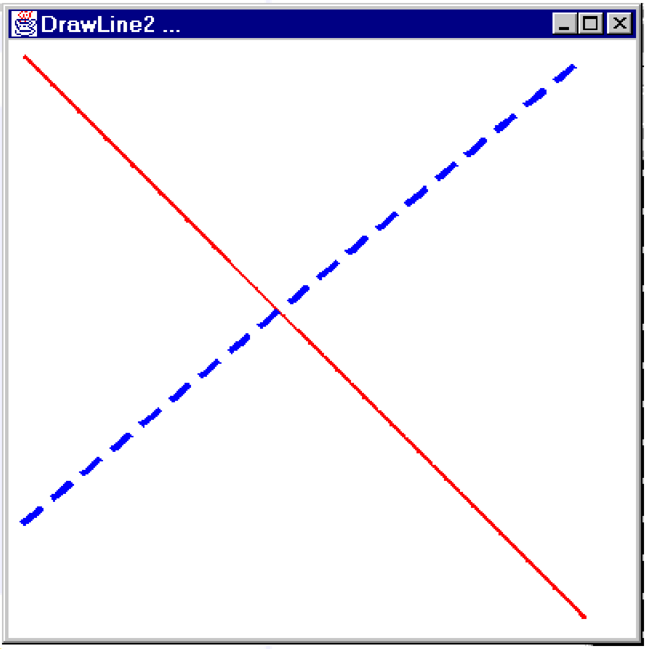
- Line width and style is controlled with a **`BasicStroke`** object
- Constructors have the form:

```
BasicStroke(float width);
BasicStroke(float width, int cap, int join,
            float miterlimit,
            float[] dash, float dash_phase);
```

- Can control line width, line cap style, line join style, and dashing pattern

Example: Setting Color and Stroke

```
public void paintComponent ( Graphics g ) {
  BasicStroke bs;                // Ref to BasicStroke
  Line2D line;                   // Ref to line
  float[] solid = {12.0f,0.0f};     // Solid line style
  float[] dashed = {12.0f,12.0f};   // Dashed line style
  // Cast the graphics object to Graph2D
  Graphics2D g2 = (Graphics2D) g;
  ...
  // Set the Color and BasicStroke
  g2.setColor(Color.red);
  bs = new BasicStroke( 2.0f, BasicStroke.CAP_SQUARE,
                 BasicStroke.JOIN_MITER, 1.0f,
                 solid, 0.0f );
  g2.setStroke(bs);
  // Draw line
  line = new Line2D.Double (10., 10., 360., 360.);
  g2.draw(line);
  // Set the Color and BasicStroke
  g2.setColor(Color.blue);
  bs = new BasicStroke( 4.0f, BasicStroke.CAP_SQUARE,
                 BasicStroke.JOIN_MITER, 1.0f,
                 dashed, 0.0f );
  g2.setStroke(bs);
  // Draw line
  line = new Line2D.Double (10., 300., 360., 10.);
  g2.draw(line);
}
```



Set color

Define stroke

Set stroke

Draw line

The `Rectangle2D` Classes

- There are two classes for creating rectangles: `Rectangle2D.Float` and `Rectangle2D.Double.` The only difference between them is the units of the calling parameters.
- Constructors:

```
Rectangle2D.Double( double x, double y,
                    double w, double h )
Rectangle2D.Float( float x, float y,
                   float w, float h )
```

- These classes create a rectangle with origin (*x*,*y*), with width *w* and height *h*

The `RoundRectangle2D` Classes

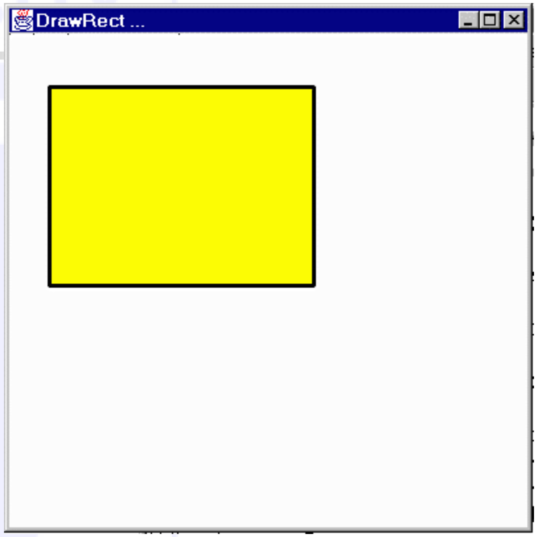
- There are two classes for creating rounded rectangles: `RoundRectangle2D.Float` and `RoundRectangle2D.Double`. The only difference between them is the units of the calling parameters.
- Constructors:

```
 RoundRectangle2D.Double( double x, double y,
    double w, double h,double arcw, double arch )
 RoundRectangle2D.Float( float x, float y,
    float w, float h, float arcw, float arch )
```

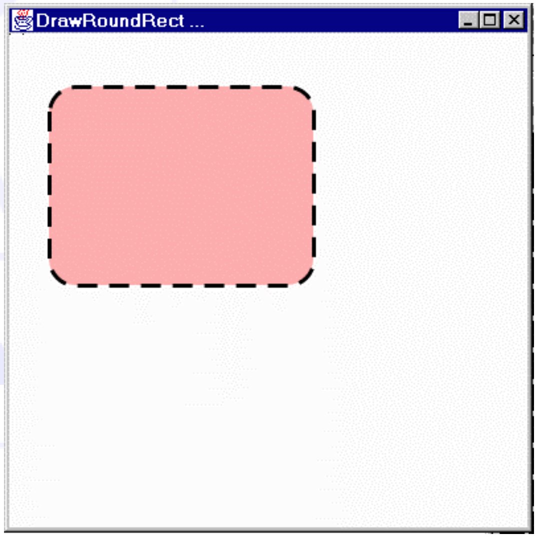
- These classes create a rectangle with origin (*x*,*y*), with width *w*, height *h*, arc width *arcw*, and arc height *arch*

Example: Creating a Rectangle and a Rounded Rectangle

```
float[] solid = {12.0f,0.0f}; // Solid line style
bs = new BasicStroke( 3.0f, BasicStroke.CAP_SQUARE,
                      BasicStroke.JOIN_MITER, 1.0f,
                      solid, 0.0f );
g2.setStroke(bs);
Rectangle2D rect = new Rectangle2D.Double
            (30., 40., 200., 150.);
g2.setColor(Color.yellow);
g2.fill(rect);
g2.setColor(Color.black);
g2.draw(rect);
```



```
float[] dashed = {12.0f,12.0f};   // Dashed line style
bs = new BasicStroke( 3.0f, BasicStroke.CAP_SQUARE,
                      BasicStroke.JOIN_MITER, 1.0f,
                      dashed, 0.0f );
g2.setStroke(bs);
RoundRectangle2D rect = new RoundRectangle2D.Double
            (30., 40., 200., 150., 40., 40.);
g2.setColor(Color.pink);
g2.fill(rect);
g2.setColor(Color.black);
g2.draw(rect);
```



The `Ellipse2D` Classes

- There are two classes for creating circles and ellipses: `Ellipse2D.Float` and `Ellipse2D.Double`. The only difference between them is the units of the calling parameters.
- Constructors:

```
Ellipse2D.Double( double x, double y,
                  double w, double h);
Ellipse2D.Float( float x, float y,
                 float w, float h);
```

- These classes create the ellipse that fits in a rectangular box with origin (*x*,*y*), with width *w* and height *h*
- Example: creating an ellipse

```
Ellipse2D rect = new Ellipse2D.Double
                (30., 40., 200., 150.);
g2.setColor(Color.black);
g2.fill(rect);
```

The `Arc2D` Classes

- There are two classes for creating arcs: **`Arc2D.Float`** and **`Arc2D.Double`**.
- Constructors:

```
Arc2D.Double( double x, double y, double w, double h,
              double start, double extent, int type );
Arc2D.Float( float x, float y, float w, float h,
             float start, float extent, int type );
```

  - These classes create an arc that fits in a rectangular box with origin (*x*,*y*), with width *w* and height *h*. The arc starts at *start* degrees and extends for *extent* degrees.
  - The type of arc is **`Arc2D.OPEN`**, **`Arc2D.CHORD`**, or **`Arc2D.PIE`**

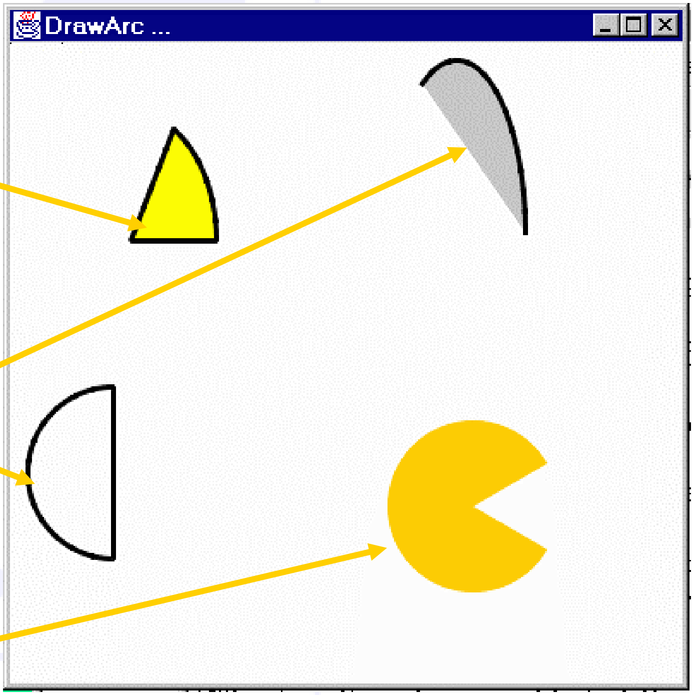
Example: Creating Arcs

```
// Define arc1
 Arc2D arc = new Arc2D.Double (20., 40., 100., 150.,
             0., 60., Arc2D.PIE);
 g2.setColor(Color.yellow);
 g2.fill(arc);   g2.setColor(Color.black);   g2.draw(arc);

 // Define arc2
 arc = new Arc2D.Double (10., 200., 100., 100.,
             90., 180., Arc2D.CHORD);
 g2.setColor(Color.black);   g2.draw(arc);

 // Define arc3
 arc = new Arc2D.Double (220., 10., 80., 200.,
             0., 120., Arc2D.OPEN);
 g2.setColor(Color.lightGray);
 g2.fill(arc);   g2.setColor(Color.black);   g2.draw(arc);

 // Define arc4
 arc = new Arc2D.Double (220., 220., 100., 100.,
            -30., -300., Arc2D.PIE);
 g2.setColor(Color.orange);   g2.fill(arc);
```



Note: Do not put multiple statements on one line!

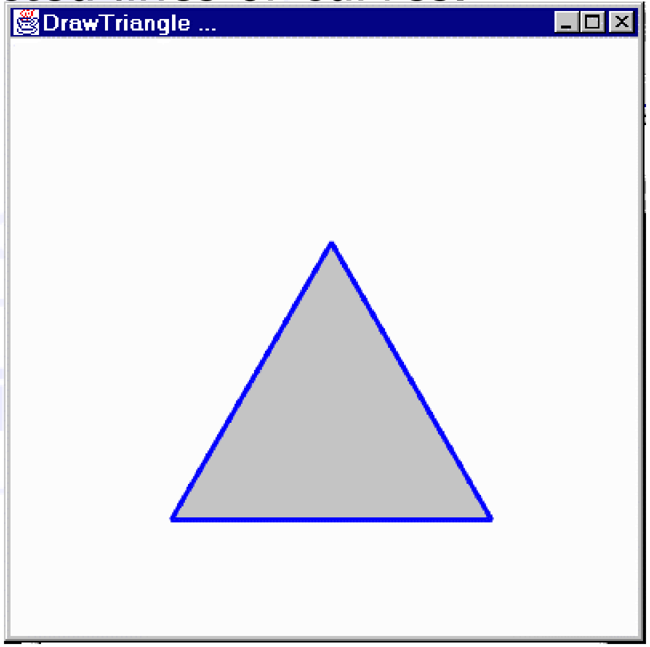
The `GeneralPath` Class

- Allows the construction of arbitrary shapes.
- Constructor: `GeneralPath();`
- Selected Methods (see Java docs for more):

```
moveTo(float x, float y);    // Move to (x,y) w/o line
lineTo(float x, float y);    // Draw line to (x,y)
quadTo(float x1, float y1, float x2, float y2);// Draw curve
closePath();                 // Close shape
```

- Creates a general shape as a series of connected lines or curves.
- Example:

```
GeneralPath p = new GeneralPath();
p.moveTo(100.0f,300.0f);
p.lineTo(300.0f,300.0f);
p.lineTo(200.0f,127.0f);
p.closePath();
g2.setColor( Color.lightGray );
g2.fill(p);
g2.setColor( Color.blue );
g2.draw(p);
```



Displaying Text

- Text is displayed with the `Graphics2D` method `drawString`. Forms:

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

- These methods write `String s` on the component. The point (*x, y*) specifies the <u>*lower-left hand corner*</u> of the text box within the component.
  - *Note that this differs from the convention for other 2D graphics objects, where (x, y) is the upper-left hand corner!*
- Example:

```
g2.setColor( Color.black );
g2.drawString("This is a test!",20,40);
```

Setting Fonts

- Fonts are created with the `java.awt.Font` class
- Constructor:

```
Font( String s, int style, int size )
```

  - `s` is the name for the font to use.
  - `style` is the style (`Font.PLAIN`, `Font.BOLD`, `Font.ITALIC`, or a combination)
  - `size` is the font size in points
- Any font on the system may be used, but certain fonts are guaranteed to be present on any system

Standard Font Names

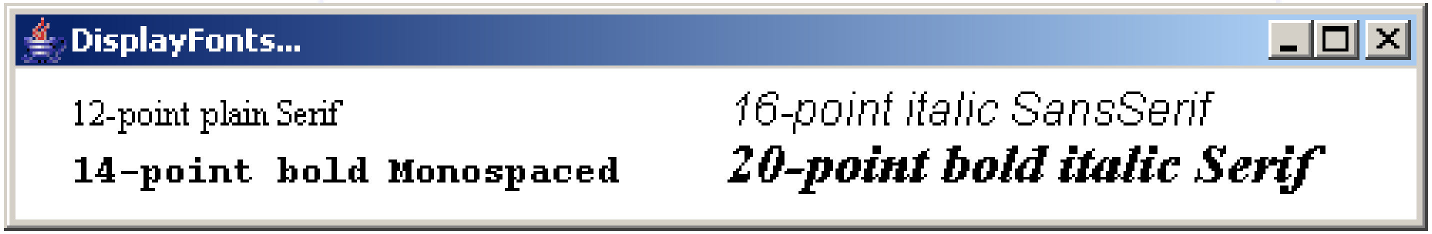
- The following standard fonts are present on any Java implementation:

Font Name	Description
`Serif`	Standard serif font for a particular system. Examples: Times and Times New Roman.
`SansSerif`	Standard sansserif font for a particular system. Examples: Helvetica and Arial.
`Monospaced`	Standard monospaced font for a particular system. Examples: Courier and Courier New.
`Dialog`	Standard font for *dialog boxes* on a particular system.
`DialogInput`	Standard font for *dialog inputs* on a particular system.

Example: Defining Fonts

```
Font f1 = new Font("Serif",Font.PLAIN,12);
Font f2 = new Font("SansSerif",Font.ITALIC,16);
Font f3 = new Font("Monospaced",Font.BOLD,14);
Font f4 = new Font("Serif",Font.BOLD+Font.ITALIC,20);
// Display fonts
g2.setColor( Color.black );
g2.setFont(f1);
g2.drawString("12-point plain Serif",20,40);
g2.setFont(f2);
g2.drawString("16-point italic SansSerif",20,80);
g2.setFont(f3);
g2.drawString("14-point bold Monospaced",20,120);
g2.setFont(f4);
g2.drawString("20-point bold italic Serif",20,160);
```

(DefineFontsDemo)



```
12-point plain Serif            16-point italic SansSerif
14-point bold Monospaced        20-point bold italic Serif
```

Getting Information About Fonts

- Class `java.awt.FontMetrics` can be used to get information about a font
- Constructor:

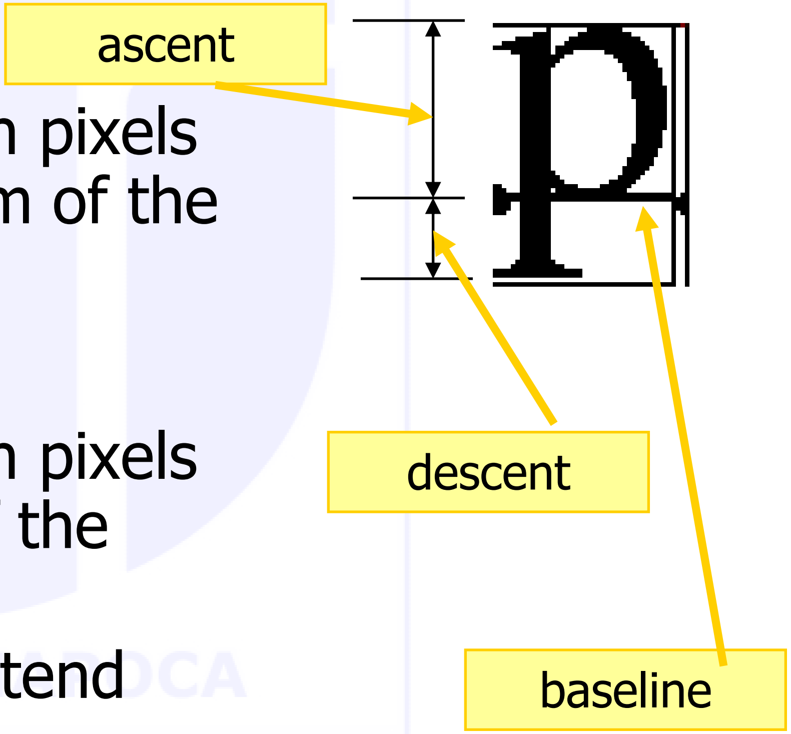
```
FontMetrics fm = new FontMetrics( Font f );
FontMetrics fm = g2.getFontMetrics();
```

- Methods:

Method Name	Description
`public int getAscent()`	Returns the ascent of a font in pixels.
`public int getDescent()`	Returns the descent of a font in pixels.
`public int getHeight()`	Returns the height of a font in pixels.
`public int getLeading()`	Returns the leading of a font in pixels.

Some Font Terminology

- Ascent:
  - Defines the nominal distance in pixels from the baseline to the bottom of the previous line of text.
- Descent:
  - Defines the nominal distance in pixels from the baseline to the top of the next line of text.
- Some font glyphs may actually extend beyond the font ascent/descent.



Some Font Terminology

- Leading, or interline spacing:is the logical amount of space to be reserved between the descent of one line of text and the ascent of the next line.
  - The height metric is calculated to include this extra space.
- Height:
  - distance between the baseline of adjacent lines of text.
  - Sum of the leading + ascent + descent.

The Affine Transform

- The *affine transform* is a transform that shifts, scales, rotates, or skews a shape while maintaining parallel lines.
- Constructor:

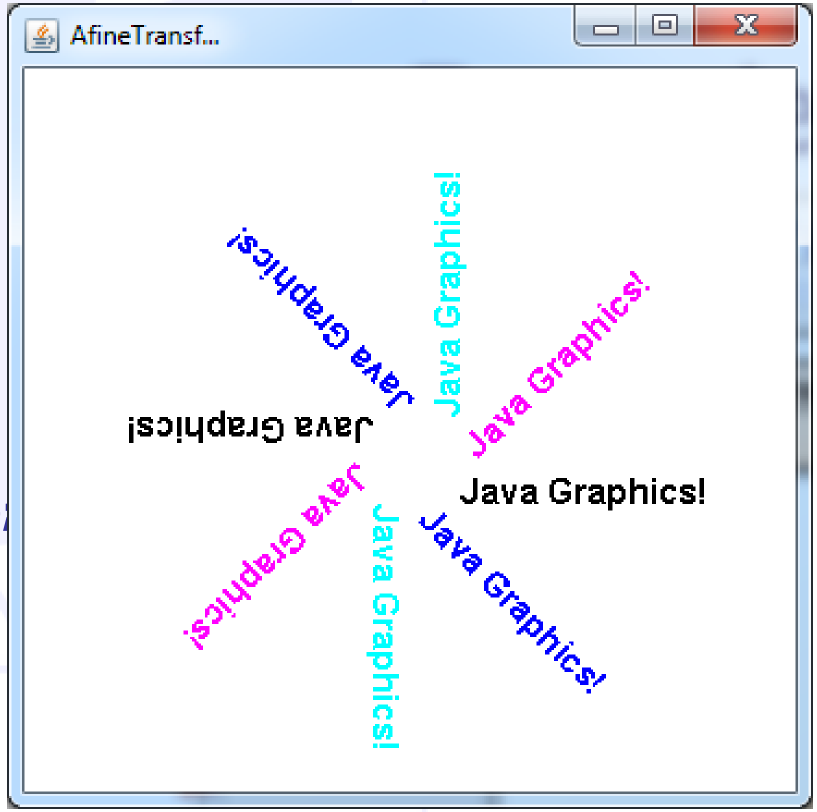
```
AffineTransform at = new AffineTransform();
```

- Methods (all are public void):

Method Name	Description
`rotate(double theta)`	Rotates data by theta radians. A positive angle corresponds to a *clockwise* rotation.
`rotate(double theta, double x, double y)`	Rotates data by theta radians about point *(x, y)*. A positive angle corresponds to a *clockwise* rotation.
`scale(double sx, double sy)`	Scales (multiplies) *x*- and *y*-axes by the specified amounts.
`void shear(double shx, double shy)`	Shears *x*- and *y*-axes by the specified amounts.
`translate(double tx, double ty)`	Concatenates this transform with a translation transformation

Example: Using Affine Transforms to Rotate Text

```
public void paintComponent ( Graphics g )
 {
   super.paintComponent(g);
   // Cast the graphics object to Graphics2D
   Graphics2D g2 = (Graphics2D) g;
   // Get the affine transform
   AffineTransform at = new AffineTransform();
   Color colorArray[] = new Color[] {
     Color.blue, Color.cyan, Color.magenta,
     Color.black, Color.blue, Color.cyan,
     Color.magenta, Color.black };
   g2.setFont(new Font("SansSerif",Font.BOLD,16));
   for ( int i = 0;  i < 8; i++)
   {
     at.rotate(Math.PI/4, 180, 180);
     g2.setTransform(at);
     g2.setColor(colorArray[i]);
     g2.drawString("Java Graphics!", 200, 200);
   }
   super.setBackground( Color.white );
 }
```



(AfineTransfDemo)

XOR Mode

- Normally, when two graphical objects overlay each other, the one on the bottom is hidden by the one lying over it.
- The `Graphics2D` method `setXORMode` overrides this behavior, so that the region of overlap appears in a different color.
- Method call:

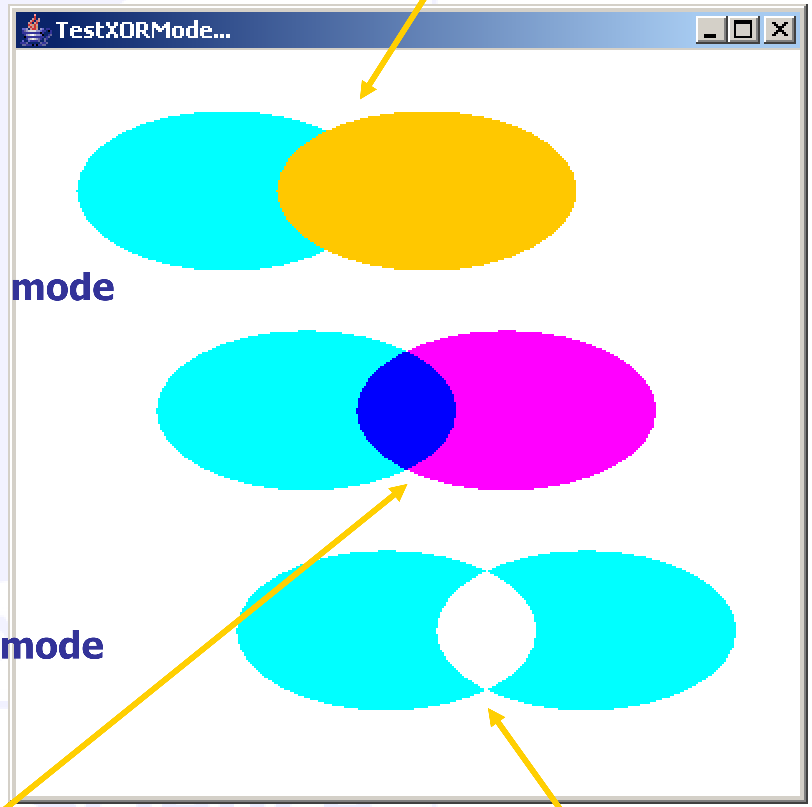
```
g2.setXORMode ( Color c );
```

where `c` is the color for the overlap region *if the two objects are of the same color*. Otherwise, `c` is ignored.

Example: XOR Mode

```
// Two ellipses plotted in normal mode
 ell1 = new Ellipse2D.Double (30., 30., 150., 80.);
 ell2 = new Ellipse2D.Double (130., 30., 150., 80.);
 g2.setColor(Color.cyan);
 g2.fill(ell1);
 g2.setColor(Color.orange);
 g2.fill(ell2);
 // Two ellipses with different colors plotted in XOR mode
 ell1 = new Ellipse2D.Double (70., 140., 150., 80.);
 ell2 = new Ellipse2D.Double (170., 140., 150., 80.);
 g2.setXORMode(Color.white);
 g2.setColor(Color.cyan);
 g2.fill(ell1);
 g2.setColor(Color.magenta);
 g2.fill(ell2);
 // Two ellipses with the same color plotted in XOR mode
 ell1 = new Ellipse2D.Double (110., 250., 150., 80.);
 ell2 = new Ellipse2D.Double (210., 250., 150., 80.);
 g2.setXORMode(Color.white);
 g2.setColor(Color.cyan);
 g2.fill(ell1);
 g2.setColor(Color.cyan);
 g2.fill(ell2);
```

(TestXORMode)



Normal mode: overlapping

XOR Mode w/different colors

XOR Mode w/same colors

Reading

- Barnes: no such topic
- Deitel: chapter 14
- Eckel: chapter 23

Summary

- Events, sources, event listeners
- Applications with buttons
- Text input processing
- Mouse events

- Graphics:
  - Components, containers
  - Graphics on a component
  - Graphics coordinate system

    Drawing:
  - lines, rectangles, ellipses, arcs, general lines
  - text
  - Controlling color and style
  - Fonts: setting, getting info
  - Affine transform
  - XOR mode