Graphics in Swing

Paint, update and repaint in Swing

- Each swing component has method
  - void paintComponent(Graphics g)
  - void paint(Graphics g)
  - void repaint()
  - paintComponent(Graphics g) is a hook method.
    - May be overridden when extending JComponent to change the pixels on the screen
    - Called from paint when part or all of the screen devoted to a Component needs redrawing.
  - paint(Graphics g)
    - See next page.
    - Don't override and don't call!
  - repaint()
    - Schedules a later call to paint.

Painting in Swing

- In Swing all components should extend JComponent, which extends Container
- In JComponent, the paint(Graphics) method follows (roughly) the following algorithm:

```
public void paint( Graphics g ) {
    paintComponent( g ) ;
    paintBorder( g ) ;
    paintChildren( g ) ; }
```

- And so the children appear to be in front of their parent.
- Thou shalt not override the paint method (in Swing)
- The same Graphics object is reused!

Example: A Red Oval.

```
public class Example extends JPanel {
    @Override public void
  paintComponent(Graphics g ){
        super.paintComponent(g) ;
        Color color = g.getColor() ;
        g.setColor( Color.red ) ;
        g.fillOval(
            0, 0, getWidth(), getHeight() ) ;
        g.setColor( color ) ; } }
```

- Thou shall leave the Graphics state as thou found it!

The Graphics class

- Abstract class (Adapter Pattern)
  - Provides a uniform interface to multiple raster devices
    - To the screen / the printer / images in memory
  - Typically we just use the Graphics object passed to paint.
  - Note that *paint* is thus "polymorphic" it operates on any underlying object that realizes the Graphics interface.
  - However we can also create Graphics objects
    - Component.getGraphics()
    - Image.getGraphics()
    - PrintJob.getGraphics()
    - Graphics.create() // A clone.
    - Graphics.create(x, y, w, h) // a translated clone
  - Always call Graphics.dispose() on any Graphics object you create!

Graphics

- Graphics objects provide two kinds of methods
  - Methods that access/mutate the state of the Graphics object:
    - setColor(Color)
    - setClip(Shape)
    - setFont(Font)
    - setPaintMode() / setXORMode()
    - hitClip(x, y, w, h)
  - Methods that affect the underlying raster:
    - drawRect(x, y, w, h) / fillRect(x, y, w, h)
    - drawOval(x, y, w, h) / fillOval(x, y, w, h)
    - drawLine(x0, y0, x1, y1)
    - drawString(x, y, string)
    - drawImage(image, x, y, observer)

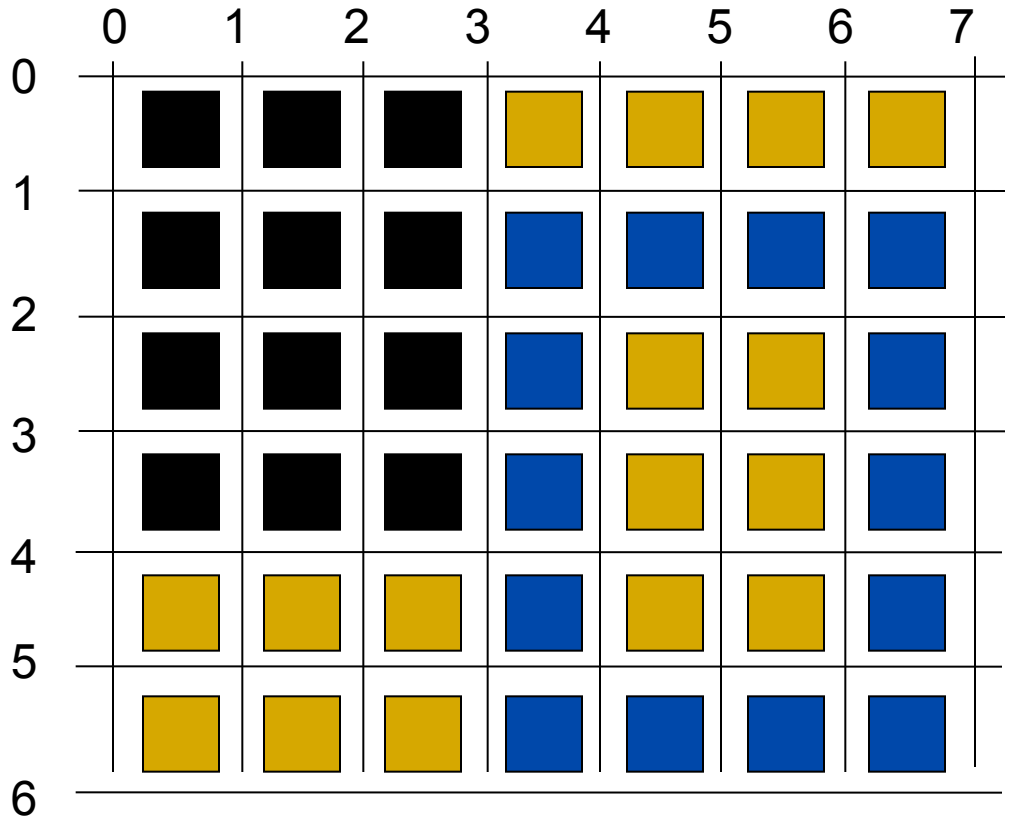
Coordinate systems

All coordinates are in terms of pixels and are ints.

- (x,y) is the point *between* four pixels:
  - (x-1,y-1), (x-1,y), (x,y-1), (x,y)
- We can obtain the width and height of the component by
    - Component.getWidth() and
    - Component.getHeight()
  - (0,0) is the top left of corner
  - (getWidth(), 0) is the top right corner
  - (0, getHeight()) is the lower left corner

Screen Coordinates

- The effect of
  g.fillRect(0, 0, 3, 4)
- and
  g.drawRect(3, 1, 3, 4)



Coordinate transformations

- The application generally should not deal with pixel based coordinates
- Thus we should translate from model ("world") coordinates to "view" coordinates.
- Example. OTHELLO
  - World coordinates are (row, col) where
    - row and col are floating point numbers.
    - (0.0,0.0) is the top left-corner of the board (not the component), (0.0, COLS) is the top right corner of the board (note reversal of coordinates.

Example (cont.)

A handy routine (in BasicView from othello-0 demo)

```
private Point worldToView( double row, double col ) {
    double margin = 2.0 ;
    int height = getHeight() ;
    double vertStretch = height / (ROWS+2*margin) ;
    int width = getWidth() ;
    double horStretch = width / (COLS+2*margin) ;
    int y = (int)( (margin+row)*vertStretch ) ;
    int x = (int)( (margin+col)*horStretch ) ;
    return new Point( x, y ) ; }
```

Example (cont)

- From the overridden paintComponent method of BasicView (g is the Graphics object)

```
/* Draw horizontal lines */ {
        g.setColor(Color.black);
        for( int r=0; r < ROWS+1 ; ++r ) {
            p0 = worldToView( (float)r, 0.0 ) ;
            p1 = worldToView( (float)r, (float)COLS ) ;
            g.drawLine(p0.x, p0.y, p1.x, p1.y) ; } }
```

- For purpose of mouse input, it is handy to have a routine viewToWorld as well.

Coordinate transformations

- Isolating coordinate transformation to one spot makes it easy to change.
- We could even use the Strategy Pattern so that one paint method is capable of employing various transformations. (See othello-1 demo).

Text

- To paint text on the screen:

  g.*setFont*(font) ;

  g.*drawString*(x, y, string) ;

  - (*x*,*y*) is the leftmost point of the string's baseline
    - I.e. descenders will go below the line *y*. The rest of each letter is above.
  - The font is an object of class Font e.g.
    - **new** *Font*(*Font*.SANS_SERIF , *Font.PLAIN*, 10) ;
    - For portability, first argument should be in {Font.DIALOG, Font.DIALOG_INPUT, Font.MONOSPACED, Font.SERIF, Font.SANS_SERIF}
    - Second should be in {Font.*PLAIN,* Font.*ITALIC,* Font.*BOLD,* Font.*ITALIC*|Font.*BOLD*}

Font Metrics

- Given a string we may want to know how much space it will take.
  - For this we use a FontMetrics object
    - *FontMetrics* fm = g.*getFontMetrics*(font) ;
    - **int** h = fm.*getHeight*() ; // Standard height
    - **int** w = fm.*stringWidth*(message) ;

Scaling a Font.

- We can pass a font through an Affine transformation (affine transformations preserve parallel lines).
  - E.g.to fit a string in a rectangle (p0, p1)

```
int width = p1.x - p0.x ;
int height = p1.y - p0.y ;
// Start with a font
    Font font = new Font(Font.SANS_SERIF, Font.PLAIN, 10 ) ;
// Calculate the height and width of the string in this font
    FontMetrics fm = g.getFontMetrics(font) ;
    int h = fm.getMaxAscent() + fm.getMaxDescent() ;
    int w = fm.stringWidth(message) ;
// Calculate the scale factor to make it fit
    double scaleFactor = Math.min( (double)height / h,
                                   (double)width / w) ;
```

Example (cont)

```
// Create an Affine transform that will scale the font.
    AffineTransform xform = new AffineTransform() ;
    xform.scale( scaleFactor, scaleFactor);
 // Create a new font
    font = font.deriveFont(xform) ;
 // Centre the message left to right.
    fm = g.getFontMetrics(font) ;
    int messWidth = fm.stringWidth(message) ;
    int x = p0.x+ (width-messWidth)/2 ;
    int y = p1.y – fm.getMaxDescent();
// Put it on the screen
    g.setFont( font ); g.setColor( Color.black ) ;
    g.drawString(message, x, y ) ;
```

Drawing Images

- *Image* is a class representing images.
- Images are often read in from files (gif and jpeg are supported) -- either local or remote

```
URL url = new URL(str) ; // if on web.
// or
URL url = file.toURI().toURL() ; // if in file.

ImageIcon icon = new ImageIcon( url );
graphics.drawImage(icon.getImage(), x, y, null);
```

Loading images from a .jar file

- It's convenient to obtain image files from the same directory where your .class files are stored.
- If your application is packaged as a .jar file, this can be inside the .jar file.

```
URL url = this.getClass().getResource("logo.gif");
ImageIcon icon = new ImageIcon(url) ;
drawImage( icon.getImage(), x, y, null ) ;
```

Graphics2D

- Graphics2D extends Graphics and adds new interface.
- As long as your runtime environment (JRE) is at least version 1.2, the Graphics object passed to paint (because of a call to repaint()) can be trusted to actually be an instance of Graphics2D.

```
@Override protected void paintComponent( Graphics g ) {
    super.paintComponent( g ) ;
    // Paint the rest
    assert g instanceof Graphics2D,
       "Graphics2D is required by "+this.getClass();
    Graphics2D g2d = (Graphics2D) g ;
    …and from here on we use g2d instead of g…
}
```

Graphics2D

- Graphics2D objects work in terms of floating point not integers.
- Graphics2D objects can be transformed

```
AffineTransform xform ;
…
g2d.transform( xform ) ;
```

- Graphics2D objects can have Shape objects drawn on them

```
Shape shape ;
…
g2d.draw( shape ) ;
```

Shapes

- Shape is an interface with lots of implementations

```
Point2D p0 = new Point2D.Double(x, y);
Point2D p1 = new Point2D.Double(x+w, y+h) ;
Shape l = new Line2D.Double( p0, p1 ) ;
Shape r = new Rectangle2D.Double(
                    x, y, w, h) ;
Shape e = new Ellipse2D.Double(
                    x, y, w, h) ;
```

- All these shapes can be drawn on a Graphics2D object

Shapes

- All these shapes can be drawn on a Graphics2D object

```
g2d.fill( e ) ;
g2d.draw( r ) ;
```