CS416

Introduction to Computer Science II

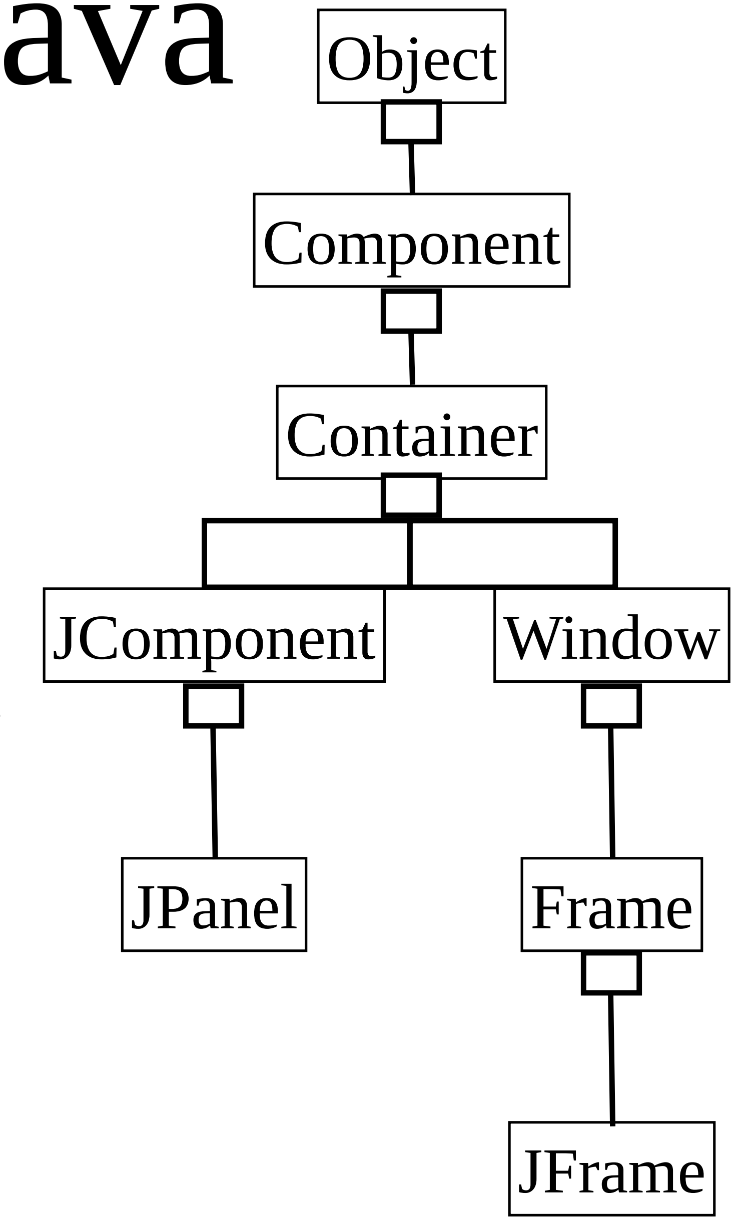
Spring 2018

3 JComponents and JWheels

- Previously, in 416
  - AWT and Swing basics
  - *wheels*-like wrappers to access AWT classes
- Preview
  - Swing JComponents
  - JWheels: wheels-like interface based on JComponent

Graphical container objects in Java

- *JPanel* is not the only class into which you can draw.
- In fact, you can draw in any class below *Component*
- *JComponent* is particularly useful for drawing small independent objects that you want to be able to click and drag



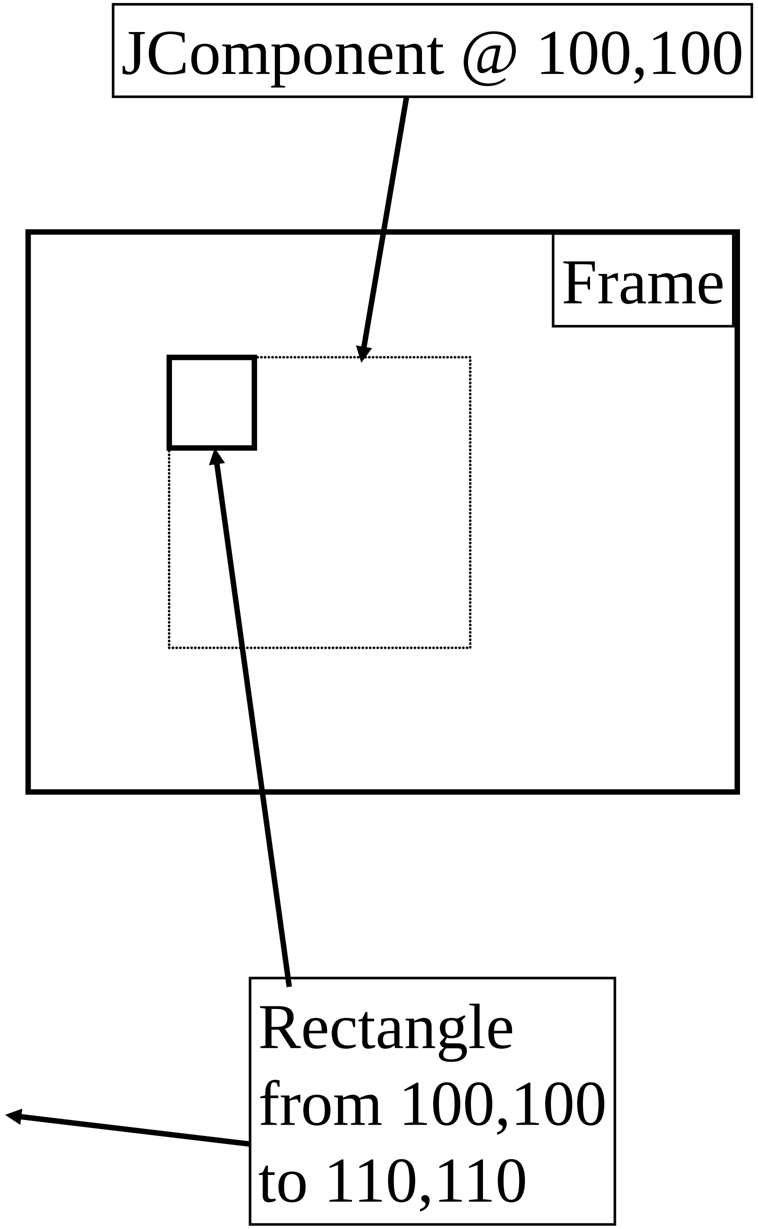
JComponent

- Most of the functionality of *JComponent* is aimed at handling <u>multiple</u> graphical objects.
- For now we want one *JComponent* to represent <u>each</u> graphical entity: ellipse, rectangle, line, etc.
- *JComponents* are convenient because
  - they get re-painted automatically (almost as magically as *wheels* objects).
  - it’s easy to get interaction events from them (we’ll do that next week)
- On the other hand, they have lots more overhead than the simpler AWT graphical objects

JComponent features

- Rectangular region you can paint in
  - Location of region is in coordinates of the frame it is in.
  - Size of rectangle is the bounds for drawing
- All drawing is relative to the *JComponent's* location
  - e.g., you can draw a *10* x *10* rectangle at (100,100) in a frame, by drawing in a *JComponent* located at (100,100) using:

```
brush.drawRect(0,0,10,10);
```



JComponent @ 100,100

Frame

Rectangle
from 100,100
to 110,110

JComponent painting

- The full Swing painting model is pretty complex. For now, we’ll settle for the simplification:
  - The *paintComponent* method of all components with non-zero area is called whenever a portion of its area may have been corrupted, or in response to a *repaint()* method call.
  - The order of invocation (by default) is the **reverse** order in which the components were added to the frame.

What do you see?

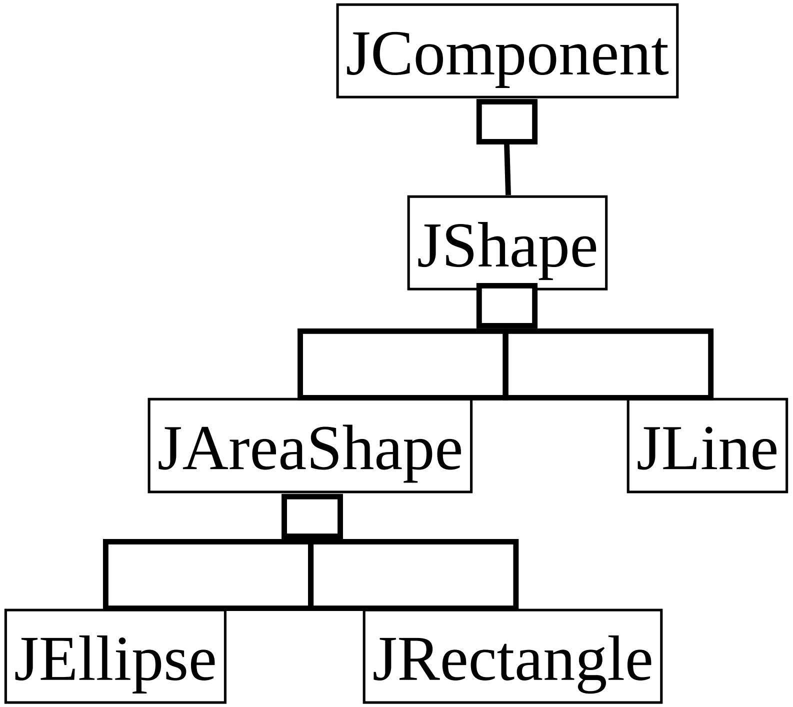
- This is 2D graphics; objects are painted *on top of* previously drawn objects. When objects overlap, the one drawn last is visible.
- *paintComponent* invocation order is key
  - painted in **reverse** order of how added to the frame

```
// JEllipse and JLine are JComponents.
// panel is a Container that holds them
JEllipse e1 = new JEllipse( ... );
panel.add( e1 );

JEllipse e2 = new JEllipse( ... );
panel.add( e2 );

JEllipse e3 = new JEllipse( ... );
panel.add( e3 );

JLine    line1 = new JLine( ... );
panel.add( line1 );
```

The painting order of the code above is:
line1, e3, e2, e1.
This is the opposite of *wheels*, where the objects are displayed in the order you create them, so the object created first appear to be "furthest" away.
In Swing, the objects you want to appear to be **closest** should be added to the frame **first**!

JWheels

- Another *wheels*-like interface based on Swing
- Graphical objects inherit from *JComponent*
- *JShape* encapsulates location, size, line (border) color
- *JAreaShape* adds *filling* and provides lots of shared code between ellipse and rectangle



JEllipse class

```
public class JEllipse extends JAreaShape
{
  // inherits methods to set fill color, border color, location,
  //    size, line width, etc.
  // only code here is in constructors and paintComponent

  public JEllipse( Color aColor )
  {
    super( aColor );
  }

  public void paintComponent( Graphics g )
  {
   super.paintComponent( g );
   Graphics2D brush2 = (Graphics2D) brush;
   int w = getWidth();
   int h = getHeight();
   brush2.setClip( 0, 0, w + 1, h + 1 );
   brush2.setColor( getFillColor() );
   brush2.fillOval( 0, 0, w, h );
   brush2.setColor( getBorderColor() );
   brush2.drawOval( 0, 0, w, h);
  }
}
```

JAreaShape extends JComponent

JEllipse has a couple constructors; all mostly just call super constructors

JEllipse.paintComponent

```
public class JEllipse extends JAreaShape
{
  . . .
  public void paintComponent( Graphics g )
  {
    super.paintComponent( g );
    Graphics2D g2 = (Graphics2D) g;
    int w = getWidth();
    int h = getHeight();
    int b = getLineWidth(); // border width

    g2.setClip( -b, -b, w + 2*b, h + 2*b );

    g2.setColor( getFillColor() );
    g2.fillOval( 0, 0, w, h );
    g2.setColor( getBorderColor() );
    g2.drawOval( 0, 0, w, h);
  }
}
```

JComponent methods to get w, h

Default clip region is same size as the extents of the *fill* region; but, the *draw* extents (border) are larger in x and y by the width of the line drawn. So, we expand the clipping region.

getFillColor and *getBorderColor* are inherited from *JAreaShape*

draw and *fill* use coordinates in the JComponent, not the frame, so location is always 0,0 -- the component is the size of the ellipse

fillOval and *drawOval* are *Graphics class methods*

JRectangle.paintComponent

```
public class JRectangle extends JAreaShape
{
  . . .
  public void paintComponent( Graphics g )
  {
    super.paintComponent( g );
    Graphics2D g2 = (Graphics2D) g;
    int w = getWidth();
    int h = getHeight();
    int b = getLineWidth(); // border width

    g2.setClip( -b, -b, w + 2*b, h + 2*b );

    g2.setColor( getFillColor() );
    g2.fillRect( 0, 0, w, h );
    g2.setColor( getBorderColor() );
    g2.drawRect( 0, 0, w, h);
  }
}
```

JRectangle is essentially identical to *JEllipse* except for painting method calls.

Note: these *paintComponent* methods do not save and restore the state of the *Graphics* object, as we did before. This is because a new *Graphics* object is created for each *JComponent*. Since the *JWheels* classes use a *JComponent* for each displayable shape, this *Graphics* object is never used again.

fillRect and *drawRect* are *Graphics* methods

JLine class

- *JLine* is a bit more complicated
  - It implements a line location (upper left corner of bounding box of the line)
    - This allows all *JShape* objects to be moved without knowing what kind of *JShape* it is.
  - It computes the bounding box of the line, which becomes the area of the *JComponent*.
  - It computes the coordinates of the line relative to the location of the *JComponent*
  - A key (private) method is *updateComponent()*

JLine.updateComponent()

```
private void updateComponent( )
{
  // _x1, _y1, _x2, _y2: absolute coords of line
  // find bounding box of line,
  int locX   = Math.min( _x1, _x2 );
  int locY   = Math.min( _y1, _y2 );
  int width  = Math.max( _x1, _x2 ) - locX;
  int height = Math.max( _y1, _y2 ) - locY;
  width      = Math.max( width, 1 );
  height     = Math.max( height, 1 );
  super.setLocation( locX, locY );
  super.setSize( width, height );

  // get line coords relative to JComponent
  _drawX1 = _x1 - locX;
  _drawY1 = _y1 - locY;
  _drawX2 = _x2 - locX;
  _drawY2 = _y2 - locY;
}
```

Upper left corner of bounding box is JComponent location

width and height are also computed for the bounding box

need to tell the JComponent its location and size; if we don’t, it has 0 area and *paintComponent* is never called.

need line coordinates relative to the origin for paintComponent.

JLine.paintComponent

```
public class JLine extends JShape
{
  . . .
  public void paintComponent( Graphics g )
  {
    super.paintComponent( g );
    Graphics2D g2 = (Graphics2D) g;
    int w = getWidth();
    int h = getHeight();
    int b = getLineWidth(); // border width

    g2.setClip( -b, -b, w + 2*b, h + 2*b );

    g2.setColor( getColor() );
    g2.setStroke( new BasicStroke( b ));
    g2.drawLine( _drawX1, _drawY1, _drawX2, _drawY2 );
  }
}
```

getLineWidth and *getColor* are inherited from *JShape*

Clip region is extended to include the bounding rectangle plus enough border for the line width

setStroke and *drawLine* are *Graphics* methods

use the coordinates relative to the origin for paintComponent.

Using *JWheels* Objects

- Using *JWheels* is nearly as easy as *wheels*
    - Just create the object and add it to its container (usually a *JPanel* or another *JComponent*)
    - *JRectangle*, *JEllipse*, *et al*. issue *repaint* calls whenever the object changes (location, color, size, line width, etc.)

```
public class DrawPanel extends JPanel
{
  // As before, by convention, the DrawPanel creates objects
  public DrawPanel( ... )
  {
    JRectangle r = new JRectangle( x, y );
    r.setColor( ... ); // and setSize, etc.
    ...
    this.add( r );
  }   // That’s it! no DrawPanel.paintComponent method needed
}
```

Composite *JWheels* Objects

- Composite as *JComponent* seems easiest

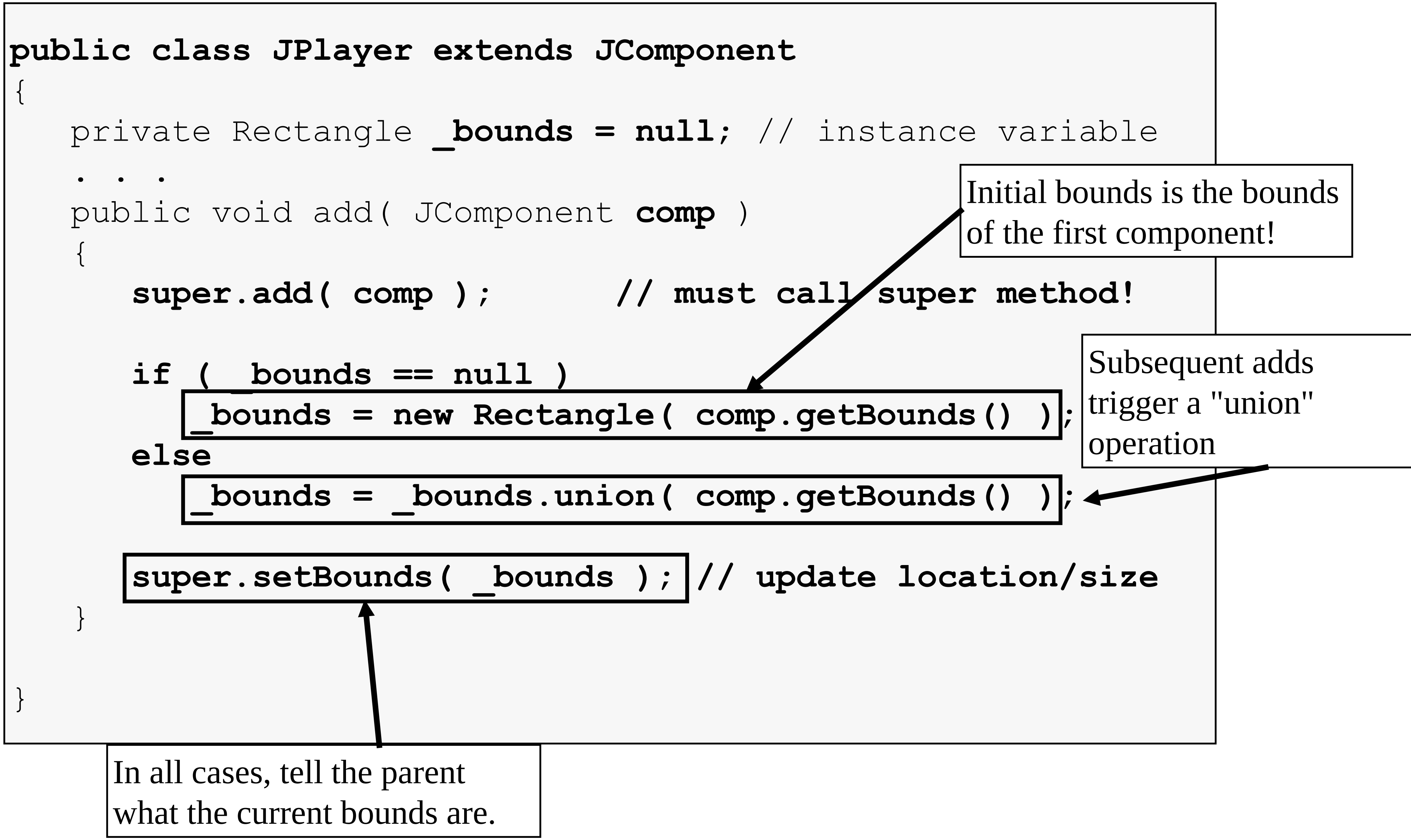
```
public class JPlayer extends JComponent
{
  . . .
  public void JPlayer( . . . )
  {
    JEllipse head = new JEllipse( ... );
    . . .
    this.add( head ); // Add JEllipse to JComponent
    JRectangle body = new JRectangle( ... );
    ...
    this.add( body ); // Add JRectangle to JComponent
  }
}
```

But, we still need to **explicitly** determine the composite **bounds**; if we don’t, it has 0 area and *paintComponent* is never called.

Computing Composite Size

- *add(Component)* is a *Container* method inherited by *JComponent* and, hence, our composite class
    - In general, the semantics for computing a *Container's* size are very complicated and time-dependent!
    - In the specific case of a *JComponent* that contains only *JShape* objects, <u>whose sizes and relative locations don't change</u>, it's tractable.
    - For the *JPlayer* composite, we override the *add* method to recompute the composite's *bounds* whenever a new component is added.

JPlayer.add(JComponent)



```
public class JPlayer extends JComponent
{
   private Rectangle _bounds = null; // instance variable
   . . .
   public void add( JComponent comp )
   {
      super.add( comp );      // must call super method!

      if ( _bounds == null )
         _bounds = new Rectangle( comp.getBounds() );
      else
         _bounds = _bounds.union( comp.getBounds() );

      super.setBounds( _bounds ); // update location/size
   }

}
```

Initial bounds is the bounds of the first component!

Subsequent adds trigger a "union" operation

In all cases, tell the parent what the current bounds are.

Mixing *AWheels* & *JWheels*

- Can use both *A-objects* and *J-objects* at the same time:
  - the *JPanel (or JComponent) paintComponent* method must explicitly "display" each of the *A-objects*.
  - the *J-objects* must be "added" to the *JPanel*; their own *paintComponent* methods will be automatically called.

Review

- *JComponent*
- *JWheels*
- AWT *Graphics*
  - *draw/fillRect*, *draw/fillOval*, *drawLine*
  - *draw/fillPolygon*

Next, in 416

- Animation in AWT/Swing (Ch. 7.4 – 7.6)
  - Swing Timer class
  - Mover interface