Graphics and Painting

Painting and the Graphics Context

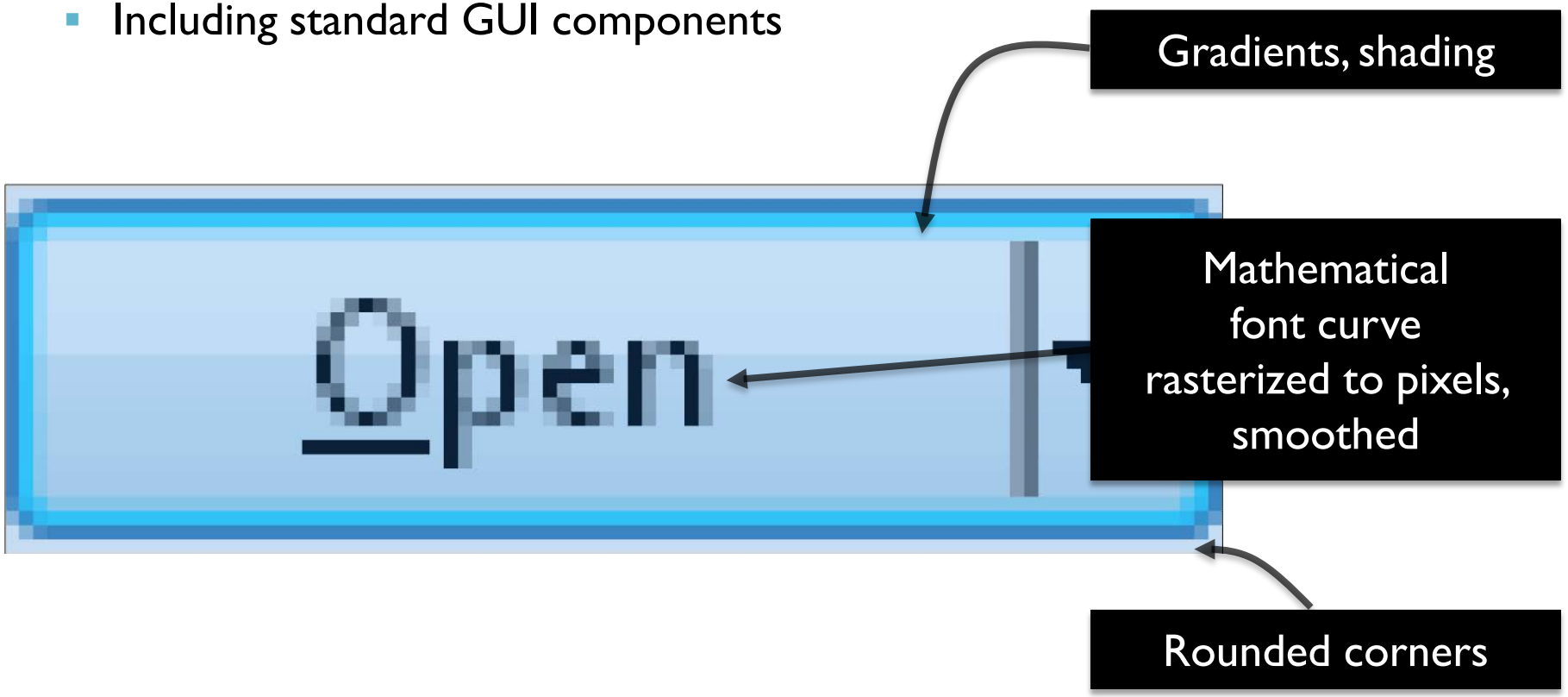
Images in Java

jonas.kvarnstrom@liu.se – 2015

Intro 1: Painting

jonkv@ida

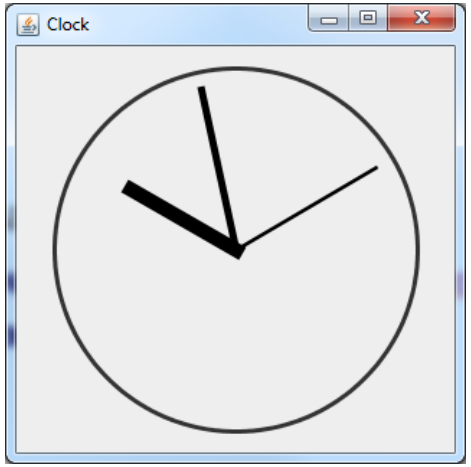
- Everything on the screen is **painted** using **pixels**
  - Including standard GUI components



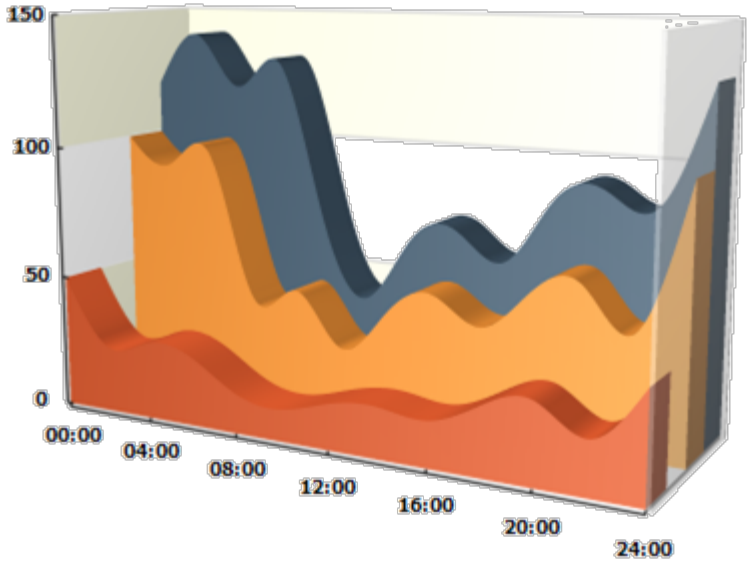
Intro 2: General Custom Components

jonkv@ida

- Painting is needed to implement **custom components**
  - Analog clock
    - **public class** ClockComponent **extends** JComponent { … }
    - Background
    - Circle
    - Three hands



  - Area chart
    - **public class** AreaChartComponent **extends** JComponent { … }

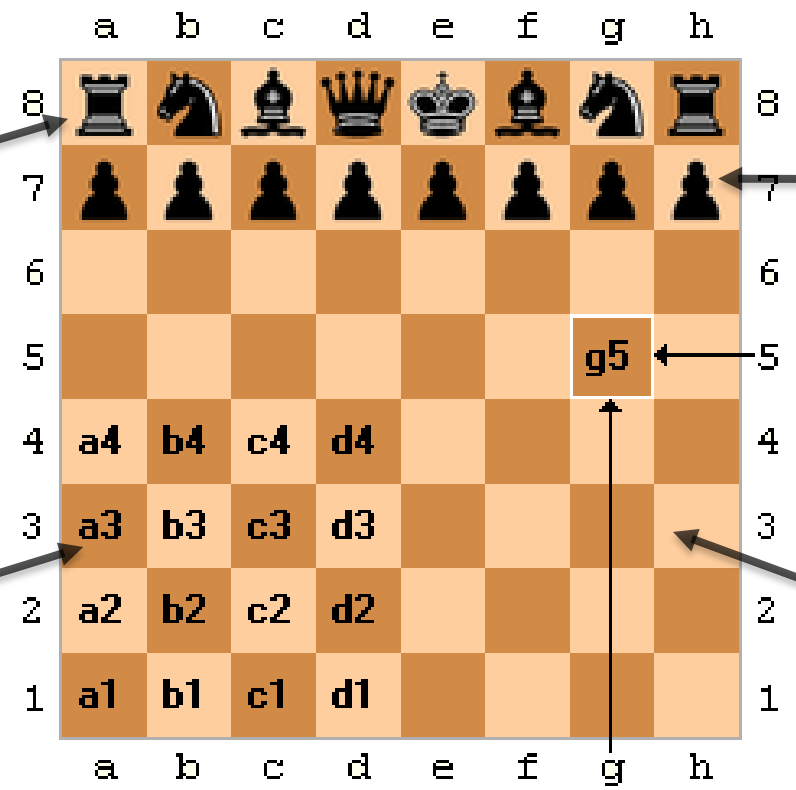


Intro 3: Specific Custom Components

jonkv@ida

- A game GUI *could* consist of multiple custom / built-in components

16 game piece *components*: transparent background



Move pieces by changing components' coordinates

Transparent JLabels

Background checkerboard *component*

Intro 4: Specific Custom Components

jonkv@ida

- A game GUI can also be a **single component**
  - You keep track of all the contents by yourself

- Possibly surrounded by some standard **Swing components**
  - Menus, buttons, …

You cannot paint without a component!

Custom Components and the paintComponent() method

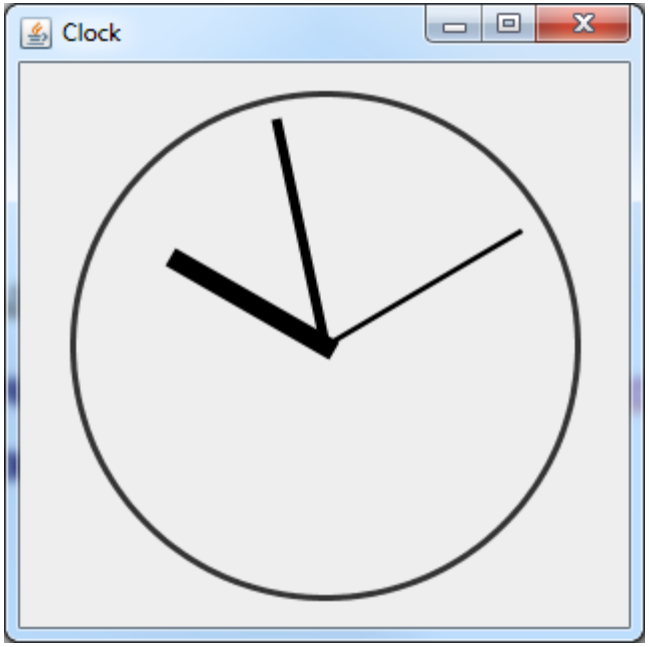
Painting 1: Custom component

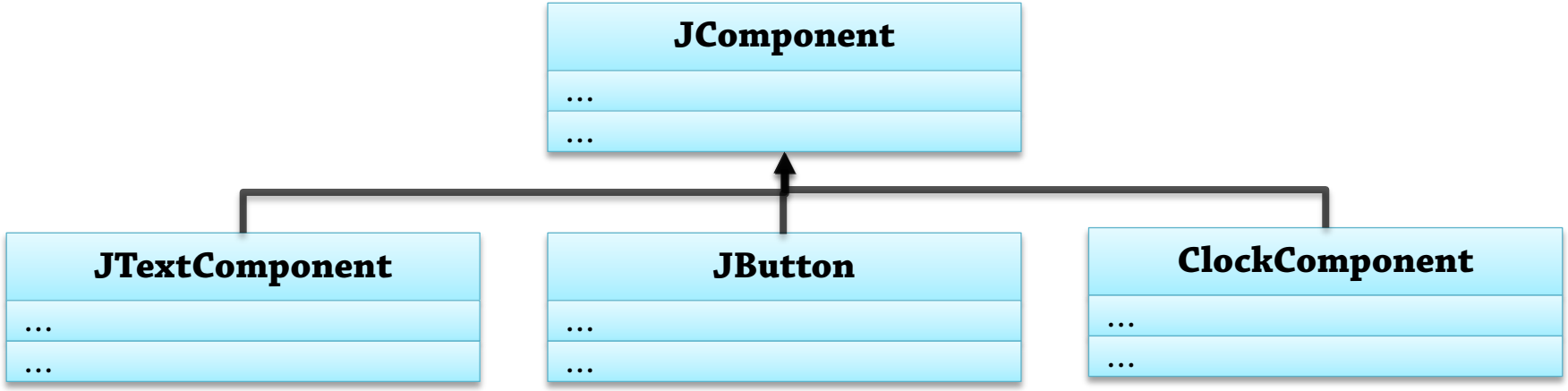
jonkv@ida

- Custom components extend **JComponent**

```
public class ClockComponent extends JComponent {
   // Which time should be shown?
   private LocalTime time;

   // Construct a clock component
   public ClockComponent(LocalTime time) {
      this.time = time;
   }
   …
}
```



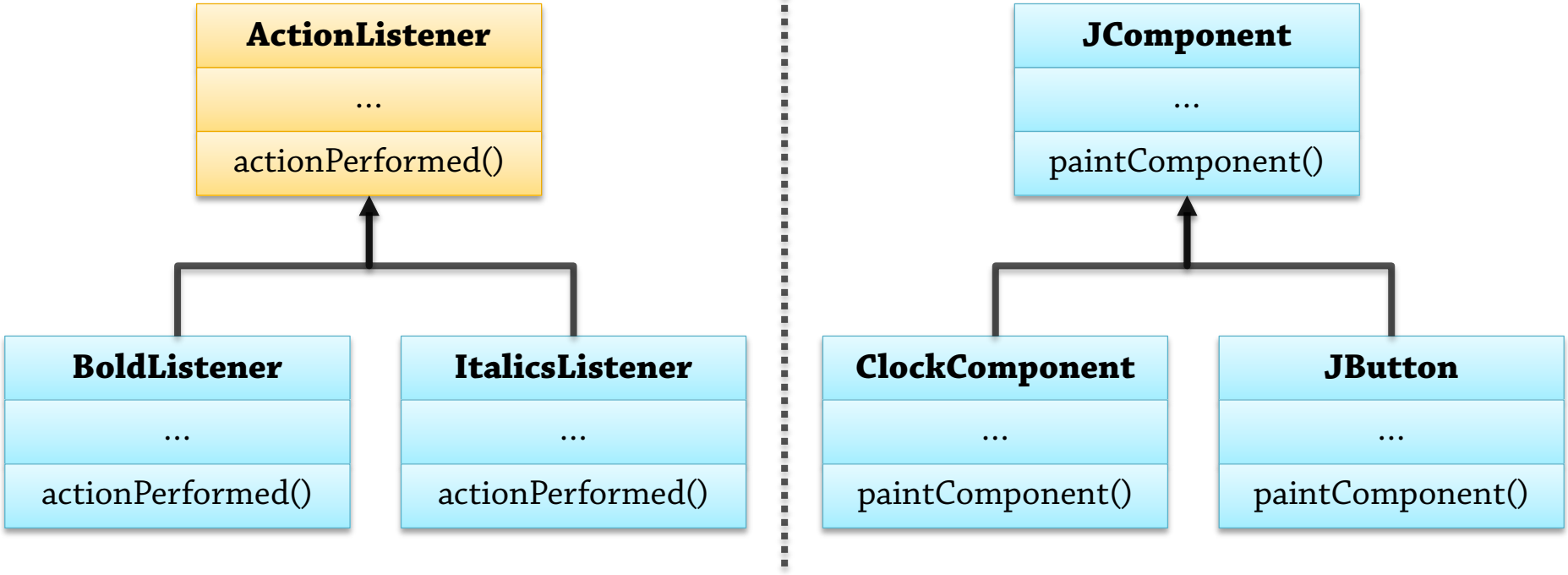


Painting 2: Callback Method

jonkv@ida

How do we *paint* a ClockComponent?

Similar to action listeners:
We implement a callback method
Swing calls the method when necessary – from the event thread



Difference: Only one paint method, implemented in the component class

Painting 3: paintComponent()

jonkv@ida

Swing's event thread **calls paintComponent()** to display the component's contents…

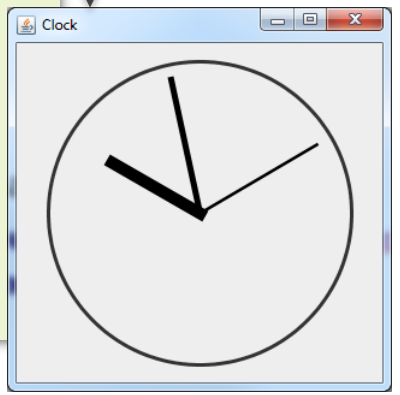
The component defines how it is painted

```
public class ClockComponent extends JComponent {
   private LocalTime time;

   public void paintComponent(Graphics g) {
      // Let JComponent do its job...
      super.paintComponent(g);
      // ...and then we add our own look
      g.drawOval(... coordinates for circle ...);
      g.drawLine(... coordinates for hours ...);
      g.drawLine(... coordinates for minutes ...);
      g.drawLine(... coordinates for seconds ...);
   } // Details to follow!
}
```

We're *overriding* paintComponent() but still want the parent to do *its* painting, if any…

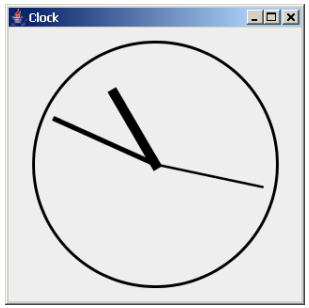
Swing gives us a **Graphics** object that can paint on **this** component



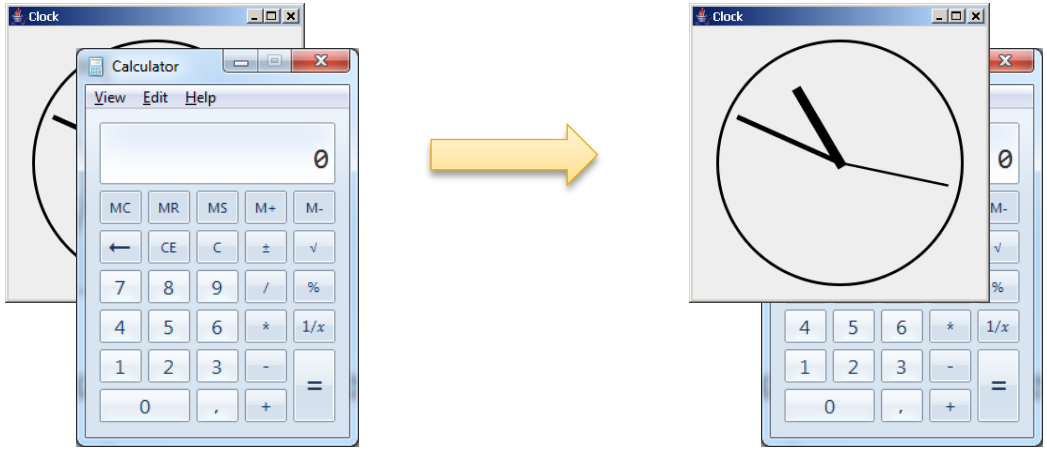
Painting 4: Swing calls paintComponent()

jonkv@ida

- So Swing (not you!) calls **paintComponent**():
  - The **first time** the window is shown



  - When the component **reappears** after having been (partially) covered



Painting 5: We call repaint()

jonkv@ida

- If **we** want to **update** the component:
  - We **ask** Swing to call **paintComponent**() for us

```
public class TextLabel extends JComponent {
   public void setText(String text) {
      this.text = text;
      this.repaint();
   }
}
```

My label text changed ➔
ask Swing to repaint me

```
public class GameMechanics {
   private GameComponent gameComponent;
   private void stepGameForward() {
      movePlayers();
      checkCollisions();
      gameComponent.repaint();
   }
```

Game clock ticks ➔
ask Swing to repaint
the game component

Queues a repaint *request* – handled when there's time

Painting 6: How Large Am I?

jonkv@ida

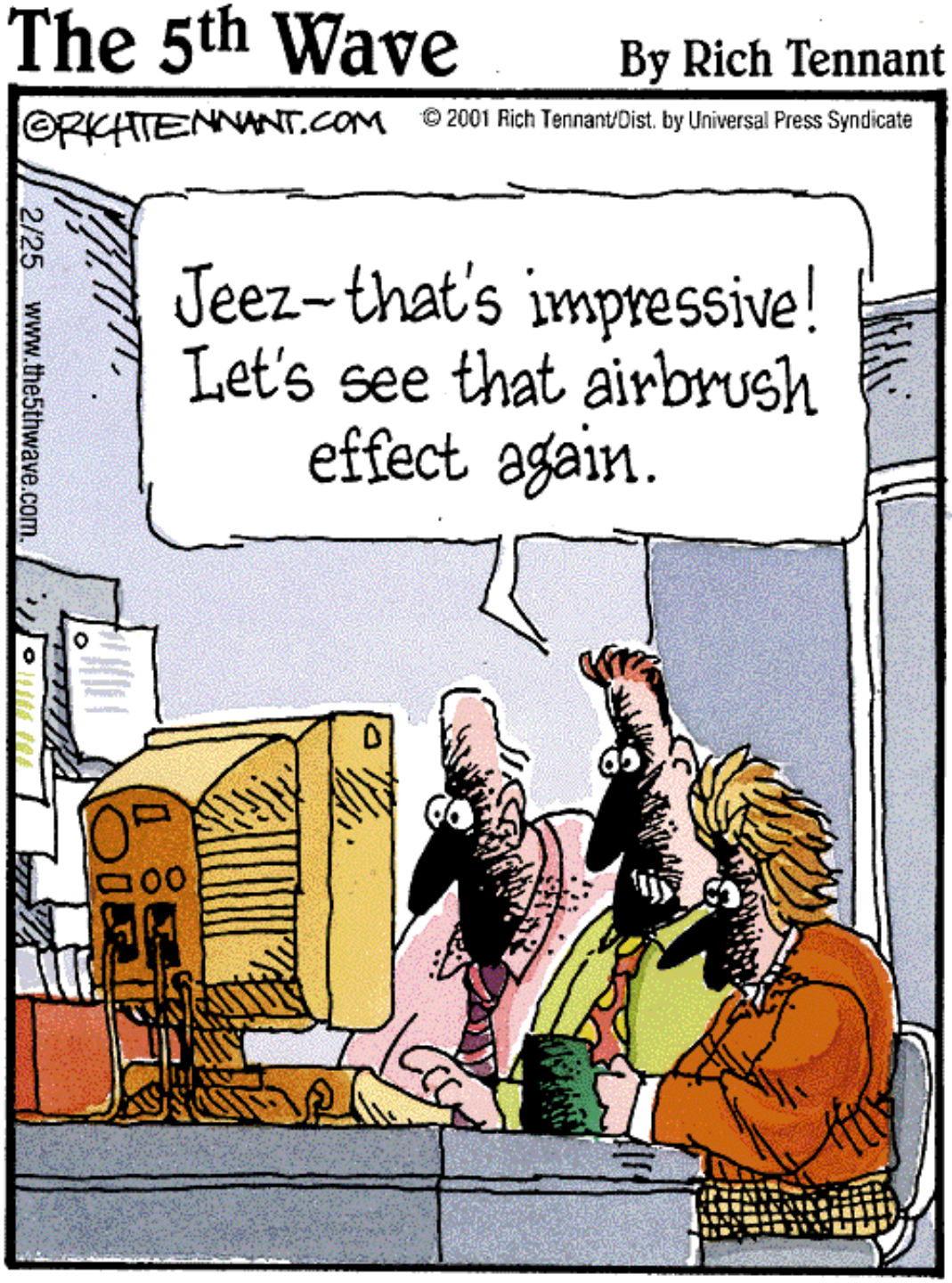
- A component may be sized by someone else
  - But can always find out its current size

```
public class ClockComponent extends JComponent {
   private LocalTime time;

   public void paintComponent(Graphics g) {
      int myWidth = this.getWidth();
      int myHeight = this.getHeight();
      …
   }
}
```

Basic Painting:

The Graphics Context

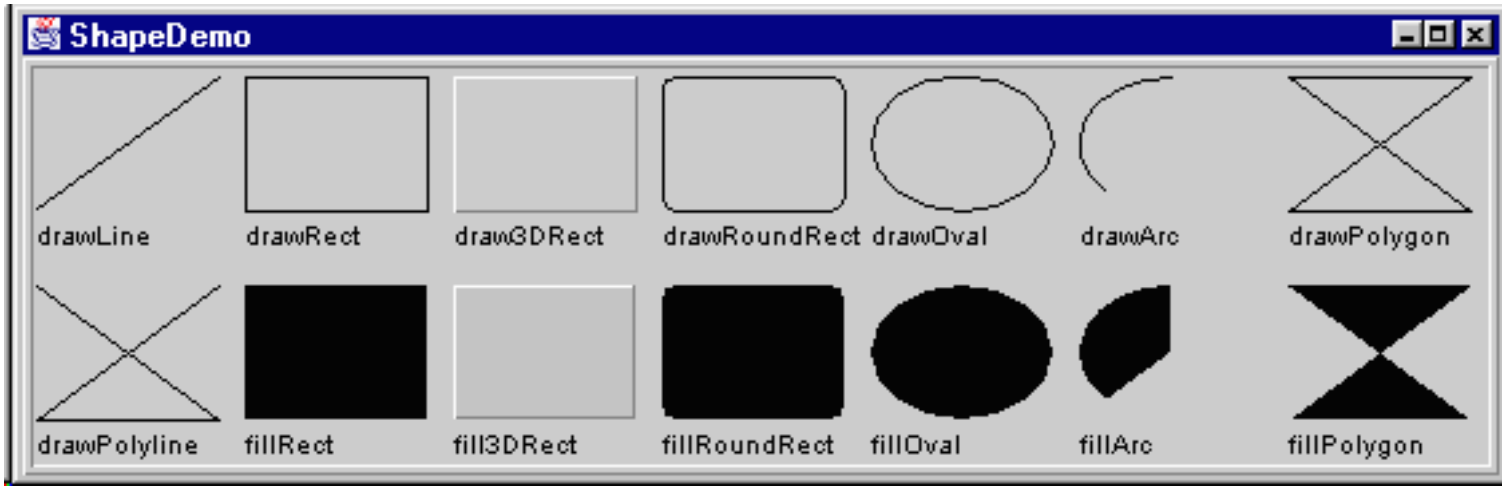


Basic Graphics 1: java.awt.Graphics

jonkv@ida

- **Graphics**: Basic **painting**
  - Limited repertoire, one method per shape

Graphics



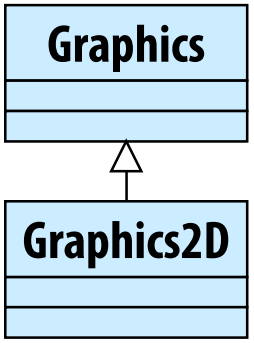

```
public class ClockComponent extends JComponent {
   public void paintComponent(Graphics g) {
      ...
      g.drawOval(... coordinates for circle ...);
      g.drawLine(... coordinates for hours ...);
   }
}
```

Basic Graphics 2: java.awt.Graphics2D

jonkv@ida

- Newer subclass **Graphics2D**: **arbitrary shapes**
  - Create a **Shape object**, then call:
    - g2.**draw**(Shape s)
    - g2.**fill**(Shape s)

For anything implementing the **Shape** interface, including *custom* shapes!



- Similarities to Python's graphics package

```
r = Rectangle(Point(20,20),Point(100,60))
r.draw(win)
```

The shape paints itself on a window

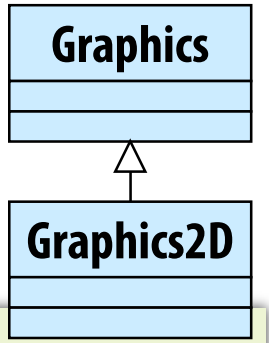
```
public void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    Shape r = new Rectangle2D.Float(20, 20, 80, 40);
    g2.fill(r);
}
```

The graphics object paints the shape

Basic Graphics 3: Example

jonkv@ida

- Painting with Graphics2D




```
public class ClockComponent extends JComponent {
   public void paintComponent(Graphics g) {
      ...
      ...
      Graphics2D g2 = (Graphics2D) g;


      Shape myCircle = new Ellipse2D.Double(
            center.x - radius, center.y - radius,
            2 * radius, 2 * radius);
      g2.draw(myCircle);
   }
}
```

Backwards compatibility:
Parameter of type Graphics…

The actual *value* is a Graphics2D,
I promise!

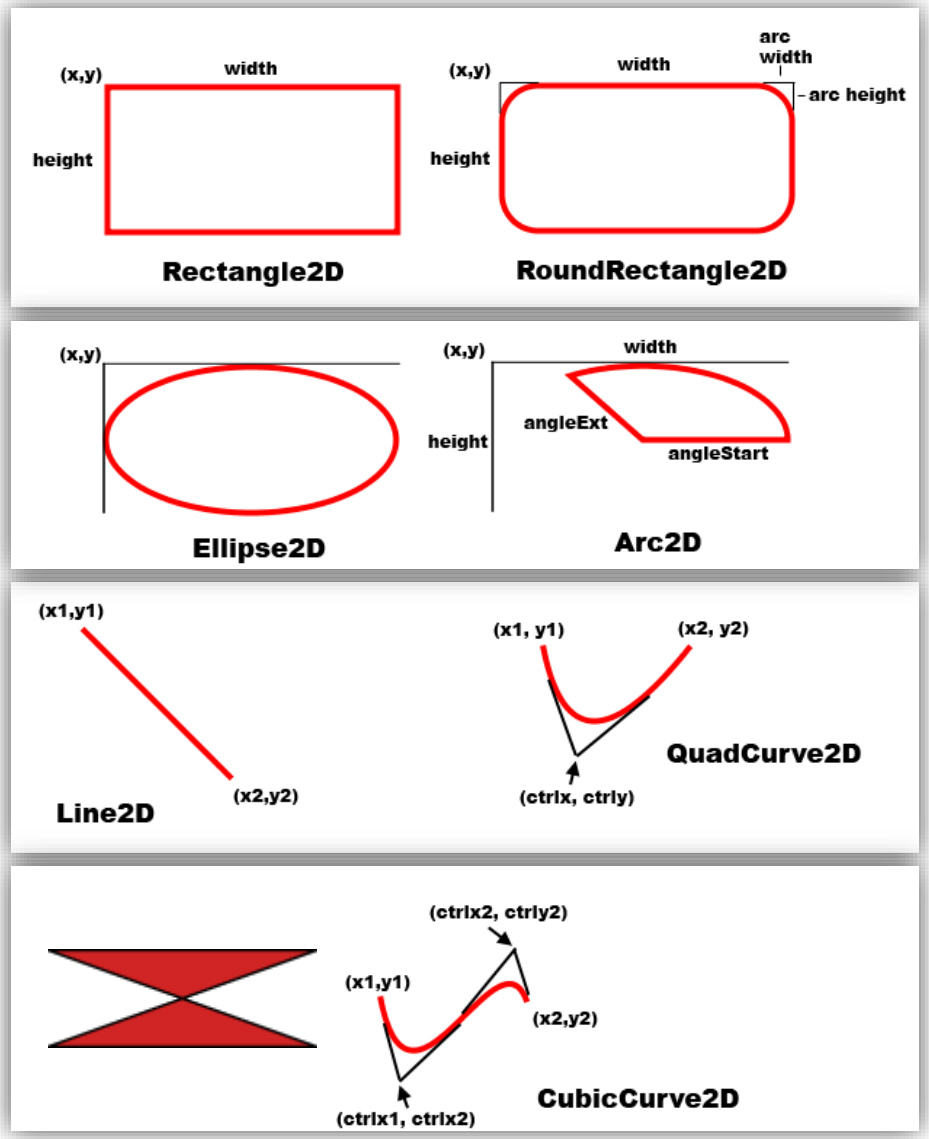
Create a Shape
(double-precision parameters)

Basic Graphics 4: Predefined Shapes

jonkv@ida

- **Predefined shapes**:
  - Rectangle2D,
    RoundRectangle2D

    Rectangle2D.Float and
    Rectangle2D.Double, etc.
  - Arc2D,
    Ellipse2D
  - Line2D,
    QuadCurve2D,
    CubicCurve2D
  - Area,
    GeneralPath



Basic Graphics 5: Combining Shapes

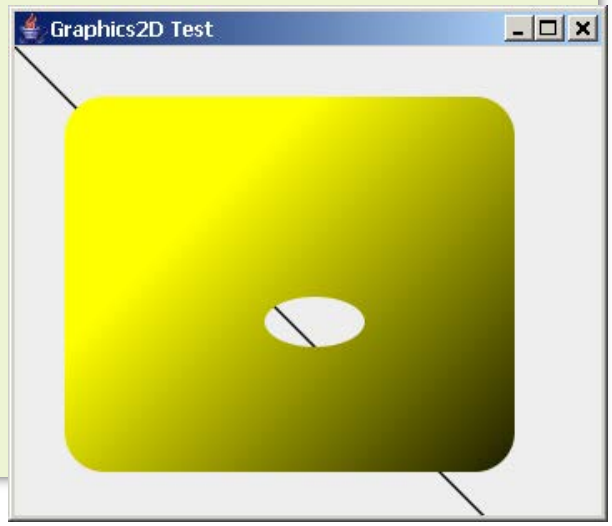
jonkv@ida

```
public class GraphicsTest extends JComponent {
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;
      g2.setColor(Color.BLACK);
      g2.drawLine(0, 0, 200, 200);

      g2.setPaint(new GradientPaint(  70, 70, Color.YELLOW.brighter(),
                                      200, 200, Color.BLACK));
      Area area = new Area(new RoundRectangle2D.Float(20, 20, 180, 150, 32, 32)));
      area.subtract(new Area(new Ellipse2D.Float(100, 100, 40, 20)));
      g2.fill(area);
   }
}
```

at x=70,y=70, the color is bright yellow

at (200,200), the color is black



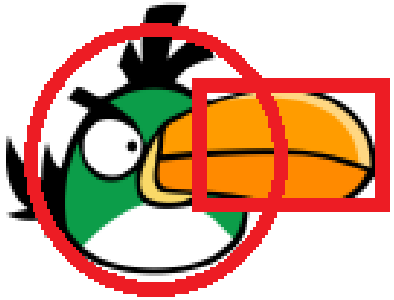
Side Note: Shapes and Collision Detection

jonkv@ida

- **Shapes** are *geometric*, can be tested for **collisions**

```
Shape s1                = ...;
Rectangle2D s2          = new Rectangle2D.Double(...);
if (s1.intersects(s2)) {
   // Collision!
}
```

- Useful for rough collision detection in 2D games
  - Even if you don't use shapes for painting
  - Create shapes corresponding to moving / stationary objects
  - Check if they intersect



Basic Graphics 6: Text

jonkv@ida

- To paint **text**:
  - Select a font (**java.awt.Font**)
    - Font myFont = **new** Font("Arial", Font.BOLD, 12);
      g2.setFont(myFont);
  - Use drawString()
    - int x = 100;
      int y = 100; // For the *baseline*!
      g2.drawString("Hello World", x, y);

  - **Finding metrics**: myFont.getLineMetrics(…)

Basic Graphics 7: Fonts

jonkv@ida

- **Which fonts are available?**
  - Always: "Logical" fonts
    - Dialog, DialogInput ➔ *mapped* to: Times?
    - Serif, SansSerif Helvetica?
    - Monospaced, Symbol **Broadway?**
  - Plus *any* font installed on your machine

```
public class MyComponent extends JComponent {

   public void checkFonts() {
      GraphicsEnvironment ge =
            GraphicsEnvironment.getLocalGraphicsEnvironment();
      String[] names = ge.getAvailableFontFamilyNames();
      …
   }
}
```

Example 1: Clock Component

jonkv@ida

- Constructing the **Clock Component**

```
class ClockComponent extends JComponent {
   private LocalTime time;

   public ClockComponent(LocalTime time) {
      this.time = time;
   }

   public LocalTime getTime() {
      return time;
   }

   public void setTime(LocalTime time) {
      this.time = time;
   }

   …
```

Example 2: Clock Component

jonkv@ida

```
public void paintComponent(Graphics g) {
   super.paintComponent(g);
   Graphics2D g2 = (Graphics2D) g;
   Point center = new Point(getWidth() / 2, getHeight() / 2);
   // Fill most of the current size, but leave some margins
   int radius = (int) (0.45 * Math.min(getWidth(), getHeight()));

   // Paint the circle representing the clock
   g2.draw(new Ellipse2D.Double(center.x - radius, center.y - radius,
                                2 * radius, 2 * radius));

   final double π = Math.PI;

   drawHand(g2, center, 2 * π * (time.getHour() / 12.0), radius * 2 / 3);
   drawHand(g2, center, 2 * π * (time.getMinute()  / 60.0), radius * 9 / 10);
   drawHand(g2, center, 2 * π * (time.getSecond() / 60.0), radius * 9 / 10);
}
```

"How wide and tall am I, the clock component, on the screen?"

Example 3: Clock Component

jonkv@ida

```
private void drawHand(Graphics2D g2, Point center,
                      double angle, int radius)
{
   g2.drawLine(center.x, center.y,
               (int) (center.x + radius * Math.sin(angle)),
               (int) (center.y - radius * Math.cos(angle)));
   // Or: g2.draw(new Line2D.Float(center.x, center.y, ..., ...));
}
```

Example 4: Clock Component

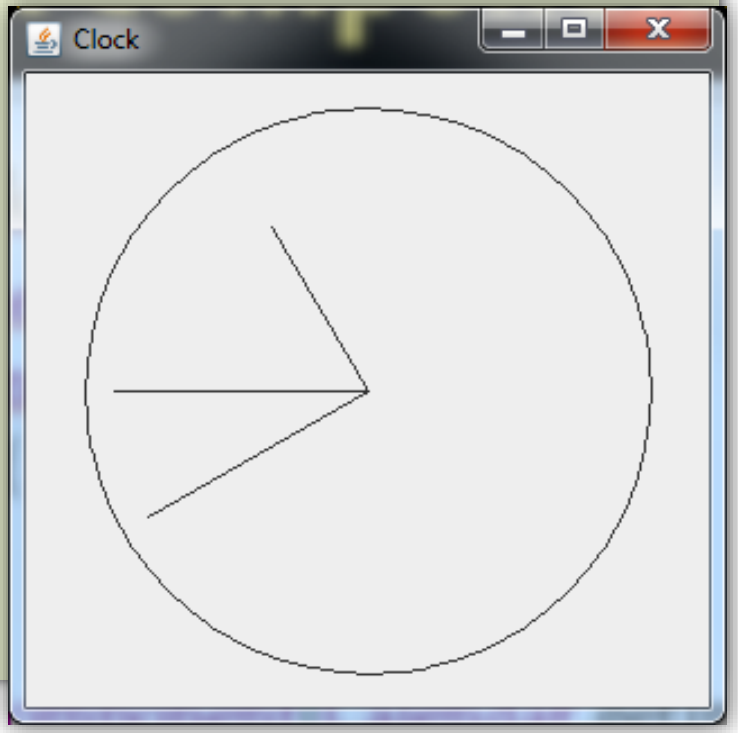
jonkv@ida

```
public class ClockFrame extends JFrame {
   private ClockComponent clock;
   public ClockFrame() {
      super("Clock");

      clock = new ClockComponent(LocalTime.now()); // Static method returns current time

      setLayout(new BorderLayout());
      add(clock, BorderLayout.CENTER);
   }
   public static void main(String[] args) {
      JFrame frame = new ClockFrame();
      frame.setSize(320,320);
      frame.setVisible(true);
   }
}
```

- Default color
- Thin lines
- Jagged lines
- …



Painting with Different Styles

Styles 1: In the Graphics object

jonkv@ida

- Python's **graphics** package: **Styles** are part of the **shape**

```
r = Rectangle(Point(20,20),Point(100,60))
r.setFill('green')
r.draw(win)
```

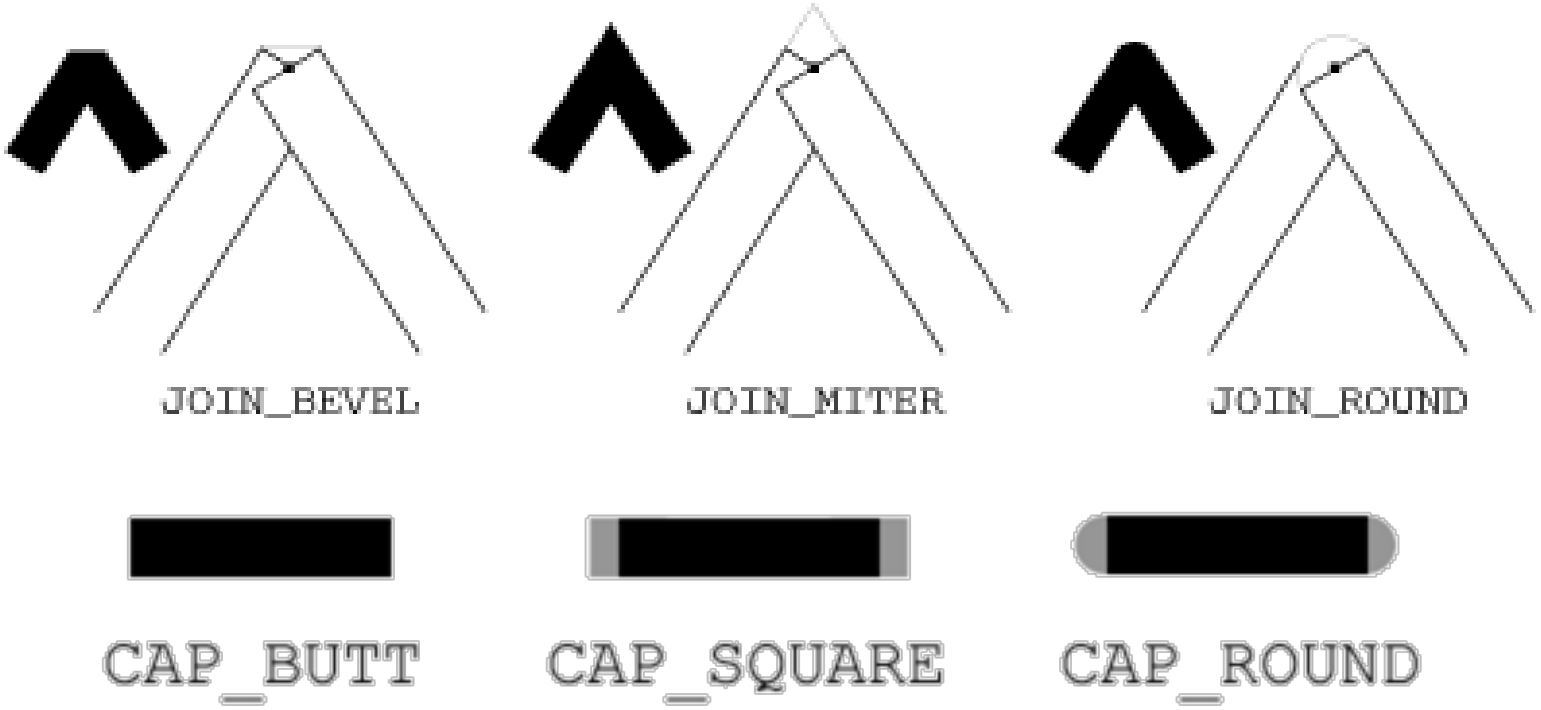
- Java: Set **styles** in the **Graphics object**
  - Applied to everything painted afterwards

```
public void paintComponent(Graphics g) {
   Graphics2D g2 = (Graphics2D) g;
   g2.setColor(...);
   g2.fill(new Rectangle2D.Float(20, 20, 80, 40));
}
```

Styles 2: Strokes and Pen Styles

jonkv@ida

- **Pen styles** implement the Stroke interface – use setStroke()
  - Line widths, patterns, join styles: BasicStroke.JOIN_BEVEL, etc.



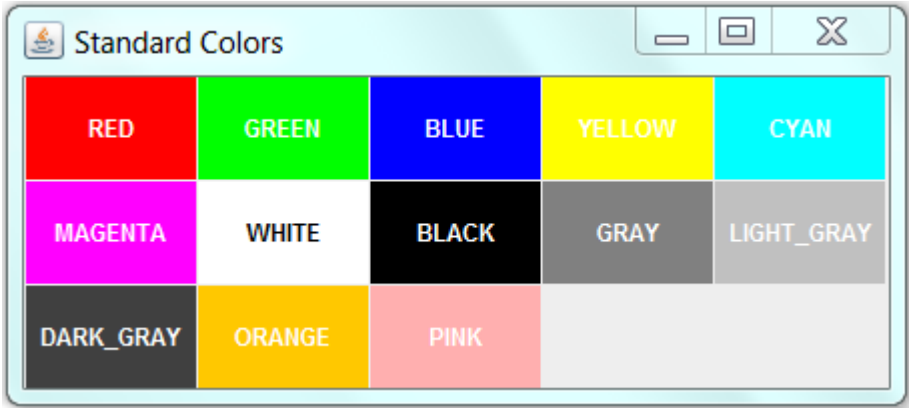
For the clock component:

```
g2.setStroke(new BasicStroke(width, BasicStroke.CAP_SQUARE,
                                    BasicStroke.JOIN_MITER));
```

Styles 3: Colors

jonkv@ida

- **Fill patterns** implement the **Paint** interface: **setPaint**(…)
  - **Color**: Solid fill, default color space sRGB (rgb + alpha)
    - Static constant fields: Color.RED, Color.GREEN, Color.BLACK, …




```
Color cyan1 = new Color(0.0, 1.0, 1.0); // Between 0.0 and 1.0
Color cyan2 = new Color(0, 255, 255); // Between 0 and 255
```

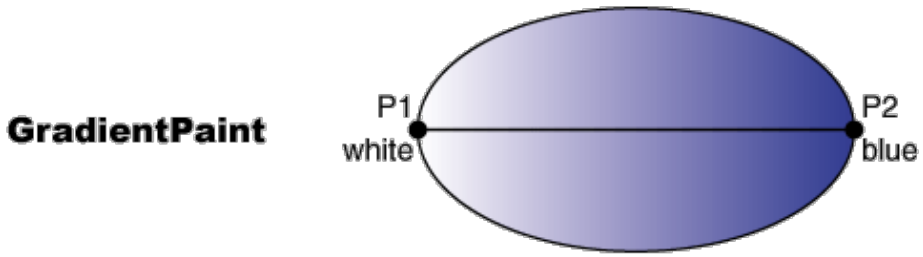
For the clock component:

g2.setPaint(Color.BLACK);

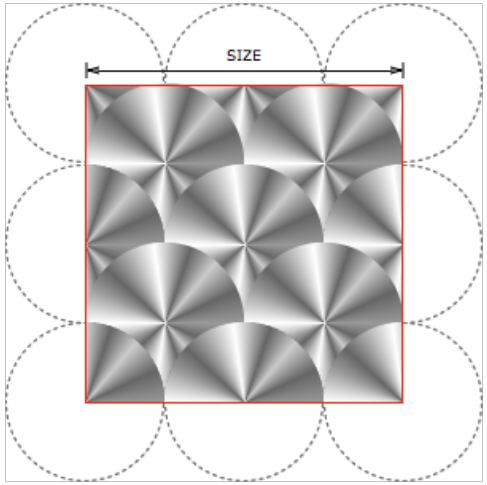
Styles 4: Gradients and Textures

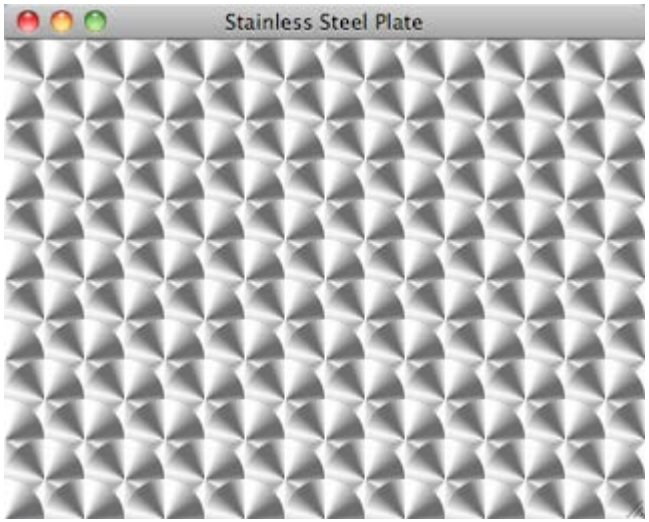
jonkv@ida

- **GradientPaint**:
  - A gradient between two colors



- **TexturePaint**:
  - Tiles a picture





Styles 5: Composition

jonkv@ida

- What happens when you **draw on top of** existing pixels?
  - Sometimes you want *semi-transparent* paint…

Styles 6: Composition in Swing

jonkv@ida

- We want to **compose** the *existing* and the *new*
  - Such rules implement the **Composite** interface
  - Plain transparency:
    - g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.7));
    - (0.0 is transparent, 1.0 is opaque)
  - Applies to everything you paint

Styles 7: Rendering Hints

jonkv@ida

- Tell Graphics2D **how to paint**:
  - **Antialiasing** for **text**
    - g2.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
      RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
  - **Antialiasing** for **non-text**
    - g2.setRenderingHint(
      RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON);

For the clock component:

g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);

Clock Revisited

jonkv@ida

```
public void paintComponent(Graphics g) {
   ...
   g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                       RenderingHints.VALUE_ANTIALIAS_ON);
   g2.setColor(Color.GREEN);
   g2.setStroke(new BasicStroke(3.0f));
   g2.draw(new Ellipse2D.Double(center.x - radius, center.y - radius, 2*radius, 2*radius));
   ...
   ...
   drawHand(g2, radius * 0.08f, center, 2 * π * (time.getHour() / 12.0), radius * 2 / 3);
   drawHand(g2, radius * 0.04f, center, 2 * π * (time.getMinute() / 60.0), radius * 9 / 10);
   drawHand(g2, radius * 0.02f, center, 2 * π * (time.getSecond() / 60.0), radius * 9 / 10);
}
private void drawHand(Graphics2D g2, float width, Point center,
                      double angle, int radius) {
   g2.setColor(Color.BLACK);
   g2.setStroke(new BasicStroke(width, BasicStroke.CAP_SQUARE,
                                BasicStroke.JOIN_MITER));
   ...
}
```

Antialiasing on

Green circumference, thicker pen

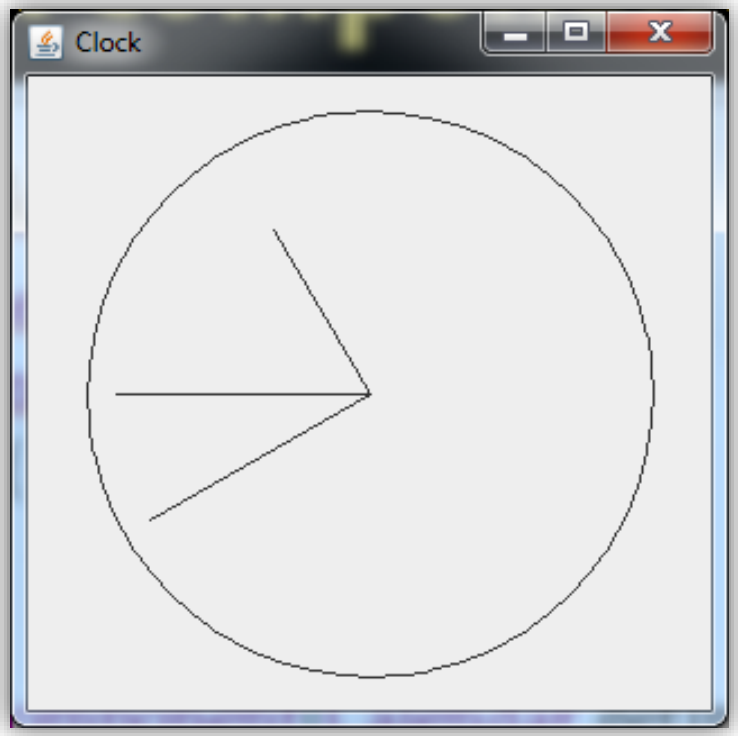
Specify line thickness…

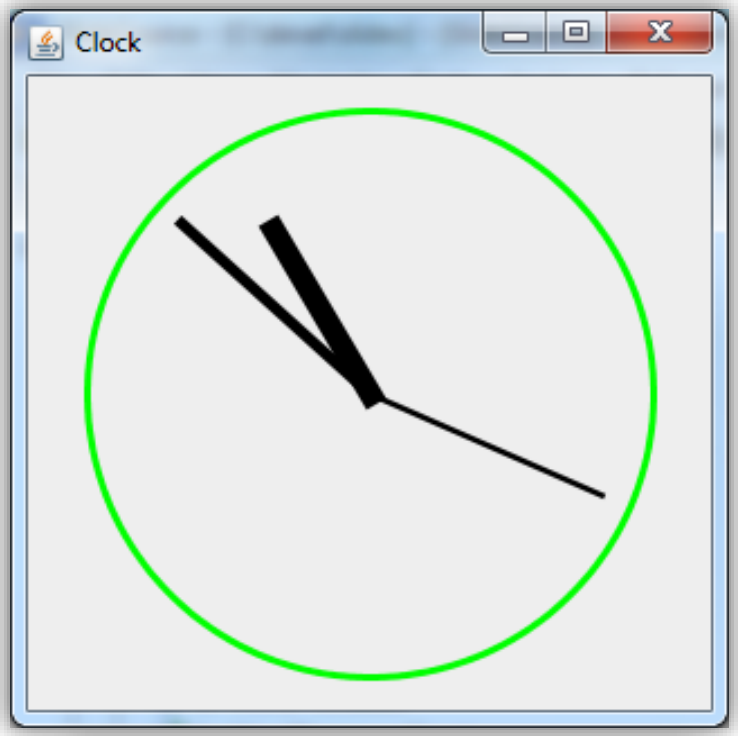
Specify line thickness and more

Clock Revisited (2)

jonkv@ida

- Result:



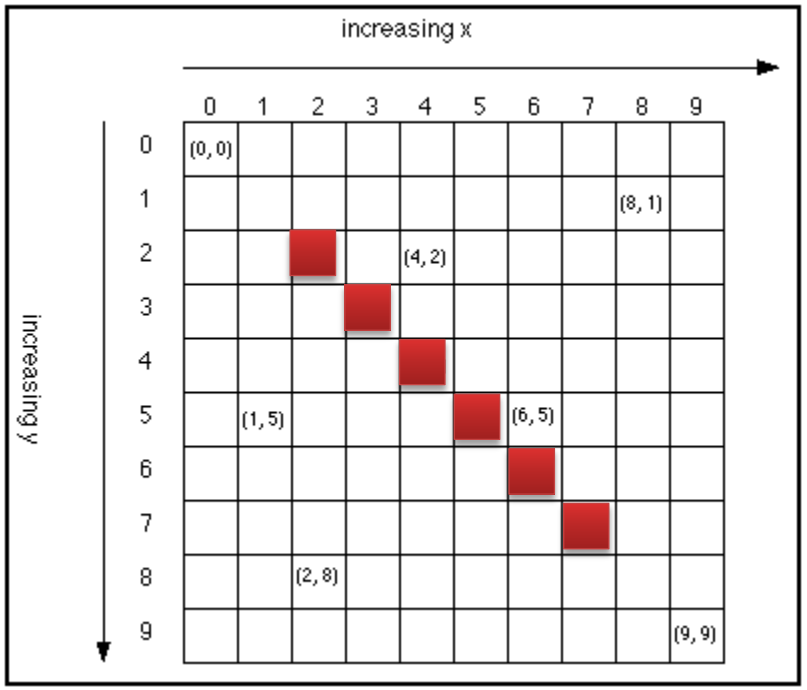


Coordinates and Transforms

Coordinate System: Default

jonkv@ida

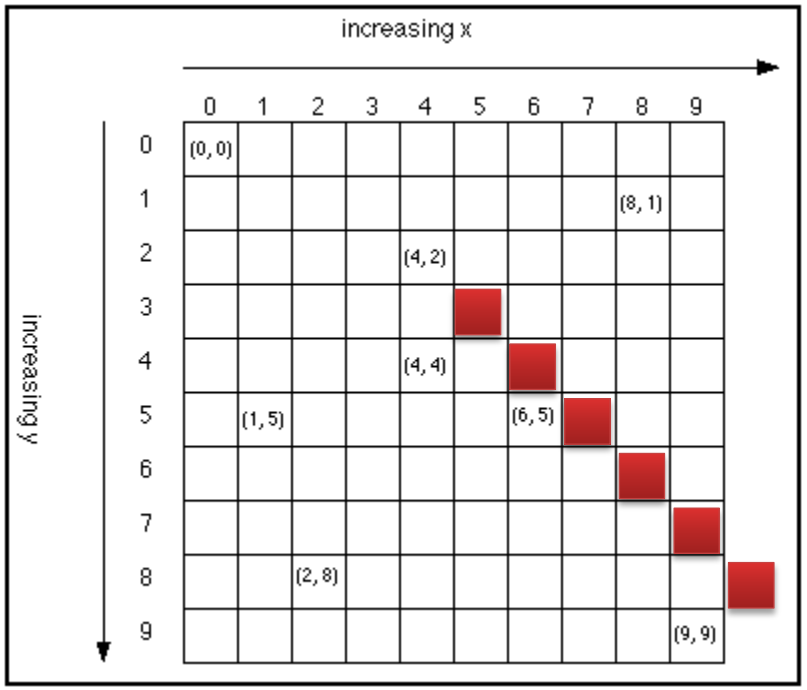
- Default coordinates:
  - Counting pixels
  - Starting in the top left corner
    - **g2.drawLine(2,2,7,7);**



Translations

jonkv@ida

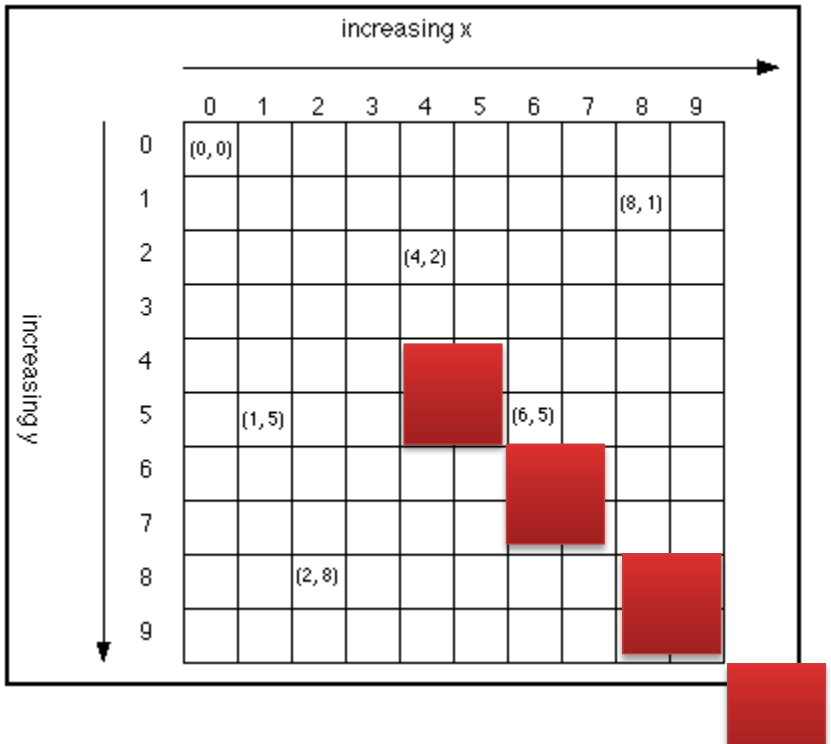
- Coordinates are transformed using **Affine Transforms**
  - Matrix math – hidden behind these concepts
    - AffineTransform at = **new** AffineTransform();
      at.**translate**(3, 1);
      g2.setTransform(at);
      g2.drawLine(2,2,7,7);



Scaling

jonkv@ida

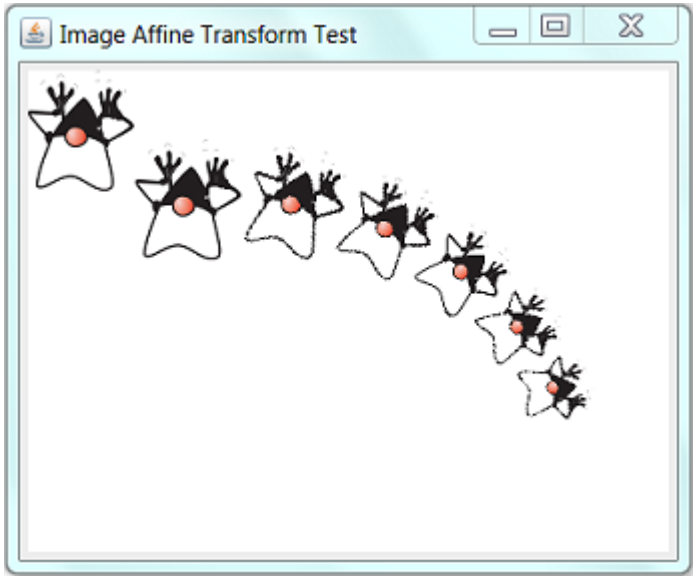
- Coordinates are transformed using **Affine Transforms**
  - Matrix math – hidden behind these concepts
    - AffineTransform at = **new** AffineTransform();
      at.**scale**(2, 2);
      g2.setTransform(at);
      g2.drawLine(2,2,7,7);

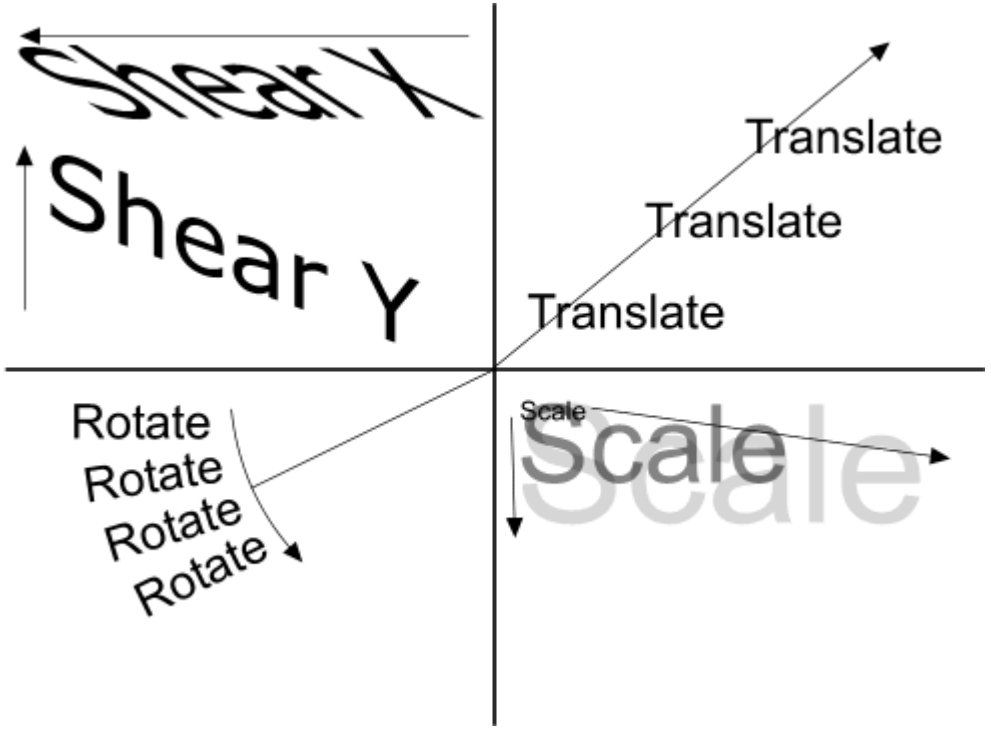


Transforms

jonkv@ida

- Coordinates are transformed using **Affine Transforms**
  - Rotate, translate, scale or shear **shapes, images, text, anything**!
    - ```
      AffineTransform at = new AffineTransform();
      at.setToRotation(Math.toRadians(45));          // 45 degrees
      at.shear(0.10, 0.00);
      at.scale(2.0, 4.0);
      at.translate(100, 200);
      g2.setTransform(at);
      ```

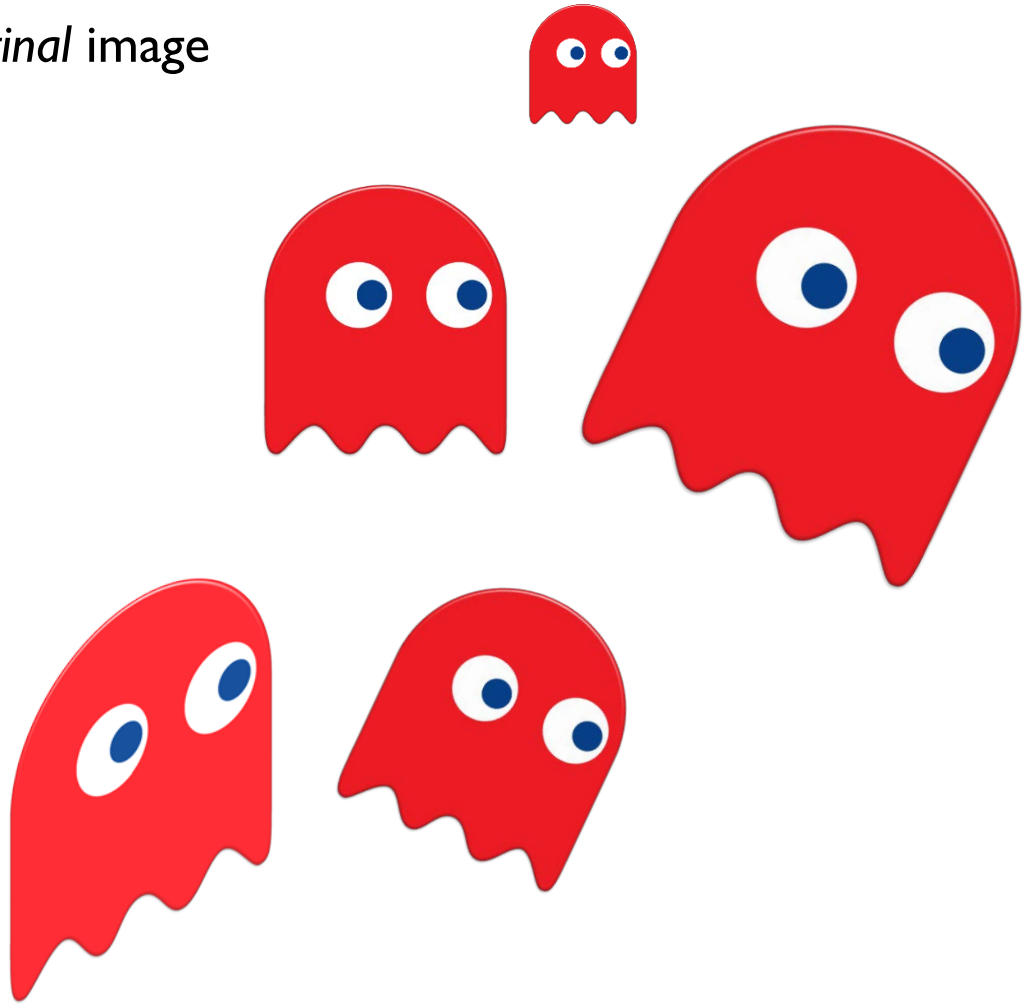



      ```

Transforms and Images

jonkv@ida

- To clarify: **This is the way to rotate and scale ANY image**!
  - Don't "get a *rotated* image"
  - Set a transform, paint the *original* image

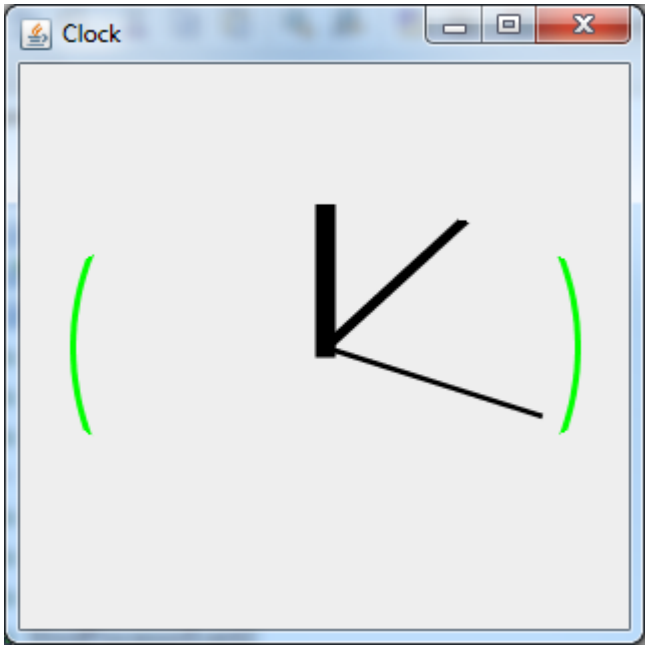


Clipping:
Where *not* to paint

Clipping 1: Intro

jonkv@ida

- A Graphics object has a **clipping area**
  - Nothing is painted outside this area
    - g2d.setClip(**new** Ellipse2D.Double(0, getHeight()/4, getWidth(), getHeight()/2));



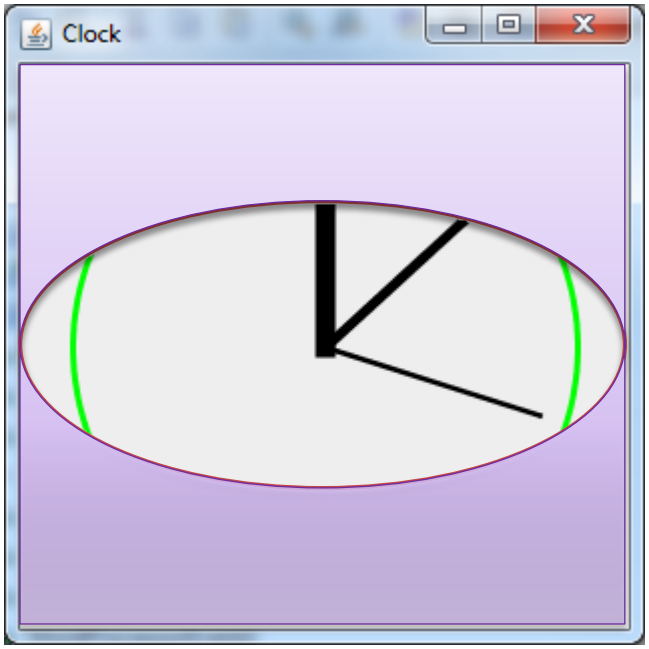


Illustration of clipping region (not visible in reality!)

Use to *protect parts of the screen*

Use to *save time when painting*

Clipping 2: For Protection

jonkv@ida

Single game component, completely repainted…

Protect the score banner when blue pieces are painted

Protect the chat bar when painting the ball

(which can disappear outside the screen)

Clipping 3: For Performance

jonkv@ida

Repainted when only the ball has moved…

Set this clip area

Ask Swing to paint the *entire* green BG gradient

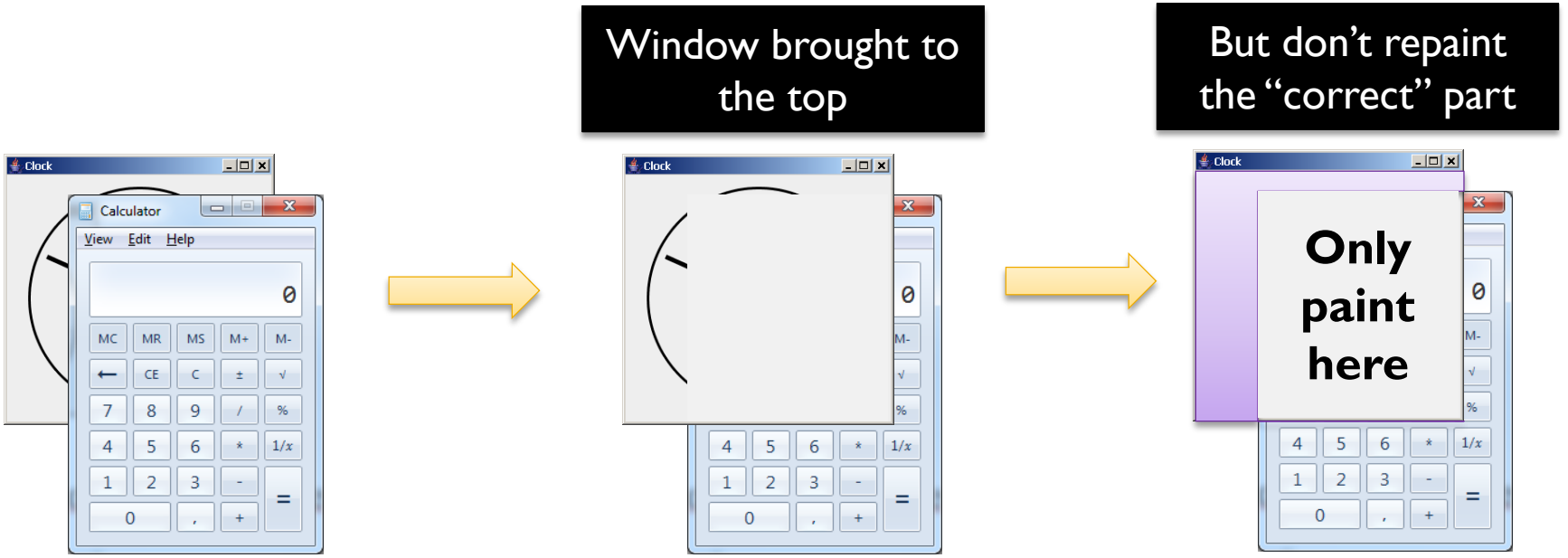
➔ only the part inside the clip area is actually painted

➔ no need to repaint the blue pieces

Clipping 4: By Swing

jonkv@ida

- When a component appears after having being covered:
  - Swing **automatically sets** the clipping area

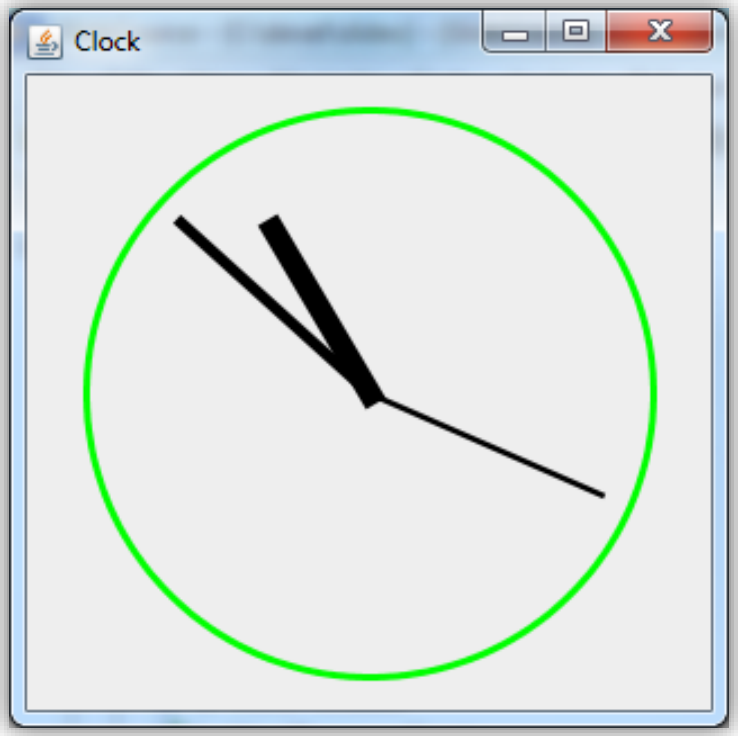


Scheduling Tasks

Motivation

jonkv@ida

- So far the clock never changes…



Regular Updates 1

jonkv@ida

- For continuous updates:
  - First create an **update action**

```
// Create a clock
JComponent clock = new ClockComponent(LocalTime.now());

// Create an action object that updates the clock when called
Action task = new AbstractAction() {
   public void actionPerformed(ActionEvent e) {
      clock.setTime(LocalTime.now());
      clock.repaint();
   }
};
```

Or update an entire game state, moving all players, checking collisions, …

Regular Updates 2

jonkv@ida

- Then **call this action** periodically
  - In the background
  - Using a **javax.swing.Timer**

Don't manipulate a GUI in a *java.util.Timer*!

```
// Every 500 ms, ask Swing to "call task as soon as possible"
// Handled when Swing has handled preceding requests in the queue
// Executed in the event thread ➔ task shouldn't take too much time...
Timer clockTimer = new Timer(500, task);

// If we get a backlog, only perform one of the queued requests
clockTimer.setCoalesce(true);

// Start the timer
clockTimer.start();

// Possibly clockTimer.stop() at some point...
```

Images in Java

Swing Icons and ImageIcons

Introduction: Two Image Classes

jonkv@ida

- Two important classes for **displaying images from files**
  - **java.awt.Image**
    - **Incremental asynchronous** loading, useful for slow network connections
    - Not covered here
  - **javax.swing.ImageIcon**
    - Synchronous loading
    - Much simpler API
  - Both classes support GIF, JPEG, PNG, possibly others
    - Can plug in support for new formats

Icons 1: Icon and ImageIcon

jonkv@ida

- Interface **javax.swing.Icon** – a picture of fixed size
  - Can be bitmapped or painted using the Graphics object!
    - int getIconHeight()
    - int getIconWidth()
    - void paintIcon(Component c, Graphics g, int x, int y)

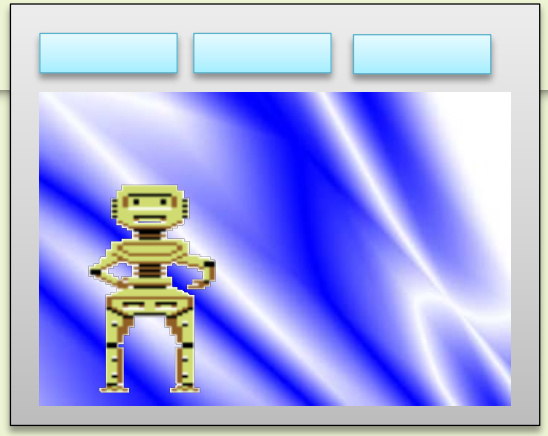
- Class **javax.swing.ImageIcon** implements Icon:
  - An icon **created from a bitmap image**
    - **new** ImageIcon(byte[] data)
    - **new** ImageIcon(String filename)
    - **new** ImageIcon(URL location)

Icons 2: A Background Image Example

jonkv@ida

- Simple example: Place player images on the screen

```
public class Player {
   private final Icon myself;  // Loaded and set in the constructor
   …
   public void showYourself(JComponent comp, Graphics2D g2) {
      myself.paintIcon(comp, g2, myCurrentX, myCurrentY);
} }
```



```
public class MyGameComponent extends JComponent {
   private List<Player> players = new ArrayList<Player>();
   …
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;
      …
      for (Player player : players)
         player.showYourself(this, g2);
      }
   }
}
```

Loading Icons

jonkv@ida

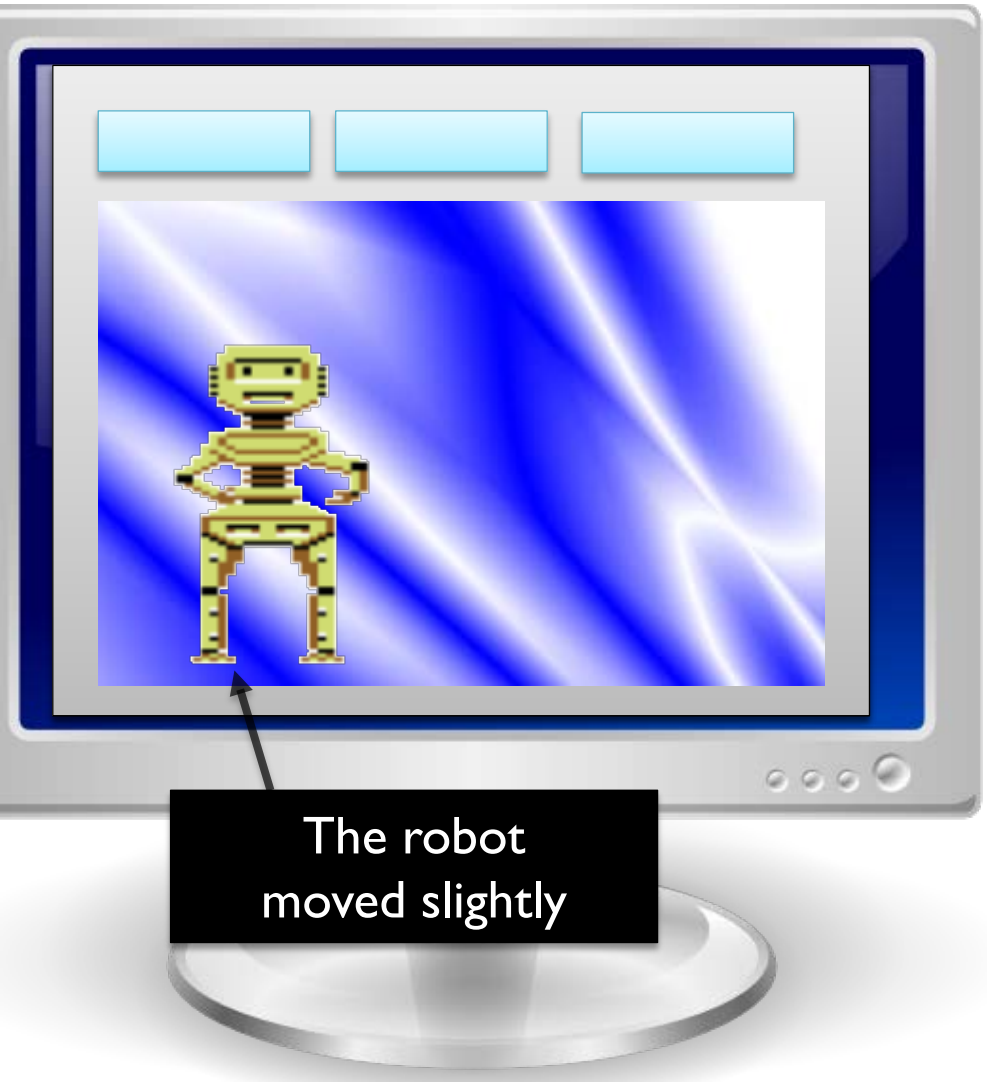
- How to *load* icons from a file?
  - Using *resources* – see the I/O lecture!

Double Buffering

Buffering 1: Single

jonkv@ida

- With **single buffering**, you paint directly onto the screen



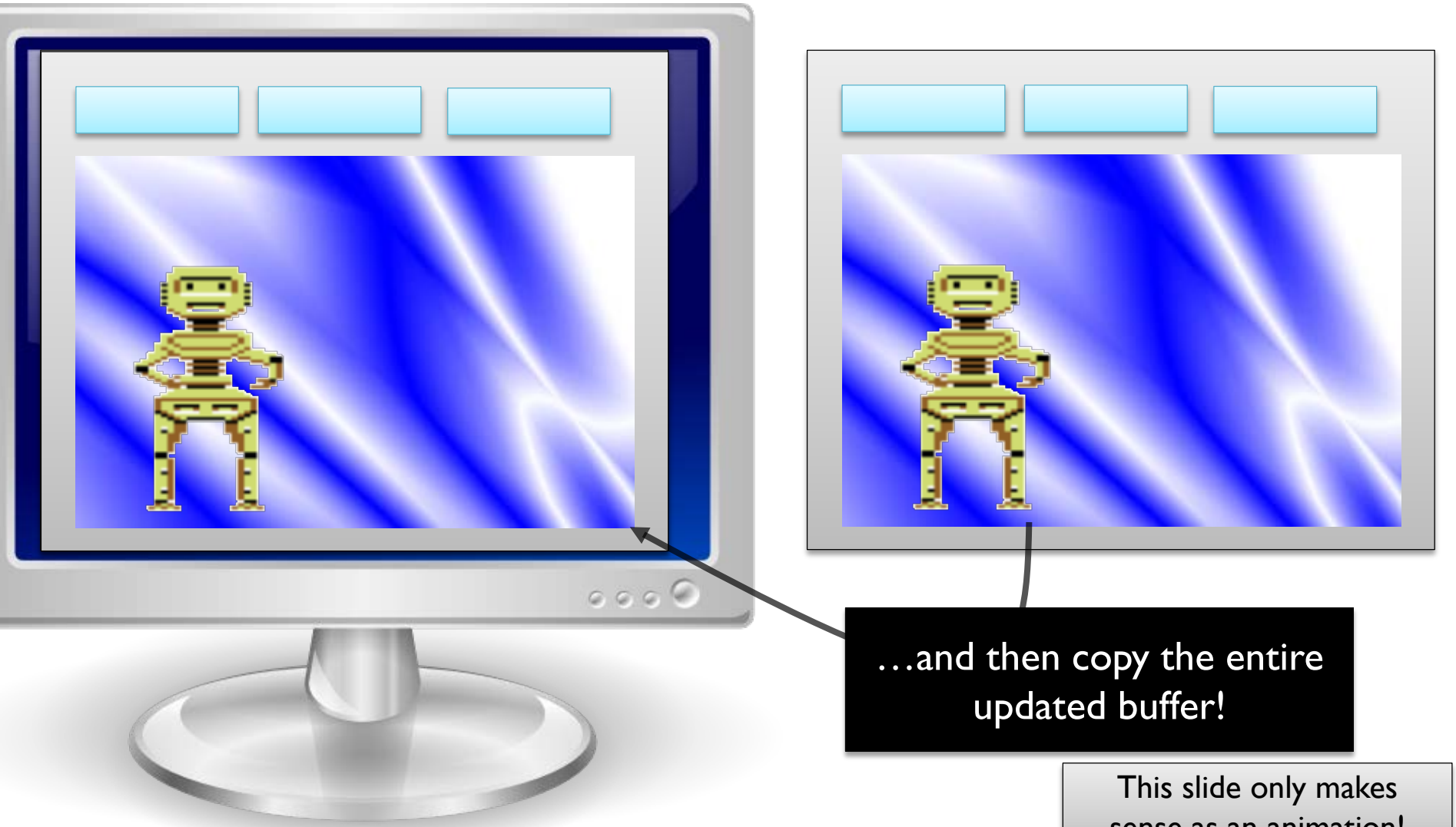
No problem if it's fast enough…
But often it isn't
➔ flickering!

This slide only makes sense as an animation!

Buffering 2: Double

jonkv@ida

- With **double buffering**, you paint onto an off-screen buffer…



…and then copy the entire updated buffer!

This slide only makes sense as an animation!

Buffering 3: Built In

jonkv@ida

- Built-in **double buffering**: setDoubleBuffered(true);
  - Turned on by default for JPanel
    - If your component is inside a JPanel, it won't flicker