Java GUI

Part 1. Shapes

Lecture 18

Component producers and consumers

- To write a good reusable component you need to use abstraction, encapsulation and make your code generic
- A client programmer will consume the components written by a server programmer and build the required system from smart blocks
- As a client programmer, sometimes you need not only to use the functionality provided to you, but adjust the components to your task

Tools to make simple cartoons

We want to be able to:

- Draw shapes on the screen
- Move shapes
- Rotate shapes
- Add text
- Animate shapes
- Start and stop animation

Each window has a frame

`javax.swing.JFrame`

- Outlines the window on the screen
- Has built-in buttons for minimizing, maximizing and closing
- The buttons look differently on different platforms

Easy GUI with javax.swing.*

```
import javax.swing.*;

public class ButtonInFramePane {
        public static void main (String [] args)
        {
                JFrame f=new JFrame("Button on the screen");
                JButton button=new JButton("Click me");
                f.getContentPane().add(button);

                f.setSize(300, 300);
                f.setVisible(true);
        }
}
```

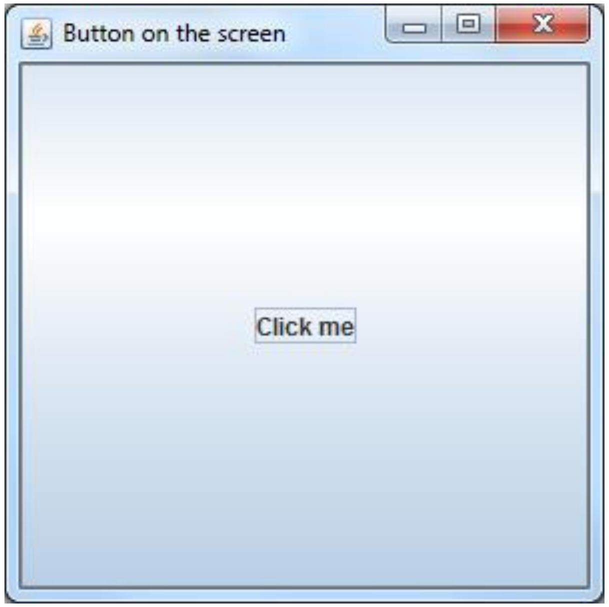
GUI component

Component is added to a Pane: each JFrame has a default content pane

Closing the frame does not stop the application

We need explicitly stop the program when the close button of our main window is pressed:

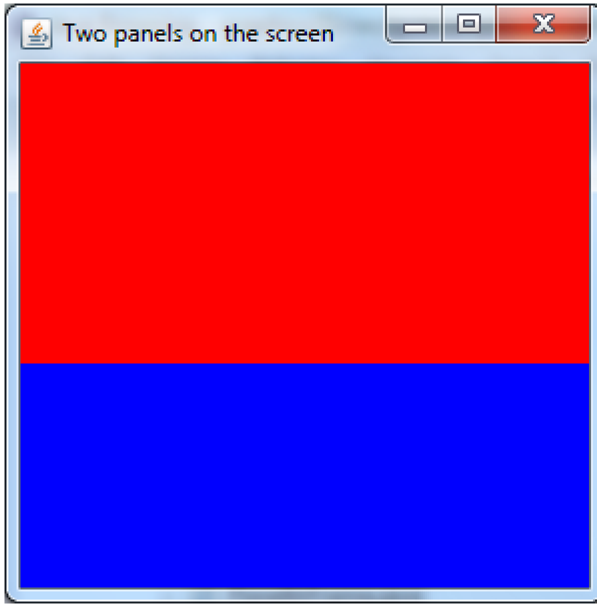
```
JFrame f=new JFrame("Button on the screen");
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```



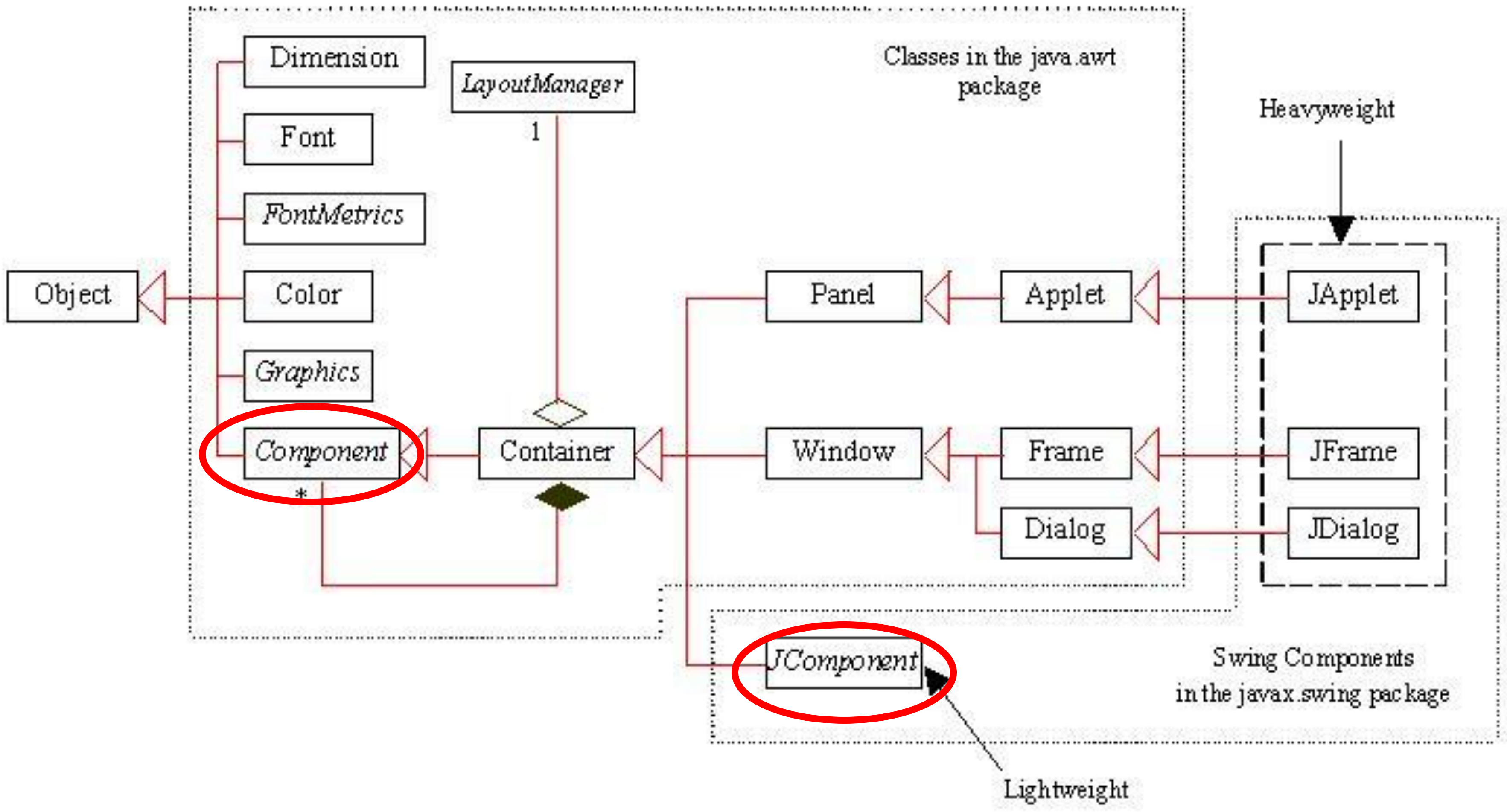
Content is added to JPanel, JPanel is added to JFrame

- We can create our own content panes and add them to a frame:

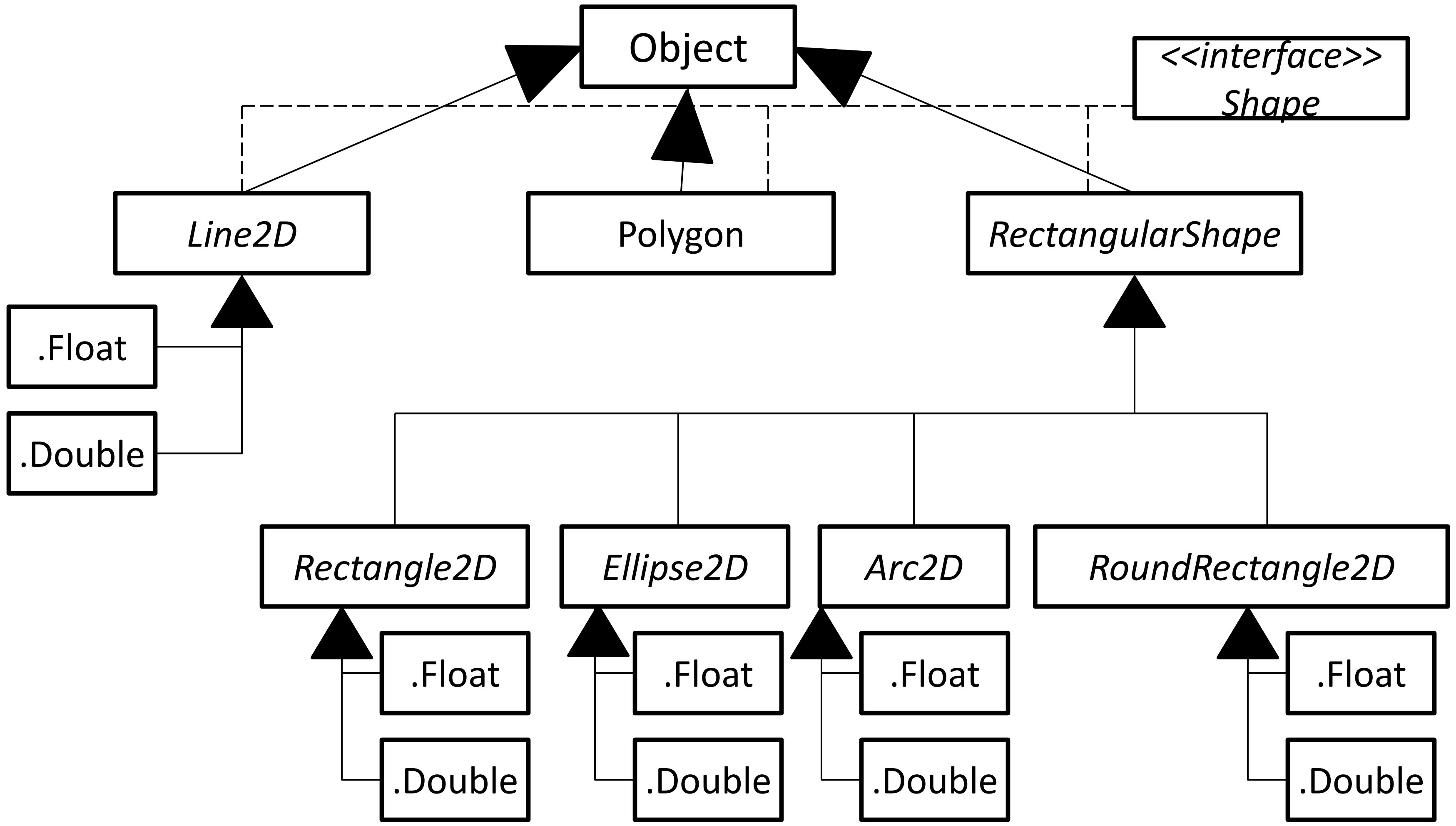
```
import javax.swing.*;

public class PanelsInFrame {
public static void main (String [] args){
        JFrame f=new JFrame("Two panels on the screen");
        JPanel upperPanel=new JPanel();
        upperPanel.setBackground(java.awt.Color.RED);

        JPanel lowerPanel=new JPanel();
        lowerPanel.setBackground(java.awt.Color.BLUE);

        f.setSize(300, 300);

        upperPanel.setBounds(0, 0, f.getWidth(), f.getHeight()/2);
        lowerPanel.setBounds(0, f.getHeight()/2, f.getWidth(), f.getHeight()/2);

        f.add(upperPanel);
        f.add(lowerPanel);
        f.setVisible(true);
    }
}
```



We can add to JPanel any widget that **is a *Component***



Java Shapes in java.awt.geom package



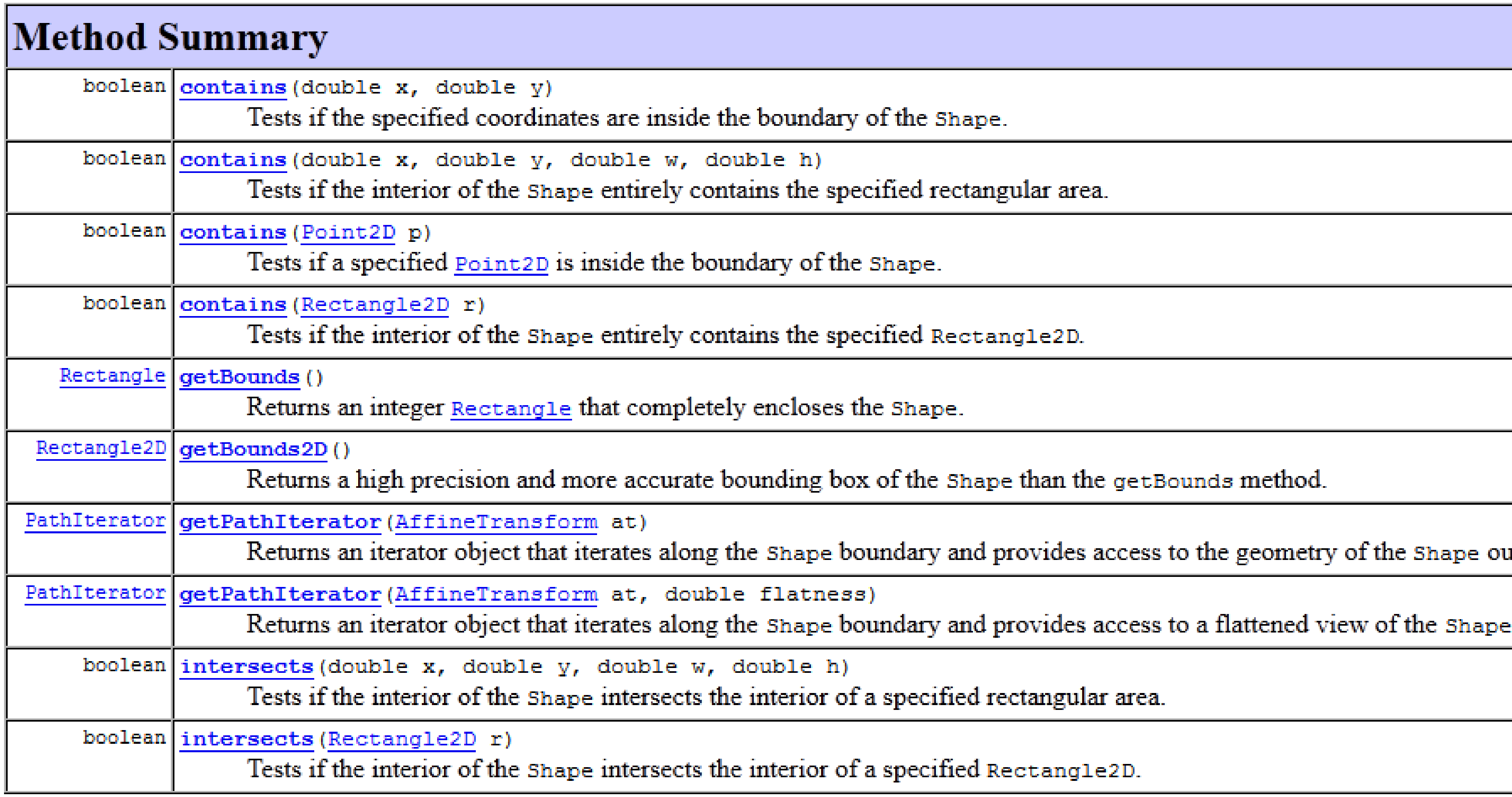
- Since Shape is not a Component, we **cannot add** it to the JPanel directly
- We **can draw** shapes on JPanel

SmartShapes and Java *Shape* interface

We want our shapes to be able to:

- Draw themselves
- Size themselves
- Position themselves
- Color themselves
- Rotate themselves

Java shape interface



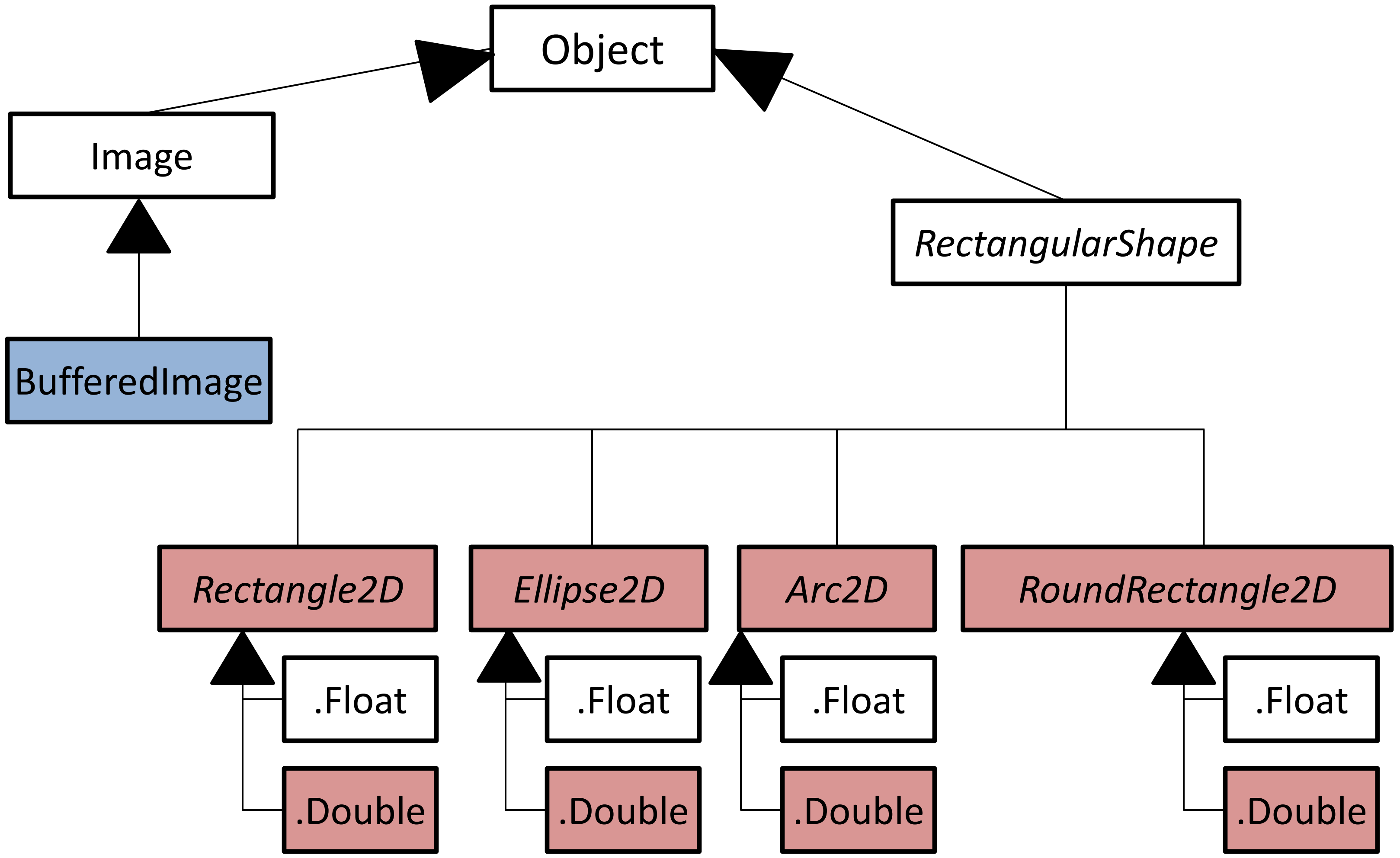
Method Summary	
boolean	`contains(double x, double y)` Tests if the specified coordinates are inside the boundary of the `Shape`.
boolean	`contains(double x, double y, double w, double h)` Tests if the interior of the `Shape` entirely contains the specified rectangular area.
boolean	`contains(Point2D p)` Tests if a specified `Point2D` is inside the boundary of the `Shape`.
boolean	`contains(Rectangle2D r)` Tests if the interior of the `Shape` entirely contains the specified `Rectangle2D`.
Rectangle	`getBounds()` Returns an integer `Rectangle` that completely encloses the `Shape`.
Rectangle2D	`getBounds2D()` Returns a high precision and more accurate bounding box of the `Shape` than the `getBounds` method.
PathIterator	`getPathIterator(AffineTransform at)` Returns an iterator object that iterates along the `Shape` boundary and provides access to the geometry of the `Shape` ou
PathIterator	`getPathIterator(AffineTransform at, double flatness)` Returns an iterator object that iterates along the `Shape` boundary and provides access to a flattened view of the `Shape`
boolean	`intersects(double x, double y, double w, double h)` Tests if the interior of the `Shape` intersects the interior of a specified rectangular area.
boolean	`intersects(Rectangle2D r)` Tests if the interior of the `Shape` intersects the interior of a specified `Rectangle2D`.

We want our elements to be able to:

- Draw themselves
- Size themselves
- Position themselves
- Color themselves
- Rotate themselves

Java *Shape* does not have methods for doing all this

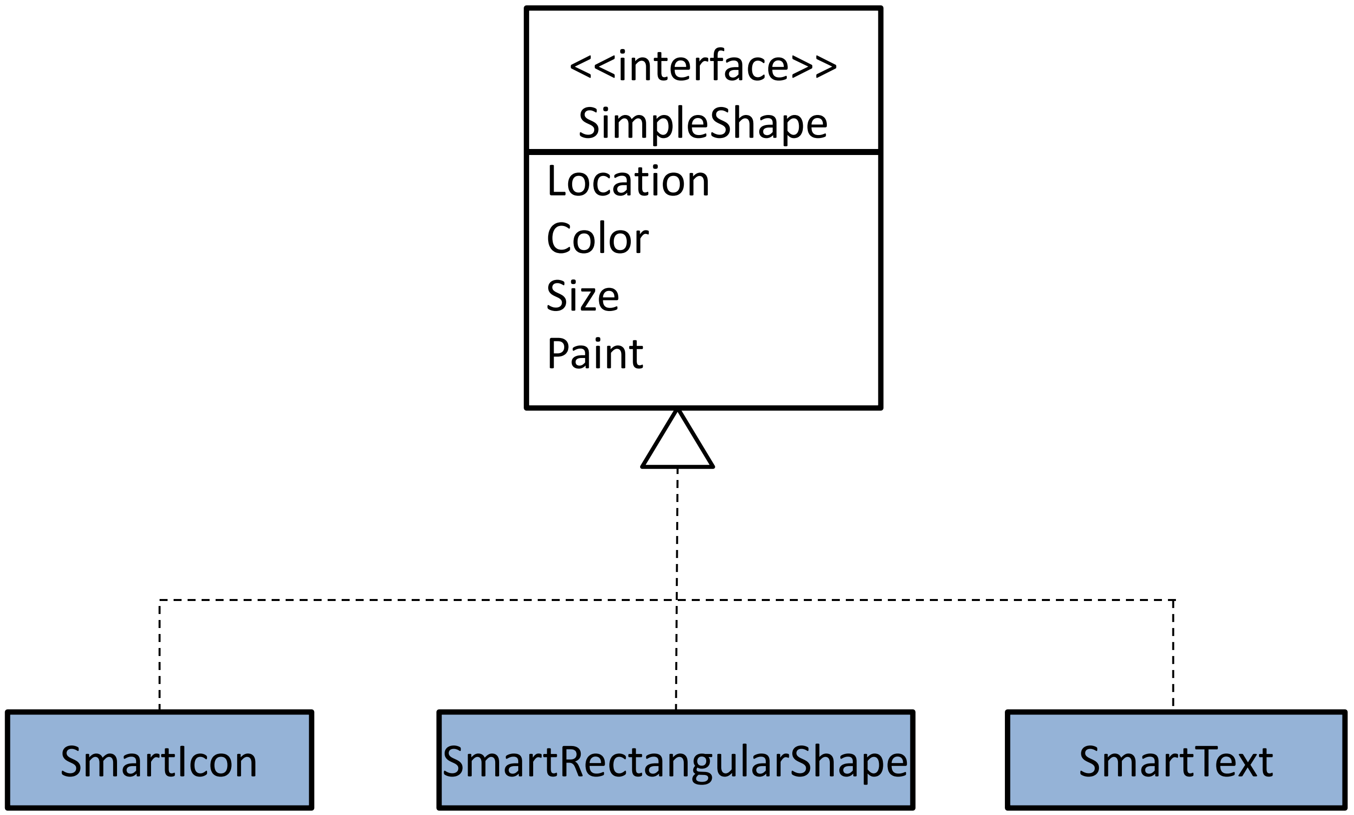
For simple animation we might need: rectangular shapes, images and text



To use classes across hierarchies: we define Interface:

```
public interface SimpleShape {
      public void setLocation (int posX, int posY);

      public void setSize (int width, int height);

      public void changeLocation (int stepX, int stepY);

//method to paint the shape

      public void draw(java.awt.Graphics2D brush);

}
```

UML diagram 1



SmartRectangularShape: wrapper class

We use **delegation** – we delegate functionality to the actual Rectangular shapes: Ellipse2D.Double, Arc2D.Double etc.

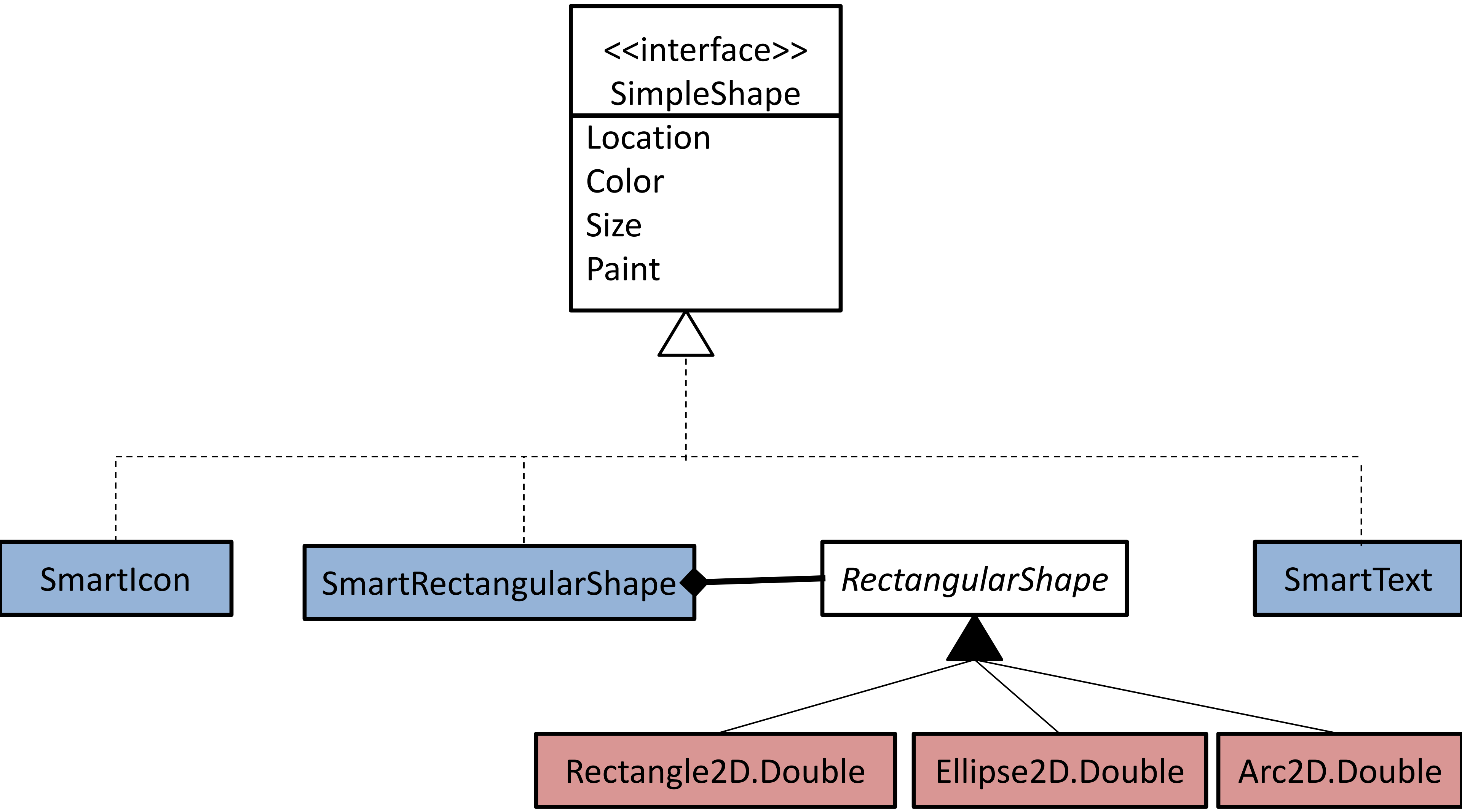
SmartRectangularShape: attributes

```
public class SmartRectangularShape
      implements SimpleShape{
      private Color  borderColor=Color.WHITE,
            fillColor=Color.WHITE;
      private int strokeWidth=1;  //default
                        stroke width 1 pixel
      private double rotation=0;
}
```

SmartRectangularShape: composition

```
public class SmartRectangularShape implements
SimpleShape{
      public final RectangularShape shape;

      public SmartRectangularShape
      (RectangularShape shape){
            this.shape=shape;
      }

}
```

UML diagram 2



SmartRectangularShape: delegation

```
public int getCenterX(){
    return (int)shape.getCenterX();
}

public int getCenterY(){
    return (int)shape.getCenterY();
}

//methods for positioning the shape
public void setLocation(int x, int y){
    shape.setFrame(x,y,shape.getWidth(),shape.getHeight());
}

public void changeLocation(int stepX, int stepY){
    this.shape.setFrame(this.getX()+stepX,
        this.getY()+stepY,this.getWidth(), this.getHeight());
}

public void setSize(int w, int h){
        shape.setFrame(shape.getX(),shape.getY(),w,h);
}
```

Painting the RectangularShape

- We still cannot add it to JPanel (not a Component)
- We need to draw it

Java Colors

- **java.awt.Color** stores color
  - RGB format

```
new
 java.awt.Color(0, 255, 0);
```

Red Green Blue

(255, 0, 0) =	🟥
(0, 255, 0) =	🟩
(0, 0, 255) =	🟦
(200, 0, 200)=	🟪

- Colors are determined by concentrations of **Red, Green** and **Blue**
  - each is given a value between 0-255
  - how many combinations are there?
    - 16,777,216
  - how many can average person distinguish?
    - about 7,000,000
- **Basic colors** come preset

java.awt.Color.ORANGE java.awt.Color.GREEN java.awt.Color.BLACK

- Colors can be
  - specify the **alpha value** [0-255]
  - 0 = completely transparent

Alpha Value

```
new java.awt.Color(239, 174, 45, 200);
```

transparent

Painting Shapes

- How do we paint on the **JPanel**?
- Shapes will have a **paint** method to **paint themselves**
- Nobody can **paint** without a **paintbrush!**

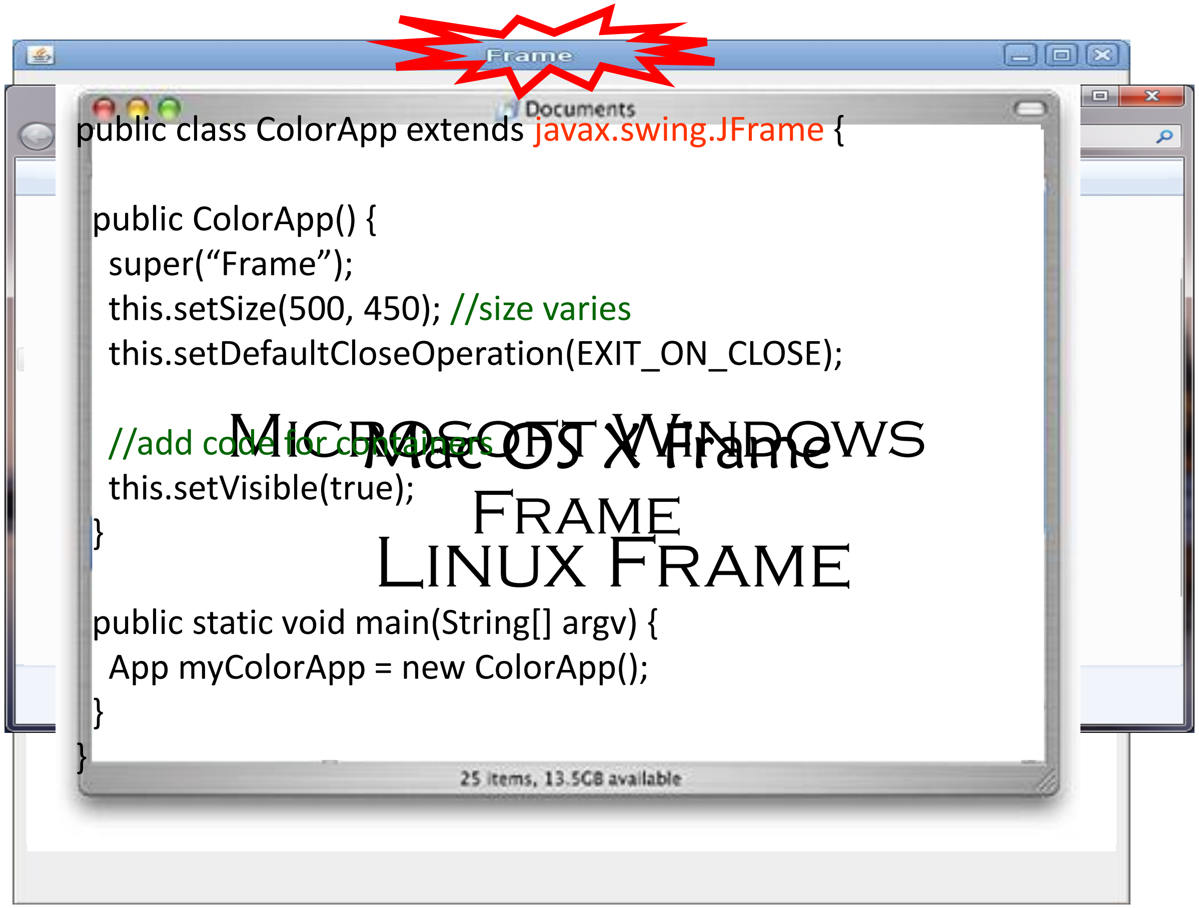
Painting Shapes - `Graphics`

- Our brush is brilliantly named **`Graphics`**
  - thanks, Java designers!
- In practice we use **`Graphics2D`**, its more powerful subclass
  - We'll see how to convert from a **`Graphics`** to a **`Graphics2D`**
- Think of **`Graphics2D`** as a **brush passed as a parameter**; now define **`SmartRectangularShape`**'s **`paint(…)`** method

```
public void paint(Graphics2D brush){

     // set the color of the brush
     brush.setColor(fillColor);
     brush.draw(shape);

     // tell brush to fill in the
     // geometric shape
     brush.fill(shape);
}
```

Frames

- When you draw graphics, you always need to draw onto some kind of container: JFrame holds containers…



```
public class ColorApp extends javax.swing.JFrame {

 public ColorApp() {
   super("Frame");
   this.setSize(500, 450); //size varies
   this.setDefaultCloseOperation(EXIT_ON_CLOSE);

   //add code for containers
   this.setVisible(true);
 }

 public static void main(String[] argv) {
   App myColorApp = new ColorApp();
 }
}
```

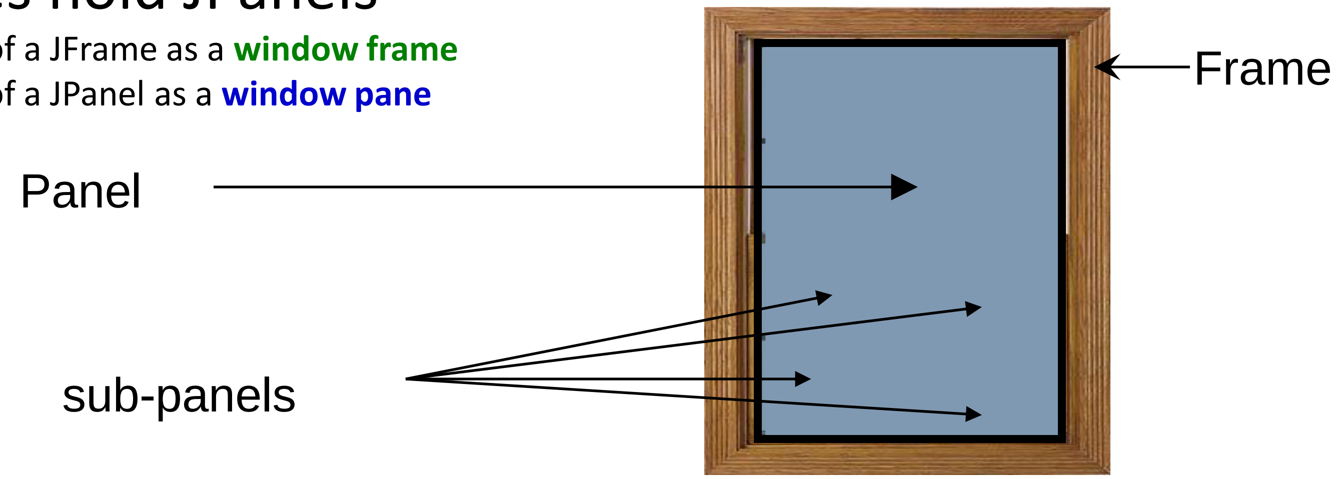
Microsoft Windows Frame

Mac OS X Frame

Linux Frame

Panels (`javax.swing.JPanel`s)

- JFrames hold JPanels
  - think of a JFrame as a **window frame**
  - think of a JPanel as a **window pane**



- Panels are the **canvas** for your program
  - panels draw graphical shapes and GUI elements
  - panels can contain sub-panels
  - we can add panels to a frame, and then add shapes and GUI elements to the panels

Drawing Panels on the screen

- **Specialized panels** hold specific types of objects
  - some hold buttons, sliders and text boxes
  - some hold shapes
  - some hold other panels

- We want to specialize a `JPanel` to hold and paint shapes
  - let's make a subclass of `JPanel` and call it `DrawingPanel`

- To paint anything on any JPanel, you must **partially override** its special method:

  ```
  protected void paintComponent(Graphics g)
  ```

  - remember that a `Graphics` is our "brush"

- The paintComponent is called implicitly
- Call `paint` on your **shape** from within this method
  - pass the brush to your shape as a parameter
  - you need to pass your shape a `Graphics2D`
  - the parameter of `paintComponent(…)` is a more general `Graphics`

Override paintComponent

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);  //partial override
    Graphics2D brush = (Graphics2D) g;
    //code to paint shapes with Graphics2D here
  }
```

To the Drawing Panel!

Add a rectangle to a **Drawing Panel**…

```
public class DrawingPanel extends
                          javax.swing.JPanel {

   private SmartRectangle rectangle;

   public DrawingPanel() {
      super();
      this.setSize(500,500);

      this.setBackground(java.awt.Color.WHITE);
      rectangle = new SmartRectangle ();
   }

   public void paintComponent(Graphics g) {
      super.paintComponent(g); //JPanel magic
      Graphics2D brush = (Graphics2D) g; //downcasting

      rectangle.paint(brush); }
}
```

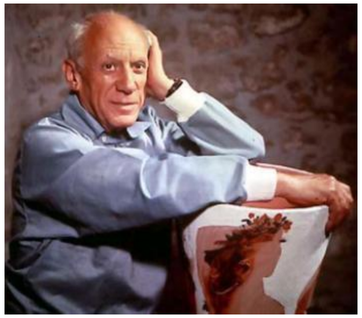
`paintComponent(…)`

- Whenever you want to execute the code in `paintComponent(…)`, call **`repaint()`** on the **`DrawingPanel`** instead

  ```
  drawingPanel.repaint();
  ```

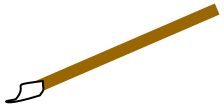
- **`repaint()`** is a method inherited by the **`DrawingPanel`** from `JPanel`
- **`repaint()`** calls `drawingPanel`'s `paintComponent(…)` for you and also creates the **`brush`** for you.
- Summary: To make a JPanel do something useful, create a subclass of it and *augment* ("partially override") `paintComponent()` to call `paint()` on the shapes in your panel

Picasso calls `repaint()` on a `DrawingPanel`!

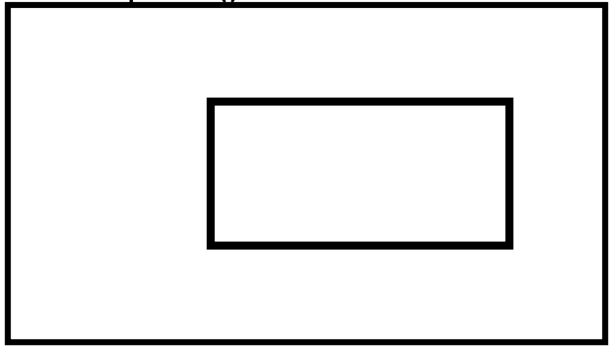


Pablo Picasso

repaint() creates a Graphics and calls DrawingPanel's paintComponent(…)



repaint()



DrawingPanel

(In DrawingPanel)

```
paintComponent(Graphics g){
  super.paintComponent(g);
  Graphics2D brush =  (Graphics2D) g;
  rectangle.paint(brush);
}
```

(In SimpleShape)

```
paint(Graphics2D brush) {
  brush.setColor(borderColor);
  brush.draw(shape);//_shape is a Rectangle2D
  brush.setColor(fillColor);
  brush.fill(shape);
}
```

Painting the RectangularShape

```
public void fill(Graphics2D brush){
//save color in a brush to restore it
        Color savedColor=brush.getColor();
        brush.setColor(this.fillColor);
        brush.fill(this.shape);
//restore the color of the brush
        brush.setColor(savedColor);
}
public void stroke(Graphics2D brush){
        //save color in a brush to restore it
        Color savedColor=brush.getColor();
        brush.setColor(this.borderColor);
        //save default stroke width to restore it
        java.awt.Stroke savedStroke=brush.getStroke();
        brush.setStroke(new java.awt.BasicStroke(this.getStrokeWidth(
        brush.draw(this.shape);
        //restore the color of the brush
        brush.setColor(savedColor);
        //restore the default stroke width
        brush.setStroke(savedStroke);}
```

SmartRectangualrShape: paint

```
public void paint(Graphics2D brush){
      this.stroke(brush);
      this.fill(brush);
}
```

Extensions 1/3: Smart Eclipse

```java
public class SmartEllipse extends
        SmartRectangularShape {

        public SmartEllipse ()      {
                super(new Ellipse2D.Double());
        }
}
```

Extensions 2/3: Smart Rectangle

```
public class SmartRectangle extends
      SmartRectangularShape {

      public SmartRectangle (){
            super(new Rectangle2D.Double());
      }
}
```

Extensions 3/3: Smart Arc

```
public class SmartArc extends SmartRectangularShape {
     public SmartArc (double aStart,
                double anExtent, int aType){
          super(new  Arc2D.Double
                (0,0,0,0,aStart,anExtent,aType));
     }
}
```

Arc2D.Double constructor parameters

Arc2D.Double

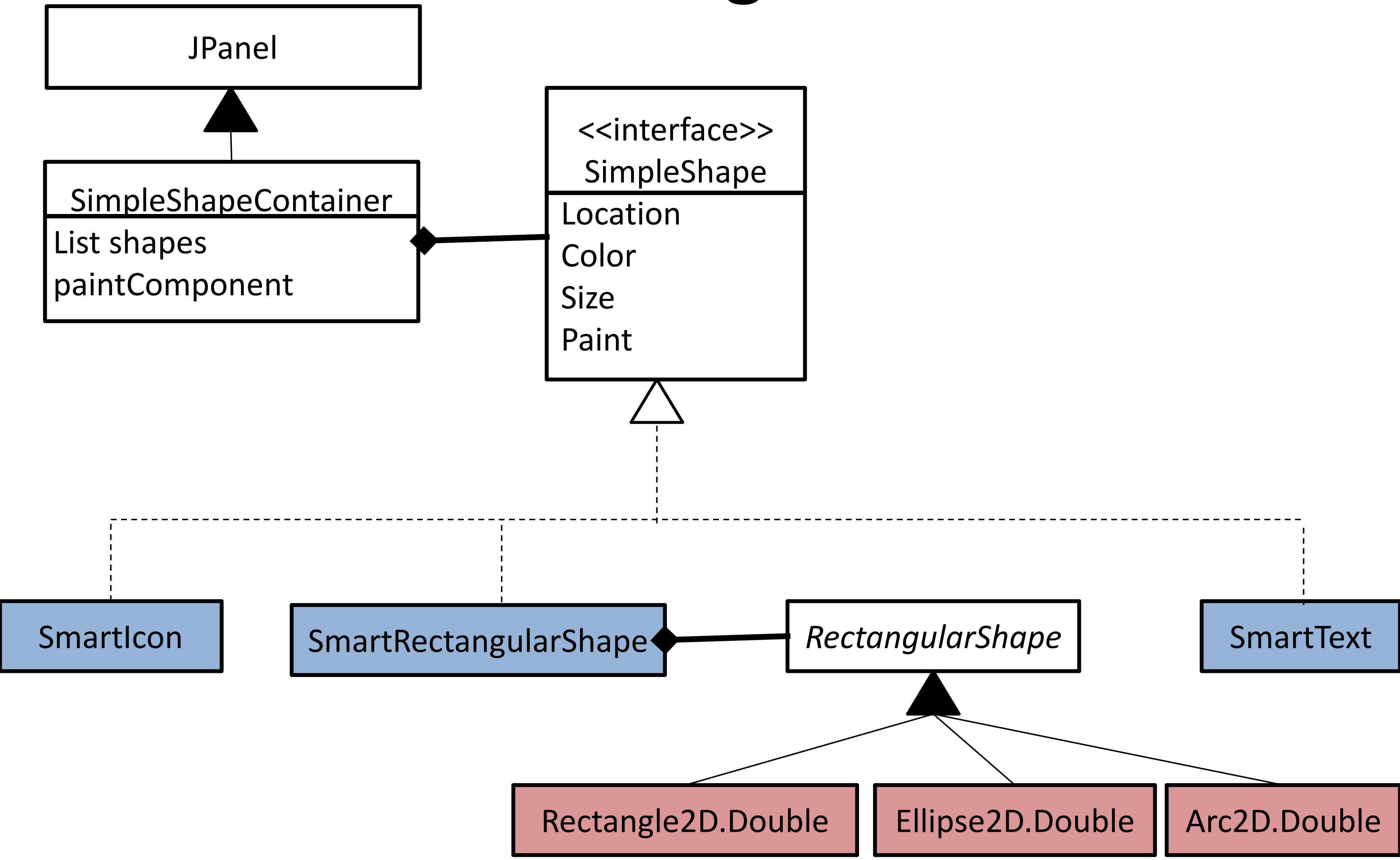
public **Arc2D.Double**(double x, double y, double w, double h, double start, double extent, int type)

- start - The starting angle of the arc in degrees.
- extent - The angular extent of the arc in degrees.
- type - The closure type for the arc: Arc2D.OPEN, Arc2D.CHORD, or Arc2D.PIE.

Panel holder for multiple shapes
(shapes should implement SimpleShape interface)

```
public class SimpleShapeContainer extends JPanel {

        private  List <SimpleShape> shapes=new LinkedList<SimpleShape>();

        public SimpleShapeContainer (){
                super();
        }

        public void addShape(SimpleShape shape){ shapes.add(shape);}

        public void paintComponent (Graphics aBrush ){
                super.paintComponent(aBrush);
                Graphics2D aBetterBrush=(Graphics2D)aBrush;
                for(SimpleShape s:shapes){
                        s.paint(aBetterBrush);
                }
        }
}
```

UML diagram 3



Adding simple shapes (1/3): Rectangular shapes

```
SmartRectangularShape arc=new SmartArc(200,300,Arc2D.PIE);
arc.setStrokeWidth(2);
arc.setBorderColor(Color.BLACK);
arc.setFillColor(Color.RED);
arc.setSize(50, 60);
arc.setLocation(200, 200);

p.addShape(arc);
```

Remains: SmartIcon and SmartText

SmartIcon implements SimpleShape

```
public class SmartIcon implements SimpleShape{
        BufferedImage image;

        public SmartIcon(BufferedImage img){
                this.image = img;
        }

        public void changeLocation (int stepX, int stepY){
                this.posX+=stepX;
                this.posY+=stepY;
        }

        public int getCenterX(){
                return this.posX+this.width/2;
        }

        public int getCenterY(){
                return this.posY+this.height/2;
        }
}
```

SmartIcon paint

```
public void paint(Graphics2D brush){
      brush.drawImage(this.image,null,posX, posY);
      Toolkit.getDefaultToolkit().sync();
}
```

SmartText implements SimpleShape

```
public class SmartText implements SimpleShape{
        Font font;
        String text;
        FontRenderContext frc;

        public SmartText(String text, Font font){
                this.text=text;
                this.font=font;
        }


        public void setSize (int width, int height){
                this.width=(int)this.font.getStringBounds(this.text,
                        frc).getBounds2D().getWidth();
                this.height=(int)this.font.getStringBounds(this.text,
                        frc).getBounds2D().getHeight();
        }
        …
}
```

SmartText paint

```java
public void paint(java.awt.Graphics2D brush)
{
      Font savedFont=brush.getFont();
      brush.setRenderingHint
            (RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);

      brush.setFont(font);
      brush.drawString(this.text, this.posX, this.posY);
      brush.setFont(savedFont);
}
```

Adding simple shapes (2/3): Icons

```
SmartIcon img=null;
try
{
        BufferedImage bi = ImageIO.read(new
        File("star.png"));
        img=new SmartIcon(bi);
        img.setLocation(200, 200);
        img.setSize(bi.getWidth(),bi.getHeight());
        p.addShape(img);
}
catch (IOException e) { e.printStackTrace(); }
```

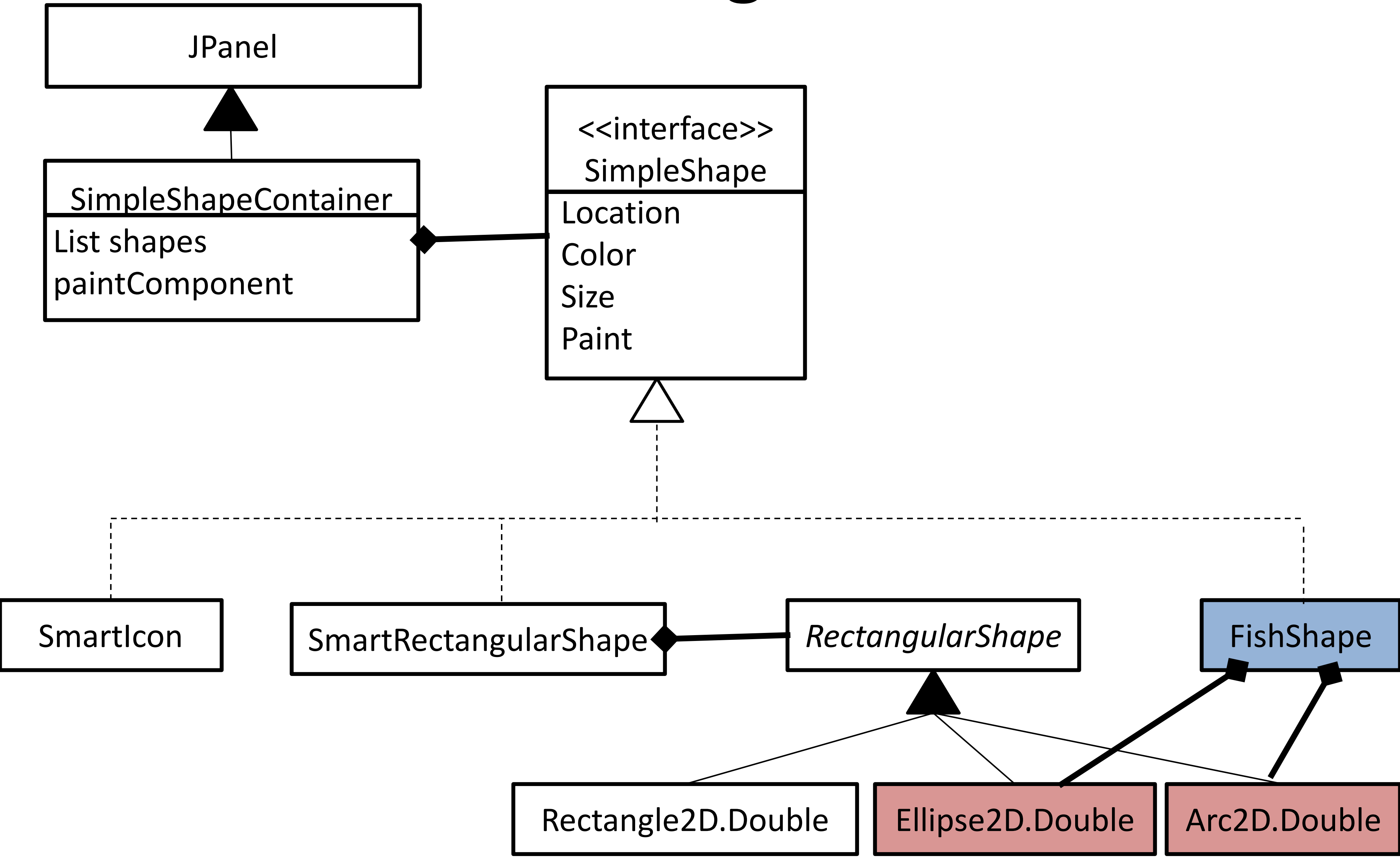
Adding simple shapes (3/3): Text

```
Font font = new Font("Serif", Font.PLAIN, 22);
SmartText textShape=new SmartText("Hello, world",font);
textShape.setLocation(30, 30);
p.addShape(textShape);
```

Creating composite shapes

- To create a composite shape, we use composition
- Composite shape should also implement the SimpleShape interface in order to be used by our animation program

UML diagram 4



FishShape implements SimpleShape

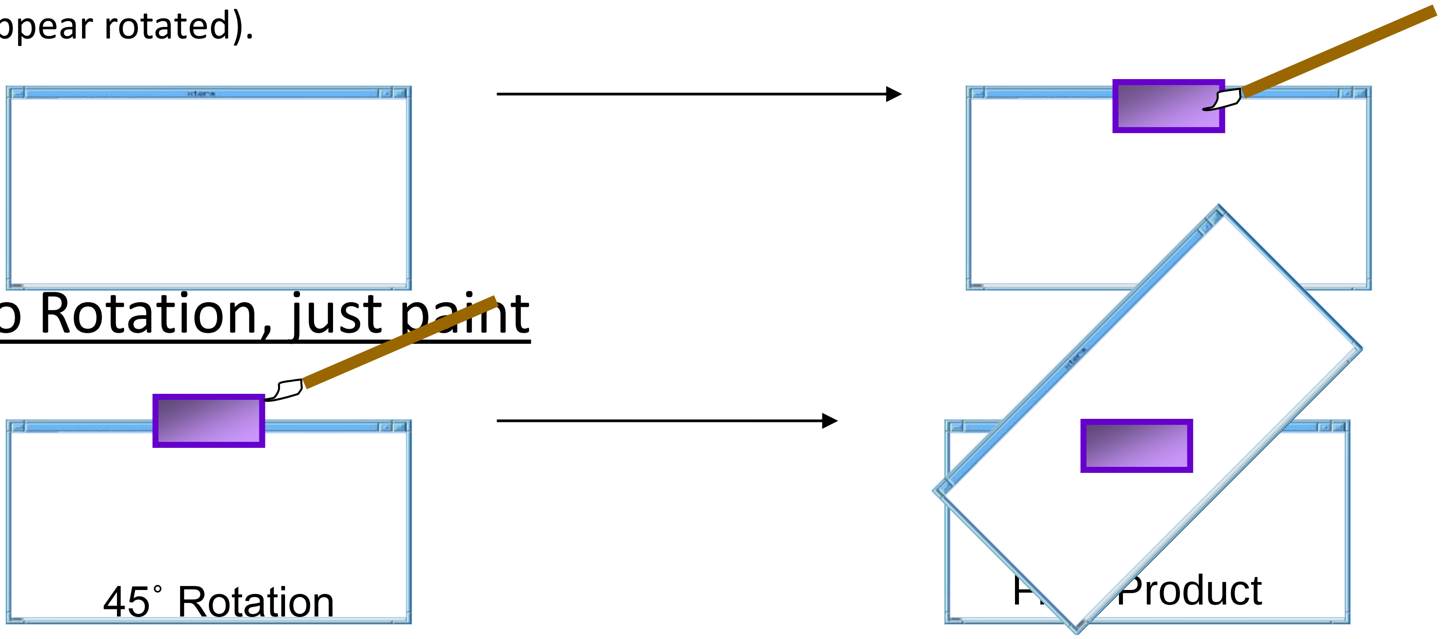
```java
SmartEllipse body=new SmartEllipse();
SmartArc tail=new SmartArc(90,180,Arc2D.Double.PIE);

public FishShape(int width, int height, Color color){
        this.setSize(width, height);
        this.setFillColor(color);
        this.setBorderColor(color);
}

public void setLocation (int posX, int posY){
        this.posX=posX;
        this.posY=posY;
        body.setLocation(posX+TAIL_WIDTH/2, posY);
        tail.setLocation(posX, posY);
}

public void setFillColor(Color color){
        this.fillColor=color;
        body.setFillColor(color);
        tail.setFillColor(color);
}
```

FishShape paint

```java
public void fill(Graphics2D brush){
	brush.fill(this.body.shape);
	brush.fill(this.tail.shape);
}

public void stroke(Graphics2D brush){
	brush.draw(this.body.shape);
	brush.draw(this.tail.shape);
}

public void paint(Graphics2D brush){
	this.stroke(brush);
	this.fill(brush);
}
```

Rotation

- Rotating shapes is easy, but slightly counter-intuitive in Swing…
  - we don't actually rotate the `Shape`
- Instead, tell `Graphics2D brush` to `rotate(…)`
  - give `rotate(…)` a positive rotation angle in radians to give the appearance of clockwise rotation
  - telling the brush to rotate is <u>equivalent to rotating the *canvas*</u> in opposite direction of specified rotation
  - **paint** shape, then tell brush to **un-rotate** (otherwise, we'll keep drawing shapes that appear rotated).



- <u>No Rotation, just paint</u>
- With Rotation of 45 degrees (clockwise)

Rotation

- Need to store rotation angles…
  - `Graphics2D` only understands radians
- Rotate method requires:
  - angle of rotation (**radians**)
  - center point of shape, i.e., point around which to rotate
    - subclasses of `RectangularShape` store center point
- Before drawing the shape, rotate the brush:

```
brush.rotate(rotation, shape.getCenterX(), shape.getCenterY());
```

- Don’t forget to *un-rotate* when you’re finished!

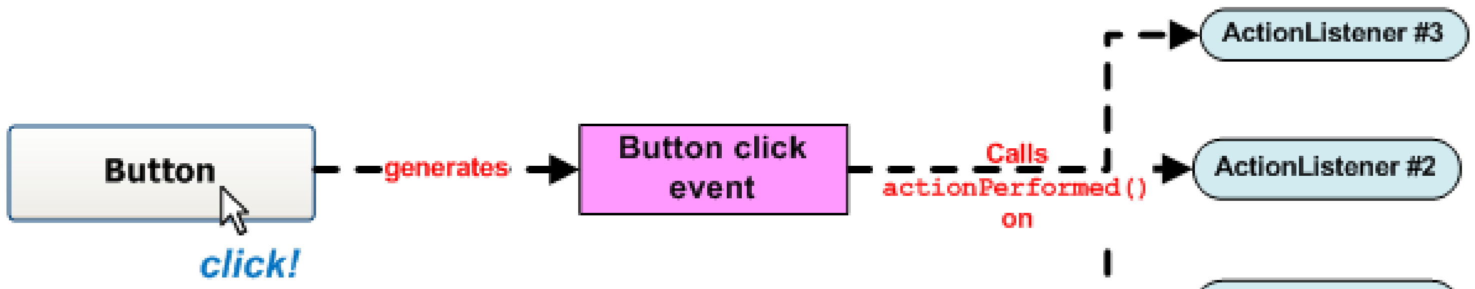
Painting with rotation example

```
public void paint(Graphics2D brush)
{
        if(this.rotation!=0){
                brush.rotate(this.rotation, this.getCenterX(),
                                this.getCenterY());
        }
        this.stroke(brush);
        this.fill(brush);
        if(this.rotation!=0){
                brush.rotate(-this.rotation, this.getCenterX(),
                                this.getCenterY());
        }
}
```

Animation

- Simple animation: redrawing simple shapes at different locations
- We need a timer to cause shape to change its location

Java event model: the Observer design pattern



- In order to handle an event, ActionListeners should register with the source of the event:

addEventListener(new ActionListener #1());

javax.swing.Timer

```
Timer timer=new Timer(100, new ActionListener1());
```

Recurrent interval in milliseconds

```
timer.addActionListener(new ActionListener2());
timer.start();
```

Bouncing Ball type event listener

```
public class BouncingBallAnimationListener implements ActionListener {

          private SimpleShape shape;
          private int maxBoundX, maxBoundY;
          private int step;

          JFrame window;

          private int signX=1, signY=1;

          BouncingBallAnimationListener(SimpleShape shape, int step, JFrame window ){
                    …
          }

          public void actionPerformed(ActionEvent e) {
          if(shape==null) return;
          if(shape.getX()+shape.getWidth()+this.step >= this.maxBoundX)
                    signX=-1;
          else if (shape.getX() -this.step<=0)
                    signX=1;
          if(shape.getY()+shape.getHeight()+this.step >= this.maxBoundY)
                    signY=-1;
          else if (shape.getY() -this.step<=0)
                    signY=1;

          shape.changeLocation(signX*step, signY*step);

          window.repaint();
      }
```

Swimming fish type event listener

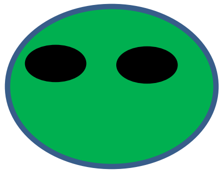
```
public void actionPerformed(ActionEvent e) {

      if(shape==null)
            return;
      if(shape.getX()+shape.getWidth()+this.step >= this.maxBoundX){
            signX=-1;
            shape.setRotation(Math.PI);
      }
      else if (shape.getX() -this.step<=0){
            signX=1;
            shape.setRotation(0);
      }

      shape.changeLocation(signX*step, 0);

      window.repaint();
   }
```

Assignment 4 (5 points)

Create a simple cartoon with animation

Some ideas:

- Falling snow
- Building a snowman
- Rainy day
- Flying bird
- Ship in the sea
- Aliens



The program should include the possibility to start and to stop the animation