Introduction to

Object Oriented Programming

Tutor: Li-Ping (Rita) Kuo
Office: Cramer 226
Phone: Ext. 5195
E-mail: rita@cs.nmt.edu

Graphics Programming

Graphical User Interface (GUI)

- Usage
  - Presents a user-friendly mechanism for interacting with an application
  - Gives an application a distinctive “look-and-feel”
  - Are built from GUI components - **controls** or **widgets**
- GUI component
  - An object with which the user interacts via the mouse, the keyboard or another form of input (e.g., voice recognition)
  - Java: **Swing** GUI components (from **`javax.swing`** package)

Swing

- Abstract Window Toolkit (AWT)
  - For basic GUI programming
  - Deals with user interface elements by delegating their creation and behavior to the native GUI toolkit on each target platform (Windows, Solaris, Macintosh, etc)
- Disadvantages in AWT
  - User interface elements such as menus, scrollbars, and textfields can have subtle differences in the behavior on different platforms
  - The application built with the AWT simply did not look as nice as native Windows or Macintosh application, nor did they have the kind of functionality that users of those platforms had come to expect
  - There were different bugs in the AWT user interface library on the different platform

Swing

- Internet Foundation Classes (IFC)
  - Developed by Netscape in 1996
  - User interface elements, such as buttons, were **painted** onto blank windows
  - The only functionality required from the underlying windowing system was a way to put up windows and to paint on the window
- Swing
  - Was available as an extension to Java 1.1 (`javax.swing.*`)
  - Became a part of the standard library in Java SE 1.2

Swing

- Benefits of Swing
  - Has a rich and convenient set of user interface elements
  - Has few dependencies on the underlying platform; it is therefore less prone to platform-specific bugs
  - Gives a consistent user experience across platform
- AWT and Swing
  - Swing is not a complete replacement for the AWT
    → it is built on top of the AWT architecture
  - Whenever you write a Swing program, you use the foundations of the AWT
  - Swing: painted user interface classes
  - AWT: The underlying mechanisms of the windowing toolkit

Frame

- Definition
  - A top-level window (a window that is not contained inside another window)
  - Swing class: **`JFrame`**, extends the **`Frame`** class
- Example:

```
public class FrameTest extends JFrame {
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;
    public FrameTest() {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    }
    public static void main(String[] args) {
        FrameTest f = new FrameTest();
        f.setTitle("Frame Test");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // f.setSize(300, 200);
        f.setVisible(true);
    }
}
```

Frame

- Example:
  - **`JFrame`** Class
    - An indirect subclass of class **`java.awt.Window`**
    - Provides the basic attributes and behaviors of a window: a title bar at the top, and buttons to minimize, maximize, and close the window
  - **`setDefaultCloseOperation`** method
    - By default, closing a window simply **hides** the window
    - Indicate the program should **terminate** when the window is closed by the user by setting **`JFrame.EXIT_ON_CLOSE`** constant in the **`setDefaultCloseOperation`** method
  - **`setVisible`** method
    - Display the window on the screen with the argument true

Frame

- Frame Properties
  - Most of the methods for working with the size and position of a frame come from the superclasses, such as **`awt.Frame`** and **`awt.Window`**
  - Important methods:
    - **`setLocation`** and **`setBounds`**: setting the position of the frame
    - **`setIconImage`**: display the icon in the title bar, task switcher window, and so on.
    - **`setTitle`**: changing the text in the title bar
    - **`setResizable`**: determine if a frame will be resizable by the user
    - **`setLocationByPlatform`**: if **`true`**, the window system picks the location (but not the size) typically with a slight offset from the last window

Frame

- Frame Properties
  - **`Toolkit`** class in the **`awt`** package
    - Is a dumping ground for a variety of methods interfacing with the native windowing system
    - Methods in the Toolkit class
    - **`getScreenSize`**: return the screen size as a Dimension object
  - Dimension class in the **`awt`** package
    - Store a width and a height in **public** instance variable **`width`** and **`height`**
  - **`ImageIcon`** class in the **`swing`** package
    - An implementation of the **`Icon`** interface that paints **`Icons`** from Images.
    - Images that are created from a URL, filename or byte array are preloaded using **`MediaTracker`** to monitor the loaded state of the image.

Frame

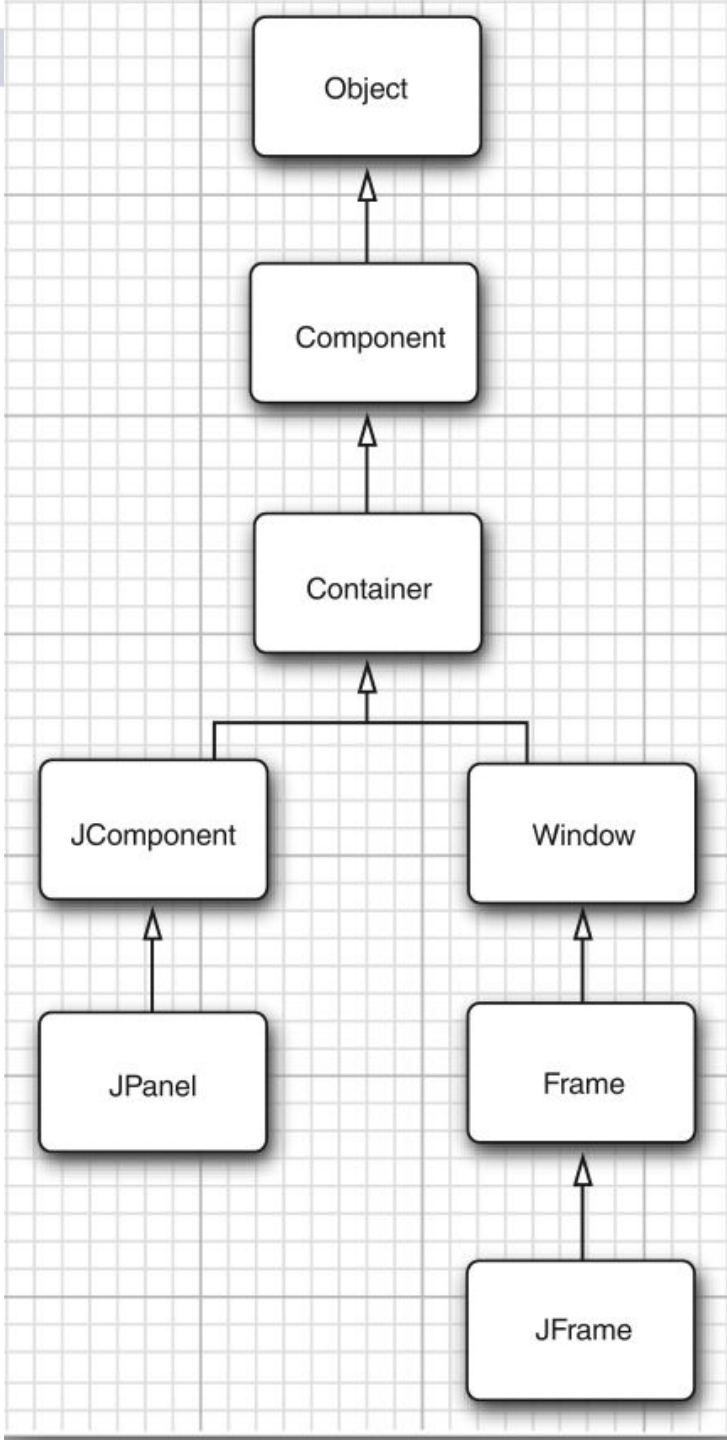
- Frame Properties Example
  - Revise the previous example as follow:

```
public class FrameTest extends JFrame {
    public FrameTest() {
        Toolkit kit = Toolkit.getDefaultToolkit();
        Dimension screenSize = kit.getScreenSize();
        int screenHeight = screenSize.height;
        int screenWidth = screenSize.width;

        setSize(screenWidth / 2, screenHeight / 2);
        setLocationByPlatform(true);

        Image img = new ImageIcon("icon.png").getImage();
        setIconImage(img);
    }
    public static void main(String[] args) {
        FrameTest f = new FrameTest();
        f.setTitle("Frame Test");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
    }
}
```

Components

- Define a new component extends **`JComponent`**
  - Add a new class, **`MsgComponent`**, in the previous example:

```
class MsgComponent extends JComponent {
    public void paintComponent(Graphics g) {
        g.drawString("Hello World", 0, 10);
    }
    public Dimension getPreferredSize() {
        return new Dimension(300, 200);
    }
}
```



Components

- Define a new component extends **`JComponent`**
  - The **`paintComponent`** method
    - Each time a window needs to be **redrawn**, the **`paintComponent`** methods of all components will be executed
    - Do **not** call the **`painComponent`** method by yourself; it is called **automatically**.
  - **`Graphic`** object
    - Measurement for screen display is down in pixel
    - The **`(0, 0)`** coordinate denotes the top left corner of the component on whose surface you are drawing
  - The **`drawString`** method in the **`Graphic`** object
    - Display the text
    - Setup the coordinate where the string start
  - The **`getPreferredSize`** method
    - Tell the users how bit the component would like to be.

Components

- Add the component in the frame
  - Revise the **`FrameTest`** constructor as follow:

```
public class FrameTest extends JFrame {
    public FrameTest() {
        Image img = new ImageIcon("icon.png").getImage();
        setIconImage(img);
        // no setSize method
        add(new msgComponent());
        pack();
    }
    public static void main(String[] args) {
        ...
    }
}
```

  - The **`pack`** method inherited from the Window class
    - Window to be sized to fit the preferred size and layouts of its subcomponents

Using Fonts

- Use **`Font`** object to specify the character fonts
  - The constructor:

    `Font (java.lang.String name, int style, int size)`
  - The font face name:
    - Five logical font names defined in AWT
      - **`SansSerif`**
      - **`Serif`**
      - **`Monospaced`**
      - **`Dialog`**
      - **`DialogInput`**
    - Are always mapped to some fonts that actually exist on the client machine
      - **`Sanserif`** → **`Arial`** in Windows

Using Fonts

- Use **`Font`** object to specify the character fonts
  - The constructor:

    ```
    Font (java.lang.String name, int style, int size)
    ```
  - The style:
    - **`Font.PLAIN`**
    - **`Font.BOLD`**
    - **`Font.ITALIC`**
    - **`Font.BOLD + Font.ITALIC`**

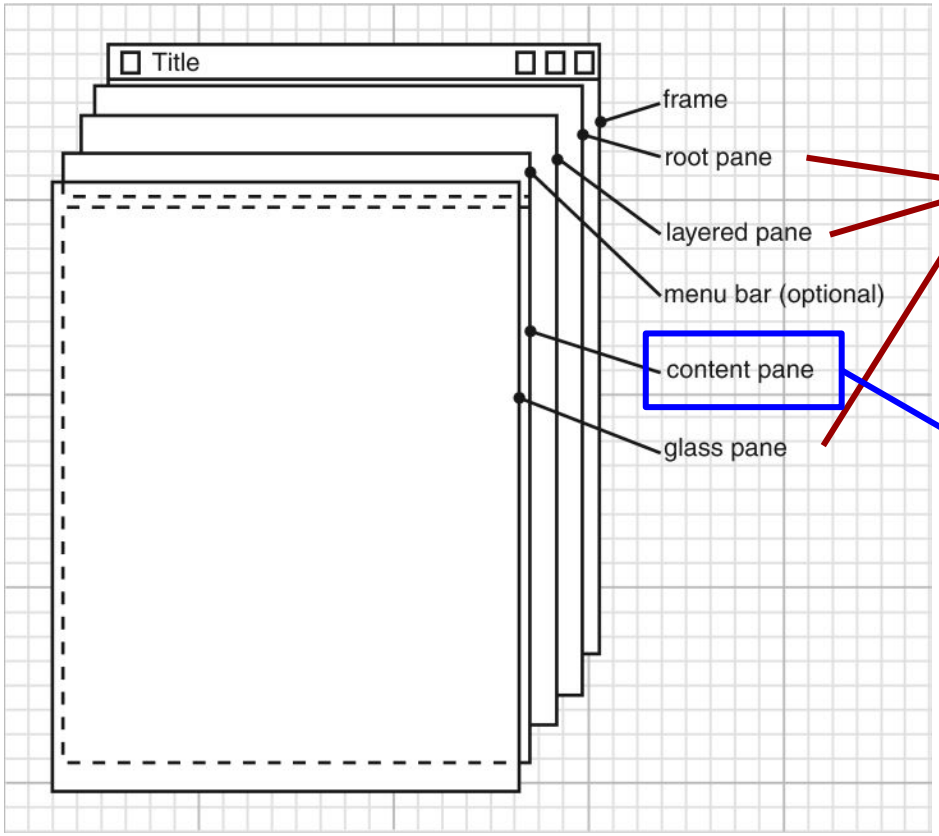
Using Fonts

- Use **`setFont`** method in **`Graphics2D`** object
  - Use **`Graphic2D`** object for drawing the string
  - Use **`setFont`** method to setup the font
  - Revise the location where to start the string
  - Revise the component's preferred size

```
class MsgComponent extends JComponent {
    public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        Font f = new Font("Serif", Font.BOLD, 36);
        g2.setFont(f);

        g2.drawString("Hello World", 0, 50);
    }
    public Dimension getPreferredSize() {
        return new Dimension(300, 200);
    }
}
```

Components

- Displaying Information in a Component
  - Frames are really designed to be **containers** for **components**
  - The structure of **`JFrame`**



Components

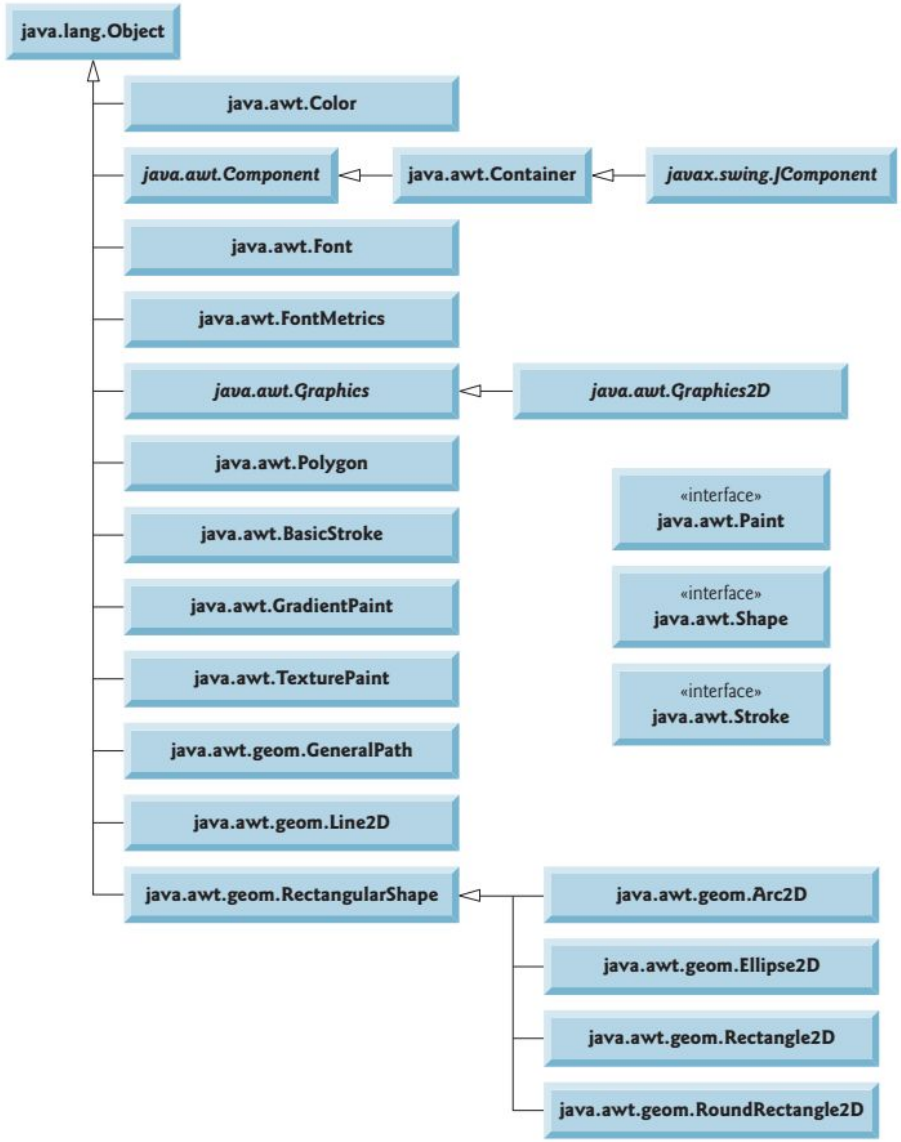
- Add the component in the frame
  - The **`add`** method
    - The traditional way to add components in the frame:

```
Container contentPane = frame.getContentPane();
Component c = ...;
contentPane.add(c);
```

    - If calling **`JFrame.add`** method, there would be an exception occurred
    - Nowadays, the **`JFrame.add`** method has given up trying to reeducate programmers and simplys **`add`** on the content pane

```
Component c = ...;
frame.add(c);
```

Working with 2D Shapes



Working with 2D Shapes

- Use **`Graphics2D`** Object
  - Create a new class, **`DrawComponent`**, in the previous example
  - Use **`Graphic2D`** object for drawing shapes in the Java 2D library
  - Set the component's preferred size is **`400 X 400`**

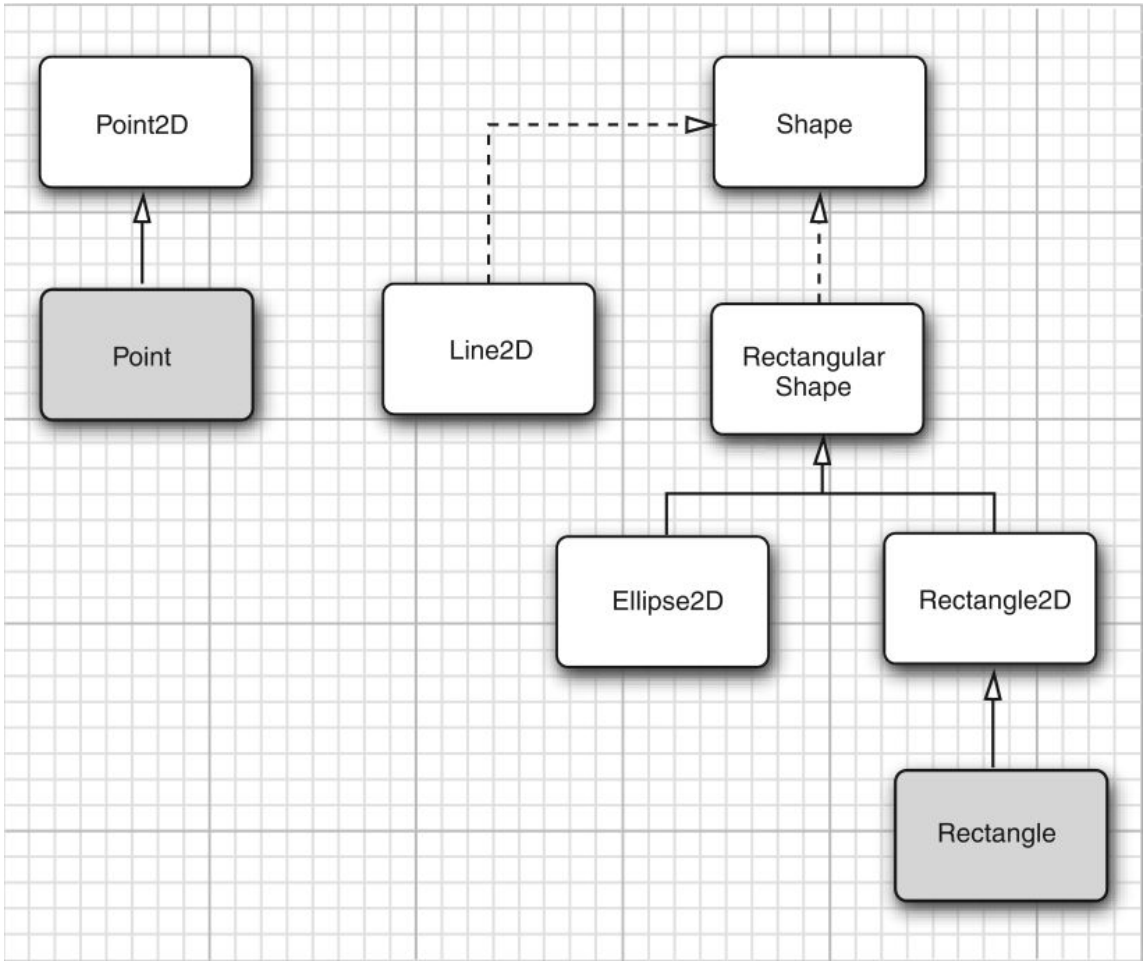
```
class DrawComponent extends JComponent {
    public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        ...
    }
    public Dimension getPreferredSize() {
        return new Dimension(400, 400);
    }
}
```

  - Revise the constructor in **`FrameTest`**

```
   public FrameTest() {
        add(new DrawComponent());
        pack();
    }
```

Working with 2D Shapes

- Shape Classes



Working with 2D Shapes

- Java 2D library - **`Rectangle2D`**
  - Revise the **`paintComponent`** method in the **`DrawComponent`**

```
public void paintComponent(Graphics g) {
     Graphics2D g2 = (Graphics2D) g;

     Rectangle2D rect = new Rectangle2D.Double(100, 100, 200, 150);
     g2.draw(rect);
 }
```

  - Java 2D shapes use floating-point coordinates
  - Supply float and double version for each shape class

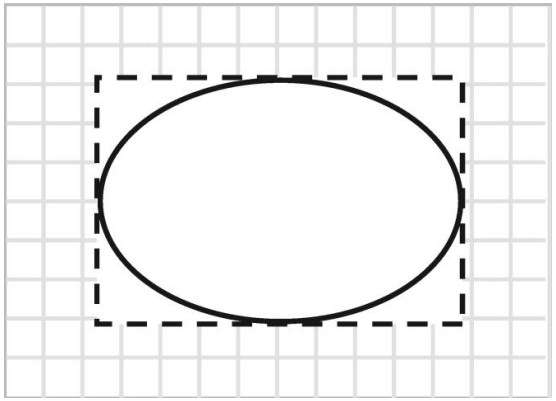
```
Rectangle2D floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);
Rectangle2D doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```

Working with 2D Shapes

- Java 2D library - **`Ellipse2D`**
  - Add an ellipse in the **`paintComponent`** method in the **`DrawComponent`**

```
public void paintComponent(Graphics g) {
      ...
      Ellipse2D ellipse = new Ellipse2D.Double();
      ellipse.setFrame(rect);
      g2.draw(ellipse);
 }
```

  - Ellipses are not rectangular, but they have a bounding rectangle



Working with 2D Shapes

- Java 2D library - **`Ellipse2D`**
  - The **`setFrame`** method
    - `setFrame(double x, double y, double w, double h)`
    - `setFrame(Point2D loc, Dimension2D size)`
    - `setFrame(Rectangle2D r)`
  - The **`setFrameFromCenter`** method
    - `setFrameFromCenter(double centerX, double centerY, double cornerX, double cornerY)`
    - `setFrameFromCenter(Point2D center, Point2D corner)`
  - The **`setFrameFromDiagonal`** method
    - `setFrameFromDiagonal(double x1, double y1, double x2, double y2)`
    - `setFrameFromDiagonal(Point2D p1, Point2D p2)`

Working with 2D Shapes

- Java 2D library - **`Ellipse2D`**
  - Add a circle with the same center in the **`paintComponent`** method in the **`DrawComponent`**

```
public void paintComponent(Graphics g) {
      ...
      double centerX = rect.getCenterX();
      double centerY = rect.getCenterY();
      double radius = 150;

      Ellipse2D circle = new Ellipse2D.Double();
      circle.setFrameFromCenter(centerX, centerY,
                                centerX + radius, centerY + radius);
      g2.draw(circle);
 }
```

Working with 2D Shapes

- Java 2D library - **`Line2D`**
  - Add a line in the **`paintComponent`** method in the **`DrawComponent`**

```
public void paintComponent(Graphics g) {
     ...
     Line2D l = new Line2D.Double(100, 100, 100+200, 100+150);
     g2.draw(l);
 }
```

  - The **`Line2D.Double`** method
    - **`Double(double x1, double y1, double x2, double y2)`**
    - **`Double(Point2D p1, Point2D p2)`**

Using Color

- The **`Color`** Class
  - Static fields
    - **`Color.BLACK`**, **`Color.BLUE`**, **`Color.CYAN`**, etc.
  - Constructors
    - **`Color(float r, float g, float b)`**
    - **`Color(float r, float g, float b, float a)`**
    - **`Color(int rgb)`**
    - **`Color(int rgba, boolean hasalpha)`**

Using Color

- Use **`setPaint`** method in **`Graphics2D`** object
  - Revise the rectangle drawing in **`DrawComponent`** class as follow

```
Rectangle2D rect = new Rectangle2D.Double(100, 100, 200, 150);
g2.setPaint(Color.RED);
g2.draw(rect);
```

- Use **`fill`** method to fill the color in the shape
  - Revise the ellipse drawing in **`DrawComponent`** class as follow

```
Ellipse2D ellipse = new Ellipse2D.Double();
ellipse.setFrame(rect);
g2.setPaint(new Color(200, 200, 255));
g2.fill(ellipse);
g2.setPaint(255);
g2.draw(ellipse);
```

Using Color

- Use **`setStroke`** method in **`Graphics2D`** object
  - Revise the line drawing in **`DrawComponent`** class as follow

```
Line2D l = new Line2D.Double(100, 100, 100+200, 100+150);
g2.setPaint(Color.GREEN);
g2.setStroke(new BasicStroke(10));
g2.draw(l);
```

- Other examples
  - https://docs.oracle.com/javase/tutorial/2d/geometry/strokeandfill.html

Displaying Images

- Use **`drawImage`** method in **`Graphics2D`** object to draw images
  - Create a new class, **`ImageComponent`**, in the previous example with an Image private field and related constructor
  - Set the component's preferred size is **`600 X 400`**
    Use **`ImageIcon`** class to get the image from the file system

```
class ImageComponent extends JComponent {
    private Image image;
    public ImageComponent() {
        image = new ImageIcon("Icon.png").getImage();
    }
    ...
    public Dimension getPreferredSize() {
        return new Dimension(600, 400);
    }
}
```

Displaying Images

- Use **`drawImage`** method in **`Graphics2D`** object to draw images
  - Add an image in the component by adding **`paintComponent`** method with **`drawImage`** method

```
public void paintComponent(Graphics g) {
      if (image == null) return;
      int imageWidth = image.getWidth(this);
      int imageHeight = image.getHeight(this);

      g.drawImage(image, 0, 0, null);
      g.drawImage(image, 0 + imageWidth, 0 + imageHeight, null);
 }
```

  - Revise the constructor in **`FrameTest`**

```
public FrameTest() {
      add(new ImageComponent());
      pack();
 }
```

Displaying Images

- Use **`drawImage`** method in **`Graphics2D`** object to draw images
  - Revise the **`paintComponent`** method with **`copyArea`** method to replace the second **`drawImage`** method

```
public void paintComponent(Graphics g) {
     if (image == null) return;
     int imageWidth = image.getWidth(this);
     int imageHeight = image.getHeight(this);

     g.drawImage(image, 0, 0, null);
     g.copyArea(0, 0, imageWidth, imageHeight,
                imageWidth, imageHeight);
 }
```

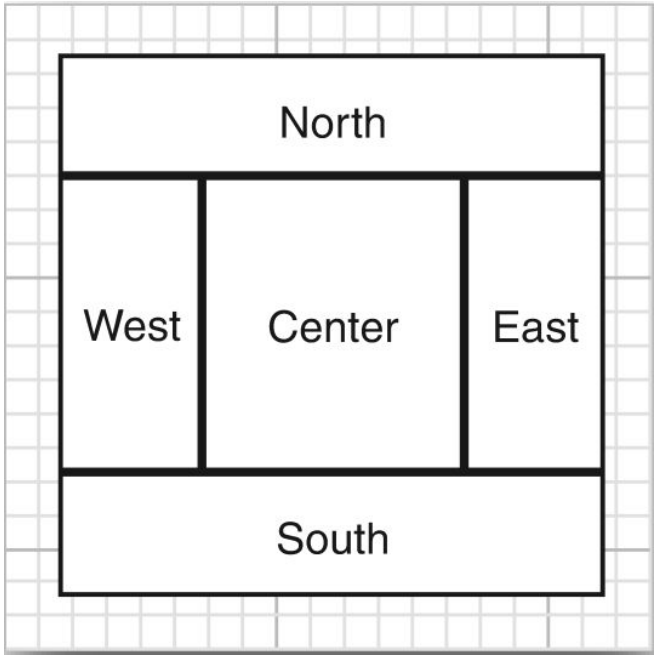
Layout Management

- The Layout Manager
  - Components are placed inside containers
  - A layout manager determines the positions and sizes of components in the container
- Layout managers provided by **`AWT`** and **`Swing`**
  - BorderLayout
  - BoxLayout
  - CardLayout
  - FlowLayout
  - GridBagLayout
  - GridLayout
  - GroupLayout
  - SpringLayout

https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

Layout Management - Border Layout

- How **`BorderLayout`** display components
  - A BorderLayout places components in up to five areas: top, bottom, left, right, and center.
  - All extra space is placed in the center area
  - Is the **default** layout manager of the content pane of every **`JFrame`**



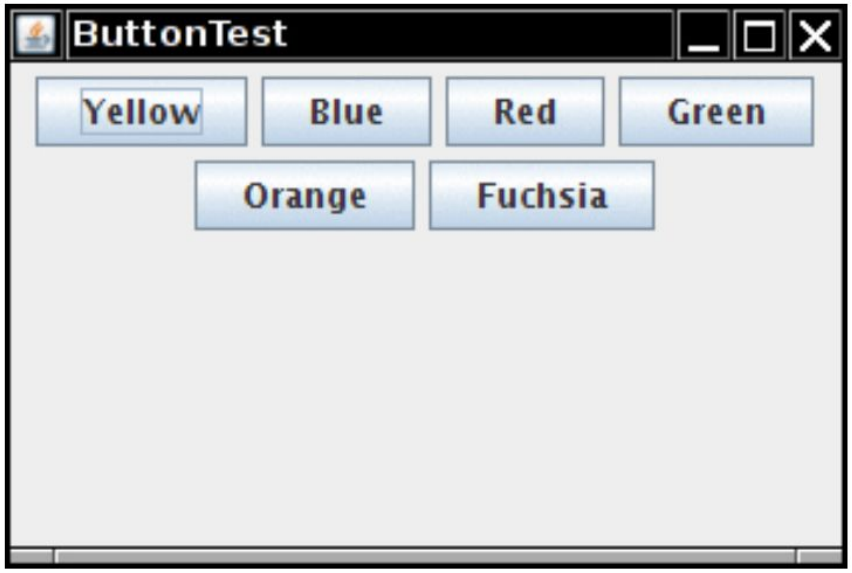
Layout Management - Border Layout

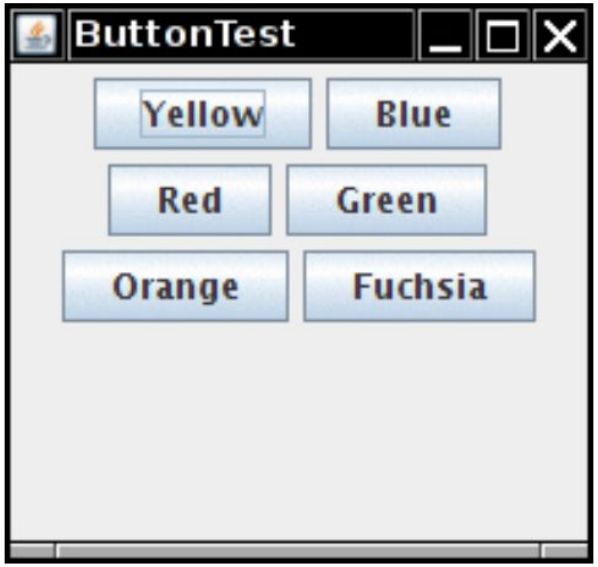
- Use **`BorderLayout`** to display the three components in the previous example
  - Revise the constructor of **`FrameTest`** as follow

```
public FrameTest() {
      add(new MsgComponent(), BorderLayout.WEST);
      add(new DrawComponent(), BorderLayout.NORTH);
      add(new ImageComponent());
      pack();
 }
```

Layout Management - Flow Layout

- How **`FlowLayout`** display components
  - Lay out components in a single row, starting a new row if its container is not sufficiently wide.
  - Is the **default** layout manager for every **`JPanel`**





Layout Management - Flow Layout

- Use **`FlowLayout`** to display the three components in the previous example
  - Revise the constructor of **`FrameTest`** as follow

```
public FrameTest() {
      add(new MsgComponent());
      add(new DrawComponent());
      add(new ImageComponent());
      pack();
 }
```

  - Revise the **`main`** method in the **`FrameTest`** as follow

```
public static void main(String[] args) {
      FrameTest f = new FrameTest();
      f.setLayout(new FlowLayout());
      f.setTitle("Frame Test");
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      f.setVisible(true);
 }
```

Layout Management - Grid Layout

- How **`GridLayout`** display components
  - Arrange all components in rows and columns like a spreadsheet
  - All components are given the same size

Layout Management - Grid Layout

- Use **`GridLayout`** to display the three components in the previous example
  - Keep the constructor of **`FrameTest`** as follow

```
public FrameTest() {
      add(new MsgComponent());
      add(new DrawComponent());
      add(new ImageComponent());
      pack();
 }
```

  - Revise the **`main`** method in the **`FrameTest`** as follow

```
public static void main(String[] args) {
      FrameTest f = new FrameTest();
      f.setLayout(new GridLayout(2, 2));
      f.setTitle("Frame Test");
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      f.setVisible(true);
 }
```

Graphics Programming (cont).

Event Handling

- Is of fundamental importance to programs with a graphical user interface
  - Any operating environment that supports GUIs constantly monitors events such as keystrokes or mouse clicks
  - The operating environment reports these events to the programs that are running
  - Each program then decides what, if anything, to do in response to these events

Event Handling

Microsoft Foundation Classes

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static TCHAR szAppName[] = TEXT ("HelloWin") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("This program requi
                      szAppName, MB_ICONERROR) ;
          return 0 ;
     }
     hwnd = CreateWindow (szAppName,
                          TEXT ("The Hello Program"),
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          NULL,
                          NULL,
                          hInstance,
                          NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}
```

```
                                   // initial x size
                                   // initial y size
                                   // parent window handle
                                   // window menu handle
                                   // program instance handle
                                   // creation parameters
```

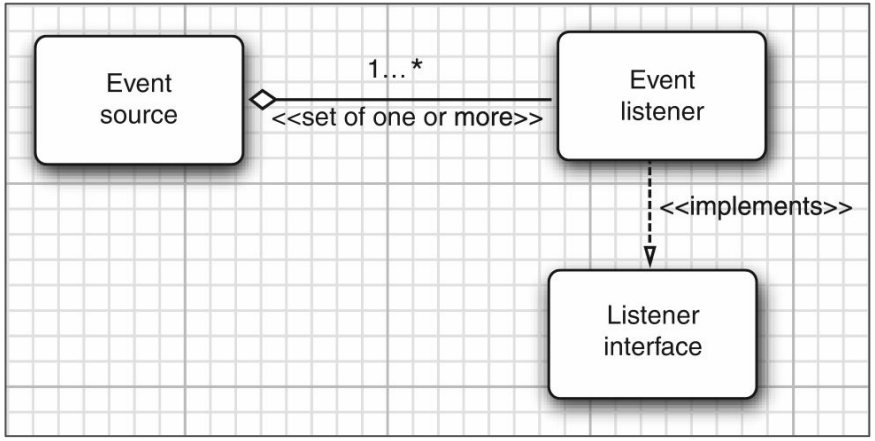
```
while (GetMessage (&msg, NULL, 0, 0))
{
     TranslateMessage (&msg) ;
     DispatchMessage (&msg) ;
}
```

Event Handling

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rect ;

     switch (message)
     {
     case WM_CREATE:
          PlaySound (TEXT ("hellowin.wav"), NULL, SND_FILENAME | SND_ASYNC) ;
          return 0 ;

     case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps) ;

          GetClientRect (hwnd, &rect) ;

          DrawText (hdc, TEXT ("Hello, Windows 98!"), -1, &rect,
                    DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY:
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

Event Handling

- Event Handling in Java
  - A listener object is an instance of a class that implements a special interface called a listener interface



  - An event source is an object that can register listener objects and send them event objects
  - The event source sends out event objects to all registered listeners when that event occurs
  - The listener objects will then use the information in the event object to determine their reaction to the event

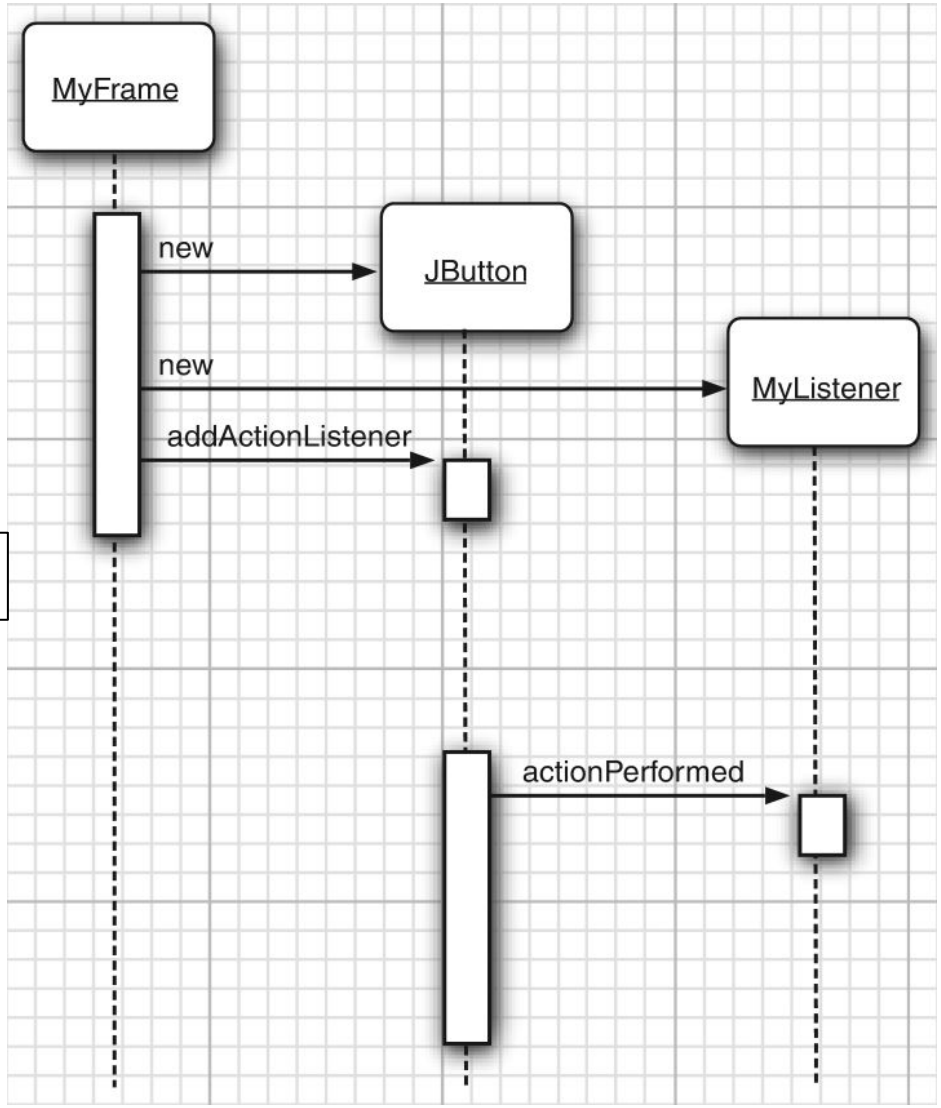
```
ActionListener listener = ...;
JButton button = new JButton("OK");
button.addActionListener(listener);
```

Event Handling

- Event Notification
  - Whenever the user click the button, the **`JButton`** object creates an **`ActionEvent`** object and calls

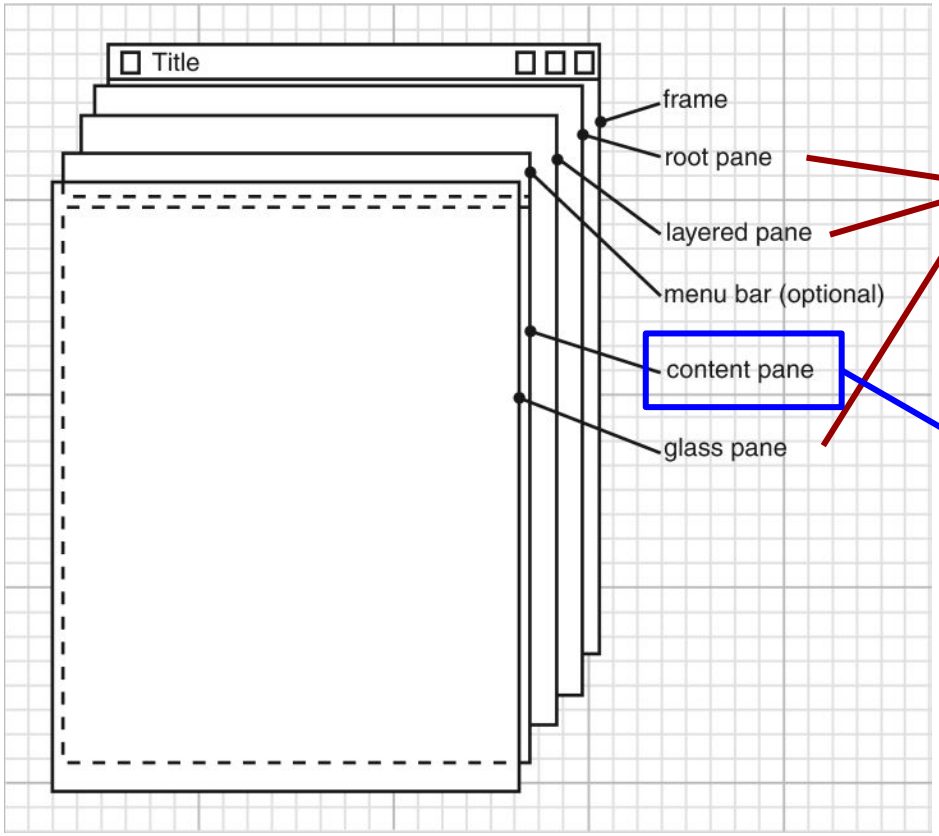
```
listener.actionPerformed(event)
```

passing that even object



Review - Components

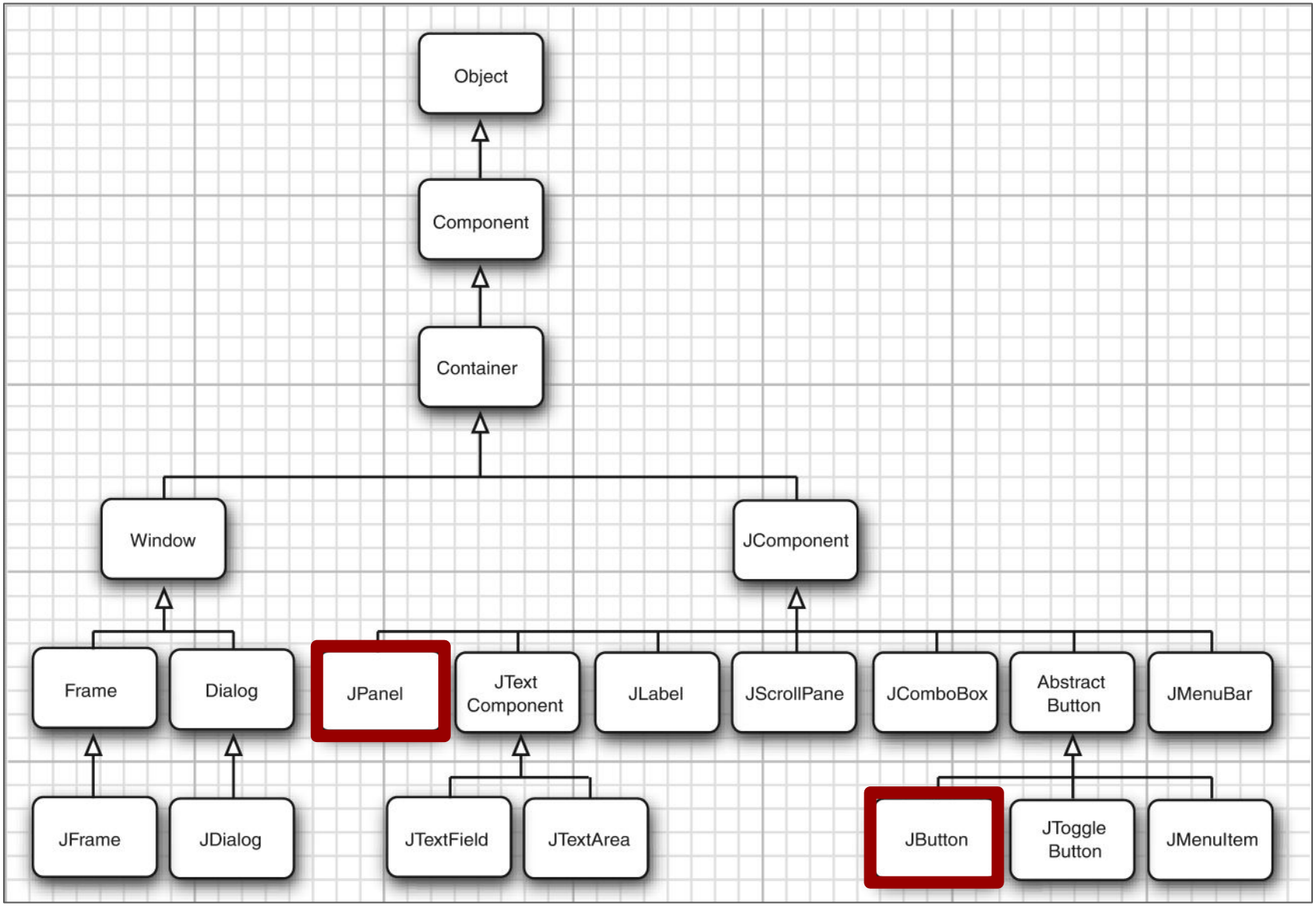
- Displaying Information in a Component
  - Frames are really designed to be **containers** for **components**
  - The structure of JFrame



- Organize the menu bar and content pane
- Implement the look-and-feel

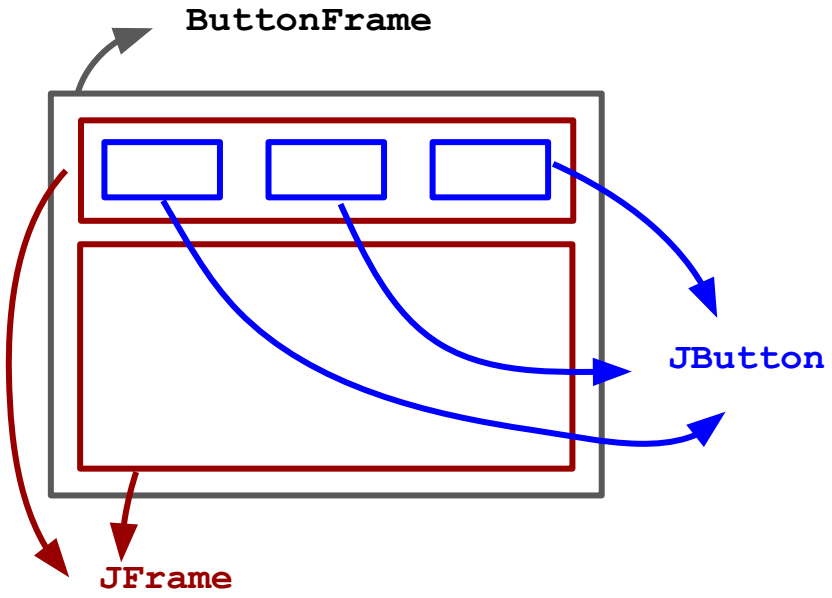
Add components into the content pane

Inheritance Hierarchy for the `Component`



Event Handling - Example

- Change the panel color based on the button click event



Event Handling - Example

- Create **`ButtonFrame`** class and its **`main`** method

```
public class ButtonFrame extends JFrame {
    public static void main(String[] args) {
        ButtonFrame f = new ButtonFrame();
        f.setTitle("Button Test");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
    }
}
```

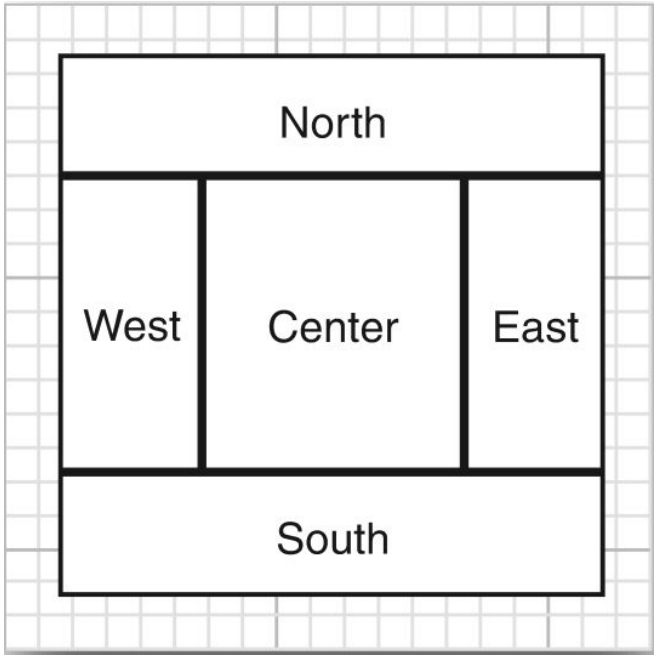
Event Handling - Example

- Define the component layouts in the constructor

```
public class ButtonFrame extends JFrame {
    private JPanel buttonPanel;
    private JPanel contentPanel;
    public ButtonFrame() {
        setSize(300, 200);

        JButton yellowButton = new JButton("Yellow");
        JButton blueButton = new JButton("Blue");
        JButton redButton = new JButton("Red");

        buttonPanel = new JPanel();
        contentPanel = new JPanel();

        buttonPanel.add(yellowButton);
        buttonPanel.add(blueButton);
        buttonPanel.add(redButton);

        add(buttonPanel, BorderLayout.NORTH);
        add(contentPanel);
    }
}
```

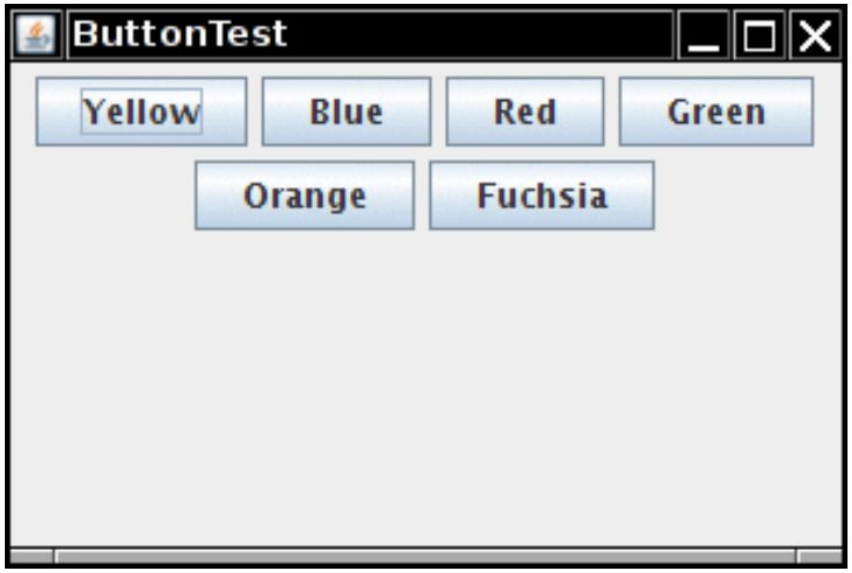
Review - Layout Management - Border Layout

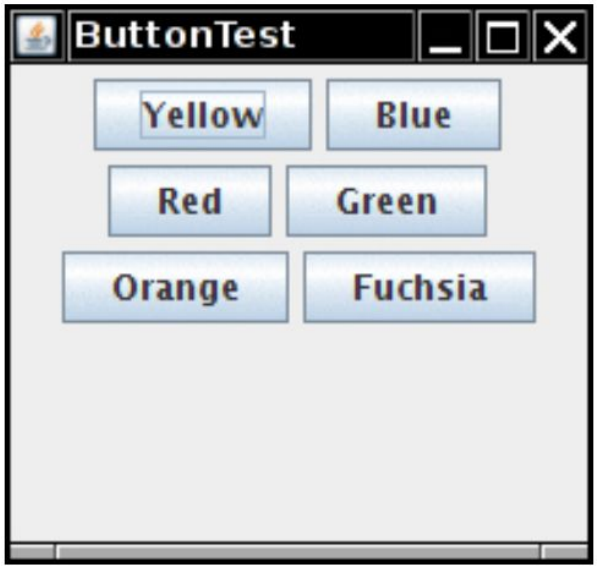
- How **`BorderLayout`** display components
  - A BorderLayout places components in up to five areas: top, bottom, left, right, and center.
  - All extra space is placed in the center area
  - Is the **default** layout manager of the content pane of every **`JFrame`**



Review - Layout Management - Flow Layout

- How **`FlowLayout`** display components
  - Lay out components in a single row, starting a new row if its container is not sufficiently wide.
  - Is the **default** layout manager for every **`JPanel`**





Event Handling - Example

- Add **`ActionListener`**, **`ColorAction`**, as an inner class in the **`ButtonFrame`** class

```
public class ButtonFrame extends JFrame {
    ...
    private class ColorAction implements ActionListener {
        private Color bgColor;
        public ColorAction(Color c) {
            bgColor = c;
        }
        public void actionPerformed(ActionEvent event) {
            contentPanel.setBackground(bgColor);
        }

    }
    ...
}
```

Inner Class

- Definition
  - A class that is defined inside another
- Usage
  - Inner class methods can access the data from the scope in which they are defined, including the data that would otherwise be private
  - Inner classes can be hidden from other classes in the same package
  - Anonymous inner classes are handy when you want to define callbacks without writing a lot of code

Event Handling - Example

- Associate actions with buttons by revising the constructor of the **`ButtonFrame`** class

```
public class ButtonFrame extends JFrame {
    ...
    public ButtonFrame() {
        ...

        ColorAction yellowAction = new ColorAction(Color.YELLOW);
        ColorAction blueAction = new ColorAction(Color.BLUE);
        ColorAction redAction = new ColorAction(Color.RED);

        yellowButton.addActionListener(yellowAction);
        blueButton.addActionListener(blueAction);
        redButton.addActionListener(redAction);
    }
    ...
}
```

Review - Lambda Expression

- Syntax
  - `(parameterList) -> {statements}`

- Example:
  - `(int x, int y) -> {return x + y;}`
- Variations
  - `(x, y) -> {return x + y;}`
    Omit the parameter types: the compiler determines the parameter and return types by the lambda's context
  - `(x, y) -> x + y`
    Omit the return keyword and the curly braces: contains only one expression. The expression's value is implicitly returned

Review - Lambda Expression

- Syntax
  - `(parameterList) -> {statements}`

- Variations
  - `value -> System.out.printf("%d", value)`
    Omit the parentheses: the parameter list contains only one parameter
  - `() -> System.out.println("Welcome!")`
    A lambda with an empty parameter list

Event Handling - Example

- Revise the action listeners associated to the buttons as lambda expression

```
public class ButtonFrame extends JFrame {
    ...
    public ButtonFrame() {
        ...
        /*ColorAction yellowAction = new ColorAction(Color.YELLOW);
        ColorAction blueAction = new ColorAction(Color.BLUE);
        ColorAction redAction = new ColorAction(Color.RED);

        yelloButton.addActionListener(yellowAction);
        blueButton.addActionListener(blueAction);
        redButton.addActionListener(redAction);*/
        yellowButton.addActionListener(event ->
                        contentPanel.setBackground(Color.YELLOW));
        blueButton.addActionListener(event ->
                        contentPanel.setBackground(Color.BLUE));
        redButton.addActionListener(event ->
                        contentPanel.setBackground(Color.RED));
    }
    ...
}
```

Event Handling - Example

- Use only one action listener in the previous example
  - Make three buttons as private fields

```
public class ButtonFrame extends JFrame {
    private JPanel buttonPanel;
    private JPanel contentPanel;
    private JButton yellowButton = new JButton("Yellow");
    private JButton blueButton = new JButton("Blue");
    private JButton redButton = new JButton("Red");
    ...
}
```

Event Handling - Example

- Use only one action listener in the previous example
  - Create a new inner class, **`AllColorAction`**

```
private class AllColorAction implements ActionListener {
    public void actionPerformed (ActionEvent event) {
        Object source = event.getSource();
        if (source == yellowButton) {
            contentPanel.setBackground(Color.YELLOW);
        } else if (source == blueButton) {
            contentPanel.setBackground(Color.BLUE);
        } else if (source == redButton) {
            contentPanel.setBackground(Color.RED);
        }
    }
}
```

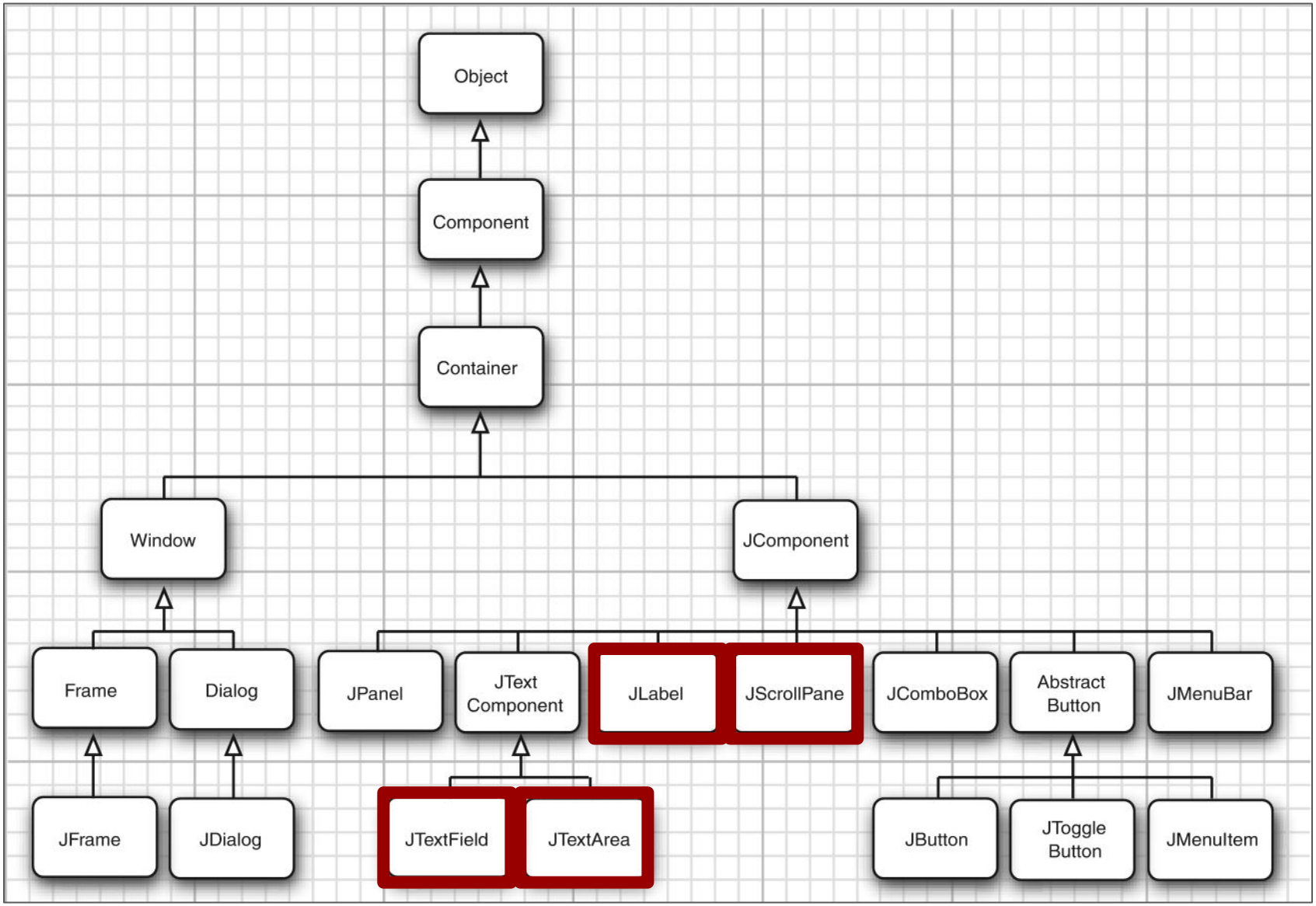
Event Handling - Example

- Use only one action listener in the previous example
  - Revise the association between buttons and action listeners as follow:

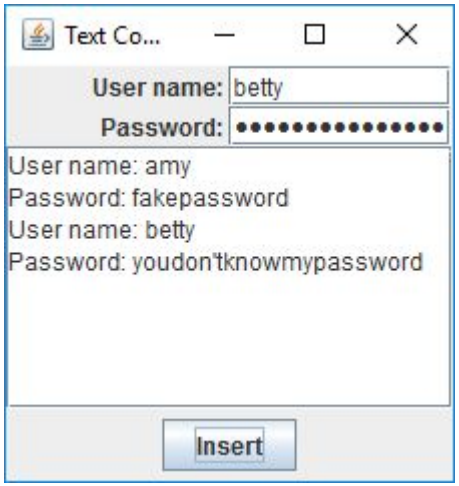
```
public class ButtonFrame extends JFrame {
    public ButtonFrame() {
        ...

        AllColorAction allAction = new AllColorAction();

        yellowButton.addActionListener(allAction);
        blueButton.addActionListener(allAction);
        redButton.addActionListener(allAction);
    }
    ...
}
```

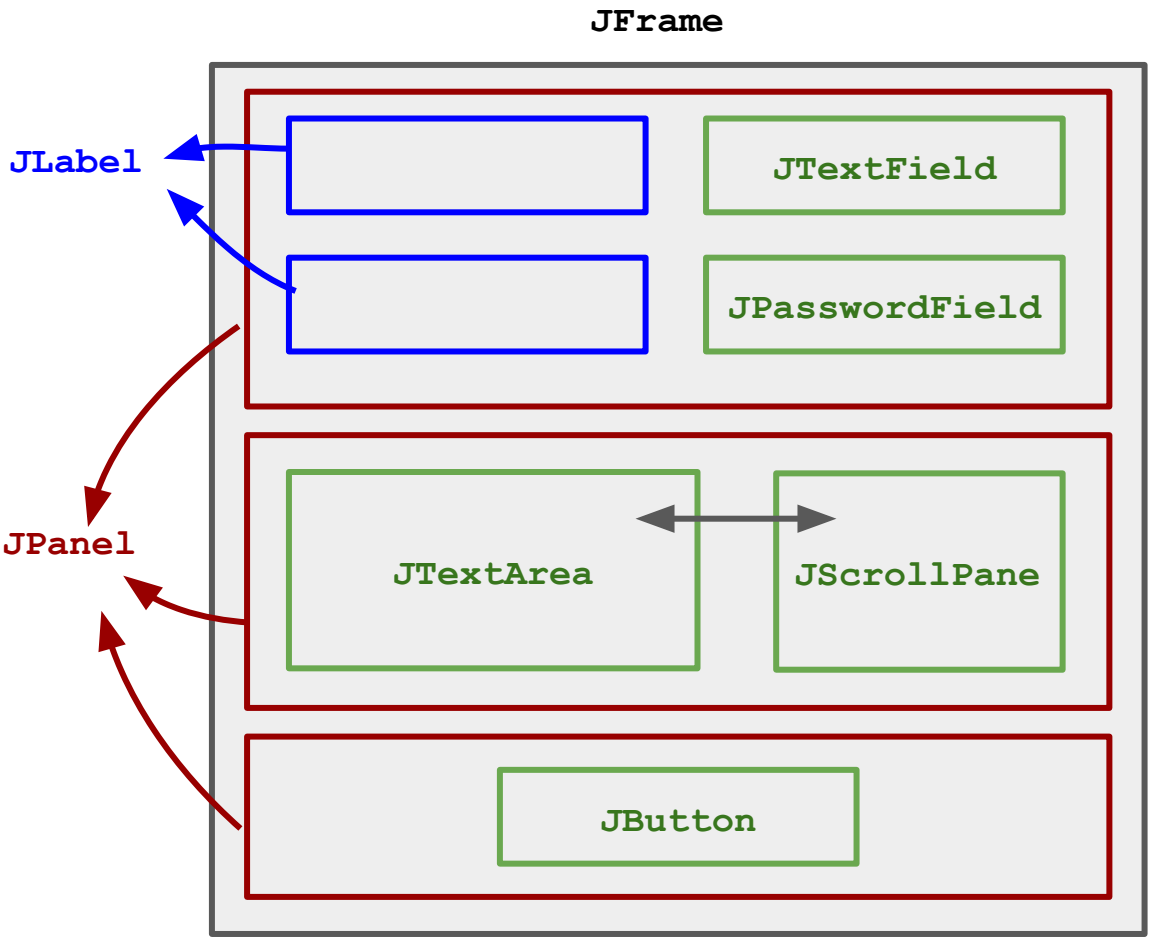
Graphics Programming (cont).

Review - Inheritance Hierarchy for the `Component`



Text Components Introduction





Text Components Introduction

- Create the frame layout in the new class
  - Create a new class, **`TextComponentFrame`**, which is the subclass of **`JFrame`** and set up the **`main`** method in the class

```
public class TextComponentFrame extends JFrame {
    public TextComponentFrame() {
        ...
        pack();
    }
    public static void main(String[] args) {
        TextComponentFrame f = new TextComponentFrame();
        f.setTitle("Text Component Frame");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
    }
}
```

Text Components Introduction

- Create the frame layout in the new class
  - Set up the north panel in the frame constructor

```
public TextComponentFrame() {
    JTextField textField = new JTextField();
    JPasswordField passwordField = new JPasswordField();

    JPanel northPanel = new JPanel();

    northPanel.setLayout(new GridLayout(2, 2));
    northPanel.add(new JLabel("User name: ", SwingConstants.RIGHT));
    northPanel.add(textField);
    northPanel.add(new JLabel("Password: ", SwingConstants.RIGHT));
    northPanel.add(passwordField);

    add(northPanel, BorderLayout.NORTH);

    ...
}
```

Text Components Introduction

- **`JLabel`**

`javax.swing.JLabel` 1.2

- `JLabel(String text)`
- `JLabel(Icon icon)`
- `JLabel(String text, int align)`
- `JLabel(String text, Icon icon, int align)`

constructs a label.

Parameters:		
	`text`	The text in the label
	`icon`	The icon in the label
	`align`	One of the `SwingConstants` constants `LEFT` (default), `CENTER`, or `RIGHT`

- `String getText()`
- `void setText(String text)`

gets or sets the text of this label.

- `Icon getIcon()`
- `void setIcon(Icon icon)`

gets or sets the icon of this label.

Text Components Introduction

- **`JTextField`**

`javax.swing.JTextField` 1.2

- `JTextField(int cols)`

  constructs an empty `JTextField` with the specified number of columns.

- `JTextField(String text, int cols)`

  constructs a new `JTextField` with an initial string and the specified number of columns.

- `int getColumns()`
- `void setColumns(int cols)`

  gets or sets the number of columns that this text field should use.

Text Components Introduction

- **`JPasswordField`**

`javax.swing.JPasswordField` 1.2

- `JPasswordField(String text, int columns)`

  constructs a new password field.

- `void setEchoChar(char echo)`

  sets the echo character for this password field. This is advisory; a particular look-and-feel may insist on its own choice of echo character. A value of `0` resets the echo character to the default.

- `char[] getPassword()`

  returns the text contained in this password field. For stronger security, you should overwrite the content of the returned array after use. (The password is not returned as a `String` because a string would stay in the virtual machine until it is garbage-collected.)

Text Components Introduction

- Create the frame layout in the new class
  - Set up the content panel in the frame constructor

```
public TextComponentFrame() {
    ...

    JTextArea textArea = new JTextArea(8, 20);
    JScrollPane scrollPane = new JScrollPane(textArea);

    add(scrollPane, BorderLayout.CENTER);

    ...
}
```

Text Components Introduction

- **`JTextArea`**

javax.swing.JTextArea 1.2

- `JTextArea()`
- `JTextArea(int rows, int cols)`
- `JTextArea(String text, int rows, int cols)`

  constructs a new text area.
- `void setColumns(int cols)`

  tells the text area the preferred number of columns it should use.
- `void setRows(int rows)`

  tells the text area the preferred number of rows it should use.
- `void append(String newText)`

  appends the given text to the end of the text already in the text area.
- `void setLineWrap(boolean wrap)`

  turns line wrapping on or off.
- `void setWrapStyleWord(boolean word)`

  If `word` is `true`, long lines are wrapped at word boundaries. If it is `false`, long lines are broken without taking word boundaries into account.
- `void setTabSize(int c)`

  sets tab stops every `c` columns. Note that the tabs aren't converted to spaces but cause alignment with the next tab stop.

Text Components Introduction

- `JScrollPane`

javax.swing.JScrollPane 1.2
• JScrollPane(Component c) creates a scroll pane that displays the content of the specified component. Scrollbars are supplied when the component is larger than the view.

Text Components Introduction

- Create the frame layout in the new class
  - Set up the south panel in the frame constructor

```
public TextComponentFrame() {
    ...

    JPanel southPanel = new JPanel();
    JButton insertButton = new JButton("Insert");
    southPanel.add(insertButton);

    add(southPanel, BorderLayout.SOUTH);

    ...
}
```

Text Components Introduction

- Define the button action
  - Use lambda function to define the action

```
public TextComponentFrame() {
    ...

    insertButton.addActionListener(event ->
            textArea.append("User name: " + textField.getText()
                + "\nPassword: "
                + new String(passwordField.getPassword()) + "\n") );

    ...
}
```

Text Components Introduction

- Define the button action
  - Use **anonymous inner class** to define the action

```
public TextComponentFrame() {
    ...

    insertButton.addActionListener(
            new ActionListener() {
                public void actionPerformed (ActionEvent event) {
                    textArea.append("User name: "
                        + textField.getText()
                        + "\nPassword: "
                        + new String(passwordField.getPassword())
                        + "\n");
                }
            }
    );

    ...
}
```

Text Components Introduction

- Define the button action
  - Another way to define the anonymous inner class

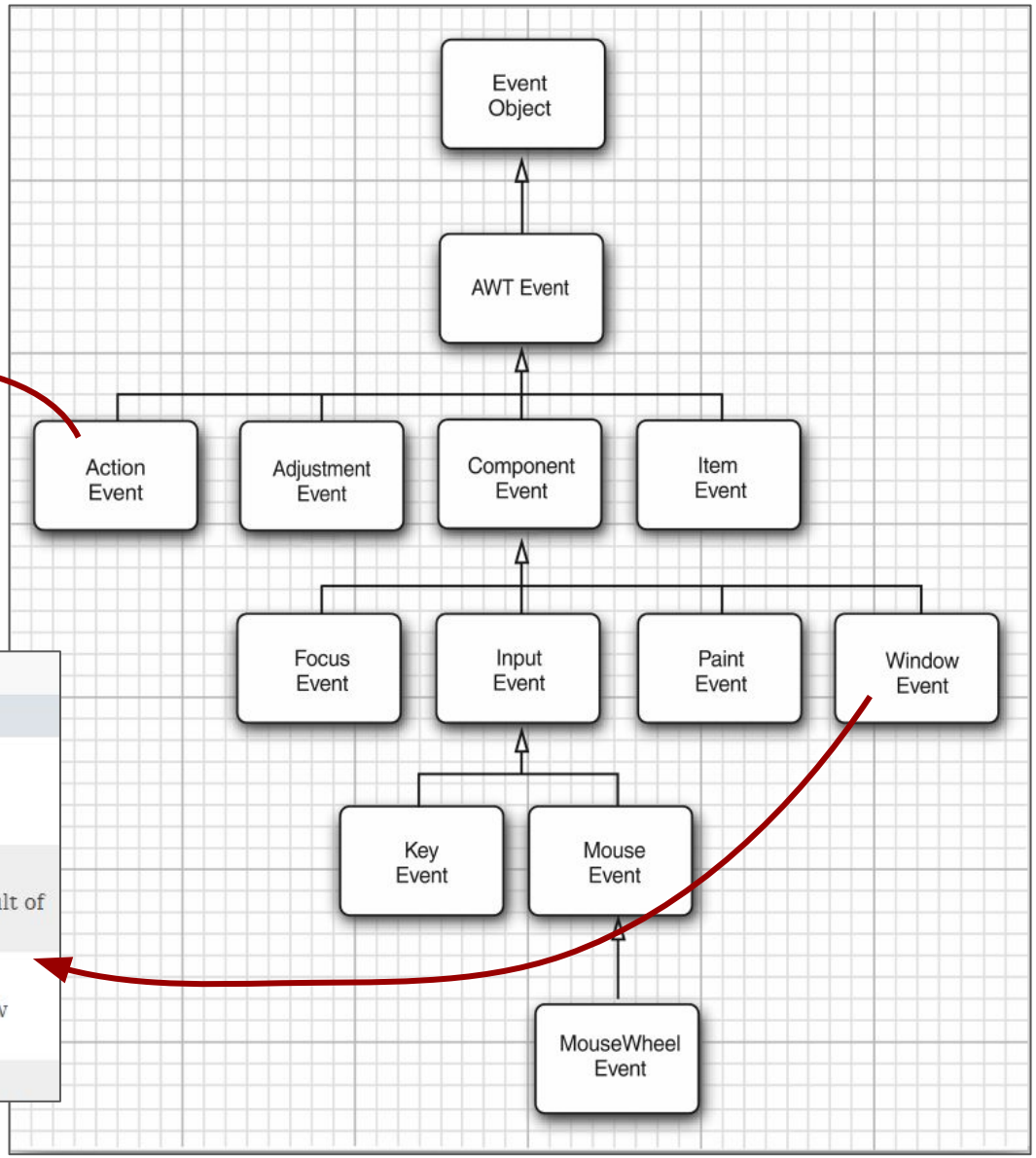
```
public TextComponentFrame() {
    ...

    ActionListener listener = new ActionListener() {
        public void actionPerformed (ActionEvent event) {
            textArea.append("User name: " + textField.getText()
                    + "\nPassword: "
                    + new String(passwordField.getPassword())
                    + "\n");
        }
    };
    insertButton.addActionListener(listener);

    ...
}
```

Adapter Classes

- AWT Event Hierarchy

All Methods	Instance Methods	Abstract Methods
Modifier and Type	**Method and Description**	
void	`actionPerformed(ActionEvent e)` Invoked when an action occurs.	

All Methods	Instance Methods	Abstract Methods
Modifier and Type	**Method and Description**	
void	`windowActivated(WindowEvent e)` Invoked when the Window is set to be the active Window.	
void	`windowClosed(WindowEvent e)` Invoked when a window has been closed as the result of calling dispose on the window.	
void	`windowClosing(WindowEvent e)` Invoked when the user attempts to close the window from the window's system menu.	
void	`windowDeactivated(WindowEvent e)`	



Adapter Classes

- **`WindowListener`** defines seven methods
  - Any class that implements WindowListener has to implement all methods
  - If we are only interested in one of the seven methods, such **`windowClosing`**
    - Write do-nothing functions for the other six methods

```
class Terminator implements WindowListener
{
   public void windowClosing(WindowEvent e)
   {
      if (user agrees)
         System.exit(0);
   }

   public void windowOpened(WindowEvent e) {}
   public void windowClosed(WindowEvent e) {}
   public void windowIconified(WindowEvent e) {}
   public void windowDeiconified(WindowEvent e) {}
   public void windowActivated(WindowEvent e) {}
   public void windowDeactivated(WindowEvent e) {}
}
```

Adapter Classes

- **`WindowListener`** defines seven methods
  - Any class that implements WindowListener has to implement all methods
  - If we are only interested in one of the seven methods, such **`windowClosing`**
    - Write do-nothing functions for the other six methods
    - Use **adapter class**: Implements all the methods in the interface but does nothing with them

```
class Terminator extends WindowAdapter
{
   public void windowClosing(WindowEvent e)
   {
      if (user agrees)
         System.exit(0);
   }
}
```

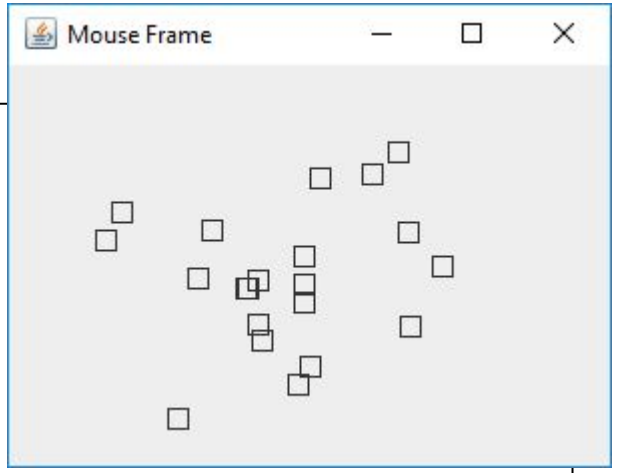
Adapter Classes

- Event-adapter classes and the interface they implement

Event-adapter class in `java.awt.event`	Implements interface
`ComponentAdapter`	`ComponentListener`
`ContainerAdapter`	`ContainerListener`
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`WindowAdapter`	`WindowListener`

Mouse Event

- Create a new class, **`MouseFrame`**, and add a defined **`MouseComponent`** in it




```
public class MouseFrame extends JFrame {
    public MouseFrame() {
        add(new MouseComponent());
        pack();
    }
    public static void main(String[] args) {
        MouseFrame f = new MouseFrame();
        f.setTitle("Mouse Frame");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
    }
}
class MouseComponent extends JComponent {
    public Dimension getPreferredSize() {
        return new Dimension(300, 200);
    }
}
```

Mouse Event

- Define the fields and the constructor of the **`MouseComponent`**

```
class MouseComponent extends JComponent {
    public static final int SIDELENGTH = 10;

    private ArrayList<Rectangle2D> squares;
    private Rectangle2D current;

    public MouseComponent() {
        squares = new ArrayList<>();
        current = null;

        addMouseListener(new MouseHandler());
    }

    ...
}
```

Mouse Event

- Define **`MouseHandler`** which detect **`mousePressed`** event

```
class MouseComponent extends JComponent {
    ...
    private class MouseHandler extends MouseAdapter {
        public void mousePressed(MouseEvent event) {
            add(event.getPoint());
        }
    }
    ...
}
```

Mouse Event

- Define **`add`** method which add a new element in the ArrayList

```
class MouseComponent extends JComponent {
    ...
    public void add(Point2D p) {
        double x = p.getX();
        double y = p.getY();

        current = new Rectangle2D.Double(x - SIDELENGTH / 2,
                                         y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);
        squares.add(current);

        repaint();
    }
    ...
}
```

Mouse Event

- Define the **`paintComponent`** method

```
class MouseComponent extends JComponent {
    ...
    public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        for (Rectangle2D r : squares) {
            g2.draw(r);
        }
    }
    ...
}
```

Graphics Programming (cont).

Animation

- Create a new class, **`TimerFrame`**, which extends **`JFrame`**
  - Setup the **`main`** method as follow

```
public class TimerFrame extends JFrame {
    public static void main(String[] args) {
        TimerFrame f = new TimerFrame();
        f.setTitle("Timer Animation");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
    }
}
```

- Define **`MotionComponent`** which extends **`JComponent`** with dimension **`400 X 400`**

```
class MotionComponent extends JComponent {
    public Dimension getPreferredSize() {
        return new Dimension(400, 400);
    }
}
```

Animation

- Add **`MotionComponent`** in the **`TimerFrame`**
  - Create the constructor of **`TimerFrame`** as follow

```
public class TimerFrame extends JFrame {
    public TimerFrame () {
        MotionComponent motionComponent = new MotionComponent();
        add(motionComponent);
        pack();
    }
    ...
}
```

Animation

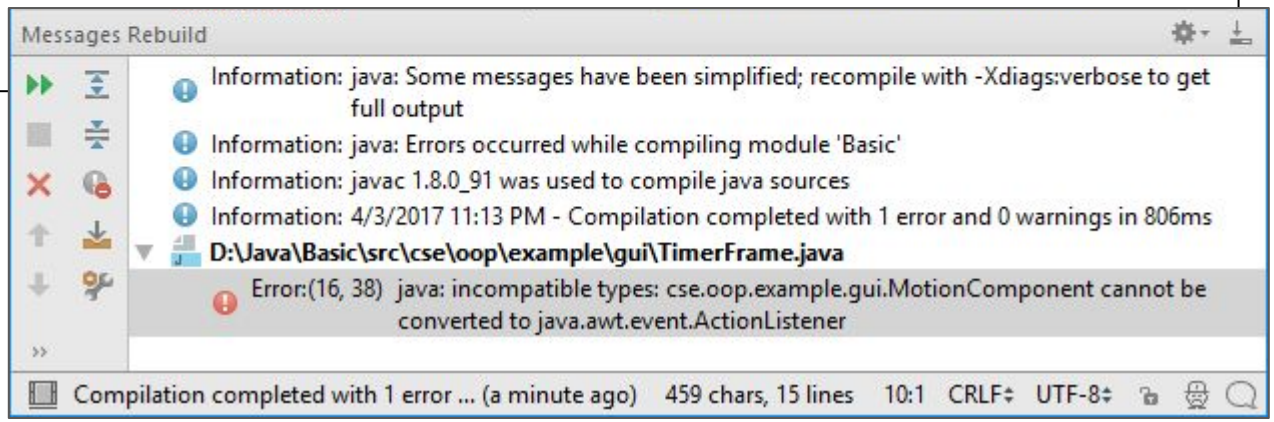
- Define the **`paintComponent`** method in the **`MotionComponent`**
  - Define the fields for the animation
  - Define the **`paintComponent`** method

```
class MotionComponent extends JComponent {
    private int initX = 0;    private int initY = 0;
    private int distX = 0;    private int distY = 0;
    private int iter = 10;
    private int length = 100;

    public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;

        Rectangle2D rect = new Rectangle2D.Double(initX + distX,
                                    initY + distY, length, length);
        g2.setPaint(Color.RED);
        g2.draw(rect);
    }
    ...
}
```

Animation

- Add a timer in the **`TimerFrame`**
  - Create a **`Timer`** object in the **`TimeFrame`** constructor
  - Attach the timer in the **`MotionComponent`**

```
public class TimerFrame extends JFrame {
    public TimerFrame () {
        MotionComponent motionComponent = new MotionComponent();
        add(motionComponent);
        pack();

        Timer timer = new Timer(100, motionComponent);
    }
    ...
}
```



Animation

- Add a timer in the **`TimerFrame`**
  - Create a **`Timer`** object in the **`TimeFrame`** constructor
  - Attach the timer in the **`MotionComponent`**

```
public class TimerFrame extends JFrame {
    public TimerFrame () {
        MotionComponent motionComponent = new MotionComponent();
        add(motionComponent);
        pack();

        Timer timer = new Timer(100, motionComponent);
    }
    ...
}
```

Constructors

Constructor and Description
`Timer(int delay, ActionListener listener)` Creates a Timer and initializes both the initial delay and between-event delay to delay milliseconds.

Animation

- Revise the **`MotionComponent`**
  - Make **`MotionComponent`** implement **`ActionListener`**
  - Add **`actionPerformed`** method
  - Call **`repaint`** method in the **`actionPerformed`** method to call the **`paintComponent`** method when event occurred

```
class MotionComponent extends JComponent implements ActionListener{
    ...
    public void actionPerformed (ActionEvent event) {
        distX += iter;
        distY += iter;
        repaint();
    }
}
```

Animation

- Start the timer in the **`TimerFrame`**
  - Create a **`Timer`** object in the **`TimeFrame`** constructor
  - Attach the timer in the **`MotionComponent`**

```
public class TimerFrame extends JFrame {
    public TimerFrame () {
        MotionComponent motionComponent = new MotionComponent();
        add(motionComponent);
        pack();

        Timer timer = new Timer(100, motionComponent);
        timer.start();
    }
    ...
}
```

Animation - Timer

- **`Timer`** class in the **`swing`** package

`javax.swing.Timer` 1.2

- `Timer(int interval, ActionListener listener)`
  constructs a timer that notifies `listener` whenever `interval` milliseconds have elapsed.
- `void start()`
  starts the timer. Once started, the timer calls `actionPerformed` on its listeners.
- `void stop()`
  stops the timer. Once stopped, the timer no longer calls `actionPerformed` on its listeners.

Animation

- Other resources
  - http://www.java2s.com/Code/Java/2D-Graphics-GUI/Animation.htm