Java GUI

Windows

Events

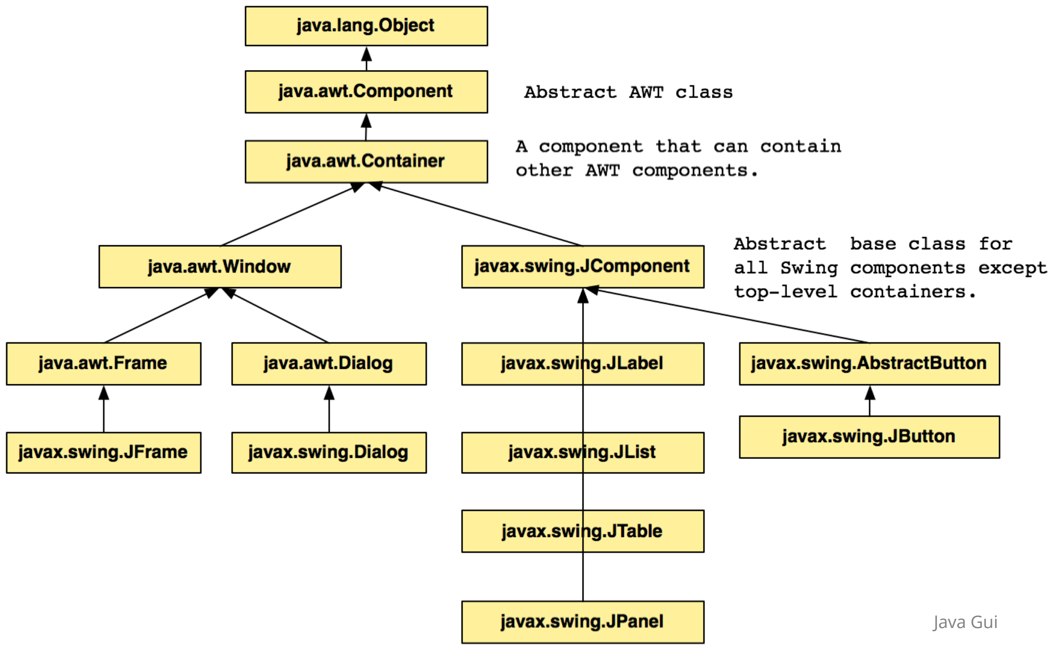
Drawing

Java GUI Toolkits

Toolkit	Description
AWT	“Heavyweight” with platform-specific widgets. AWT applications were limited to common-functionality that existed on all platforms.
Swing	“Lightweight”, full widget implementation. Commonly used and deployed cross-platform.
Standard Window Toolkit / SWT	”Heavyweight” hybrid model: native, and tied to specific platforms. Used in Eclipse.
Java FX	Intended for rich desktop + mobile apps. Still in development.

Swing Component Hierarchy

- java.awt.Window is the base for all containers.
- javax.swing.Jcomponent is the root for all widgets.

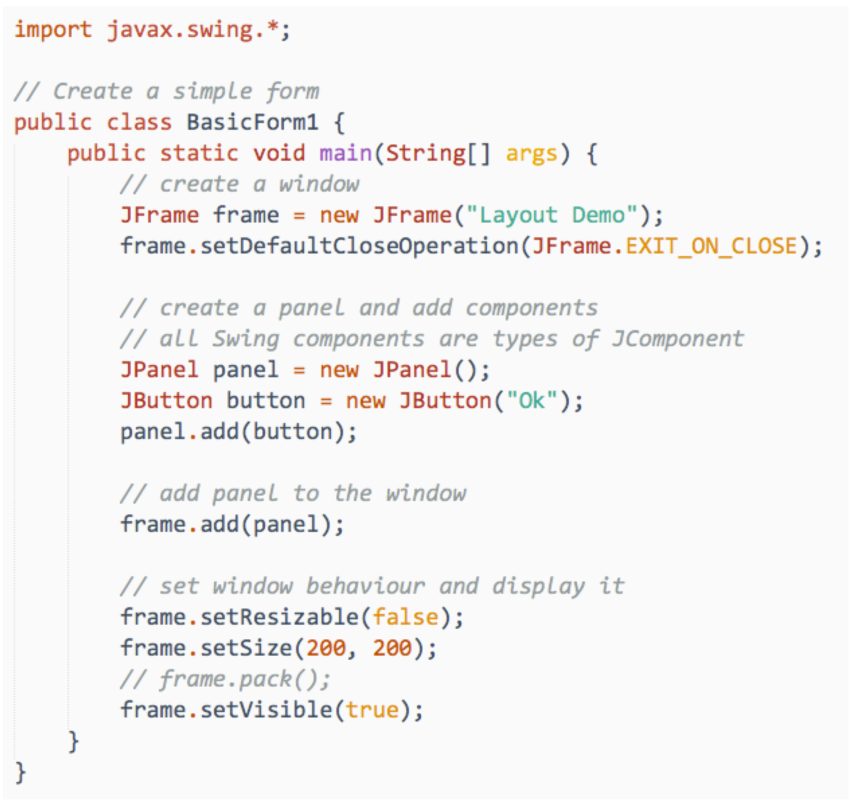


How to build a Swing UI

- Create a top-level application window, using a Swing container (JFrame or JDialog).
- Add Swing components to this window.
  - Typically, you create a smaller container (like a JPanel) and add components to the panel.
  - This makes dynamic layouts easier (more on that later in the course!)
- Register for events: add listeners, like keyboard (press), mouse (down, up, move)
- Write code to respond to these events.
- Make components update and paint themselves based on events.

Creating a Window

BasicForm1.java



```
import javax.swing.*;

// Create a simple form
public class BasicForm1 {
    public static void main(String[] args) {
        // create a window
        JFrame frame = new JFrame("Layout Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // create a panel and add components
        // all Swing components are types of JComponent
        JPanel panel = new JPanel();
        JButton button = new JButton("Ok");
        panel.add(button);

        // add panel to the window
        frame.add(panel);

        // set window behaviour and display it
        frame.setResizable(false);
        frame.setSize(200, 200);
        // frame.pack();
        frame.setVisible(true);
    }
}
```

Open a Window

```
package guis_1.v1;

import javax.swing.*;

public class GUIs1v1 {

  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      @Override
      public void run() {

        JFrame frame = new JFrame("Window Title");
        frame.setDefaultCloseOperation(
                                          JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 500);
        frame.setVisible(true);


      }
    });
  }
}
```

JFrame is a container for components.
Paint on components (coming).

Open a Window

```
package guis_1.v1;

import javax.swing.*;

public class GUIs1v1 {

  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      @Override
      public void run() {

        JFrame frame = new JFrame("Window Title");
        frame.setDefaultCloseOperation(
                                        JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 500);
        frame.setVisible(true);

      }
    });
  }
}
```

invokeLater ensures that the program can't start accepting events from the user before it's ready to start processing them.

Adding a Component

```
package guis_1.v2;

import javax.swing.*;

public class GUIs1v2 {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      @Override
      public void run() {
        JFrame frame = new JFrame("Window Title");
        frame.setDefaultCloseOperation(
                                        JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 500);

        frame.add(new ColouredX());
        frame.setVisible(true);
      }
    });
  }
}

class ColouredX extends JComponent {
}
```

Painting the ColouredX

```
class ColouredX extends JComponent {
  private BasicStroke stroke = new BasicStroke(30.0f);

  public void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    int w = this.getWidth();
    int h = this.getHeight();

    g2.setStroke(this.stroke);
    g2.setRenderingHint(
                                    RenderingHints.KEY_ANTIALIASING,
                                    RenderingHints.VALUE_ANTIALIAS_ON);
    g2.setColor(Color.RED);
    g2.drawLine(0, 0, w, h);

    g2.setColor(Color.BLUE);
    g2.drawLine(0, h, w, 0);
  }
}
```

Graphics vs. Graphics2D

paintComponent is called automatically. You never call it yourself.*

*Except, maybe, for pedagogical reasons in part 1 of assignment 1.

Animation in Java

```
package guis_1.v2;

import javax.swing.*;

public class GUIs1v2 {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      @Override
      public void run() {
        JFrame frame = new JFrame("Window Title");
        frame.setDefaultCloseOperation(
                                            JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 500);

        frame.add(new ColouredX());
        frame.setVisible(true);
      }
    });
  }
}

class ColouredX extends JComponent {
}
```

Animation using a Timer

```
class ColouredX extends JComponent {
  …
  private Point ballPos = new Point(100, 0);
  private final int FPS = 40;
  private Timer timer;

  public ColouredX() {
    this.addMouseListener(…);

    this.timer =

      new Timer(1000/FPS, new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
        ballPos.y += 2;
        repaint();
      }
    });
    this.timer.start();
  }


  public void paintComponent(Graphics g) {
    …

    g2.setColor(Color.ORANGE);
    g2.fillOval(this.ballPos.x, this.ballPos.y, 30, 30);
  }

}
```

FPS = Frames Per Second

javax.swing.Timer

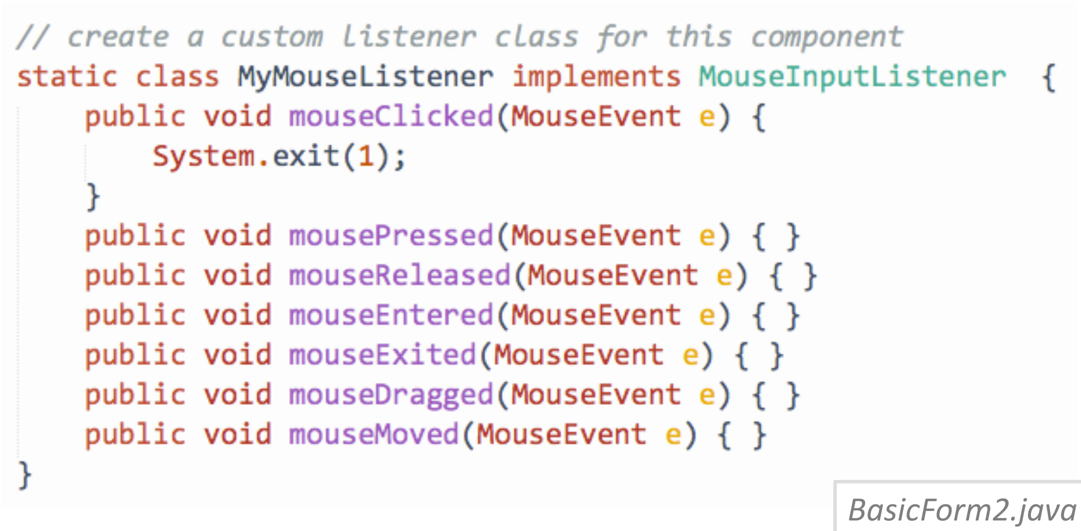
Paint the ball at its new location.

Java Listener Model

- Java has interfaces specialized by event type.
  - Each interface lists the methods that are needed to support that device’s events
- To use them, write a class that implements this interface, and override the methods for events you care about.
- Because it’s an interface, you have to override all of these methods – even for events you don’t care about!

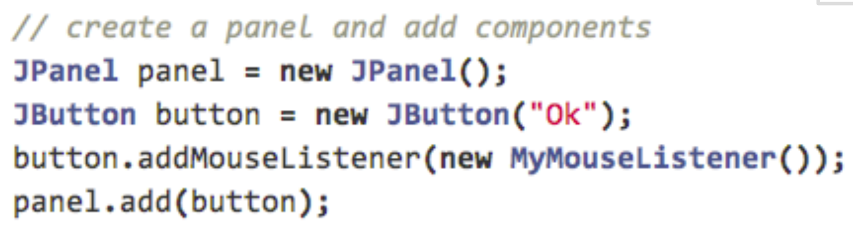
```
interface MouseInputListener {
    public void mouseClicked(MouseEvent e);
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
    public void mouseDragged(MouseEvent e);
    public void mouseMoved(MouseEvent e)
}
```

Using Listeners



```java
// create a custom listener class for this component
static class MyMouseListener implements MouseInputListener  {
    public void mouseClicked(MouseEvent e) {
        System.exit(1);
    }
    public void mousePressed(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
    public void mouseDragged(MouseEvent e) { }
    public void mouseMoved(MouseEvent e) { }
}
```

BasicForm2.java

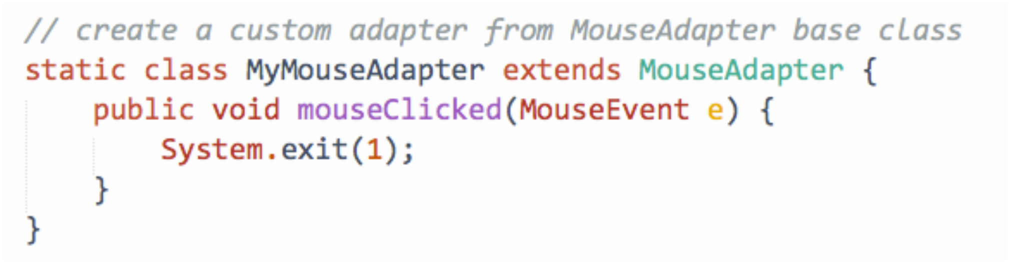


```java
// create a panel and add components
JPanel panel = new JPanel();
JButton button = new JButton("Ok");
button.addMouseListener(new MyMouseListener());
panel.add(button);
```

What's wrong with this approach?

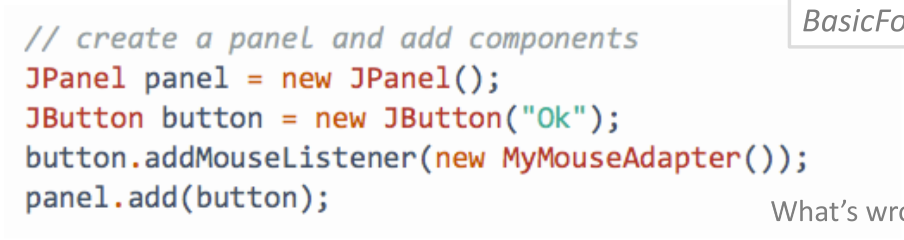
Adapters vs. Listeners

- Java also has adapters, which are base classes with empty listeners.
  - Extend the adapter and override the event handlers that you care about; avoids bloat.



```java
// create a custom adapter from MouseAdapter base class
static class MyMouseAdapter extends MouseAdapter {
    public void mouseClicked(MouseEvent e) {
        System.exit(1);
    }
}
```

BasicForm3.java



```java
// create a panel and add components
JPanel panel = new JPanel();
JButton button = new JButton("Ok");
button.addMouseListener(new MyMouseAdapter());
panel.add(button);
```

What's wrong with this approach?

Anonymous Inner Classes

- We really, really don't want to create custom adapters for every component.
  - Solution? Anonymous inner class.

BasicForm4.java

```java
public static void main(String[] args) {
    // create a window
    JFrame frame = new JFrame("Window Demo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // create a panel and add components
    JPanel panel = new JPanel();
    JButton button = new JButton("Ok");
    button.addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
        System.exit(1);
    }});
    panel.add(button);
```

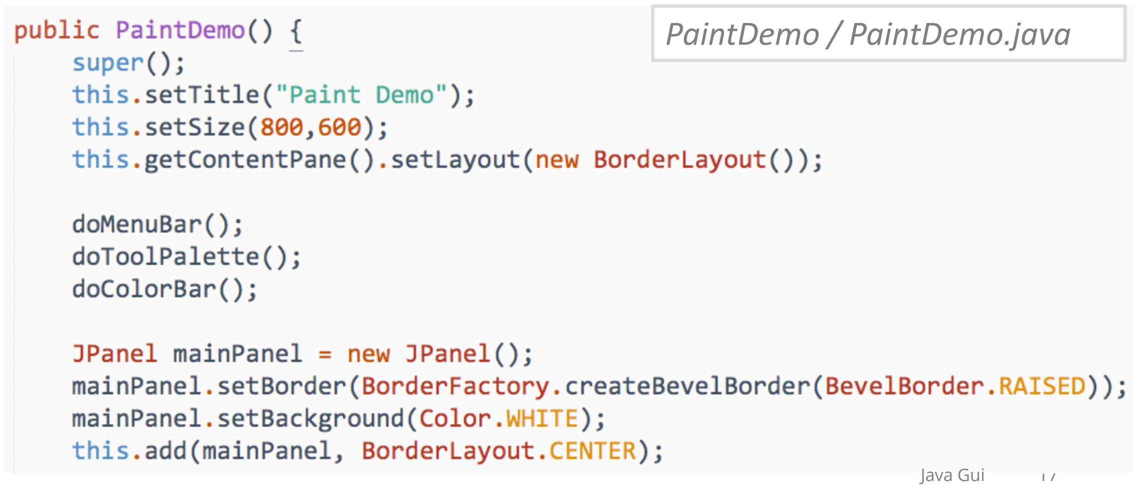
Swing UI Thread

- Swing needs to make sure that all events are handled on the Event Dispatch thread.
- If you just "run" your application from main, as we've been doing in the examples, you risk the main program accepting input before the UI is instantiated!
  - Use `invokeLater()` to safely create the UI.

```java
public static void main(String[] args)
{
    SwingUtilities.invokeLater(new Runnable()
    {
        public void run()
        {
            runProgram();
        }
    });
}
```

PaintDemo.java

- PaintDemo is an example of a UI hierarchy.
  - Demonstrates how to nest containers and components to build a more sophisticated application.
  - Uses LayoutManager, which we will discuss later in the term.



PaintDemo / PaintDemo.java

```
public PaintDemo() {
    super();
    this.setTitle("Paint Demo");
    this.setSize(800,600);
    this.getContentPane().setLayout(new BorderLayout());

    doMenuBar();
    doToolPalette();
    doColorBar();

    JPanel mainPanel = new JPanel();
    mainPanel.setBorder(BorderFactory.createBevelBorder(BevelBorder.RAISED));
    mainPanel.setBackground(Color.WHITE);
    this.add(mainPanel, BorderLayout.CENTER);
```

Drawing in Java

Overriding paintComponent()

Graphics object

Graphics and Painting

- Applications consist of a JFrame (window) containing one or more Swing components.
- We often define a top-level canvas (container)
  - This can hold other components (like text fields, buttons, scroll bars etc).
  - We can also draw directly on this canvas.

```java
// JComponent is a base class for custom components
public class SimpleDraw4 extends JComponent {

    public static void main(String[] args) {
        SimpleDraw4 canvas = new SimpleDraw4();
        JFrame f = new JFrame("SimpleDraw"); // jframe is the app window
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(400, 400); // window size
        f.setContentPane(canvas); // add canvas to jframe
        f.setVisible(true); // show the window
    }
```

Graphics and Painting

- Each component has a paintComponent() method, which describes how it paints itself.
  - You can override this paintComponent() method and draw primitive objects using the java.awt.Graphics object (basically, the Graphics Context).
  - This is a common technique for defining drawables in Java.

```java
// custom graphics drawing
public void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;    // cast to get 2D methods
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                        RenderingHints.VALUE_ANTIALIAS_ON);
    g2.setStroke(new BasicStroke(32)); // 32 pixel thick stroke
    g2.setColor(Color.BLUE);           // make it blue
    g2.drawLine(0, 0, getWidth(), getHeight());  // draw line
    g2.setColor(Color.RED);
    g2.drawLine(getWidth(), 0, 0, getHeight());
```

What’s left?

- Topics that we’ll cover in later lectures
- Animation
- Advanced graphics
- Design patterns
- Features (undo-redo, copy-paste)