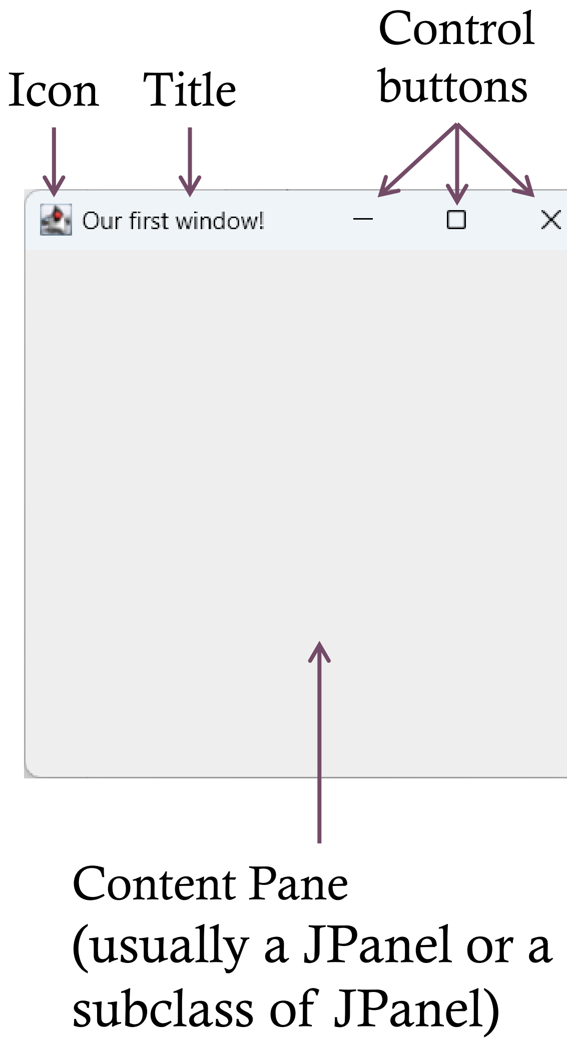
WEEK FOUR

Acknowledgements: Slides created based off material provided by Dr. Michael Raymer and Dr. Travis Doom

CREATING A WINDOW

```
import javax.swing.*;
```

```
public static void main(String args[]){
    JFrame theWindow = new JFrame("Our first window!");
    theWindow.setSize(300, 300);
    theWindow.setLocation(200, 400);
    theWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    theWindow.setVisible(true);
}
```



OUR FIRST EXECUTION THREAD

- Why is the program still running when main() is done?

```
public static void main(String args[]){
     JFrame theWindow = new JFrame("Our first window!");
     theWindow.setSize(300, 300);
     theWindow.setLocation(200, 400);
     theWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     theWindow.setVisible(true);
     System.out.println("Done!");
 }
```

The Event Dispatch Thread (EDT) waits for clicks, drags, re-sizes, keyclicks, and other events and responds to them.

OUR FIRST EXECUTION THREAD

```
public static void main(String args[]){
    JFrame theWindow = new JFrame("Our first window!");
    theWindow.setSize(300, 300);
    theWindow.setLocation(200, 400);
    theWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    theWindow.setVisible(true);
    System.out.println("Done!");
}
```

```
while (window-open){
    if (button-clicked){
        do something;
    }
    else if (window-resized){
        do something;
    }
    else if …
}
```

KEEPING TRACK OF DATA

- If main() is going to exit, where do we keep all our variables and data?

- There are several approaches, but we'll usually create a **subclass of JFrame** and use **instance variables** for all our persistent data.
- This data will live for as long as our main application window is not closed.

KEEPING TRACK OF DATA

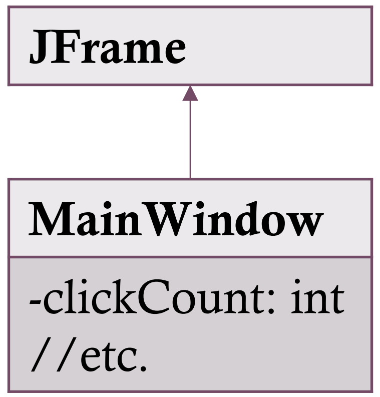
```
class MainWindow extends JFrame{
    private int clickCount;

    public MainWindow(String title){
        super(title);
        clickCount = 0;
    }
    public static void main(String args[]){
        JFrame theWindow = new MainWindow("Our first window!");
        theWindow.setSize(300, 300);
        theWindow.setLocation(200, 400);
        theWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        theWindow.setVisible(true);
        System.out.println("Done!");
    }
}
```

$\leftarrow$ Data I want to keep around until the main window is closed.

Utilizing inheritance in this way is extremely useful!

KEEPING TRACK OF DATA

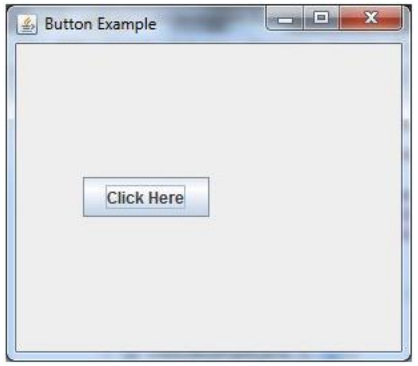
```
class MainWindow extends JFrame{
    private int clickCount;

    public MainWindow(String title){
        super(title);
        clickCount = 0;
    }
    public static void main(String args[]){
        JFrame theWindow = new MainWindow("Our first window!");
        theWindow.setSize(300, 300);
        theWindow.setLocation(200, 400);
        theWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        theWindow.setVisible(true);
        System.out.println("Done!");
    }
}
```

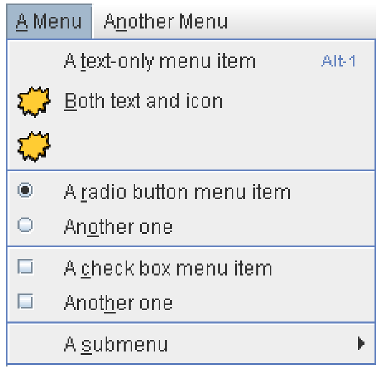


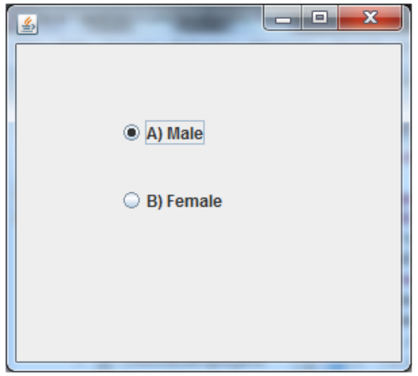
- I can store any data I want to be persistent here
- It is accessible to any method of MainWindow

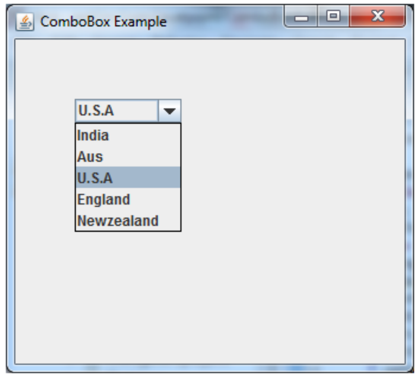
GUI ELEMENTS IN SWING

- Includes many useful GUI elements:
  - JButton
  - JLabel
  - JCheckBox
  - JRadioButton and ButtonGroup
  - JList
  - JMenuBar, Jmenu, and JMenuItem
  - JComboBox
  - JSlider, JScrollBar









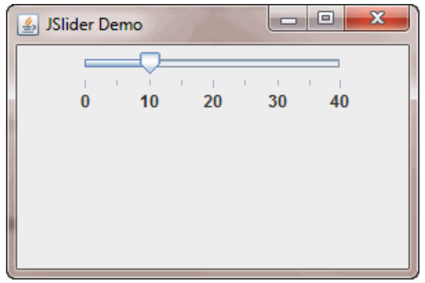


Image source: https://web.mit.edu/6.005/www/sp14/psets/ps4/java-6-tutorial/components.html
https://www.javatpoint.com

JBUTTONS

- Clickable buttons
- Can be labeled, arranged, etc.
- We can listen for the button to be pressed

```
public MainWindow(String title){
    super(title);
    clickCount = 0;

    JButton aButton = new JButton("Click me!");
    this.add(aButton);
}
```

ADDING GUI ELEMENTS

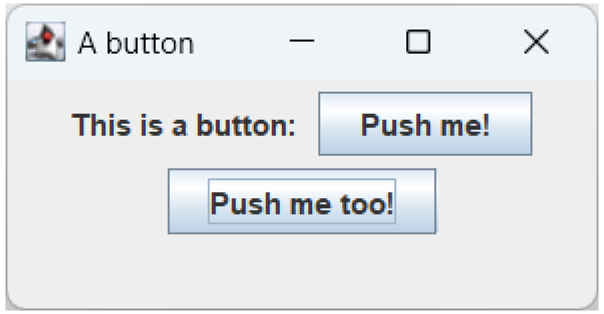
- However, the frame will only show the most recently added item

```
public MainWindow(String title){
    super(title);
    clickCount = 0;

    JButton aButton = new JButton("Click me!");
    this.add(aButton);
    JButton otherButton = new JButton("Click me too!");
    this.add(otherButton);

}
```

- How can we display more than one element?

JPANEL (AND LAYOUT MANAGERS) TO THE RESCUE!

- JPanel can hold many GUI elements and allows you to set a Layout Manager to keep them neatly arranged

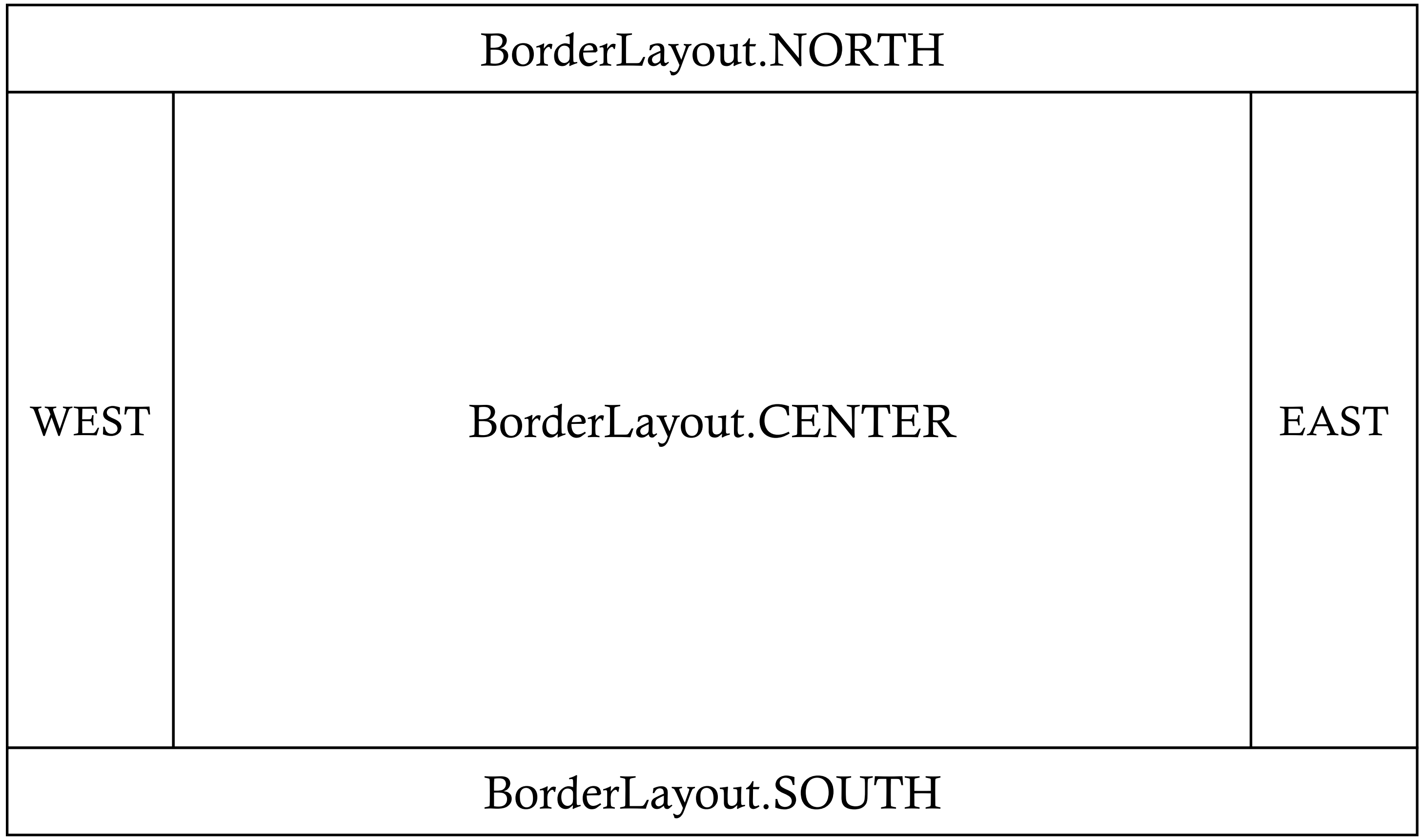


JPANEL (AND LAYOUT MANAGERS) TO THE RESCUE!

- The default layout manager is called FlowLayout
- Elements are added to a row until there is no more space, then starts another row

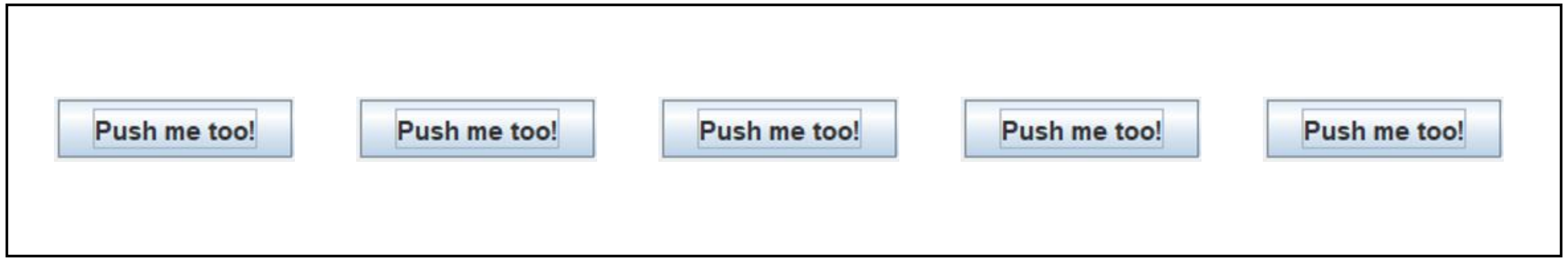
```java
public MainWindow(String title){
    super(title);
    clickCount = 0;

    JPanel content = new JPanel();
    this.setContentPane(content);

    JButton aButton = new JButton("Click me!");
    content.add(aButton);
    JButton otherButton = new JButton("Click me too!");
    content.add(otherButton);

}
```

BORDERLAYOUT

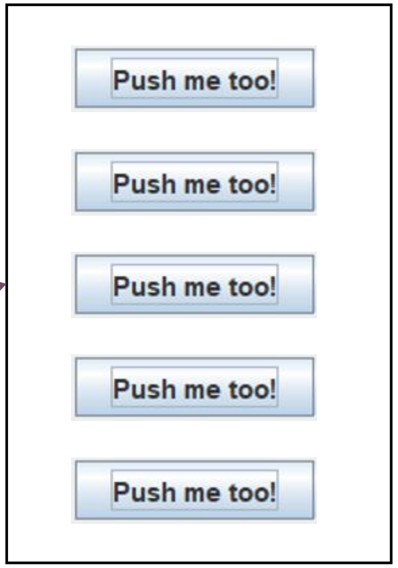


BOXLAYOUT

```
content.setLayout(new BoxLayout(content, BoxLayout.X_AXIS));
```

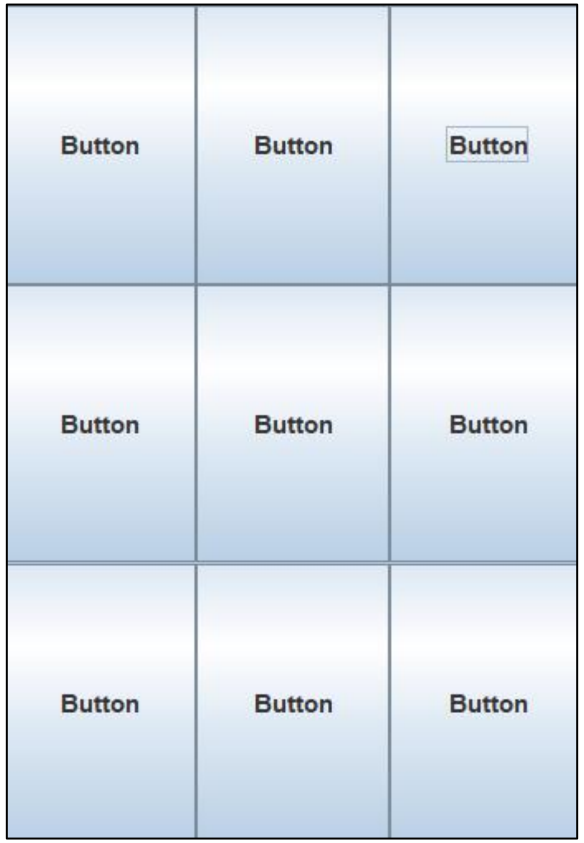


```
content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));
```



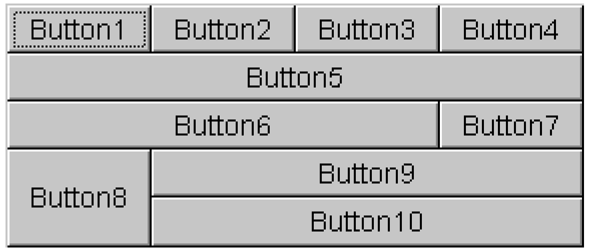
GRIDLAYOUT

```
content.setLayout(new GridLayout(3, 3));
```



GRIDBAGLAYOUT

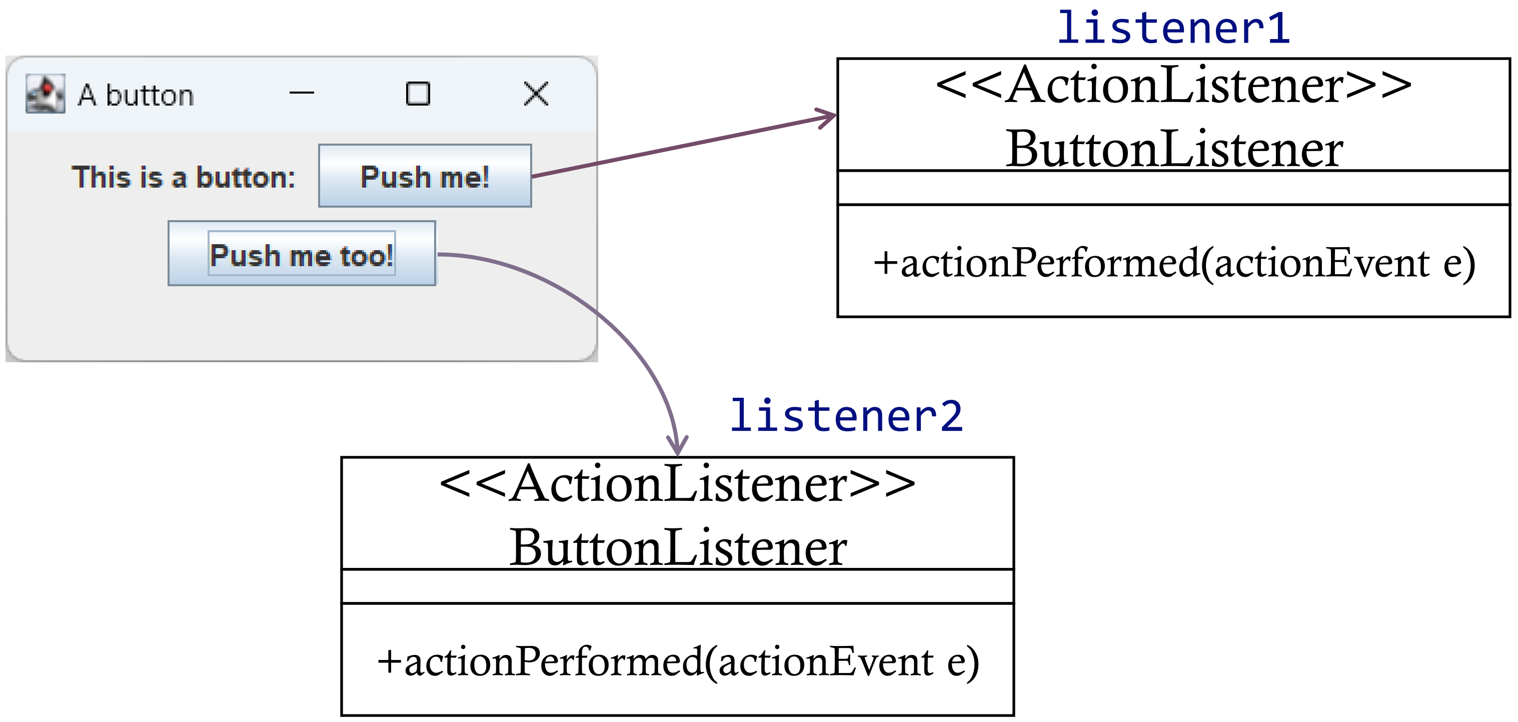
- Dynamic Grid – created as you add elements
- Set Sizes and insets (padding) as you go
- You can span rows and columns
- Complex, but powerful
- See examples in ZyBooks



https://docs.oracle.com/javase/8/docs/api/java/awt/GridBagLayout.html

LISTENING FOR EVENTS

- Wouldn't it be nice if our buttons did more than just look like buttons?



```
ButtonListener listener1 = new ButtonListener();
aButton.addActionListener(listener1);
```

LISTENING FOR EVENTS

- The `ActionListener` interface has just one method: `actionPerformed`
- `@Override` prevents errors caused by typos
- The `ActionEvent`s hold information about the event

```
public class ButtonListener implements ActionListener{
    @Override
    public void actionPerformed(ActionEvent e){
        System.out.println("The button was pressed!");
    }
}
```

WHY WON'T THIS WORK?

```
public static void main(String args[]){
     JFrame theWindow = new MainWindow("Our first window!");
     theWindow.setSize(300, 300);
     theWindow.setLocation(200, 400);
     theWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

     JPanel content = new JPanel();
     theWindow.setContentPane(content);

     int clickCount = 0;
     JButton aButton = new JButton("Click me!");
     ButtonListener listener1 = new ButtonListener();
     aButton.addActionListener(listener1);
     content.add(aButton);

     theWindow.setVisible(true);
}
```

```
public class ButtonListener implements ActionListener{
   @Override
   public void actionPerformed(ActionEvent e){
      clickCount++;
      System.out.println("The button has been clicked " + clickCount + " times");
   }
}
```

OK, WHAT ABOUT THIS?

```java
class MainWindow extends JFrame{
   private int clickCount;

   public MainWindow(String title){
      super(title);

      JPanel content = new JPanel();
      this.setContentPane(content);

      clickCount = 0;
      JButton aButton = new JButton("Click me!");
      ButtonListener listener1 = new ButtonListener();
      aButton.addActionListener(listener1);
      content.add(aButton);
   }
}
```

```java
public class ButtonListener implements ActionListener{
   @Override
   public void actionPerformed(ActionEvent e){
      clickCount++;
      System.out.println("The button has been clicked " + clickCount + " times");
   }
}
```

INNER CLASSES

- We need a way to access clickCount from actionPerformed!
- If we create the class ButtonListener *inside* the class MainWindow, it can access every field of MainWindow, **even private ones**
- This is called an Inner Class in Java

USING AN INNER CLASS

```
class MainWindow extends JFrame {
    private int clickCount;

    public MainWindow(String title) {
        super(title);
        clickCount = 0;
        JPanel content = new JPanel();
        this.setContentPane(content);

        // This Inner Class can see all of MainWindow's private variables:
        class ButtonListener implements ActionListener {
            @Override
            public void actionPerformed(ActionEvent e) {
                clickCount++;
                System.out.println("The button has been clicked " + clickCount + " times");
            }
        }

        JButton aButton = new JButton("Click me!");
        aButton.addActionListener(new ButtonListener());
        content.add(aButton);
    }
```

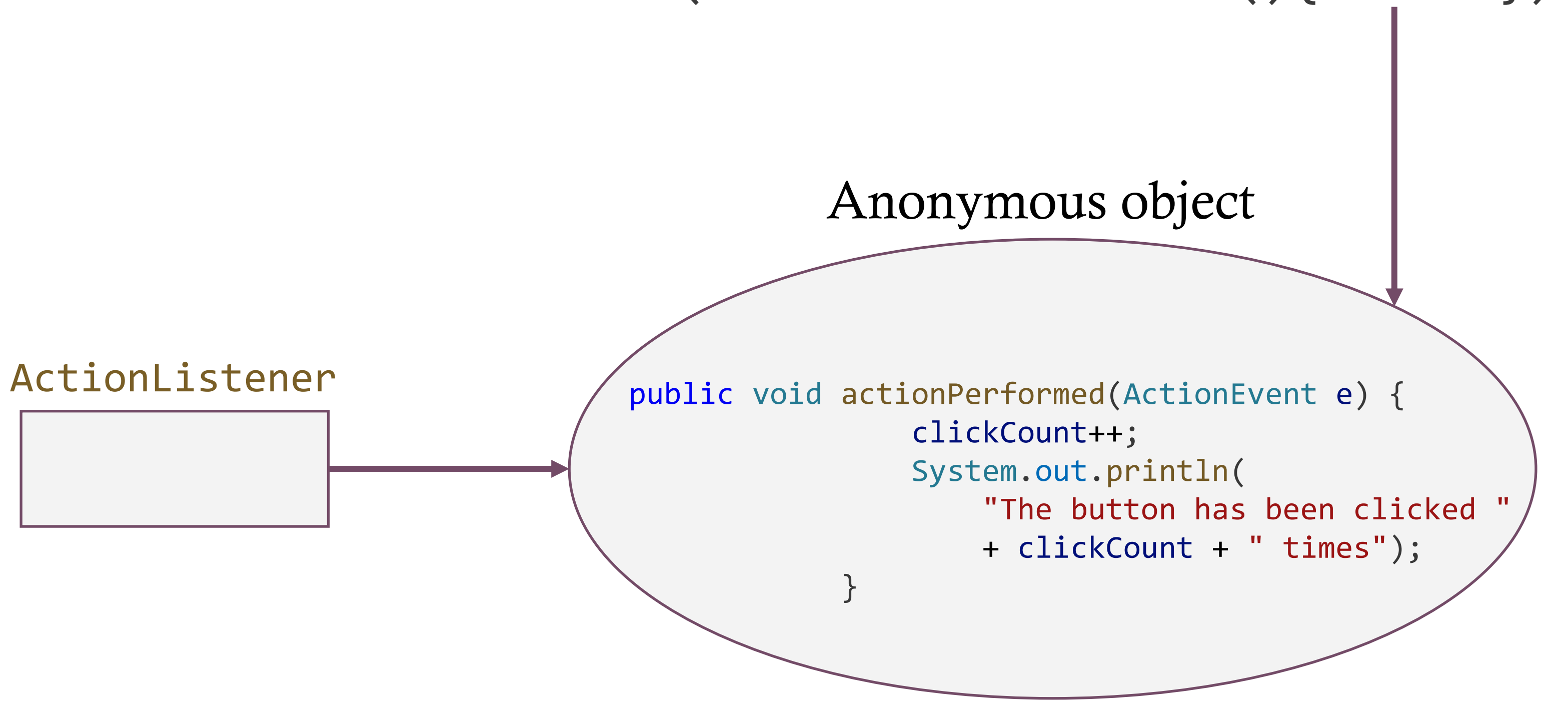
ANONYMOUS INNER CLASSES

- If we aren’t going to re-use the class, we can create it **inside** the call to addActionListener()

```
JButton aButton = new JButton("Click me!");
aButton.addActionListener(new ActionListener(){
     @Override
     public void actionPerformed(ActionEvent e) {
         clickCount++;
         System.out.println("The button has been clicked "
             + clickCount + " times");
     }});
 content.add(aButton);
```

ANONYMOUS INNER CLASSES

```
aButton.addActionListener(new ActionListener(){ code });
```



```
public void actionPerformed(ActionEvent e) {
                clickCount++;
                System.out.println(
                    "The button has been clicked "
                    + clickCount + " times");
            }
```

LAMBDA EXPRESSIONS

- Can we make this even more concise?

```java
JButton aButton = new JButton("Click me!");
aButton.addActionListener(e -> {
    clickCount++;
    System.out.println("The button has been clicked "
        + clickCount + " times");
});
```

HOW DOES JAVA KNOW?

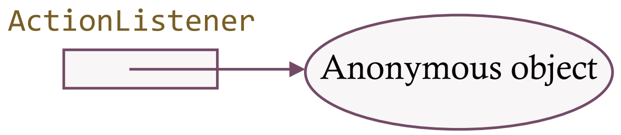
addActionListener expects one parameter, an ActionListener, so I need to create an ActionListener reference variable.

ActionListener

```
aButton.addActionListener(e -> {code});
```

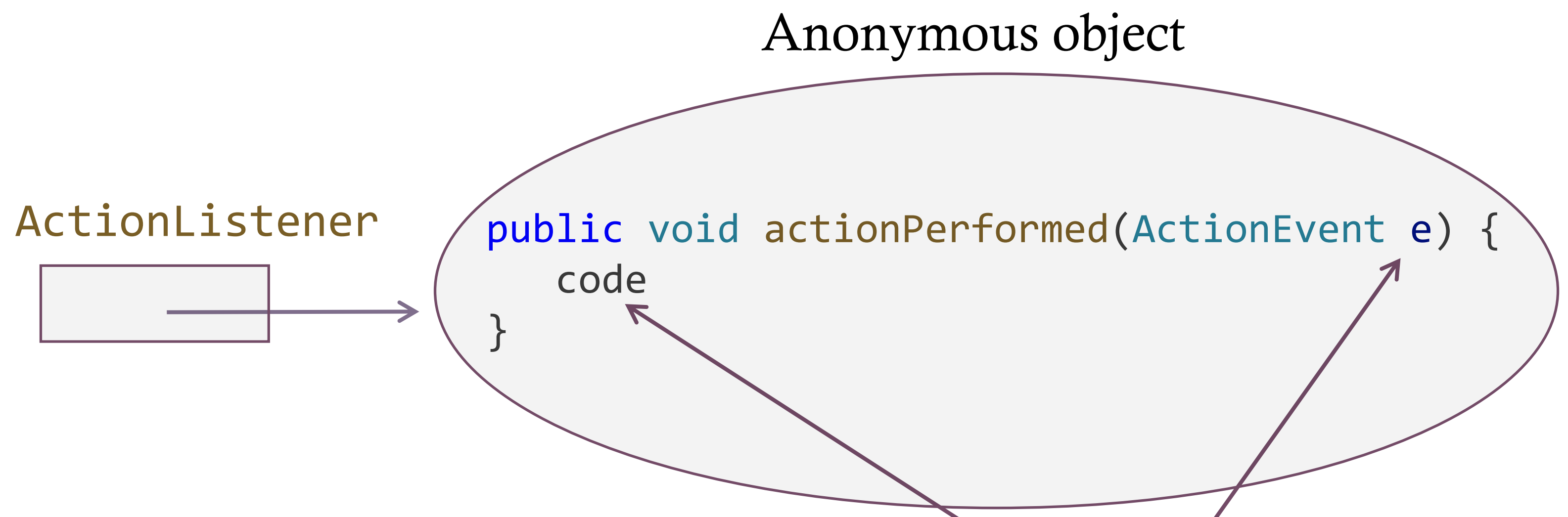
HOW DOES JAVA KNOW?

What kind of object should I point it to? I guess an anonymous one, but an ActionListener **must** have actionPerformed(ActionEvent e), so I guess that's where the code goes.



```
aButton.addActionListener(e -> {code});
```

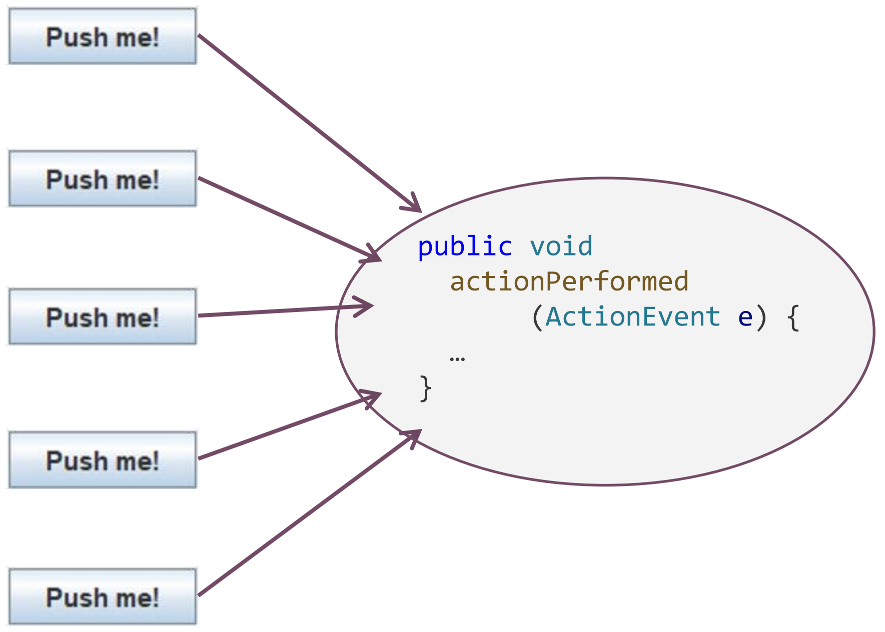
HOW DOES JAVA KNOW?



```
public void actionPerformed(ActionEvent e) {
    code
}
```

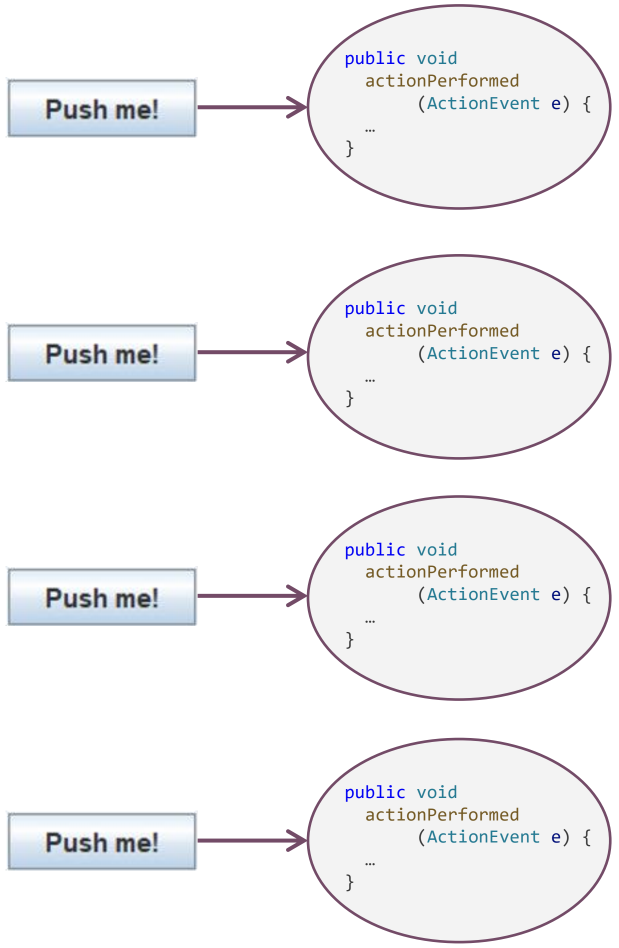
```
aButton.addActionListener(e -> {code});
```

ONE ACTIONLISTENER OR MANY?



```
public void
  actionPerformed
        (ActionEvent e) {
    …
}
```

```
if (e.getSource() == aButton){
        ...
}
```



```
public void
  actionPerformed
        (ActionEvent e) {
    …
}
```

```
public void
  actionPerformed
        (ActionEvent e) {
    …
}
```

```
public void
  actionPerformed
        (ActionEvent e) {
    …
}
```

```
public void
  actionPerformed
        (ActionEvent e) {
    …
}
```

ACTIONLISTENERS AND THE EDT

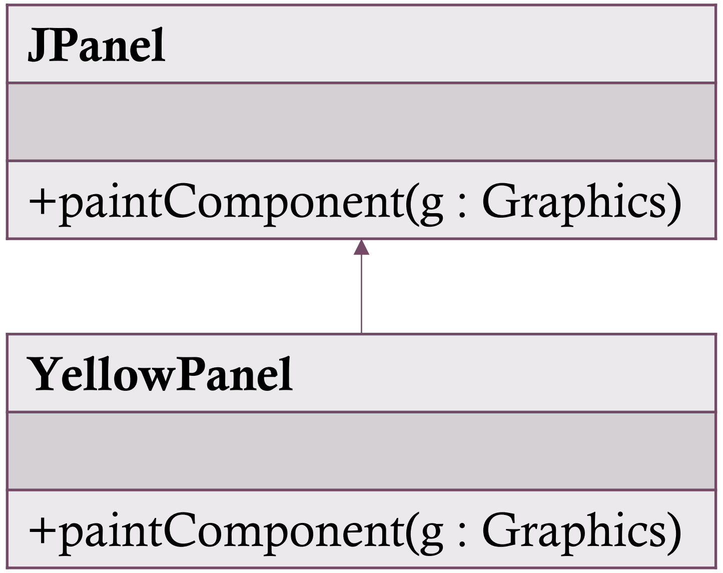
- ActionListener code runs in the EDT, which means it is **dangerous** to perform long calculations. Your program will lag, and user events will pile up.

```java
countButton.addActionListener(e -> {
    for (int i = 1; i <= 10; i++){
        System.out.println("Counting: " + i);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
});
```

A SIDE EXPERIMENT

Java calls `void paintComponent(Graphics g)` when a GUI component needs to be re-drawn

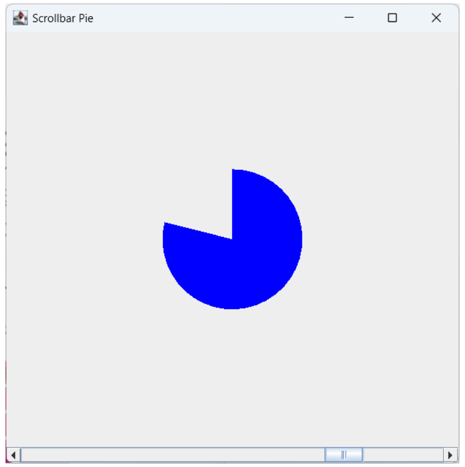
Think of a `Graphics` object like a pen, it can be used to change colors and draw things like lines and shapes.



DRAWING YOUR OWN OBJECTS

- **JComponent** is the parent (or grandparent) class for most Swing GUI components
- Override **paintComponent()** to draw graphics

```
public class Drawing extends JComponent{

    // When we re-paint this JComponent, draw a few simple shapes:
    @Override
    public void paintComponent(Graphics g){
        g.setColor(Color.BLACK);
        g.drawOval(50, 50, 75, 75);
        g.drawRect(this.getWidth()/2, this.getHeight()/2, 50, 50);
    }
}
```



ANIMATION

- Animation can be done using the main() thread, using timers, or by creating new threads, which we will discuss later.