COM6516
Object Oriented Programming and Software Design

The contents of this module has been developed by Adam Funk, Kirill Bogdanov, Mark Stevenson, Richard Clayton and Heidi Christensen

7. Graphical User Interfaces (GUI)

Aims

Show how to build simple GUIs in Java and introduce the idea of inner classes in Java

Objectives

At the end of this lecture you will be able to understand the difference between an event, an event source and an event listener and to make a JButton an event source and implement an ActionListener

7. Graphical User Interfaces (GUI)

Outline

- …

Readings

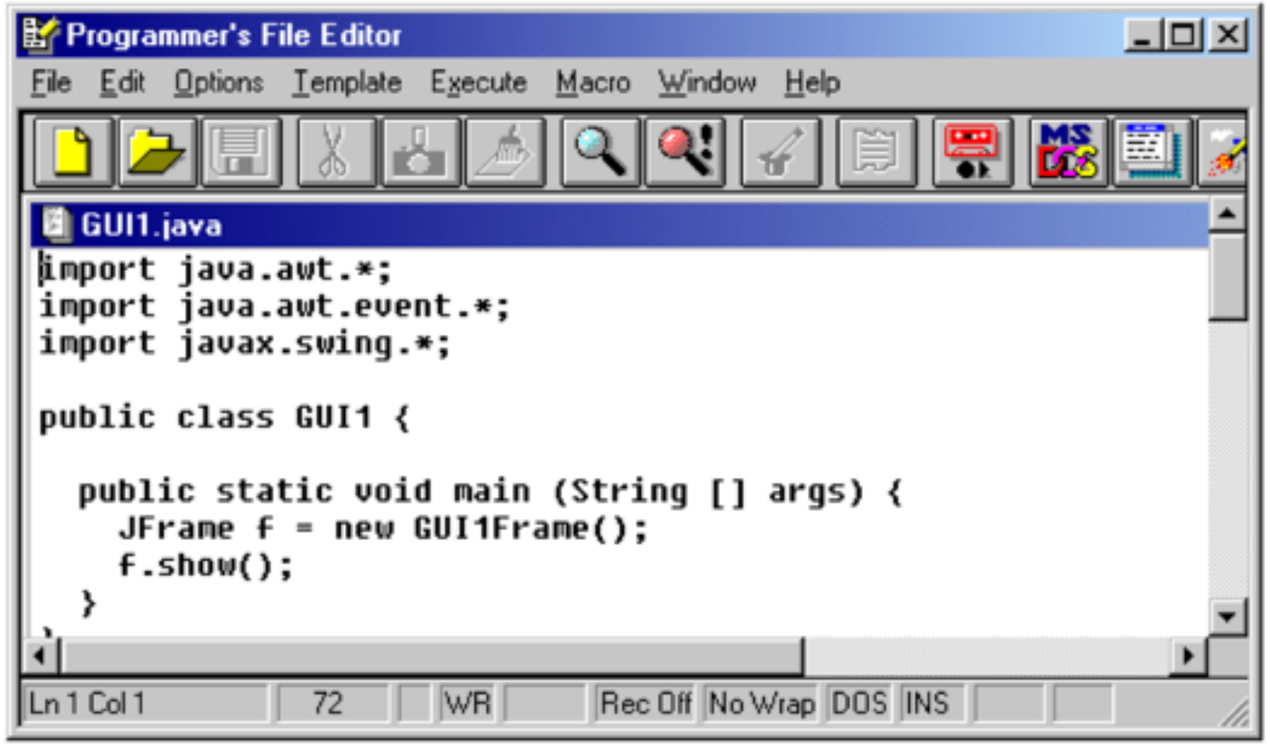
Core Java, vol.1, chapters 7,8, and 9

`http://docs.oracle.com/javase/tutorial/ui/features/components.html`

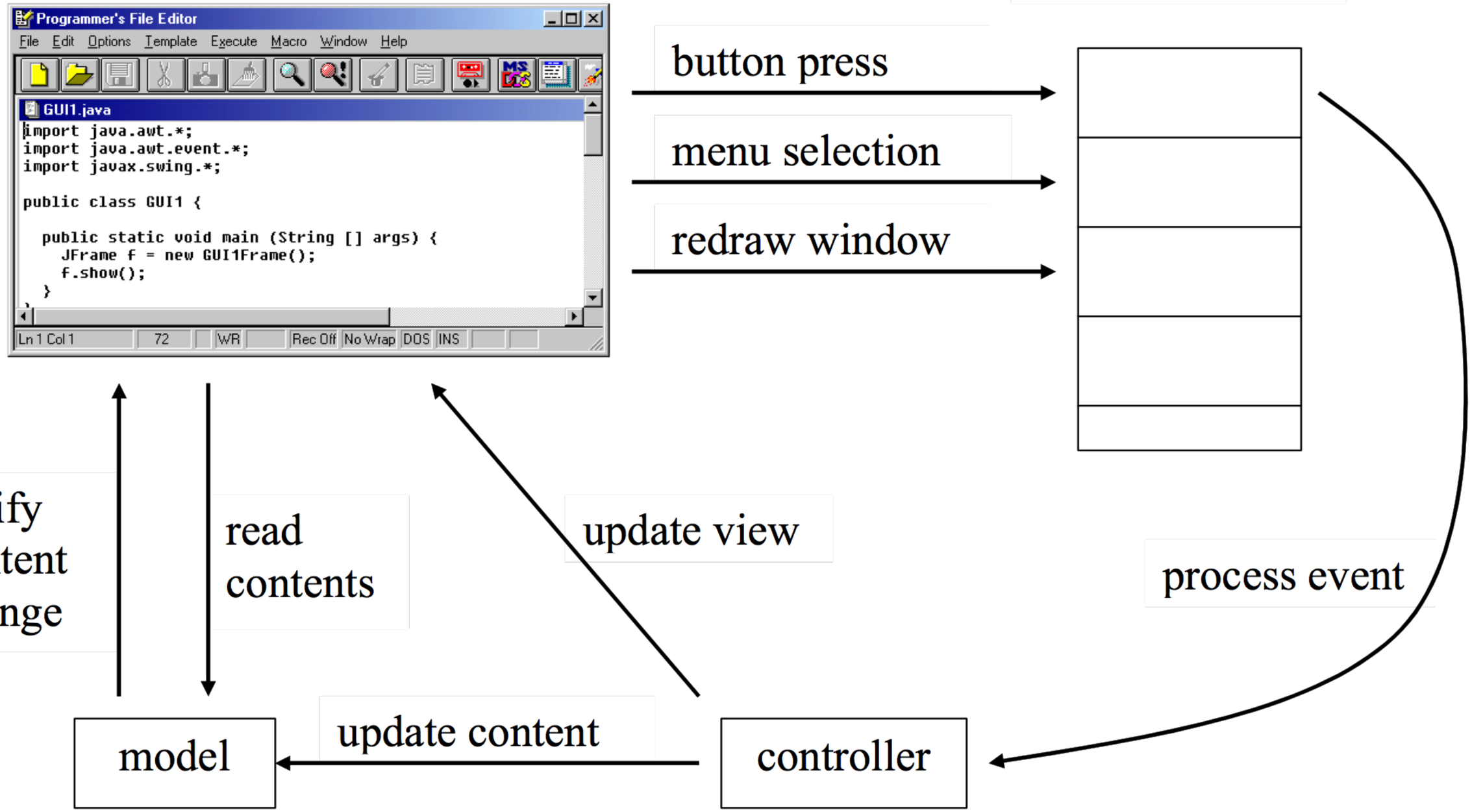
Introduction

To create a Graphical User Interface

- We need to put pieces (***components***) together as part of an interface, e.g. buttons and menus.
- We need to respond to actions (***events***) initiated by the user or as part of the working of the system.

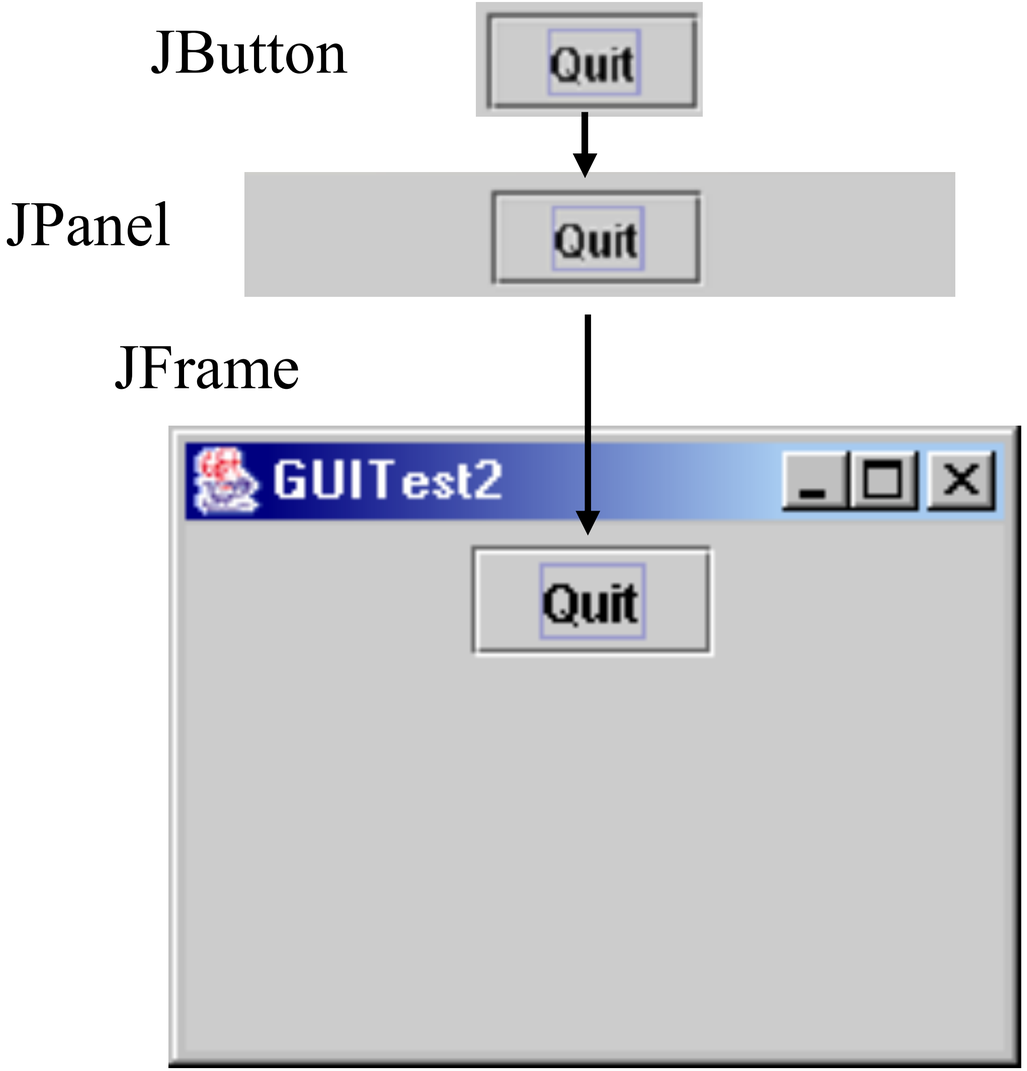


Introduction



Event handling in Java

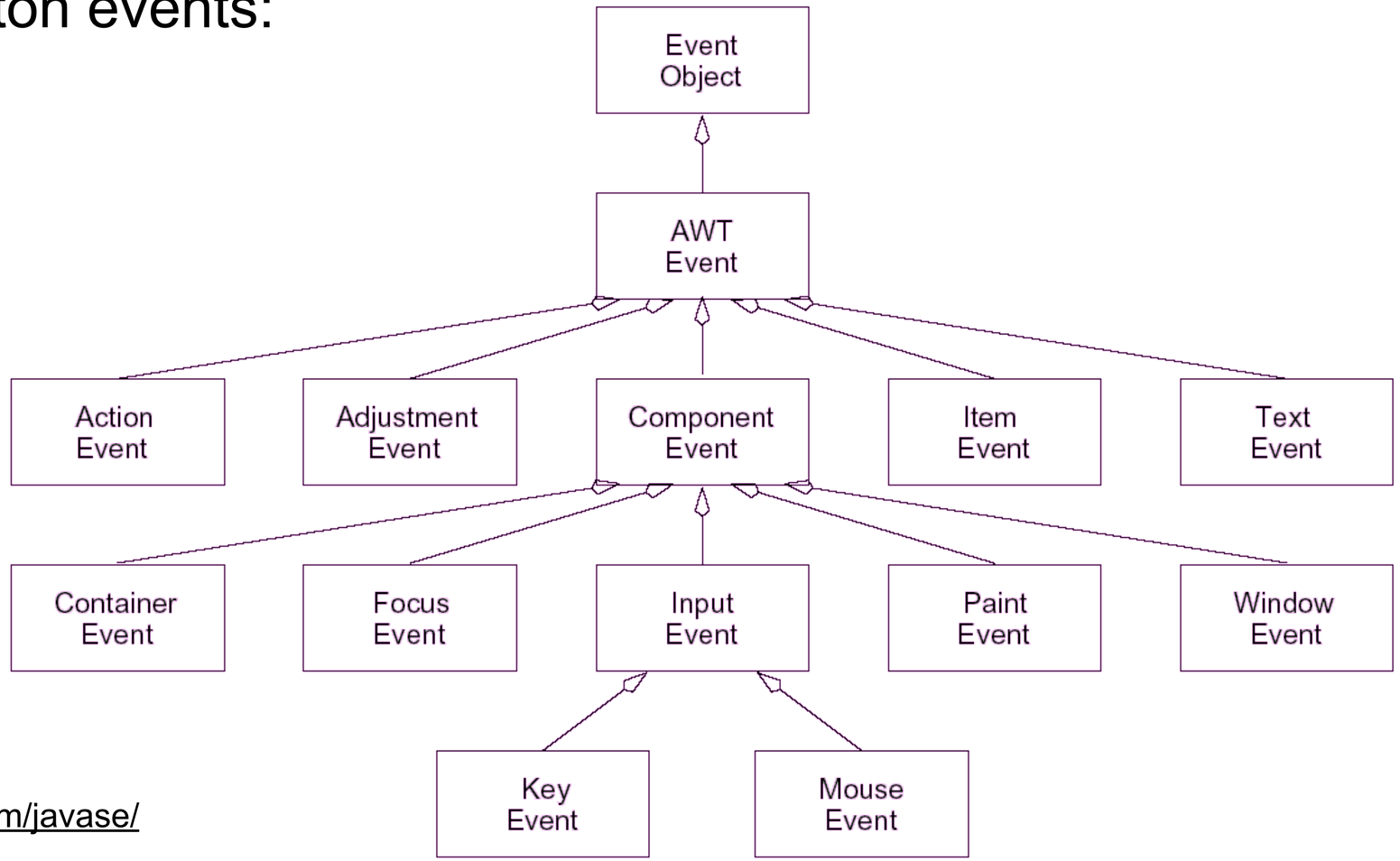
- A listener object implements a **listener interface**
- Event sources have methods to register listener objects
- When an event occurs, an *event object* is sent from the event source to every registered listener
- Each listener object reacts to the event using the information in the event object
- The event handling classes are part of the `java.awt.event` package.



AWT event classes

The different event types used within the AWT classes follow a similar pattern to handling button events:

- An interface must be implemented, so certain methods must be written.
- The parameter, i.e. the event class, returned by each of these methods can be queried.



For detailed background to event http://docs.oracle.com/javase/tutorial/uiswing/events/index.html

Stage 1: Matching event source to event listener

Example – a simple quit button (see EventHandling.java)

An `ActionListener` is associated with the `JButton` and the `JFrame`:

```
class GUIFrame extends JFrame implements ActionListener {
  private JButton quitButton;

  public GUIFrame() {
       Container contentPane = this.getContentPane();
    JPanel p = new JPanel();
    quitButton = new JButton("Quit");    // create Quit button
    quitButton.addActionListener(this); // link button and frame
    p.add(quitButton);                  // add button to JPanel
    contentPane.add(p);                 // add JPanel to JFrame
  }
  // public void actionPerformed (Event e) to be implemented
}
```

The `quitButton` (event source) will notify `this` (event listener) when it is pressed

Stage 2: The method that responds to the event

GUIFrame implements the `ActionListener` interface and so must supply a method called `actionPerformed`:

```
class GUIFrame extends JFrame implements ActionListener {
  private JButton quitButton;
  // constructor for GUIFrame as on previous slide, etc
  // …

  // respond to an event
  public void actionPerformed(ActionEvent event) {
    Object source = event.getSource();
    if (source == quitButton) {
        System.out.println(”Quit button pressed");
        System.exit(0);
    }
  }
}
```

Stage 3: The event loop is started

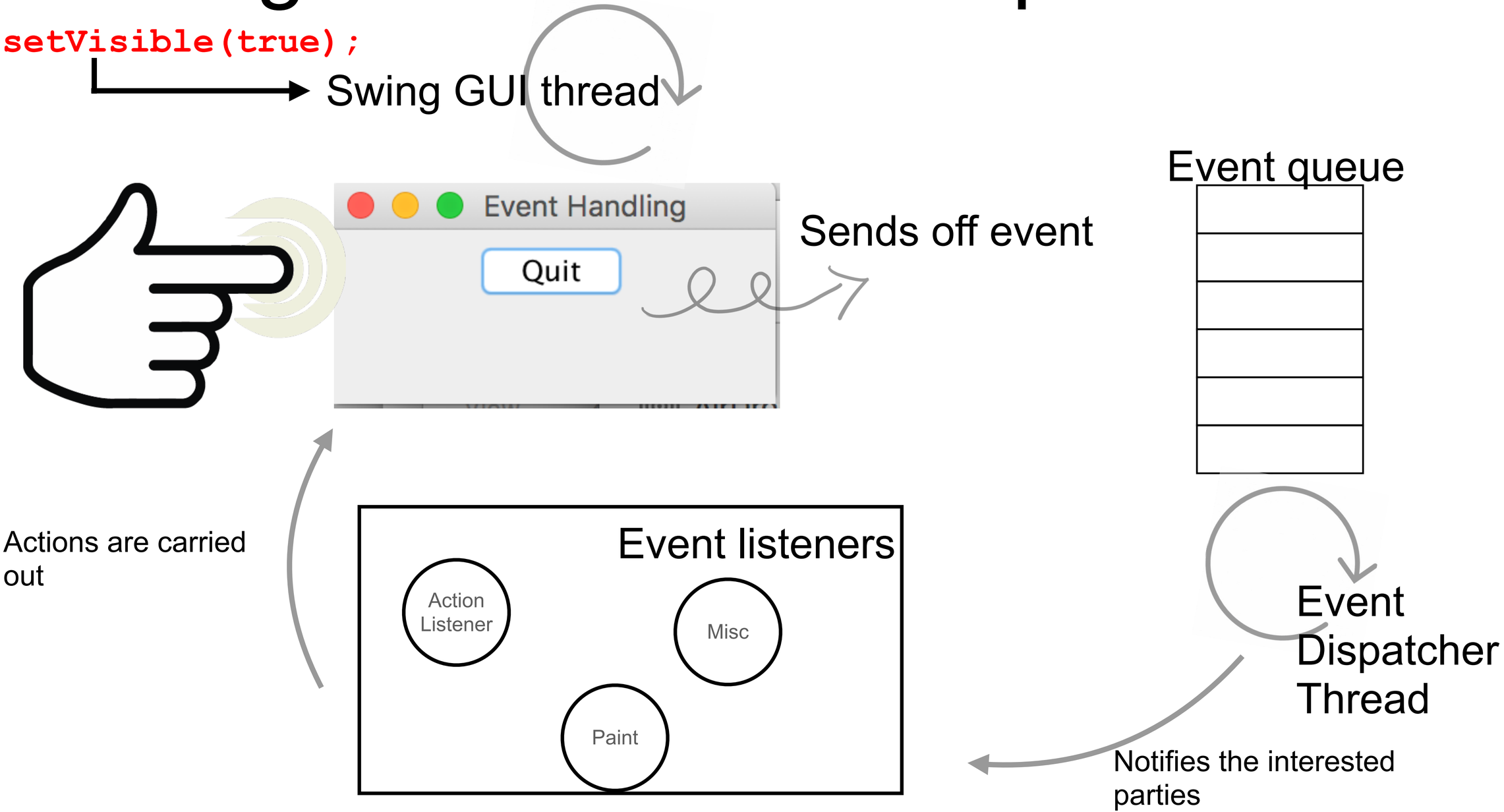
A call to `f.setVisible(true)` starts a Swing GUI thread for the application :

When an action event occurs, i.e. the `JButton` is pressed, the system creates a new instance of the `ActionEvent` class (defined in `java.awt.event.*`) and this is passed to the method `actionPerformed(…)`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class EventHandling {
    public static void main (String [] args) {
        JFrame f = new GUIFrame();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
    } …
}
```

Stage 3: The event loop is started

```
f.setVisible(true);
```



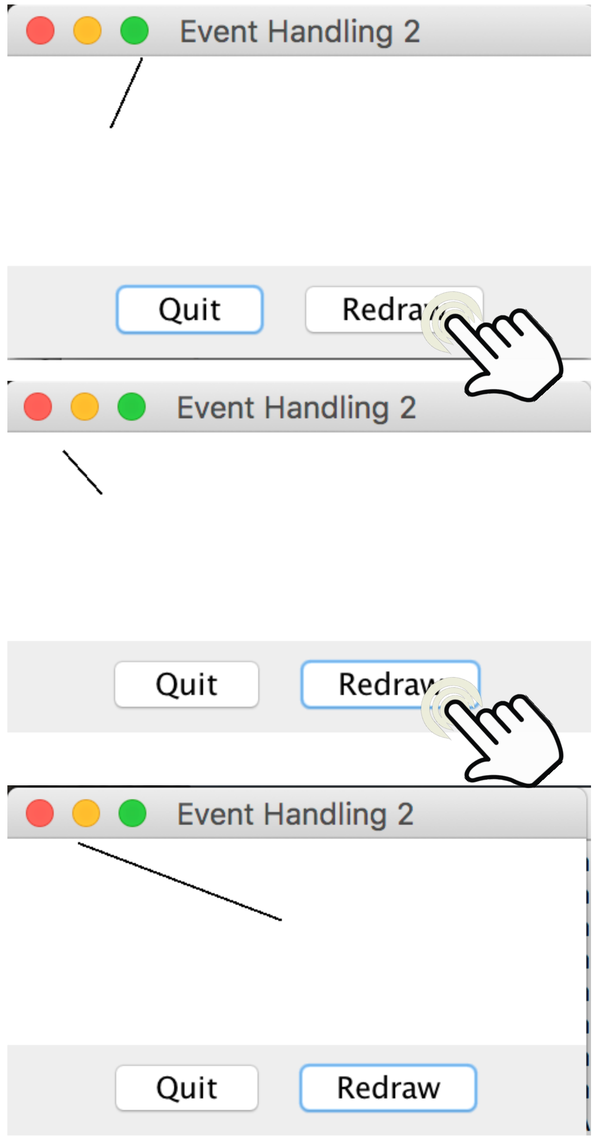
(useful method: factory method for Buttons)

To create a lot of buttons, we can use a factory method:

```
...
  JPanel p = new JPanel();
  quitButton = makeJButton("Quit", this);
  p.add(quitButton);
  redrawButton = makeJButton("Redraw", this);
  p.add(anotherButton);
...

private JButton makeJButton(String s, ActionListener a)
{
  JButton b = new JButton(s);
  b.addActionListener(a);
  return b;
}
```

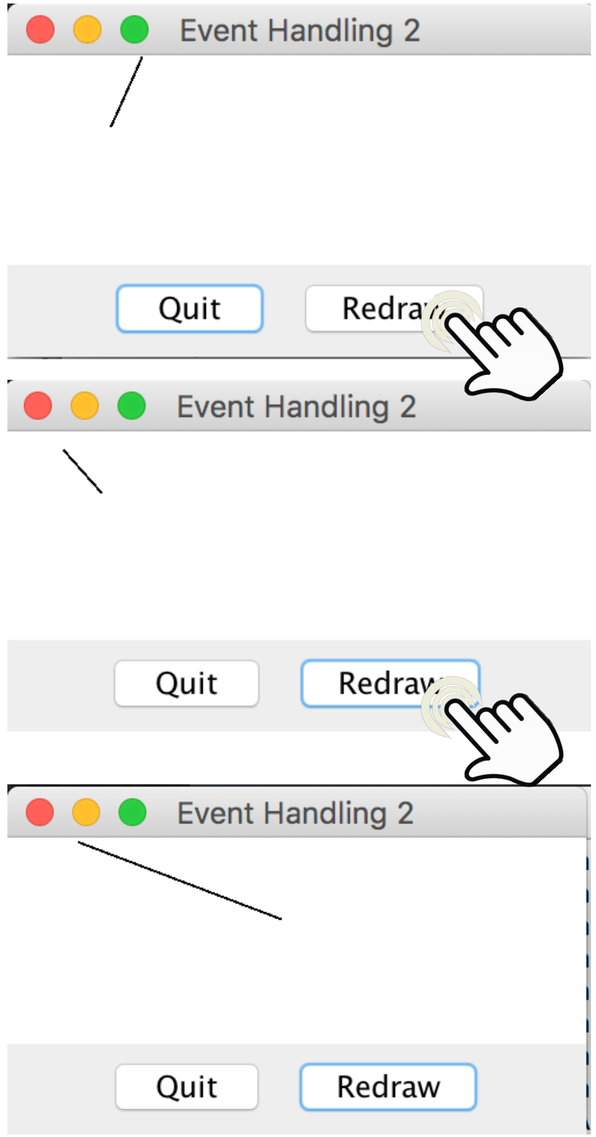
Example - `EventHandling2.java`

```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class EventHandling2 {
    public static void main (String [] args) {
    JFrame f = new GUI2Frame();
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setVisible(true);
    }
}

class GUI2Frame extends JFrame implements ActionListener {
    private JButton quitButton, redrawButton;
    private GUI2Panel drawingPanel;

    // constructor
    public GUI2Frame() {
        setTitle("Event Handling 2");
        setSize(250, 150);
        Container contentPane = this.getContentPane();
        JPanel p = new JPanel();
        quitButton = makeJButton("Quit", this);
        p.add(quitButton);
        redrawButton = makeJButton("Redraw", this);
```



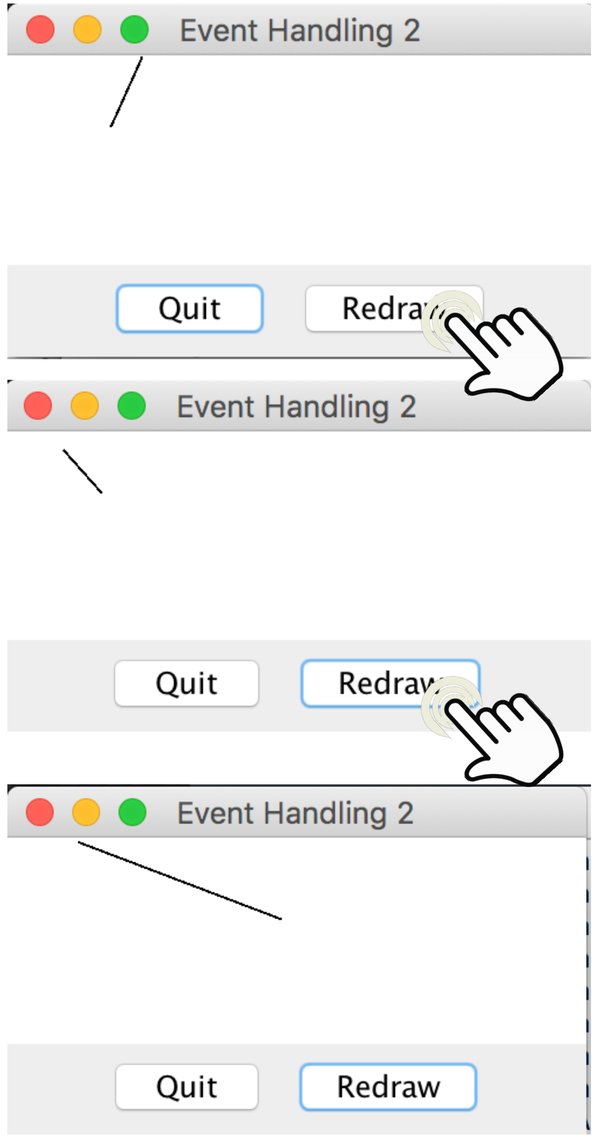
Example - EventHandling2.java

```java
class GUI2Frame extends JFrame implements ActionListener {
    private JButton quitButton, redrawButton;
    private GUI2Panel drawingPanel;

    // constructor
    public GUI2Frame() {
        setTitle("Event Handling 2");
        setSize(250, 150);
        Container contentPane = this.getContentPane();
        JPanel p = new JPanel();
        quitButton = makeJButton("Quit", this);
        p.add(quitButton);
        redrawButton = makeJButton("Redraw", this);
        p.add(redrawButton);
        contentPane.add(p, BorderLayout.SOUTH);
        drawingPanel = new GUI2Panel();
        contentPane.add(drawingPanel, BorderLayout.CENTER);
    }

    // create a button
    private JButton makeJButton(String s, ActionListener a) {
        JButton b = new JButton(s);
        b.addActionListener(a);
        return b;
    }

    // respond to an event
    public void actionPerformed(ActionEvent event) {
        Object source = event.getSource();
        if (source == quitButton) {
```

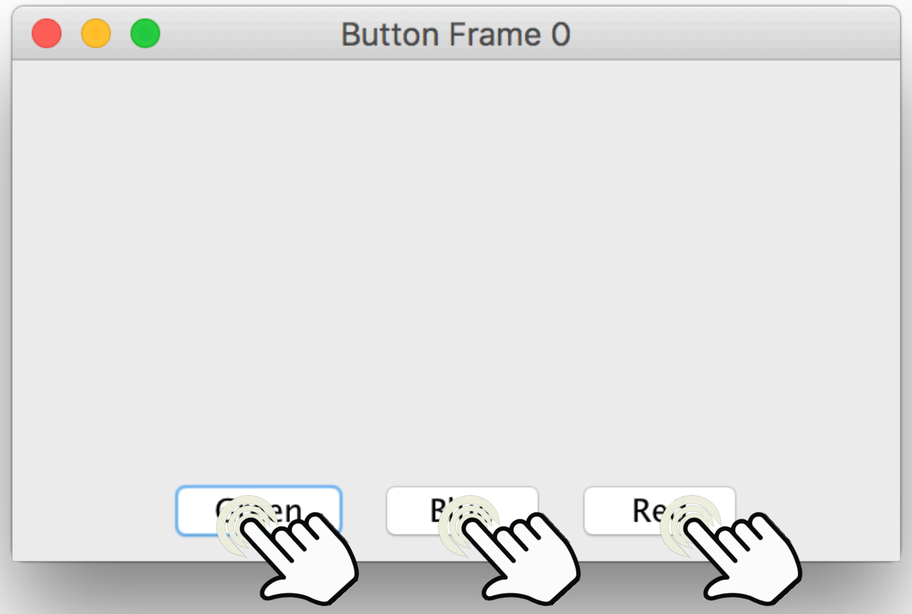


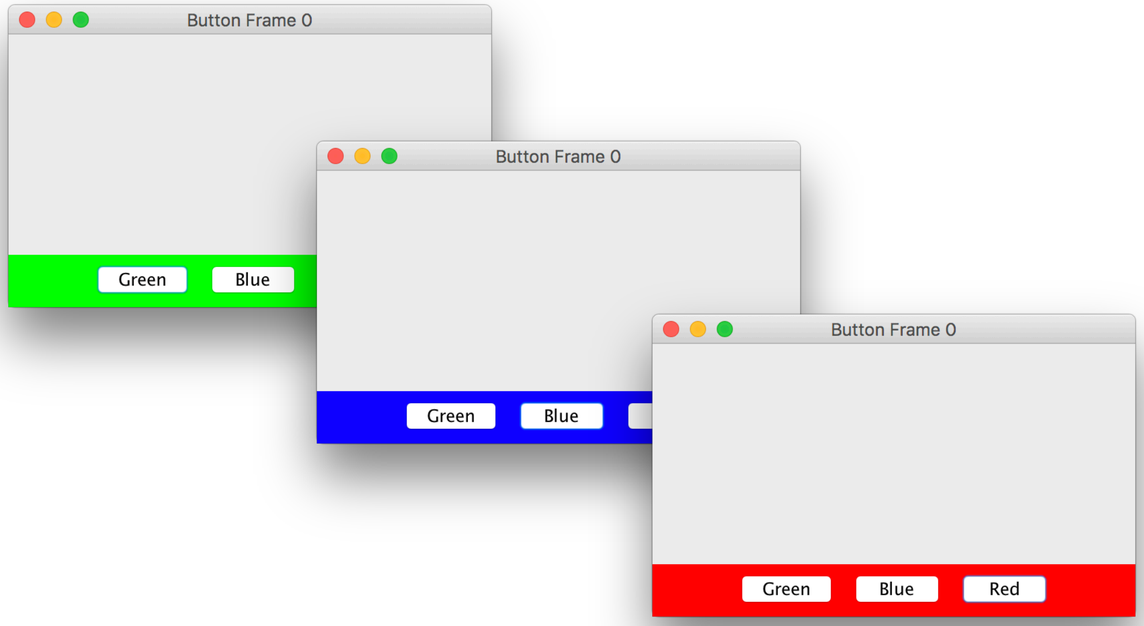
Example - EventHandling2.java

```java
        Container contentPane = this.getContentPane();
        JPanel p = new JPanel();
        quitButton = makeJButton("Quit", this);
        p.add(quitButton);
        redrawButton = makeJButton("Redraw", this);
        p.add(redrawButton);
        contentPane.add(p, BorderLayout.SOUTH);
        drawingPanel = new GUI2Panel();
        contentPane.add(drawingPanel, BorderLayout.CENTER);
    }

    // create a button
    private JButton makeJButton(String s, ActionListener a) {
        JButton b = new JButton(s);
        b.addActionListener(a);
        return b;
    }

    // respond to an event
    public void actionPerformed(ActionEvent event) {
        Object source = event.getSource();
        if (source == quitButton) {
            System.exit(0);
        }
        else if (source == redrawButton) {
            drawingPanel.repaint();
        }
    }
}
```



A button panel

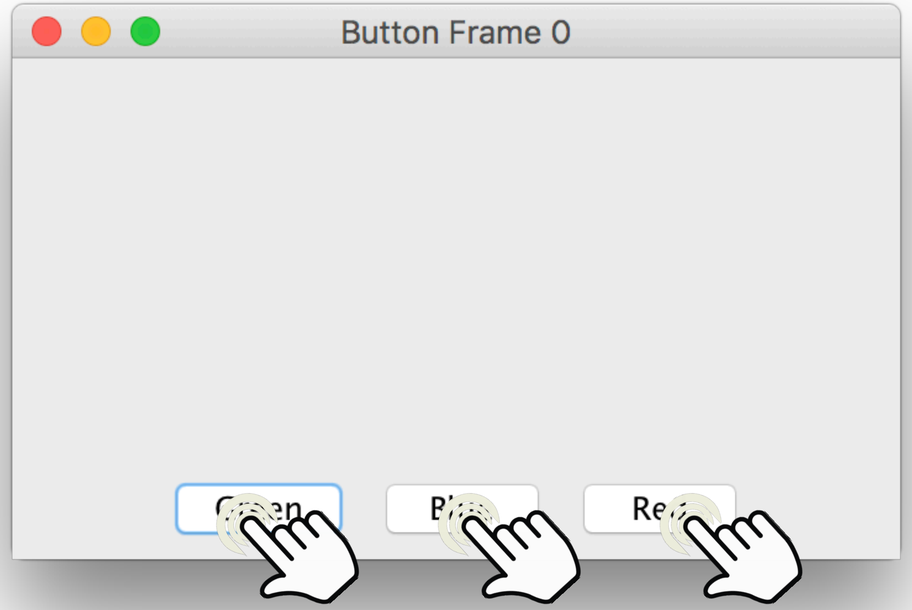


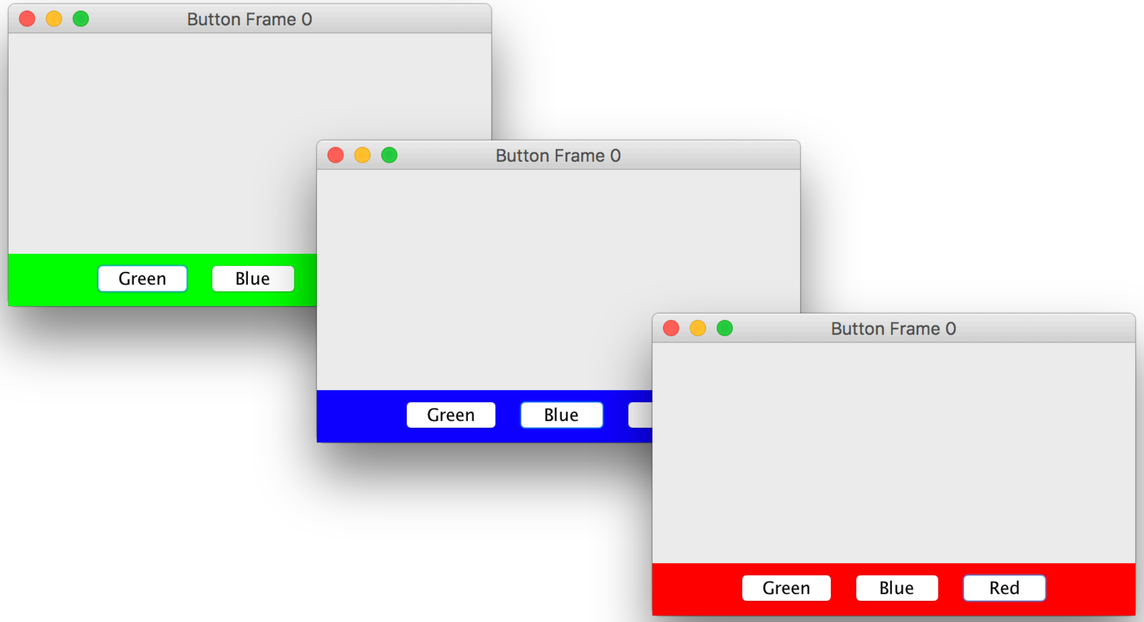


What’s the best design? A button panel example

The requirement:

- A panel with three buttons, labelled “red”, “blue”, “green”;
- Clicking the red button (for example) changes the background colour of the panel to red.
- To do this *each button (event source) must have an attached listener* to carry out the colour change action.





A button panel

The requirement:

- A panel with three buttons, labelled “red”, “blue”, “green”;
- Clicking the red button (for example) changes the background colour of the panel to red.
- To do this *each button (event source) must have an attached listener* to carry out the colour change action.

Option 0: make the `JFrame` the `ActionListener`

- Code in **`ButtonFrame0.java`**
- Works correctly.
- But ….

ButtonFrame0.java

```
/*
    ButtonFrame0.java

    Solution for Button Panel problem

    This class sets the JFrame to be the listener for events.
    However, to JFRAME ends up with many responsibilities
    (i.e. low coherence), and so it is better to have a
    separate class to handle the event listening. This idea
    is pursued in the ButtonFrame1, 2, and 3 classes

*/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ButtonFrame0 extends JFrame implements ActionListener {

    private JButton greenButton;
    private JButton redButton;
    private JButton blueButton;
    private Container c;
    private JPanel buttonPanel;
```

ButtonFrame0.java

```java
public class ButtonFrame0 extends JFrame implements ActionListener {

    private JButton greenButton;
    private JButton redButton;
    private JButton blueButton;
    private Container c;
    private JPanel buttonPanel;

    public ButtonFrame0() {
        setTitle("Button Frame 0");

        // size, position and icon
        Toolkit tk = Toolkit.getDefaultToolkit();
        Dimension dim = tk.getScreenSize();
        setSize(dim.width/4, dim.height/4);
        setLocation(new Point(dim.width/4, dim.height/4));

        c = getContentPane();
        buttonPanel = new JPanel();

        // create buttons
        greenButton = new JButton("Green");
        blueButton = new JButton("Blue");
        redButton = new JButton("Red");

        // add buttons to panel
        buttonPanel.add(greenButton);
        buttonPanel.add(blueButton);
        buttonPanel.add(redButton);
```

ButtonFrame0.java

```
    // associate actions to buttons
    greenButton.addActionListener(this);
    blueButton.addActionListener(this);
    redButton.addActionListener(this);

    // add panel to container
    c.add(buttonPanel,"South");
  }

  public void actionPerformed(ActionEvent actionEvent) {
    Object source = actionEvent.getSource();
    System.out.println(actionEvent.getActionCommand());
    if (source == greenButton)
         buttonPanel.setBackground(Color.GREEN);
    else if (source == blueButton)
         buttonPanel.setBackground(Color.BLUE);
    else if (source == redButton)
         buttonPanel.setBackground(Color.RED);
  }

  public static void main(String args[]) {
     JFrame frm = new ButtonFrame0();
     frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     frm.setVisible(true);
  }
}// ButtonFrame
```

A button panel

The requirement:

- A panel with three buttons, labelled “red”, “blue”, “green”;
- Clicking the red button (for example) changes the background colour of the panel to red.
- To do this *each button (event source) must have an attached listener* to carry out the colour change action.

Option 0: make the `JFrame` the `ActionListener`

- Code in **`ButtonFrame0.java`**
- Works correctly.
- But, **`ButtonFrame0`** has a lot of responsibilities and instance fields.
- Buttons are exposed as instance fields in **`ButtonFrame0`**.
- Better to have a ButtonPanel class, which extends `JPanel`.

A button panel

The requirement:

- A panel with three buttons, labelled “red”, “blue”, “green”;
- Clicking the red button (for example) changes the background colour of the panel to red.
- To do this *each button (event source) must have an attached listener* to carry out the colour change action.

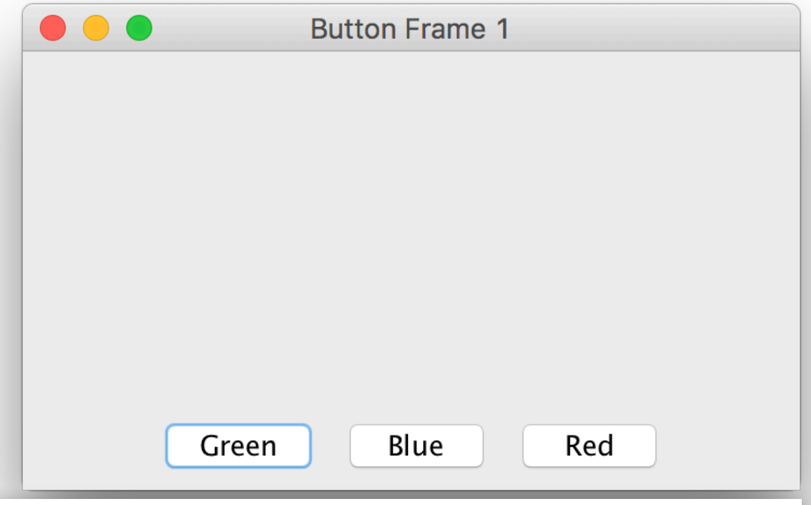
Creating buttons and adding them to a panel is straightforward

```
import javax.swing.*;
import java.awt.*;
public class ButtonPanel1 extends JPanel {
    public ButtonPanel1(){
        // create buttons
        JButton greenButton = new JButton("Green");
        JButton blueButton = new JButton("Blue");
        JButton redButton = new JButton("Red");

        // add buttons to panel
        this.add(greenButton);
        this.add(blueButton);
        this.add(redButton);
    }
}
```

A button panel

`ButtonFrame1` is a test class to put these buttons in a `JFrame`

```java
/*

    Solution for Button Panel problem

    This class reduces the responsibility of the JFrame,
    and uses a ButtonPanel1 object to hold the three
    buttons. This class has more coherence than ButtonFrame0,
    but cannot act as the ActionListener becuase it does not
    have a reference to the buttons.

*/

import javax.swing.*;
import java.awt.*;

public class ButtonFrame1 extends JFrame {

    public ButtonFrame1() {
        setTitle("Button Frame 1");

        // size, position and icon
        Toolkit tk = Toolkit.getDefaultToolkit();
        Dimension dim = tk.getScreenSize();
        setSize(dim.width/4, dim.height/4);
        setLocation(new Point(dim.width/4, dim.height/4));
```

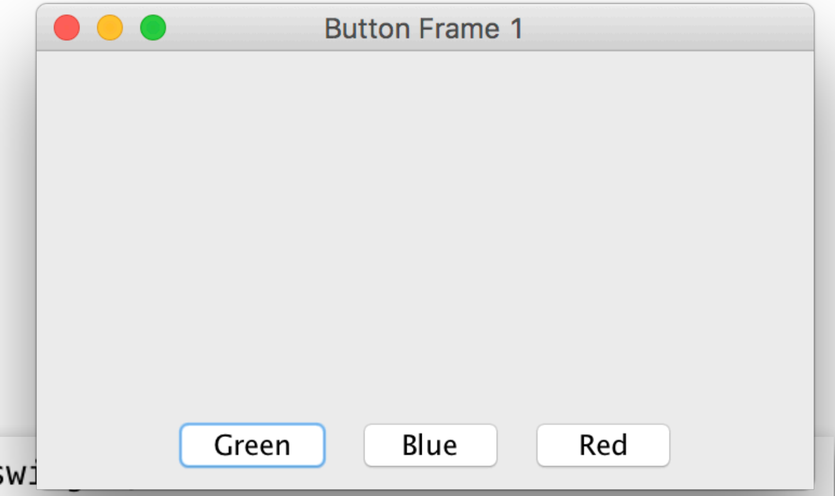



```java
import javax.swing.*;
import java.awt.*;
public class ButtonPanel1 extends JPanel {
    public ButtonPanel1(){
        // create buttons
        JButton greenButton = new JButton("Green");
        JButton blueButton = new JButton("Blue");
        JButton redButton = new JButton("Red");

        // add buttons to panel
        this.add(greenButton);
        this.add(blueButton);
        this.add(redButton);
    }
}
```

A button panel

`ButtonFrame1` is a test class to put these buttons in a `JFrame`

```java
    public ButtonFrame1() {
        setTitle("Button Frame 1");

        // size, position and icon
        Toolkit tk = Toolkit.getDefaultToolkit();
        Dimension dim = tk.getScreenSize();
        setSize(dim.width/4, dim.height/4);
        setLocation(new Point(dim.width/4, dim.height/4));

        Container c = getContentPane();
        JPanel centrePanel = new JPanel();

        // add panels
        ButtonPanel1 bp1 = new ButtonPanel1();
        c.add(bp1, "South");
        c.add(centrePanel,"Center");
    }

    public static void main(String args[]) {
        JFrame frm = new ButtonFrame1();
        frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frm.setVisible(true);
    }
}// ButtonFrame
```




```java
import javax.swi
import java.awt.*;
public class ButtonPanel1 extends JPanel {
    public ButtonPanel1(){
        // create buttons
        JButton greenButton = new JButton("Green");
        JButton blueButton = new JButton("Blue");
        JButton redButton = new JButton("Red");

        // add buttons to panel
        this.add(greenButton);
        this.add(blueButton);
        this.add(redButton);
    }
}
```

The `ActionListener`

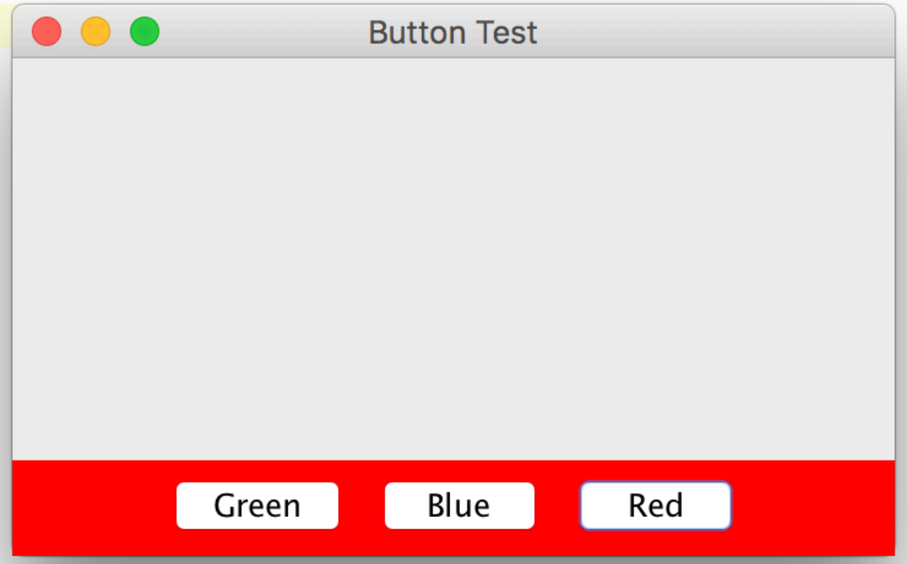
The listener is a class that implements the `ActionListener` interface (in our example it's able to change the colour of the panel)

Which class should be the listener?

1. The `ButtonPanel` itself? Easy, but this breaks encapsulation, giving `ButtonPanel` more responsibilities. Example code : `ButtonPanel2.java` and `ButtonFrame2.java.`
2. A separate `ColorAction` class? This is well encapsulated, but `ColorAction` must refer to the panel it acts on. Example code: `ButtonPanel3.java`, `ButtonFrame3.java`, and `ColorAction3.java`
3. Make `ColorAction` a private ***inner class*** of `ButtonPanel?` Encapsulated and elegant, if `ColorAction` always acts on `ButtonPanel.` Example code: `ButtonPanel4.java` and `ButtonFrame4.java.`

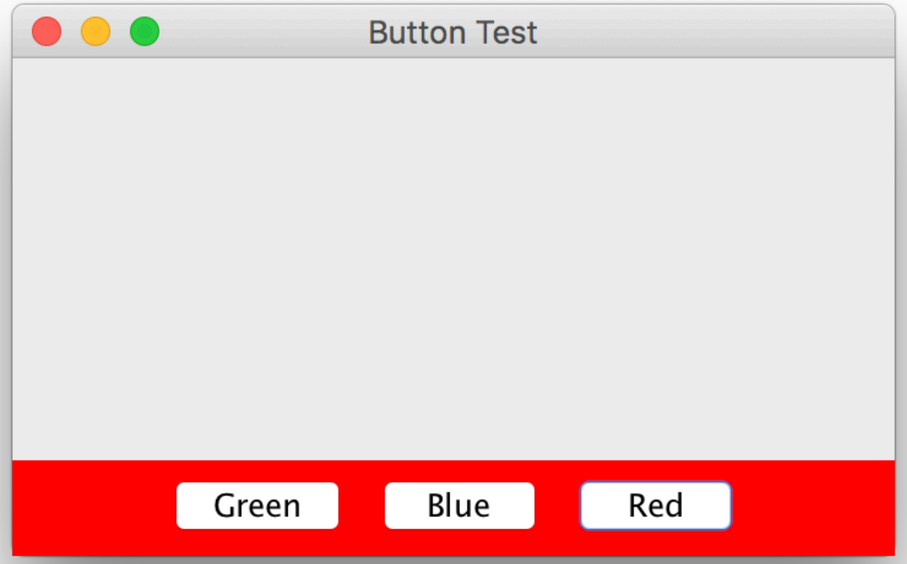
First option: `ButtonPanel2` as listener

```
buttons, and act as the ActionListener.

*/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ButtonFrame2 extends JFrame {

    public ButtonFrame2() {
    setTitle("Button Test");

    // size, position and icon
    Toolkit tk = Toolkit.getDefaultToolkit();
    Dimension dim = tk.getScreenSize();
    setSize(dim.width/4, dim.height/4);
    setLocation(new Point(dim.width/4, dim.height/4));

        Container c = getContentPane();
        JPanel centrePanel = new JPanel();

    // add panels
    c.add(new ButtonPanel2(), "South");
        c.add(centrePanel, "Center");
    }

    public static void main(String args[]) {
    JFrame frm = new ButtonFrame2();
    frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frm.setVisible(true);
    }
}// ButtonFrame
```



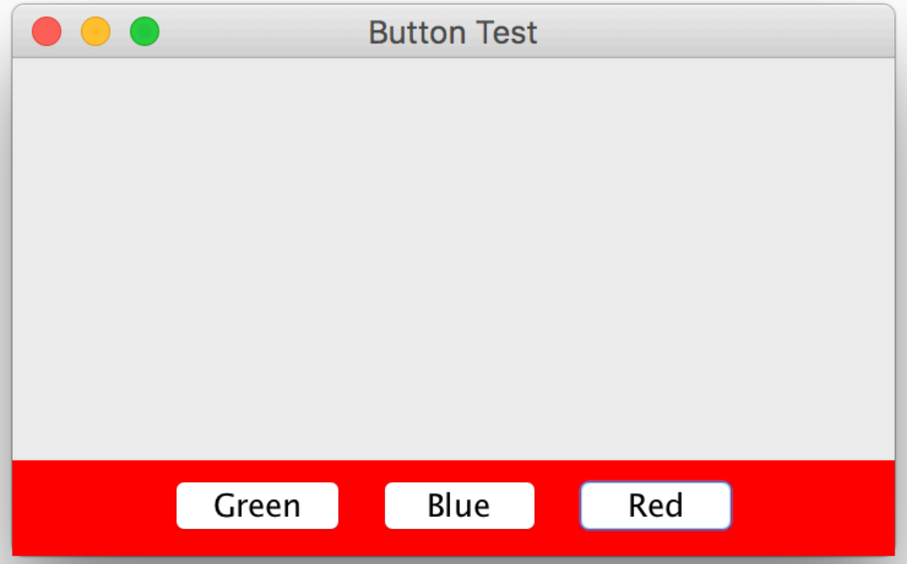
First option: `ButtonPanel2` as listener

```java
/*
    ButtonPanel2.java

    Solution for Button Panel problem

    This class is a ButtonPanel which assembles
    three buttons and adds them to a panel. It
    also acts as a listener, so includes an
    actionPerformed method to take actions
    following a button press.

*/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ButtonPanel2 extends JPanel implements ActionListener {
    private JButton greenButton;
    private JButton blueButton;
    private JButton redButton;

    public ButtonPanel2() {

        // create buttons
        greenButton = new JButton("Green");
        blueButton = new JButton("Blue");
        redButton = new JButton("Red");
```



First option: `ButtonPanel2` as listener

```java
public class ButtonPanel2 extends JPanel implements ActionListener {
    private JButton greenButton;
    private JButton blueButton;
    private JButton redButton;

    public ButtonPanel2() {

        // create buttons
        greenButton = new JButton("Green");
        blueButton = new JButton("Blue");
        redButton = new JButton("Red");

        // add buttons to panel
        add(greenButton);
        add(blueButton);
        add(redButton);

        // associate actions to buttons
        greenButton.addActionListener(this);
        blueButton.addActionListener(this);
        redButton.addActionListener(this);
    }

    public void actionPerformed(ActionEvent actionEvent) {
        Object source = actionEvent.getSource();
        if (source == greenButton)
          this.setBackground(Color.GREEN);
        else if (source == blueButton)
          this.setBackground(Color.BLUE);
        else if (source == redButton)
          this.setBackground(Color.RED);
        }
}
```



The `ActionListener`

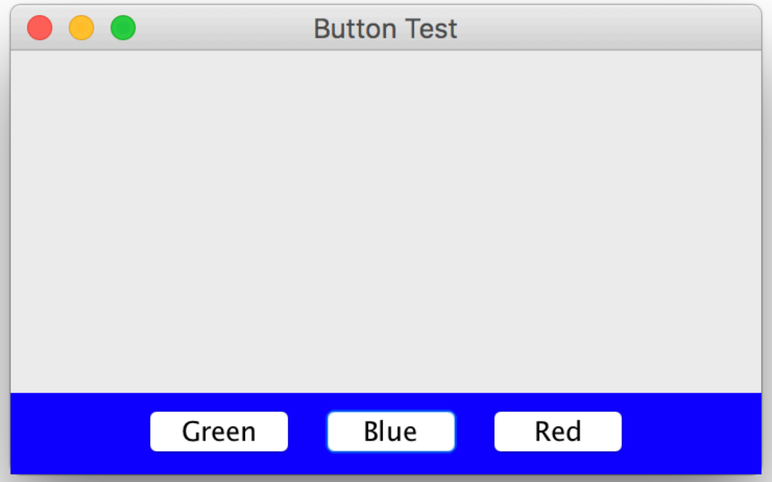
The listener is a class that implements the `ActionListener` interface (in our example it's able to change the colour of the panel)

Which class should be the listener?

1. The `ButtonPanel` itself? Easy, but this breaks encapsulation, giving `ButtonPanel` more responsibilities. Example code : `ButtonPanel2.java` and `ButtonFrame2.java.`
2. A separate `ColorAction` class? This is well encapsulated, but `ColorAction` must refer to the panel it acts on. Example code: `ButtonPanel3.java`, `ButtonFrame3.java`, and `ColorAction3.java`
3. Make `ColorAction` a private ***inner class*** of `ButtonPanel?` Encapsulated and elegant, if `ColorAction` always acts on `ButtonPanel.` Example code: `ButtonPanel4.java` and `ButtonFrame4.java.`

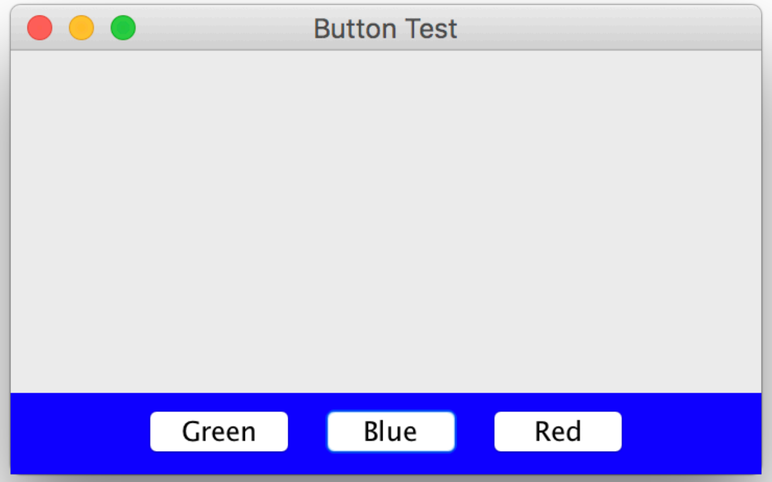
Second option: a `ColorAction` class

```
import javax.swing.*;
import java.awt.*;

public class ButtonFrame3 extends JFrame {

    public ButtonFrame3() {
    setTitle("Button Test");

    // size, position and icon
    Toolkit tk = Toolkit.getDefaultToolkit();
    Dimension dim = tk.getScreenSize();
    setSize(dim.width/4, dim.height/4);
    setLocation(new Point(dim.width/4, dim.height/4));

        Container c = getContentPane();
        JPanel centrePanel = new JPanel();

    // add panels
    c.add(new ButtonPanel3(), "South");
        c.add(centrePanel,"Center");
    }

    public static void main(String args[]) {
    JFrame frm = new ButtonFrame3();
    frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frm.setVisible(true);
    }
}// ButtonFrame
```



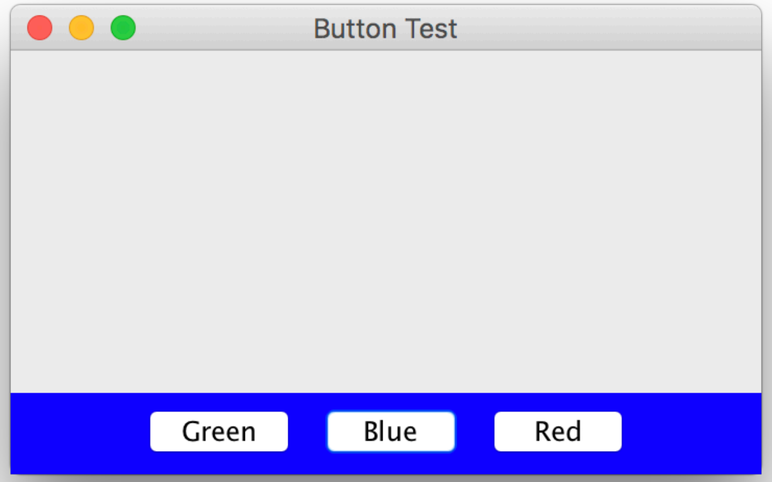
Second option: a `ColorAction` class

```
*/

import javax.swing.*;
import java.awt.*;

public class ButtonPanel3 extends JPanel {
    public ButtonPanel3() {

        // create buttons
        JButton greenButton = new JButton("Green");
        JButton blueButton = new JButton("Blue");
        JButton redButton = new JButton("Red");

        // add buttons to panel
        add(greenButton);
        add(blueButton);
        add(redButton);

        // create button actions
        ColorAction3 greenAction = new ColorAction3(Color.GREEN, this);
        ColorAction3 blueAction = new ColorAction3(Color.BLUE, this);
        ColorAction3 redAction = new ColorAction3(Color.RED, this);

        // associate actions to buttons
        greenButton.addActionListener(greenAction);
        blueButton.addActionListener(blueAction);
        redButton.addActionListener(redAction);
    }

}
```



Second option: a `ColorAction` class

```java
/*

    ColorAction3.java

    Listener class for ButtonPanel3

*/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ColorAction3 implements ActionListener {
    private Color theColor;
    private JComponent theComponent;

    public ColorAction3(Color col, JComponent comp) {
        theColor = col;
        theComponent = comp;
    }

    public void actionPerformed(ActionEvent actionEvent) {
        theComponent.setBackground(theColor);
    }
}
```



The `ActionListener`

The listener is a class that implements the `ActionListener` interface (in our example it's able to change the colour of the panel)

Which class should be the listener?

1. The `ButtonPanel` itself? Easy, but this breaks encapsulation, giving `ButtonPanel` more responsibilities. Example code : `ButtonPanel2.java` and `ButtonFrame2.java`.
2. A separate `ColorAction` class? This is well encapsulated, but `ColorAction` must refer to the panel it acts on. Example code: `ButtonPanel3.java`, `ButtonFrame3.java`, and `ColorAction3.java`
3. Make `ColorAction` a private ***inner class*** of `ButtonPanel`? Encapsulated and elegant, if `ColorAction` always acts on `ButtonPanel`. Example code: `ButtonPanel4.java` and `ButtonFrame4.java`.

Third option: an inner class

```
    This class is a ButtonPanel which assembles
    three buttons and adds them to a panel. It
    uses a ColorAction class to listen, and this
    is an inner class

*/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ButtonPanel4 extends JPanel {
    public ButtonPanel4() {
        // create buttons
        JButton greenButton = new JButton("Green");
        JButton blueButton = new JButton("Blue");
        JButton redButton = new JButton("Red");

        // add buttons to panel
        add(greenButton);
        add(blueButton);
        add(redButton);

        // create button actions
```

All the buttons are encapsulated in the ButtonPanel 4 constructor

Third option: an inner class

```
        add(blueButton);
        add(redButton);

        // create button actions
        ColorAction4 greenAction = new ColorAction4(Color.GREEN);
        ColorAction4 blueAction = new ColorAction4(Color.BLUE);
        ColorAction4 redAction = new ColorAction4(Color.RED);

        // associate actions to buttons
        greenButton.addActionListener(greenAction);
        blueButton.addActionListener(blueAction);
        redButton.addActionListener(redAction);
    }

    private class ColorAction4 implements ActionListener {
        private Color theColor;

        public ColorAction4(Color col) {
            theColor = col;
        }

        public void actionPerformed(ActionEvent actionEvent) {
            // long reference to method in ButtonPanel4
            // ButtonPanel4.this.setBackground(theColor);
            // short reference to method on ButtonPanel4
            setBackground(theColor);
        }
    }
}
```

ColorAction is an ***inner class*** that implements the ActionListener interface

The `ActionListener`

The listener is a class that implements the `ActionListener` interface (in our example it's able to change the colour of the panel)

Which class should be the listener?

1. The `ButtonPanel` itself? Easy, but this breaks encapsulation, giving `ButtonPanel` more responsibilities. Example code : `ButtonPanel2.java` and `ButtonFrame2.java`.
2. A separate `ColorAction` class? This is well encapsulated, but `ColorAction` must refer to the panel it acts on. Example code: `ButtonPanel3.java`, `ButtonFrame3.java`, and `ColorAction3.java`
3. Make `ColorAction` a private ***inner class*** of `ButtonPanel?` Encapsulated and elegant, if `ColorAction` always acts on `ButtonPanel`. Example code: `ButtonPanel4.java` and `ButtonFrame4.java`.

Inner classes

An *inner class* is a class defined inside another class.

Inner class methods can access data from the scope within which they are defined, and this includes data that would otherwise be private.

Why would you want to do this?

- Some classes are only used by one other class, and it makes sense to group them together.
- Inner classes increase encapsulation as the inner class is hidden.
- Inner classes *can* therefore simplify code – if they are used carefully.

For example: ColorAction4 enables the display (ButtonPanel4) to be separated from the ActionListener, without giving the ButtonPanel too many responsibilities.

Only ButtonPanel4 objects can create ColorAction4 objects.

More at http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

Inner class example

```
public class ButtonPanel4 extends JPanel {
    public ButtonPanel4() {
        // create buttons
        JButton greenButton = new JButton("Green");
        JButton blueButton = new JButton("Blue");
        JButton redButton = new JButton("Red");
        // add buttons to panel
        add(greenButton);
        add(blueButton);
        add(redButton);
        // create button actions
        ColorAction4 greenAction = new ColorAction4(Color.GREEN);
        ColorAction4 blueAction = new ColorAction4(Color.BLUE);
        ColorAction4 redAction = new ColorAction4(Color.RED);
        // associate actions to buttons
        greenButton.addActionListener(greenAction);
        blueButton.addActionListener(blueAction);
        redButton.addActionListener(redAction);
    }
    private class ColorAction4 implements ActionListener {
        private Color theColor;
        public ColorAction4(Color col) {
            theColor = col;
        }
        public void actionPerformed(ActionEvent actionEvent) {
            setBackground(theColor);
        }
    }
}
```

`ColorAction4` could be public, so we could access methods from another class by `ButtonPanel4.ColorAction4.actionperformed()`

ColorAction4 objects can also access the other private instance fields of ButtonPanel4 objects.

ColorAction4 can access the setBackground method of the JPanel superclass (ButtonPanel4.this.setBackground()).

Local inner classes

- Local inner classes are defined locally in a block of program code, e.g. in a method.
- Local inner classes are ***completely hidden***.
- Local classes are defined without an access specifier (`public` or `private`) because their scope is restricted to the code block where they are declared.
- Local classes are useful for
  - Implementing interfaces locally
  - User interface call-backs – e.g. a specific response to a button press
  - Local extension of an existing class

```
public class OuterClass extends Something{
   public void someMethod(){
      class LocalInnerClass extends SomethingElse{ … }

         LocalInnerClass c = new LocalInnerClass();
         do stuff with c …
      }
   }
```

Anonymous inner classes

- An anonymous inner class is a local class with no name
- Used when only single objects of a class are needed
- Anonymous inner classes can simplify coding, especially for multiple actions, but can be very confusing

This is the most elegant and concise coding for the ButtonPanel problem. But, it is not easy to figure out what the code does at first glance.

```
public class ButtonPanel5 extends JPanel {
  public ButtonPanel5() {
    makeButton("blue", Color.BLUE);
    makeButton("green", Color.GREEN);
    makeButton("red", Color.RED);
  }
  void makeButton(String name, final Color col) {
    JButton button = new JButton(name);
    add(button);
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent actionEvent) {
        setBackground(col);
      }
```

Any local variables accessed by an instance of an inner class have to be declared *final*.

Summary

Event objects are transmitted from event sources to event listeners.

Range of approaches to implementing actions

Inner classes:

- Can be used to ‘localise understanding’ of the code;
- May make some parts of the code look ‘messy’;
- Some people like them; some don’t.