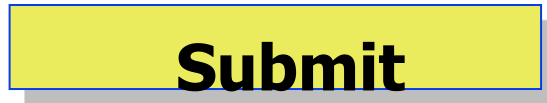
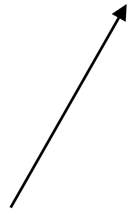
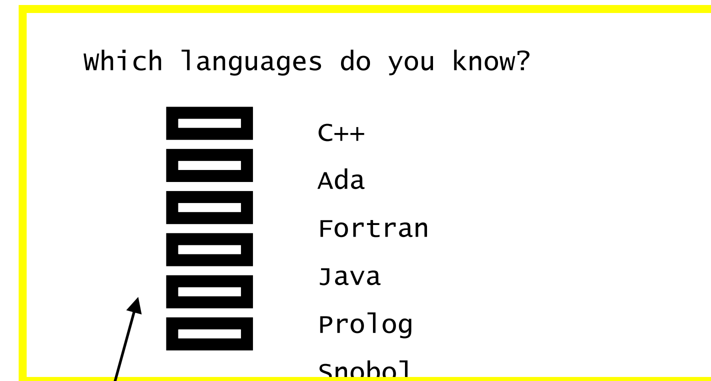
Event driven Programming

Events

- GUIs are event driven
  - Components can wait idly for something to happen - an event
  - A button may wait to be pressed
  - A checkbox may wait to be checked
- An event is a user action: pressing a key, clicking the mouse button, pressing return in a *textfield*







Pressing a Button generates an Event

Selecting a Checkbox generates an Event

The Java Event Model

The Java Event model is based on :

Event Source:

- An object that generates an **Event** (**Button**, **Checkbox**, etc)

Event Listener:

- An object that can receive and process an event

Print

Pressing the button causes a method in the Listener to be invoked

Listener

Code for processing the Event

Event Driven Programming

- To process an event
  1. Create a GUI component

     Example: `submit = new Button("Submit");`
  2. Create an Event Handler

     Example: `b = new ButtonHandler( );`
  3. Register the Event Handler with the component

     Example: `submit.addActionListener(b);`
  4. Sit back and wait for the fun to begin

     Example:

Event Driven Programming

- *Events* are represented by objects that give information about the *event* and identify the *event source*
  - Event sources are typically components, but other kinds of objects can also be event sources
- A *listener* is an object that wants to be notified when a particular event occurs
  - An event source can have multiple *listeners* registered on it.
  - A single listener can register with multiple event sources
- In order for an object to be notified when a particular event occurs, the object
  - Must implement the appropriate `Listener` interface
  - Be registered as an event listener on the appropriate event source
- Event driven programming involves writing the handlers and arranging for the handler to be notified when certain events occur

Listeners for Swing Components

Action	Listener Type
User clicks a button, presses return while typing in a text filed, or chooses a menu item	`ActionListener`
User handling a frame (main window)	`WindowListener`
User presses a mouse button while the cursor is over a component	`MouseListener`
User moves the mouse over a component	`MouseMotionListener`
A component moves, changes size, or changes visibility (low-level event)	`ComponentListener`
A component gains or looses the input focus (low-level event)	`FocusListener`

Action Events

- All classes handling events (in general) must implement the corresponding `Listener` interface, i.e. all its methods
- A class handling action events must implement the `ActionListener` interface
- Consequently, it has to implement the `void actionPerformed(ActionEvent e)` method
- Buttons are widely used ActionEvent generators
- Generally one `ActionListener` will be responsible for handling the events generated by a group of buttons
  - You can tell which button got pressed using the event's `String getActionCommand()` method
  - You can also tell that by using the event's `Object getSource()` method, which returns the Object element that produced the event

Example: Source & Listener in Different Classes

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

public class EventDemo extends Frame
{
  private Button but;
  private ButtonHandler b;

  public EventDemo() {
    this.but = new Button("MyButton");
    this.b = new ButtonHandler( );
    this.but.addActionListener(this.b);
    this.add(but);
  }
}
```

You must include this package

The but button is a Event Source

b is an Event Listener

b is registered to receive Events generated by the but button

Example: Source & Listener in Different Classes

Whenever a button is pressed, this method is called

```
public class ButtonHandler implements ActionListener
{
   public void actionPerformed(ActionEvent e)
   {
      System.out.println("You pressed the button");
   }
}
```

Example: Source & Listener in Same Class

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

public class EventDemo extends Frame
implements ActionListener
{
   private Button but1, but2;
   public EventDemo()
   {
      this.but1= new Button(“MyButton1”);
      this.but2= new Button(“MyButton2”);
      this.but1.addActionListener(this);
      this.but2.addActionListener(this);
      this.add(this.but1);
      this.add(this.but2);
   }
```

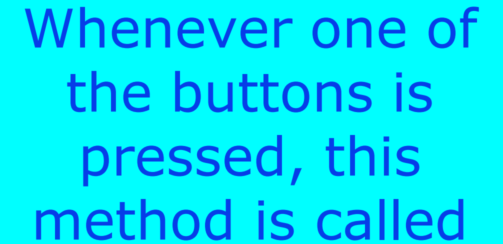
You must include this package

EventDemo is an Event Listener

The but1, but2 buttons are Event Sources

‘this’ is registered to receive Events generated by the but1, but2 buttons

Example: Source & Listener in Same Class




```
public void actionPerformed(ActionEvent e)
{
   Object source = e.getSource();
   if (source == this.but1)
      System.out.println("You pressed MyButton1");
   else if (source == this.but2)
      System.out.println("You pressed MyButton2");
}
}
```

Window Events

- In order to handle window events, a class must implement the `WindowListener` interface
- The `WindowListener` interface contains
  - `void windowActivated(WindowEvent e);`
  - `void windowClosed(WindowEvent e);`
  - `void windowClosing(WindowEvent e);`
  - `void windowDeactivated(WindowEvent e);`
  - `void windowDeiconified(WindowEvent e);`
  - `void windowIconified(WindowEvent e);`
  - `void windowOpened(WindowEvent e);`
- A class that implements `WindowListener` must implement all of these methods!

WindowAdapter

- A class that implements the `WindowListener` interface
  - The methods in this class are empty. The class exists as convenience for creating listener objects
- To use the `WindowAdapter` class:
  - Extend this class to create a `WindowEvent` listener
  - Override the methods for the events of interest
  - Create a listener object using the extended class and then register it with a Window using the window's `addWindowListener()` method
- When the window's status changes the appropriate method in the listener object is invoked, and the `WindowEvent` is passed to it

Example

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

public class SwingFrame
{
   public static void main(String args[]) {
      JFrame win = new JFrame("My First GUI Program");
      win.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e)
         {
            System.exit(0);
         }
      });

      win.setSize(250, 150);
      win.setVisible(true);
   }
}
```

Another Example

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

public class SwingFrame implements ActionListener
{
   private JFrame win;

   public SwingFrame(String title) {
      this.win = new JFrame(title);
      this.win.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e)
         {
            System.exit(0);
         }
      });
      this.win.getContentPane().setLayout(new
FlowLayout());
```

Another Example

```
        for (int i = 0; i < 10; i++) {
            JButton b = new JButton(String.valueOf(i));
            b.addActionListener(this);
            this.win.getContentPane().add(b);
        }
        this.win.pack();
        this.win.show();
    }
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button " +
                 e.getActionCommand() + " was pressed " );
    }
    public static void main(String args[]) {
        SwingFrame f = new SwingFrame("My First GUI");
    }
}
```

Mouse Events

- What about mouse events?

Component	Listener Interface	Listener Methods

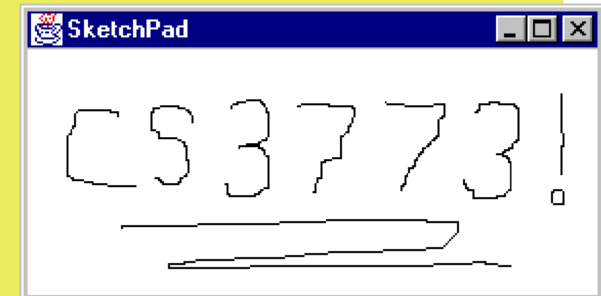
`Component`	`MouseListener`	`mousePressed(MouseEvent e)`
		`mouseClicked(MouseEvent e)`
		`mouseReleased(MouseEvent e)`
		`mouseEntered(MouseEvent e)`
		`mouseExited(MouseEvent e)`

`Component`	`MouseMotionListener`	`mouseDragged(MouseEvent e)` `mouseMoved(MouseEvent e)`

- For example, if you want to process `Mouse` events for a `JFrame`, you must
  - use a class that implements the `MouseListener` and/or `MouseMotionListener`
  - Register the `Listener` with the `JFrame`
  - Whenever a mouse event occurs, the appropriate method is called

Example

```
import java.awt.*;
import java.awt.event.*;
public class SketchPad extends Panel
       implements MouseListener, MouseMotionListener {

   private int x, y;

   public SketchPad() {
      this.addMouseListener(this);
      this.addMouseMotionListener(this);
   }

   // MouseMotionListener methods
   public void mouseDragged(MouseEvent e) {
      getGraphics().drawLine(this.x, this.y, e.getX(),
e.getY());
      this.x = e.getX();
      this.y = e.getY();
   }
```



Example

```
public void mouseMoved(MouseEvent e) {}

// MouseListener methods

public void mousePressed(MouseEvent e) {
   this.x = e.getX();
   this.y = e.getY();
}
public void mouseClicked(MouseEvent e) {}

public void mouseReleased(MouseEvent e) {}

public void mouseEntered(MouseEvent e) {}
```

Example

```
    public void mouseExited(MouseEvent e) {}

    public static void main(String args[]) {
        Frame f = new Frame("SketchPad");
        f.setSize(300, 150);
        f.add(new SketchPad());
        f.setVisible(true);
    }
}
```

MouseEvents

- Useful `MouseEvent` methods
  - `int getClickCount()` returns the number of clicks associated with this event
  - `int getX()`, and `int getY()` return the x and y coordinates of the location of the event
  - `Point getPoint()` returns the x, y location of the event as a Point object
  - `boolean isPopupTrigger()` indicates whether the event is the trigger for popup menus on this platform (usually a right mouse button release)
  - `boolean isShiftDown()` returns true if the shift key was down when this event occurred. This method is inherited from `InputEvent`
  - `boolean isMetaDown()` returns true if this was a right mouse button event. This method is inherited from `InputEvent`

Keyboard Events

- Any `Component` that has the input focus can generate `KeyEvent` objects
  - The object with the input focus is the object to which keyboard keystrokes are sent
  - Only one object at a time may have the input focus
- A `KeyListener` object can process `KeyEvent` objects
- Three methods are required by the `KeyListener` interface to process the three types of `KeyEvent` objects
  - A key pressed event is generated when any key is pressed, and before the key is released
  - A key released event is generated when a key is released
  - A key typed event is generated when the key that is being released is one of the Ascii printable characters
    - Not generated for alt, ctrl, insert, home, F1, etc… These are called the *action* keys
- Useful methods of the `KeyEvent` class

Keyboard Events

- `char getKeyChar()` returns the Unicode character value of the key that was typed
  - Only useful for key typed events since the action keys do not have corresponding Unicode values
- `int getKeyCode()` returns the virtual key code of the key that was pressed
  - This is the way to determine which action key was pressed
  - Virtual key codes are defined as constants of the `KeyEvent` class
  - KeyEvent.VK_ALT, KeyEvent.VK_CONTROL etc…
- `boolean isActionKey()` returns true if the key associated with the event is one of the action keys
- `boolean isAltDown()`,
  `boolean isControlDown()`,
  `boolean isShiftDown()` return true if the corresponding key is down during this event

Example

```
import java.awt.Component;
import java.awt.event.*;
public class KeyCloser extends KeyAdapter {
   public void keyTyped(KeyEvent e){
      char typed = e.getKeyChar();
      System.out.print("You typed an " + typed +"...");
      switch (typed){
         case 'x': case 'X':
            System.out.println("and I shut down.");
            System.exit(0);
            break;
         case 'v': case 'V':
            System.out.println("and I hide the
  component.");
            Component c = e.getComponent();
            c.setVisible(false);
            break;
         default:
            System.out.println("and I ignored your
  command.");
      }
   } // end keyTyped
}
```

Example

```
import java.awt.*;

public class KeyFrame extends Frame {

   public static void main (String[] args){
      KeyFrame win = new KeyFrame();
      win.setSize(200,200);
      win.setLocation(300,300);

      KeyCloser closer = new KeyCloser();

      win.addKeyListener(closer);
      win.setVisible(true);
   } // end main
}
```

TextArea

- A `TextArea` object is a multi-line text editing area
- `TextArea` inherits from `TextComponent`, has similar capabilities to `TextField and` handles internally scrolling
- `TextAreas` do not generate `ActionEvent` objects, unlike `TextFields`
  - `TextArea` objects generate **`TextEvent`** objects when the text changes (`TextField` objects generate `TextEvents` as well)
  - `TextListener` objects handle `TextEvent` objects
- Creating `TextArea` objects
  - Default no-arg constructor gives a default size usually dictated by the `Layout Manager` of the `TextAreas` `Container`
  - Two-arg constructor takes an integer number of rows and columns of text to display
  - Other constructors allow specification of an initial String to display, whether scrollbars should be displayed or not

TextArea

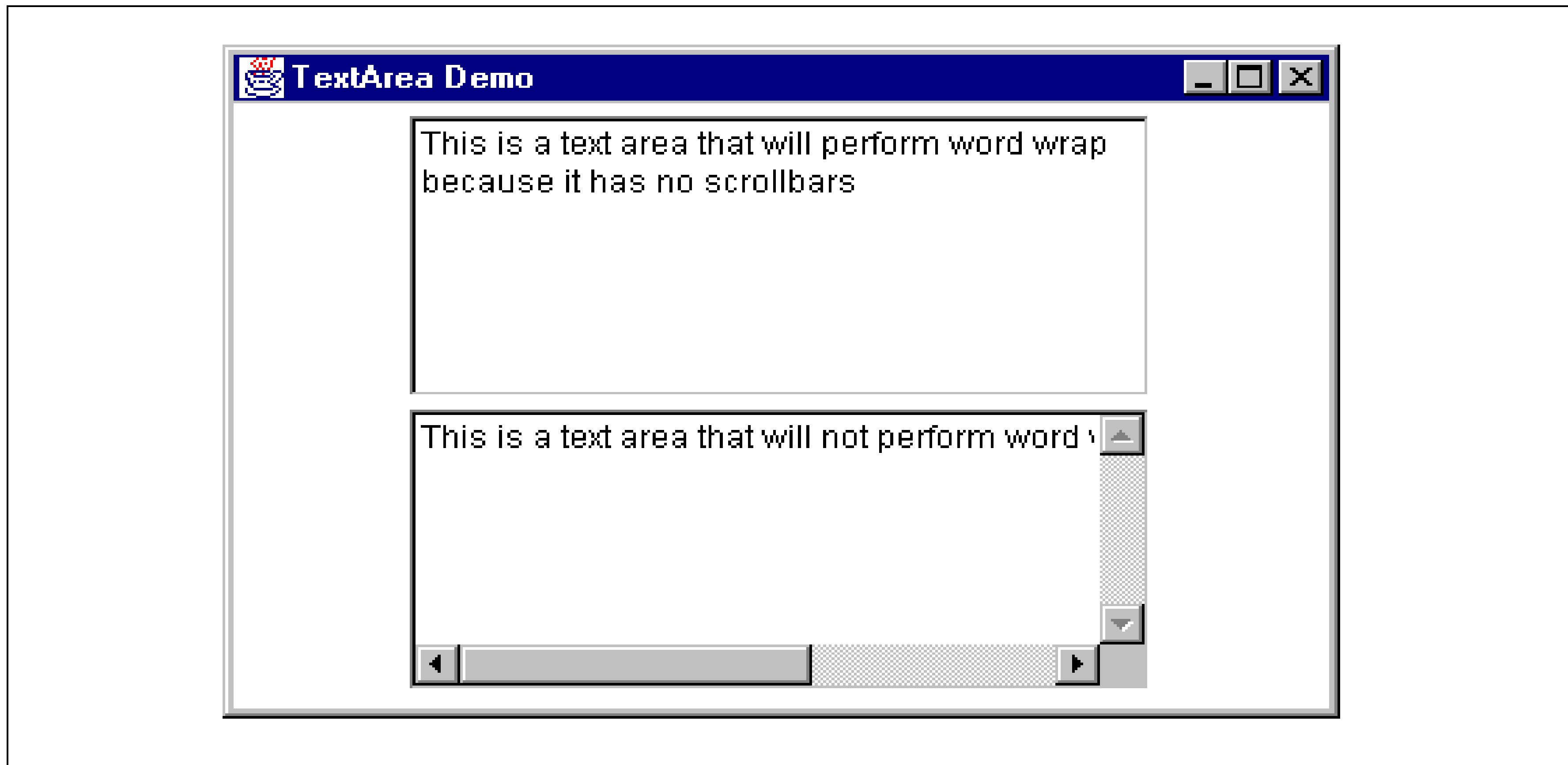
- Scrollbars
  - Can be horizontal, vertical or both
  - The use of a horizontal scrollbar dictates whether words will be wrapped or not
  - See the `TextArea` documentation for more info

- Note: The `JTextArea` class
  - Does not handle internally scrolling
  - Implements the swing `Scrollable` interface
  - This allows it to be placed inside a `JScrollPane` if scrolling behavior is desired

TextArea Example

```
import java.awt.*;

public class TextAreaDemo {

   public static void main(String args[]) {
      Frame f = new Frame("TextArea Demo");
      f.setSize(450, 250);

      f.setLayout(new FlowLayout());

      f.add(new TextArea("", 6, 35,
                         TextArea.SCROLLBARS_NONE));
      f.add(new TextArea("Initial text", 6, 35,
                         TextArea.SCROLLBARS_BOTH));

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

TextArea Example



ScrollPane: A Scrolling Container Class

- Sometimes an object is too large to be displayed entirely
  - Example: a 1000 x 1000 component in a 600 x 600 Frame
- A `ScrollPane` is a `Container` used to hold objects that are too large to be displayed
- `ScrollPanes` provide horizontal and vertical scroll bars as needed to “pan” around the object that they contain

JTextArea Example

```
import javax.swing.*;
public class JTextAreaDemo {
   public static void main(String args[]) {
      JFrame f = new JFrame("ScrollPane Demo");
      JScrollPane s = new JScrollPane();
      JTextArea t = new JTextArea("Test", 5, 20);
      s.getViewport().add(t);
      f.getContentPane().add(s);
      t.setVisible(true);
      s.setVisible(true);
      f.pack();
      f.setVisible(true);
   }
}
```

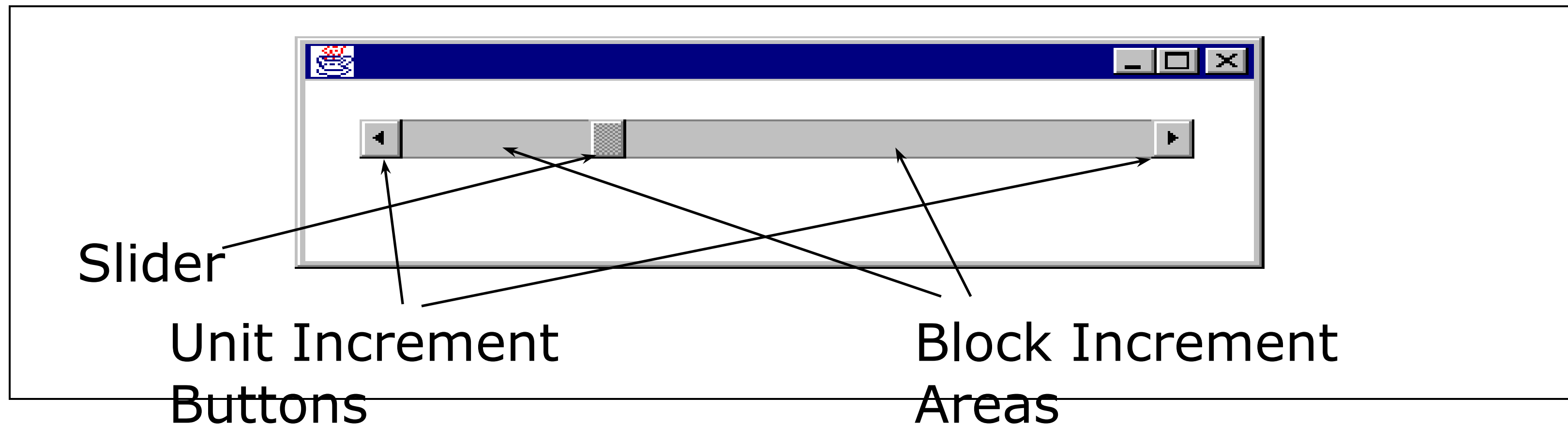
JTextArea Example

Scrollbar

- A stand alone `Component` that allows user to scroll through a range of values
- `Scrollbars` implement the interface `Adjustable`
  - Generate `AdjustmentEvent` objects
  - `AdjustmentEvent` objects are handled by `AdjustmentListener` objects
- A registered `AdjustmentListener` is notified each time the `Scrollbar` is moved
- Scrollbars are oriented either horizontally or vertically
  - Specified in a constructor with the class constants `Scrollbar.HORIZONTAL` or `Scrollbar.VERTICAL`

Scrollbar

```
Scrollbar(int orientation, int value, int visible,
          int minimum, int maximum)

Scrollbar s = new Scrollbar(Scrollbar.HORIZONTAL, 10,
                            1, 0, 255);
```



JScrollBar Example

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JColorMixer extends JFrame
                implements AdjustmentListener {

   JScrollBar red;
   JScrollBar blue;
   JScrollBar green;

   JLabel redLabel = new JLabel("Red: 128");
   JLabel greenLabel = new JLabel("Green: 128");
   JLabel blueLabel = new JLabel("Blue: 128");

   JPanel sliders = new JPanel();

   public static void main(String args[]) {
      JColorMixer cm = new JColorMixer();
      cm.setVisible(true);
   }

   public void adjustmentValueChanged(AdjustmentEvent e) {
      repaint();
   }
```

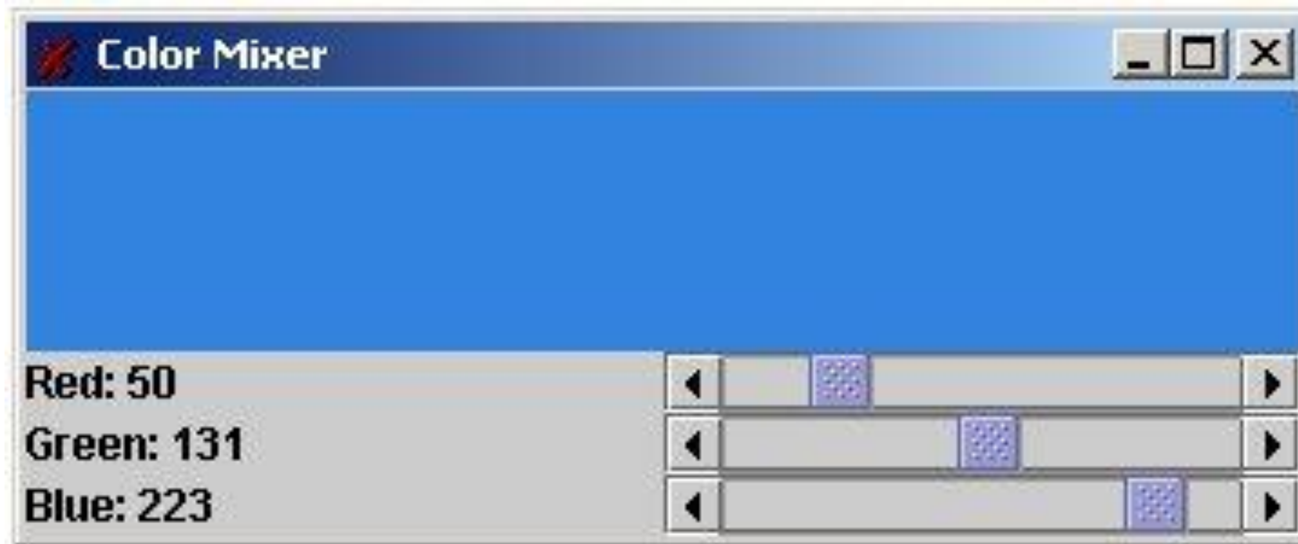
JScrollBar Example

```
public JColorMixer() {
   super("Color Mixer");

   this.sliders.setLayout(new GridLayout(3,2));
   this.red = new JScrollBar(JScrollBar.HORIZONTAL,
                     128, 1, 0, 256);
   this.red.addAdjustmentListener(this);
   this.green = new JScrollBar(JScrollBar.HORIZONTAL,
                     128, 1, 0, 256);
   this.green.addAdjustmentListener(this);
   this.blue = new JScrollBar(JScrollBar.HORIZONTAL,
                     128, 1, 0, 256);
   this.blue.addAdjustmentListener(this);

   this.sliders.add(this.redLabel);
   this.sliders.add(this.red);
   this.sliders.add(this.greenLabel);
   this.sliders.add(this.green);
   this.sliders.add(this.blueLabel);
   this.sliders.add(this.blue);
```

JScrollBar Example

```
        this.getContentPane().add(this.sliders, BorderLayout.SOUTH);
    }

    public void paint(Graphics g) {
        this.redLabel.setText("Red: " + this.red.getValue());
        this.greenLabel.setText("Green: " +
                                          this.green.getValue());
        this.blueLabel.setText("Blue: " +
                                       this.blue.getValue());
        this.getContentPane().setBackground(new
             Color(this.red.getValue(), this.green.getValue(),
                                      this.blue.getValue()));
        super.paint(g);
    }
}
```

JScrollBar Example



Using Menus with Frames

- A `JFrame` can contain a single `JMenuBar` object
  - **`public void setJMenuBar(JMenuBar)`**
- A `MenuBar` is a container for `Menu` objects
- A `Menu` represents a drop down menu GUI object
  - Think of the File, Edit, and Help menus available in most applications
- A `Menu` is a container for `MenuItem` objects
  - `Menu` is a subclass of `MenuItem`
  - By adding `Menus` to `Menus`, you get a cascading menu effect

Using Menus with Frames

- A `MenuItem` represents a menu choice
  - Selection of a `MenuItem` may invoke some action (if this is a simple choice type `MenuItem`) or it may open another `Menu`
  - `MenuItems` generate `ActionEvent` objects when they are selected
    - By adding `ActionListener` objects to each `MenuItem`, we can define behaviors for each `MenuItem`
- There are an infinite number of ways to build `Menus` and assign `ActionListeners` to the `MenuItems`
- Example
  - Note that the constructor for `MenuTest` gets very long when dealing with all of the `MenuItems`

JMenu Example

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

public class JMenuTest extends JFrame {

   JMenuItem fileNew = new JMenuItem("New");
   JMenuItem fileOpen = new JMenuItem("Open...");
   JMenuItem fileSave = new JMenuItem("Save...");
   JMenuItem fileExit = new JMenuItem("Exit");

   public JMenuTest() {
      super("Menu Test");
      this.setSize(300, 200);

      JMenuBar menuBar = new JMenuBar();
      this.setJMenuBar(menuBar);
```

JMenu Example

```
        JMenu fileMenu = new JMenu("File");
        menuBar.add(fileMenu);
        fileMenu.add(this.fileNew);
        fileMenu.add(this.fileOpen);
        fileMenu.add(this.fileSave);
        fileMenu.addSeparator();
        fileMenu.add(this.fileExit);

        this.fileOpen.addActionListener
          (new OpenHandler(this));
        this.fileExit.addActionListener(new ExitHandler());
        this.addWindowListener(new ExitHandler());
    } // end constructor

    public static void main(String args[]) {
        JMenuTest mt = new JMenuTest();
        mt.setVisible(true);
    }
}
```

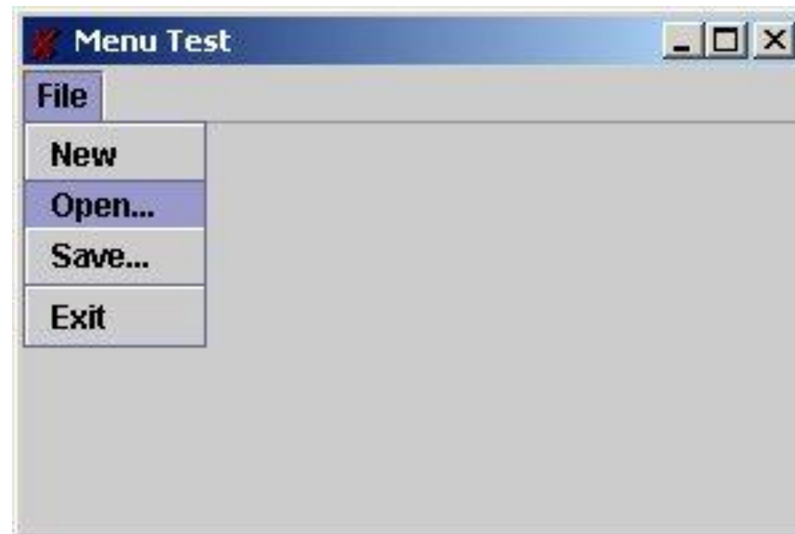
JMenu Example

```
public class ExitHandler extends WindowAdapter
 implements ActionListener {

   public void actionPerformed(ActionEvent e) {
      System.err.println("Exiting normally.");
      System.exit(0);
   }

   public void windowClosing(WindowEvent e){
      System.err.println("Exiting normally.");
      System.exit(0);
   }
}
```

JMenu Example

```
public class OpenHandler implements ActionListener {

   JFrame parent;

   public OpenHandler(JFrame f){
      parent = f;
   }

   public void actionPerformed(ActionEvent e) {
      FileDialog fd;
      fd = new FileDialog(parent, "Open",
                          FileDialog.LOAD);
      fd.setVisible(true);
      System.out.println("Open dialog selected: "
                         + fd.getFile());
   }
}
```

JMenu Example



SubMenus

```
public SubMenuTest() {
        super("SubMenu Test");
        setSize(300, 200);

        MenuBar menuBar = new MenuBar();
        setMenuBar(menuBar);

        Menu fileMenu = new Menu("File");
        Menu subMenu = new Menu("Submenu");

        menuBar.add(fileMenu);

        fileMenu.add(fileNew);
        fileMenu.add(fileOpen);
        fileMenu.add(fileSave);
        fileMenu.addSeparator();
        fileMenu.add(subMenu);

        subMenu.addSeparator();
        subMenu.add(fileExit);
        subMenu.addSeparator();

 // ... rest of code is the same
```

Dialogs

- A `Dialog` is a `Window` with a title bar, commonly used to collect information needed by an application
- A `Dialog` can be either modal or non-modal
  - Modal dialogs prevent any other window in an application from receiving input until the dialog is dismissed (closed)
    - Use a modal dialog when the user should not be allowed to perform other operations in the application until the dialog is complete (a file open dialog for example)
    - Example: Asking for confirmation for deleting a file
  - Non-modal dialogs allow the user to work in other windows of the application while the dialog remains open
    - Use a non-modal dialog when a user is allowed to do other work in the application while leaving the dialog open

Dialogs

- To create a `Dialog`, you must supply a parent `Frame` object
  - This represents the "parent" application object
- `Dialogs` can also have `Dialogs` as parents
- `Dialogs` are `Containers` and typically contain other awt/swing `Component` objects (like `TextFields`) used to collect information for the application
- Steps for creating a Dialog:
  - write a class MyDialog that extends Dialog, which displays/retrieves appropriate information
  - create a MyDialog instance, attach it to the Frame in your program, and call MyDialog.setVisible(true) to display it.
  - use a data retrieval method, such as actionListening, to obtain the information

Using PopupMenu Objects

- In GUI applications, a `PopupMenu`, or context menu is often used to provide context sensitive menu choices
- The available choices depend upon where the mouse has been clicked
- `PopupMenus` can belong to any GUI `Component`, not just a `Frame` object
- `PopupMenus` generally appear in response to a right mouse click on a `Component`
    - Given a `MouseEvent` object e, you may invoke `e.isPopupTrigger()` to determine if this event is the popup trigger for your platform
- Example: A menu of choices to manipulate a Vehicle when a user right clicks on a Vehicle object in a simulation display