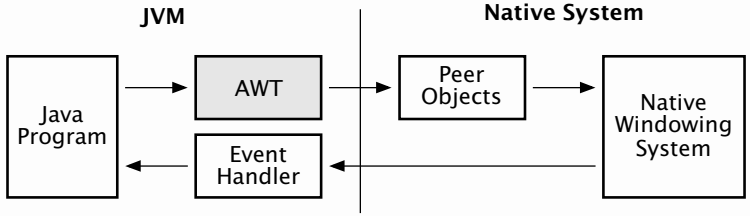
GUI

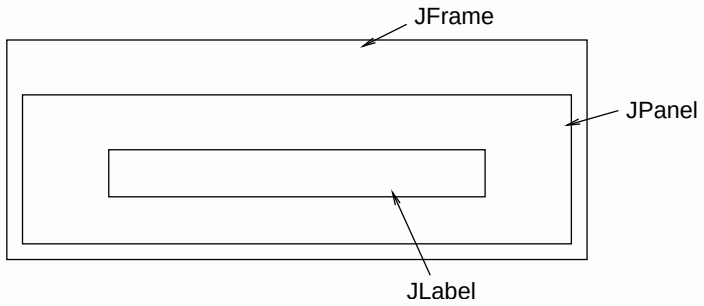
Java supports GUI development through the AWT and the JFC Swing packages.

- **Abstract Window Toolkit (AWT)**
  The AWT provides connection Java application and native GUI. The AWT components depend on native code counterparts (called peers) to handle their functionality.



- **JFC Swing**
  Swing implements a set of GUI components that build on AWT technology and provide a pluggable look and feel. Swing is implemented entirely in the Java and do not depend on peers to handle their functionality.

Simple GUI Application



Simple GUI Application (cont.)

At first, we create a label:

```
JLabel text = new JLabel("Hello, world...");
text.setForeground(Color.RED);
Font  font = new Font("serif", Font.BOLD|Font.ITALIC, 24);
text.setFont(font);
```

Then we create a panel and add the label to the panel:

```
JPanel panel = new JPanel();
panel.setBackground(Color.WHITE);
panel.setBorder(BorderFactory.createEmptyBorder(10, 30,
                10, 30));
panel.add(text);
```

Note: The classes **Color** and **Font** belong to the **java.awt** package, the classes **JLabel** and **JPanel** to the **javax.swing** package.

Simple GUI Application (cont.)

Finally, we create and show a main frame:

```
JFrame.setDefaultLookAndFeelDecorated(true);
JFrame frame = new JFrame("GUI Application");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.getContentPane().add(panel);
frame.pack();
frame.setVisible(true);
```

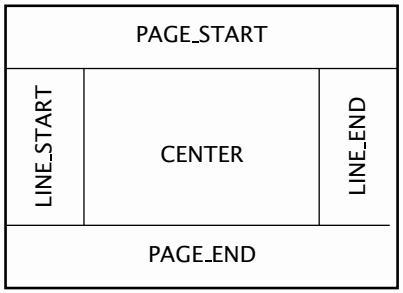
We put all the previous code in a (static) method `createGUI()` and call it from the `main()` method:

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

Layout Management

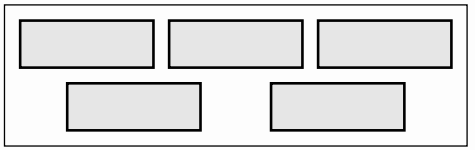
Layout management is a process of determining size and position of components. By default, each container has a layout manager – an object that performs layout management for the components within the container. Java provides several standard layout managers.

- **BorderLayout**
  The `BorderLayout` has five cells called `PAGE_START`, `PAGE_END`, `LINE_START`, `LINE_END` and `CENTER`.

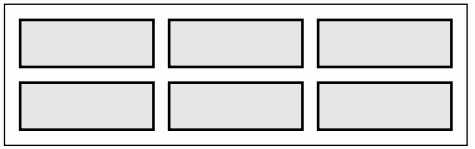


Layout Management (cont.)

- **FlowLayout**
  The simple layout manager "flows" components into the window. The components can be aligned and space between them can be specified.



- **GridLayout**
  The `GridLayout` manager's strategy is to make each cell exactly the same size so that rows and columns line up in a regular grid.

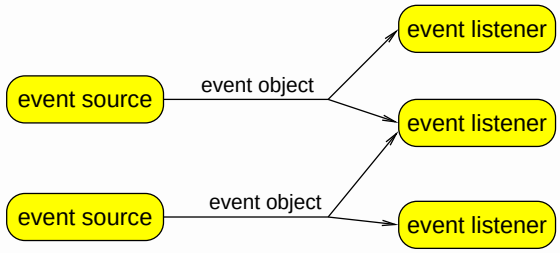


Events and Event Listeners

Evens are instances of subclasses of the **java.util.EventObject** class. They represent informations about particular events that occurred.

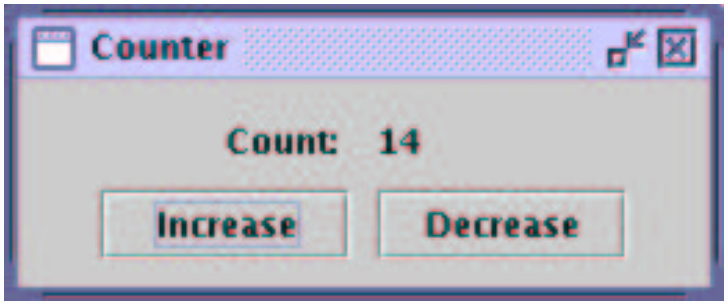
Event Listeners are objects implementing subinterfaces of the **java.util.EventListener** interface.

Most of events and event listener interfaces concerning GUI it defined in the **java.awt.util** package.



Note: Multiple listeners can register to notified of events of a particular type from a particular source. Also the same listener can listen to notifications from different objects.

Example – Counter




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

public class Counter implements ActionListener {
    private int count = 0;
    private JLabel text, value;
    private JButton increase, decrease;
```

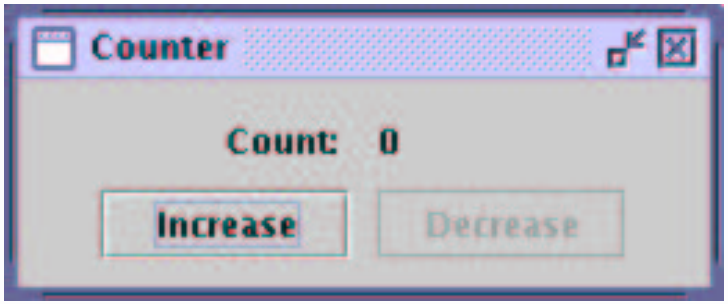
Example – Counter (cont.)

```
public void createGUI() {
    text = new JLabel("Count: ");
    text.setHorizontalAlignment(SwingConstants.RIGHT);
    value = new JLabel("0");
    increase = new JButton("Increase");
    decrease = new JButton("Decrease");
    increase.addActionListener(this);
    decrease.addActionListener(this);
    updateLabel();

    JPanel panel = new JPanel();
    panel.setBorder(BorderFactory.createEmptyBorder(10, 30,
                    10, 30));
    panel.setLayout(new GridLayout(2, 2, 10, 5));
    panel.add(text);
    panel.add(value);
    panel.add(increase);
    panel.add(decrease);
      . . .
```

Example – Counter (cont.)

```
    JFrame.setDefaultLookAndFeelDecorated(true);
    JFrame frame = new JFrame("Counter");
    frame.getContentPane().add(panel);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);
}

private void updateLabel() {
    value.setText(Integer.toString(count));
    decrease.setEnabled(count > 0);
}

public void actionPerformed(ActionEvent e) {
    Object source = e.getSource();
    if (source == increase) {
        count++; updateLabel();
    } else if (source == decrease) {
        if (count > 0) { count--; updateLabel(); }
    }
}
```

Example – Counter (cont.)



Example – Counter (cont.)

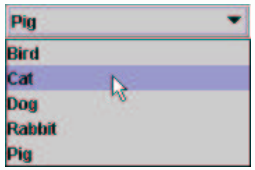
The other possible way how to add action listeners to buttons:

```
increase.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        count++; updateLabel();
    }
});

decrease.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (count > 0) { count--; updateLabel(); }
    }
});
```

Components

Overview of the most important components from the **javax.swing** package:

JButton, JCheckBox, JRadioButton



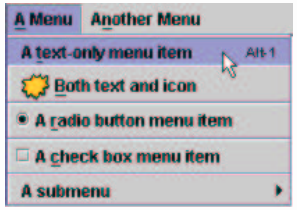
JComboBox

JSlider

JList

JSpinner

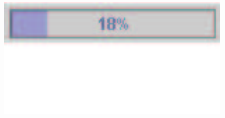
JTextField



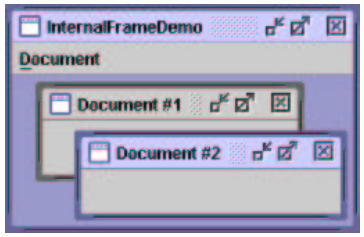
JMenu, JMenuItem
JMenuBar

Components (cont.)

JLabel



JProgressBar

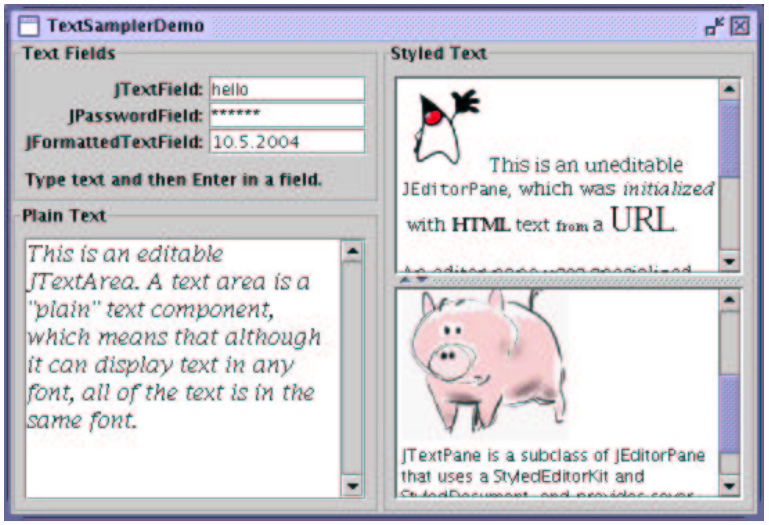


JInternalFrame

JTable

JTree

Components (cont.)



JTextField, JFormattedTextField, JPasswordField
JTextArea, JEditorPane, JTextPane

Containers

Containers are components that can contain other components.

JPanel

JScrollPane

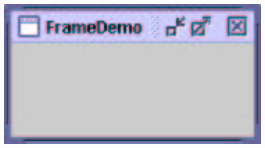
JToolBar

JSplitPane

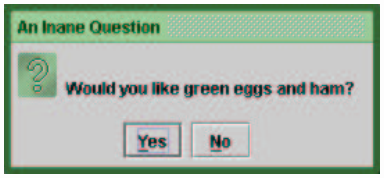
JTabbedPane

Top-Level Containers

Top level containers:

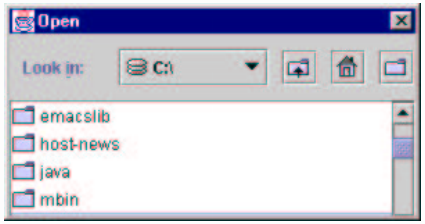


JFrame



JDialog

Predefined dialogs:



JFileChooser

JColorChooser

Important Classes from java.awt

Some important classes from the **java.awt** package:

- **Dimension** – width, height
- **Rectangle** – x, y, width, height
- **Insets** – left, right, top, bottom
- **Point** – x, y
- **Color** – e.g, Color.RED, ...
- **Cursor** – e.g., Cursor.WAIT_CURSOR
- **Font**, **FontMetrics**

The JComponent Class

The **JComponent** class is a common superclass of all Swing components.

Overview of methods:

- void setForeground(Color fg), Color getForeground()
- void setBackground(Color bg), Color getBackground()
- void setFont(Font font), Font getFont()
- void setCursor(Cursor cursor), Cursor getCursor(), boolean isCursorSet()
- void setName(String name), String getName()
- void setToolTipText(String text), String getToolTipText()
- void setEnabled(boolean enabled), boolean isEnabled()
- void setVisible(boolean aFlag), boolean isVisible(), boolean isShowing()

The JComponent Class (cont.)

- void repaint(), void repaint(int x, int y, int width, int height), void repaint(Rectangle r)
- void revalidate()
- void paintComponent(Graphics g)
- void setPreferredSize(Dimension preferredSize), Dimension getPreferredSize()
- void setMaximumSize(Dimension maximumSize), Dimension getMaximumSize()
- void setMinimumSize(Dimension minimumSize), Dimension getMinimumSize()
- boolean isMinimumSizeSet(), boolean isPreferredSizeSet(), boolean isMaximumSizeSet()

The JComponent Class (cont.)

- int getWidth(), int getHeight()
- Dimension getSize(), Dimension getSize(Dimension rv)
- int getX(), int getY(), Point getLocation(),
  Point getLocation(Point rv), Point getLocationOnScreen()
- Rectangle getBounds(), Rectangle getBounds(Rectangle rv)
- Insets getInsets(), Insets getInsets(Insets insets)
- void setLocation(int x, int y), void setLocation(Point p)
- void setSize(int width, int height), void setSize(Dimension d)
- void setBounds(int x, int y, int width, int height),
  void setBounds(Rectangle r)
- void setBorder(Border border), Border getBorder()

The JComponent Class (cont.)

Dealing with component hierarchy:

- Component add(Component comp), Component add(Component comp, int index), void add(Component comp, Object constraints), void add(Component comp, Object constraints, int index)
- void remove(int index), void remove(Component comp), void removeAll()
- Component getComponent(int n), Component[] getComponents(), int getComponentCount()
- Container getParent(), Container getTopLevelAncestor()

Events and Event Listeners

Events can be divided into two groups:

- **low-level** events – low-level input, window-system occurrences, e.g., KeyEvent, MouseEvent, MouseWheelEvent, PaintEvent, WindowEvent
- **semantic** events – everything else,
  e.g., ActionEvent, FocusEvent, ItemEvent, TextEvent

Overview of the most important listeners:

- KeyListener, MouseListener, MouseMotionListener, MouseWheelListener
- WindowListener, WindowFocusListener, WindowStateListener
- ActionListener, ChangeListener, ItemListener
- ComponentListener
- MenuListener, MenuKeyListener, PopupMenuListener

Adapter Classes

The adapter class implements its corresponding listener class by providing all of the required methods, but which have bodies that do nothing.

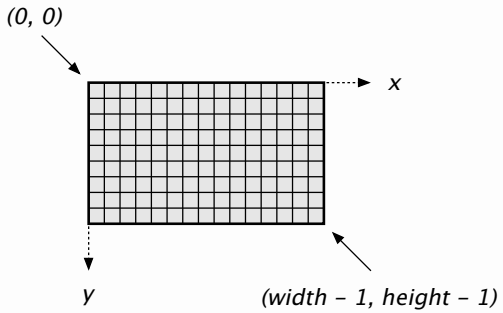
```
public abstract class MouseMotionAdapter
      implements MouseMotionListener {

   public void mouseDragged(MouseEvent e) {
   }

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

The adapter classes serve as base classes for event handlers.

```
addMouseMotionListener(new MouseMotionAdapter() {
   public void mouseMoved(MouseEvent e) {
      ...
   }
});
```

Graphics Programming

Each component has its own integer coordinate system, ranging from *(0,0)* to *(width-1, height-1)*, with each unit representing the size of one pixel. The upper left corner of a component's painting area is *(0,0)*. The *x* coordinate increases to the right, and the *y* coordinate increases downward.



Note: A pixel is a "picture element". It is a dot on a computer screen.

The Graphics Object

The `Graphics` object provides both a context for painting and methods for performing the painting, for example `drawLine()`, `drawRect()`, `fillRect()`, `drawPolygon()`, `drawString()`, etc. It encapsulates information needed for the basic rendering operations. The information includes the following properties.

- a component on which to draw,
- a translation origin for rendering and clipping coordinates,
- current clip,
- current color,
- current font,
- ...

Note: The `Graphics` objects take up operating system resources (more than just memory) and a window system may have a limited number of them.

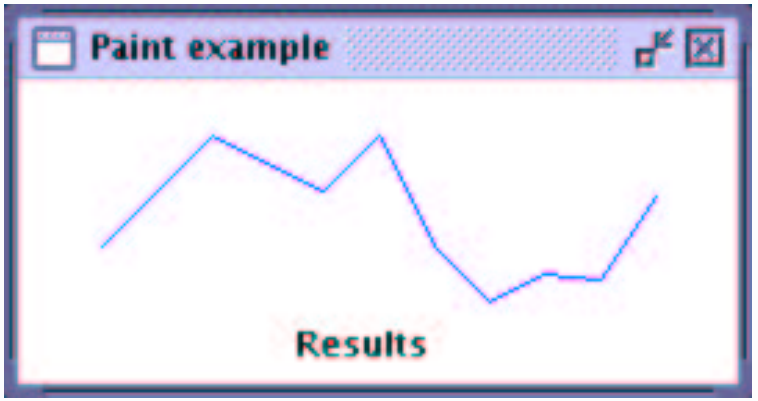
Painting

Painting mechanism schedules painting of visible components. It takes care of details such as damage detection, clip calculation and *z*-ordering. There are two kinds of painting operations.

- **System-triggered**
  The system requests a component to render its contents, usually for one of the following reasons:
  - the component is first made visible on the screen,
  - the component is resized,
  - the component has damaged that needs to be repaired.
- **Application-triggered**
  A component decides it needs to update its contents because its internal state has changed. An application invokes `repaint()` method on a component, which registers an **asynchronous** request that this component needs to be repainted. The component is then repainted by invocation of `paintComponent()` method.

  Note: If multiple calls to `repaint()` occur on a component before the initial repaint request is processed, the multiple requests may be collapsed into a single call to `paintComponent()`.

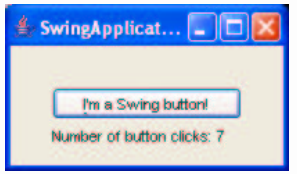
Painting Example



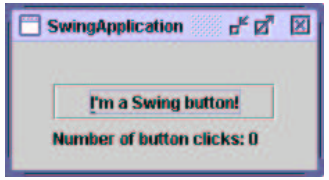
Painting Example (cont.)

```
public class Graph extends JComponent {
   private int[] values = {
      60, 40, 20, 30, 40, 20, 60, 80, 70, 72, 42
   };

   public Dimension getPreferredSize() {
      return new Dimension(260, 110);
   }

   public void paintComponent(Graphics g) {
      g.setColor(Color.BLUE);
      int step = 20;
      int x = 10;
      for (int i = 0; i < values.length-1; i++) {
         x += step;
         g.drawLine(x, values[i], x+step, values[i+1]);
      }
      g.setFont(new Font("sans", Font.BOLD, 12));
      g.setColor(Color.BLACK);
      g.drawString("Results", 100, 100);
   }
}
```

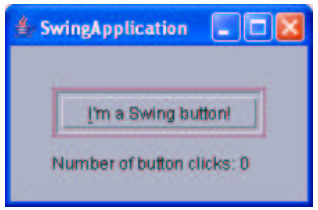
Look and Feel



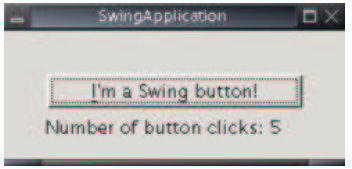
Windows look and feel



Java look and feel



Motif look and feel



GTK+ look and feel

Look And Feel (cont.)

Setting the look and feel:

```
UIManager.setLookAndFeel(
        UIManager.getCrossPlatformLookAndFeelClassName());
```

Possible values of look and feel argument:

- `UIManager.getCrossPlatformLookAndFeelClassName()`
- `UIManager.getSystemLookAndFeelClassName()`
- `"javax.swing.plaf.metal.MetalLookAndFeel"`
- `"com.sun.java.swing.plaf.windows.WindowsLookAndFeel"`
- `"com.sun.java.swing.plaf.motif.MotifLookAndFeel"`
- `"com.sun.java.swing.plaf.gtk.GTKLookAndFeel"`

Look And Feel (cont.)

It is also possible to specify the look and feel at the command line:

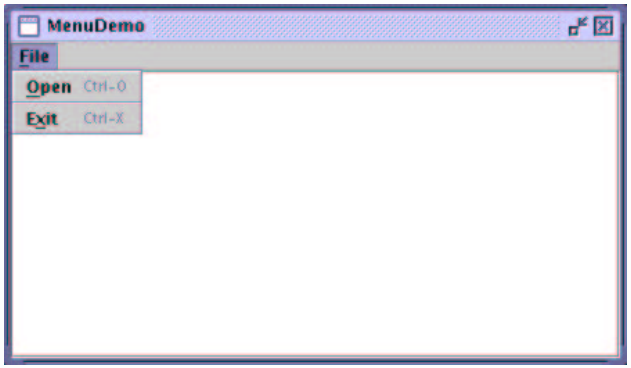
```
$ java -Dswing.defaultlaf=... MyApp
```

Another possibility is to modify the configuration file `swing.properties` in the `jre/lib` directory:

```
# Swing properties

swing.defaultlaf=...
```

Using Menus

The following example illustrates the use of menus:



Using Menus (cont.)

```
private JMenuBar createMenuBar() {
    JMenuBar menuBar = new JMenuBar();

    JMenu menu = new JMenu("File");
    menu.setMnemonic(KeyEvent.VK_F);
    menuBar.add(menu);

    JMenuItem miOpen = new JMenuItem("Open", KeyEvent.VK_O);
    miOpen.setAccelerator(KeyStroke.getKeyStroke(
            KeyEvent.VK_O, ActionEvent.CTRL_MASK));
    miOpen.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            output.append("Action: Open\n");
        }
    });
    menu.add(miOpen);

    menu.addSeparator();
```

Using Menus (cont.)

```
    JMenuItem miExit = new JMenuItem("Exit", KeyEvent.VK_X);
    miExit.setAccelerator(KeyStroke.getKeyStroke(
            KeyEvent.VK_X, ActionEvent.CTRL_MASK));
    miExit.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.exit(0);
        }
    });
    menu.add(miExit);

    return menuBar;
}
```

Using Menus (cont.)

```
private Container createContentPane() {
    JPanel contentPane = new JPanel(new BorderLayout());
    contentPane.setOpaque(true);
    output = new JTextArea(5, 30);
    output.setEditable(false);
    JScrollPane scrollPane = new JScrollPane(output);
    contentPane.add(scrollPane, BorderLayout.CENTER);
    return contentPane;
}
```

```
JFrame frame = new JFrame("MenuDemo");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

MenuDemo demo = new MenuDemo();
frame.setJMenuBar(demo.createMenuBar());
frame.setContentPane(demo.createContentPane());

frame.setSize(450, 260);
frame.setVisible(true);
```

Handling Key Events

There are three methods declared in the **KeyListener** interface:

- void keyTyped(KeyEvent e)
- void keyPressed(KeyEvent e)
- void keyReleased(KeyEvent e)

A **KeyEvent** object contains the following information:

- the key code (constants such as VK_A, VK_LEFT, VK_PAGE_DOWN, VK_F5, ...)
- the character associated with the key
- the modifiers – Shift, CTRL, Meta (Alt)
- key location – left, right, standard, numpad
- if it is an action key

Icons

An **icon** is a fixed-sized picture. An icon is an object that implements the **Icon** interface.

The **ImageIcon** class is an implementation of the **Icon** interface that paints icon from a GIF, JPEG, or PNG image.

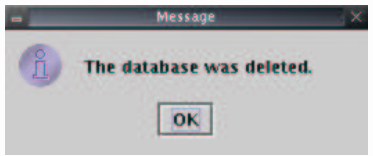
```
protected static Icon createImageIcon(String path) {
    java.net.URL imgURL = IconDemo.class.getResource(path);
    if (imgURL != null) {
        return new ImageIcon(imgURL);
    } else {
        System.err.println("Couldn't find file: " + path);
        return null;
    }
}
```

```
Icon icon = createImageIcon("images/my_image.png");
JLabel label = new JLabel(icon);
```

Simple Dialogs

To create and show simple dialogs, we can use the **JOptionPane** class:

```
JOptionPane.showMessageDialog(frame,
                              "The database was deleted.");
```



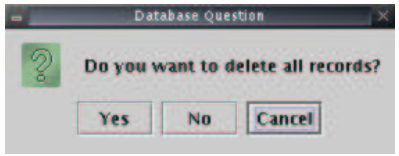
Simple Dialogs (cont.)

```
JOptionPane.showMessageDialog(frame, "The database is empty.",
      "Database warning", JOptionPane.WARNING_MESSAGE);
```


```
JOptionPane.showMessageDialog(frame, "The database is empty.",
      "Database error", JOptionPane.ERROR_MESSAGE);
```

Simple Dialogs (cont.)

```
Object[] options = { "Yes", "No", "Cancel" };
int n = JOptionPane.showOptionDialog(frame,
        "Do you want to delete all records?",
        "Database Question",
        JOptionPane.YES_NO_CANCEL_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null, options, options[2]);
```



Timers

A Swing timer (instance of the **javax.swing.Timer**) fires one or more action events after a specified delay.

Swing timer can be used in two ways:

- to perform a task once, after a delay
- to perform a task repeatedly

```
Timer timer = new Timer(1000, new ActionListener() {
    public void actionPerformed(ActionEvent e) {
            . . .
    }
});

timer.start();
```