

Chapter 5: JFC Applications

Outline

Goals and Objectives.....	3
Section 1: Java Foundation Classes	4
Introduction	5
Basic JFrame Window	6
setDefaultCloseOperation Method.....	7
Creating a GUI Interface.....	8
The JPanel	9
Events	11
ActionListener Interfaces	12
getSource Method	13
Event Interface	14
Event Objects	15
Java Resources	16
Section 2: AWT and Swing	17
AWT.....	18
Swing.....	19
JComponents	20
Containers and Components	21
Cursors	22
JPanel	23
Section 3: Windows, JFrames and Dialogs	24
Windows, Frames and Dialogs	25
Window	26
JFrame	27
JFrame Dynamics.....	28
JRootPane.....	29
ToolTip Text	30
Icon.....	31
Creating Images Using the ImageIcon Class	32
JDialog	33
Dialog Example.....	34
JDialog Based Class Demo	35
JFileChooser.....	39
JFileChooser Example	40
Summary	42

Goals and Objectives

After completing this unit the student will:

- Learn how to create a JFC application
- Add components to containers
- Create a basic GUI interface
- Manipulate common controls
- Handle application events

Section 1: Java Foundation Classes

Introduction

The Java Foundation Classes (JFC) consists of five Application Programming Interfaces (APIs)

- Abstract Windowing Toolkit (AWT)
- Java™ 2D
- Accessibility
- Drag and Drop
- Swing

We will focus on the AWT and Swing. AWT (Abstract Windowing Toolkit) was part of the original Java packages. It is still used and is the basis for the new Swing packages. Swing was introduced with JDK 1.2 in order to provide a richer and more complete set of GUI (Graphical User Interface) components (controls).

The emphasis here will be on the development of window applications using Swing. The basis for windows in Java is the JFrame class. An instance of JFrame is created and then displayed on the screen. While it is possible to draw in this window using the Graphics object and other 2D graphic techniques, the emphasis here is on the use of components.

Components are added to a container. JFrame is a container but it is more common to add components to an intermediate container, such as a JPanel, and then to add the JPanel to the JFrame. The use of the intermediate JPanel container provides more flexibility in the development of a GUI interface. Each of the containers uses a layout manager to control how components are positioned within the container. This approach makes it easier to control the appearance of an application, especially when the application window is resized.

Basic JFrame Window

There are several way of creating a window. A common approach is to extend the JFrame class. This derived class will have a main method and a single argument constructor. The purpose of the main method is to instantiate the derived class and to display it. The constructor is used to create the GUI interface. Three packages are frequently used in a JFrame based application.

- java.awt – Contains the basic GUI classes
- java.awt.event – Contains the classes used to handle events such as a mouse move
- javax.swing – Contains the new swing components

The java.awt.event is not used in this example but will be used later.

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

public class Demo extends JFrame {

    public static void main(String args[]) {
        Demo app = new Demo("New Title");
        app.setSize(300,100);
        app.setVisible(true);
    }

    public Demo(String frameTitle) {
        super(frameTitle);
        // Create GUI interface
    }
}
```



The constructor calls the base JFrame class and passes it a string. This string is used by the base class constructor as the caption of the window. In the main method, the **setSize** method determines the size of the window when it is first displayed. The **setVisible** method will make the window visible.

Other methods can be added to provide functionality to the application. The application is compiled as usual and executed using the javac and java command respectively. You will notice that the application does not terminate when the window is closed. One approach that you can use to stop the application from the DOS window is to use the Ctrl-C command. However, this is not a very good technique.

setDefaultCloseOperation Method

A more graceful way of terminating an application when the window is closed uses the **setDefaultCloseOperation** method. This method has a single argument that specifies the action to take when the window closes. Adding the following code before the **setSize** method will terminate the application when the window closes.

```
app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Other arguments can be used that will not terminate the application. In addition, there are other approaches to handle the closing of a window. These will be covered in more detail later.

Creating a GUI Interface

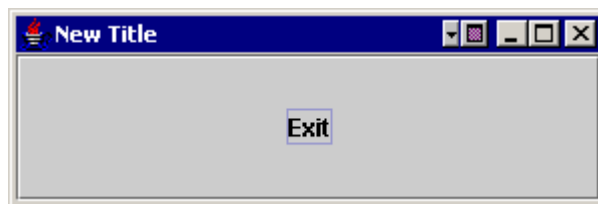
Components provide the GUI interface to an application. There are many different types of components available and learning Swing is largely about learning what components are available and how to use them. Initially, we will look at three of these components:

- **JLabel** – Provides a text label that the user cannot change directly. As the name implies it is used to label other components.
- **JTextField** – Provides a way for the user to enter information that the program can then use.
- **JButton** – This is a button that the user will press. It normally has a caption.

The **add** method is used to add a component to a container. However, it can't be added directly to a container. Swing containers have multiple panes like a multiple pane glass windows has more than one pane. Components must be added to what is known as the content pane. To get a reference to the content pane, use the **getContentPane** method. This method returns a reference to the content pane.

To create a JButton, the JButton constructor is used. One version of this overloaded constructor has a string argument. This argument is the text that will be displayed on the button. A JButton is added to the Demo class:

```
public Demo(String frameTitle) {
    super(frameTitle);
    // Create GUI interface
    Container c = getContentPane();
    JButton btnExit = new JButton("Exit");
    c.add(btnExit);
}
```



You may notice that as this window is resized, the button is automatically repositioned within the window so that it is always centered. This behavior is result of the default layout manager for a JFrame, the FlowLayout manager. This Layout manager will center any components inside of its container.

The JPanel

As mentioned earlier, a JPanel object is used as the container for most interfaces. As will become clear later on, this approach is a more flexible approach. The following illustrates the adding of a button to a JPanel and then adding the JPanel to the content pane.

```
public Demo(String frameTitle) {
    super(frameTitle);
    // Create GUI interface
    Container c = getContentPane();

    JPanel demoPanel = new JPanel();
    JButton btnExit = new JButton("Exit");
    demoPanel .add(btnExit);

    c.add(demoPanel);
}
```

It also has a default layout manager of FlowLayout. The behavior of this layout manager is better illustrated using multiple components. The program has been modified to include a JLabel and a JTextField. The constructors for these components are overloaded. In the example, the JLabel's argument contains the caption of the label and the JTextField's argument specifies the size of the text box.

```
public Demo(String frameTitle) {
    super(frameTitle);
    // Create GUI interface
    Container c = getContentPane();
    JPanel demoPanel = new JPanel();

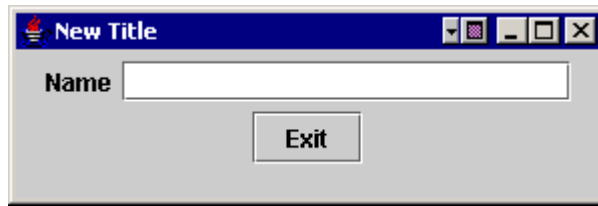
    JLabel lblName = new JLabel("Name");
    demoPanel .add(lblName);

    JTextField txtName = new JTextField(20);
    demoPanel .add(txtName);

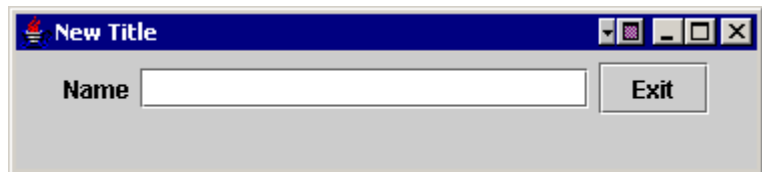
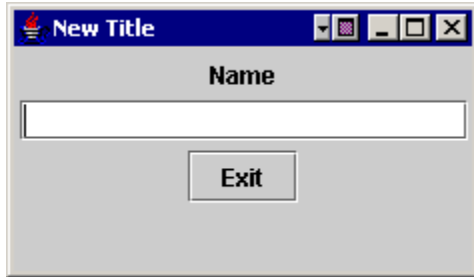
    JButton btnExit = new JButton("Exit");
    demoPanel .add(btnExit);

    c.add(demoPanel);
}
```

The initial appearance of the window is:



However, by resizing the window the positions of the components will change.



This behavior may not be desirable for all applications. Later we learn how to control the position of the components within a container.

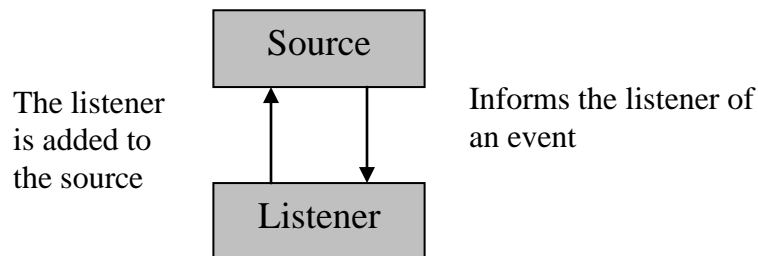
Note that the order in which the components are added to the container is important. The `FlowLayout` manager will add components to the container left to right and then top to bottom when it runs out of space on a row. When the container is resized, the components are rearranged in the same way.

Events

When a button or similar component is used as part of our interface, the question arises as to how the programmer knows when the user has pressed the button. Programs are written to respond to events. Events include actions such as:

- Clicking or movement of the mouse
- Keyboard entry
- Window closing or resizing

Java 1.1 added a source/listener paradigm to model events. The source will inform listeners when specific events occur.



In the previous application, the button is the source of events. The user may press the button, move the mouse over the button, or perform similar actions. The listener is the application itself. The application is interested when a specific event occurs.

In Java, the listener must tell the source of the event that it is:

- 1) Interested in listening to the source, and
- 2) Which events it is interested in listening to

Once the source is aware that a listener is interested in its events, it will add the listener to its list of listeners. When the event of interest occurs, the source will examine its list and then inform each listener that the event has occurred. While it may seem that there is a need for only one listener per program, later we will see that multiple listeners are useful at times.

ActionListener Interfaces

Interfaces are an important part of the event model. The `addSomeListener` informs the source that an object is interested in certain events. The naming convention is quite consistent among the different interfaces. Each event interface supports a specific number and types of events.

Each interface is of the form *SomeListener*. Each method of an interface passes an object derived from the `Event` class and is of the form *SomeEvent*. This object can be used to obtain more information regarding the event and its specific capability is determined by the nature of the event. For example, a `MouseEvent` object has the X and Y position of the mouse.

One of the advantages to this approach is that a listener can specify only those types of events that it is interested in listening to and will not be troubled by other non-relevant events. It can listen to mouse events but ignore key board events.

Of immediate interest to us is the `ActionListener` interface. This interface has a single method, **`actionPerformed`**. It is passed a single argument, an `ActionEvent` object. When the user presses a button, the button will call the **`actionPerformed`** method of any of its listeners. So when the listener's **`actionPerformed`** method is executed, it knows the user has pressed the button.

First, the listener must let the source know that it is interested in listening to the source's events. To do this it uses the source's **`addActionListener`** method. The argument passed with this method is a reference to the listener. Frequently, the `this` keyword is used as the argument as it is a reference to the current object, the listener.

```
JButton btnExit = new JButton("Exit");
demoPanel.add(btnExit);
btnExit.addActionListener(this);
```

However, for the `JButton` to call the **`actionPerformed`** method, the listener class must implement the `ActionListener` interface.

```
public class Demo extends JFrame implements ActionListener{
```

The actual implementation of the actual `actionPerformed` method is at the discretion of the programmer.

```
public void actionPerformed(ActionEvent e) {
    System.exit(0);
}
```

getSource Method

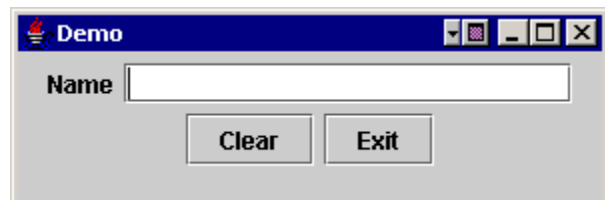
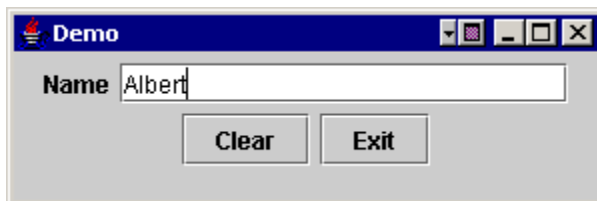
If multiple buttons, or other components that have events of interest, are used then there needs to be a way to distinguish between the buttons or components. The **getSource** method returns a reference to the source object. This can then be used to determine the specific object that caused the event to occur.

In the next example, the declaration of the JButton has been moved and declared as an instance variable. This changes the scope of the button and allows it to be accessed from both the constructor and the **actionPerformed** methods. The button is created and initialized in the constructor as before. In the **actionPerformed** method, its caption is changed using the **setText** method.

```
private JButton btnExit;    // Instance variable
...
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == btnExit) {
        System.exit(0);
    }
}
```

If we had two JButtons, btnExit and btnClear, we could distinguish between them similar to what is shown below:

```
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == btnExit) {
        System.exit(0);
    } else {
        txtName.setText("");
    }
}
```



Event Interface

Event Interface	Class Listener	Listener Methods
ActionEvent	ActionListener	actionPerformed()
AdjustmentEvent	AdjustmentListener	adjustmentValueChanged()
ComponentEvent	ComponentListener	componentHidden() componentMoved() componentResized() componentShown()
ContainerEvent	ContainerListener	componentAdded() componentRemoved()
FocusEvent	FocusListener	focusGained() focusLost()
ItemEvent	ItemListener	itemStateChanged()
KeyEvent	KeyListener	keyPressed() keyReleased() keyTyped()
MouseEvent	MouseListener	mouseClicked() mouseEntered() mouseExited() mousePressed() mouseReleased() mouseDragged() mouseMoved()
	MouseMotionListener	
TextEvent	TextListener	textValueChanged()
WindowEvent	WindowListener	windowActivated() windowClosed() windowClosing() windowDeactivated() windowDeiconified() windowIconified() windowOpened()

Event Objects

Each event results in a method of a listener interface being called. Each method is invoked with an event object. The event object contains information regarding the event. Each event has a specific meaning.

Component	Events	Generated Meaning
Button	ActionEvent	User clicked on the button
Checkbox	ItemEvent	User selected or deselected an item
CheckboxMenuItem	ItemEvent	User selected or deselected an item
Choice	ItemEvent	User selected or deselected an item
Component	ComponentEvent	Component moved, resized, hidden, or shown
	FocusEvent	Component gained or lost focus
	KeyEvent	User pressed or released a key
	MouseEvent	User pressed or released mouse button, mouse entered or exited component, or user moved or dragged mouse.
Container	ContainerEvent	Component added to or removed from container
List	ActionEvent	User double-clicked on list item
	ItemEvent	User selected or deselected an item
TextComponent	TextEvent	User changed text
TextField	ActionEvent	User finished editing text
Window	WindowEvent	Window opened, closed, iconified, deiconified, or close requested

Java Resources

Resources include those elements that are used in support of an application. These can include files containing images, sounds and video. They can also include the data files used to support an application.

To facilitate the use of images Java has a `getResource` method that is used to load a resource. Resources are assumed to be stored in a subdirectory off of the directory where the application's class files are stored. The resources can be stored as individual files or grouped together in a JAR file. IN developing real world applications it is desirable to place of the applications classes in a package.

The `getResource` method uses the path to the resource to create an URL object.

```
URL imageURL = getClass().getResource("images\\cross.gif");

if(imageURL != null) {
    ImageIcon imageIcon = new ImageIcon(imageURL, "Alt Text");
} else {
    // Could not load the image
}
```

The `getResource` method uses the directories and JAR files in the program's call path. An URL object is returned if the resource is found otherwise a null is returned.

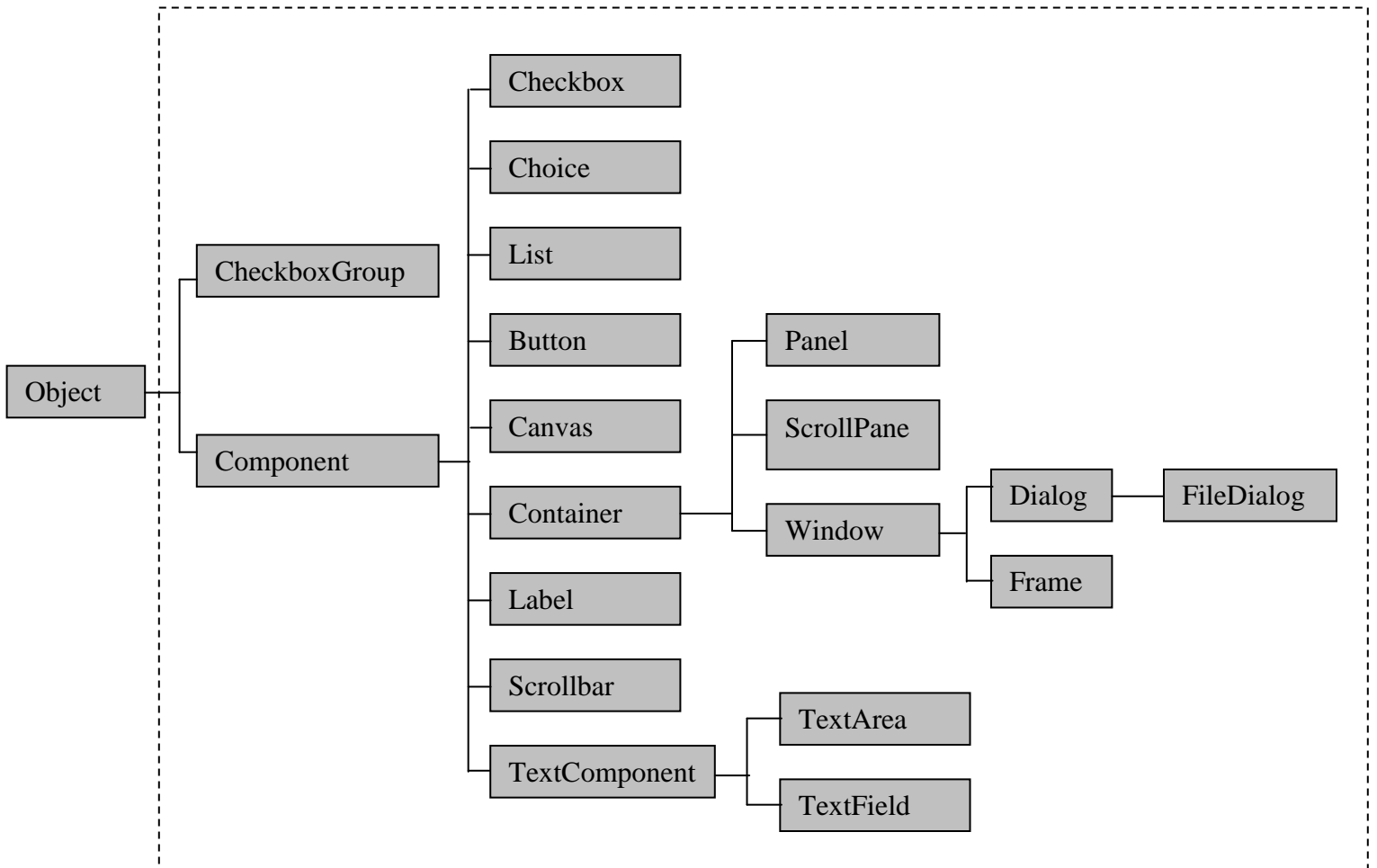
The application's class path can be specified using (see <http://java.sun.com/docs/books/tutorial/uiswing/components/icon.html>):

- `-cp` or `-classpath` command line argument
- The program's JNLP file
- Setting the `CLASSPATH` environmental variable

Section 2: AWT and Swing

AWT

The AWT package contains a number of classes. Container classes are used to hold components. These components generally represent visible graphics elements such as a text box or a label. AWT components are “heavy” components that use some native code.



* Component and Container are abstract classes

Swing

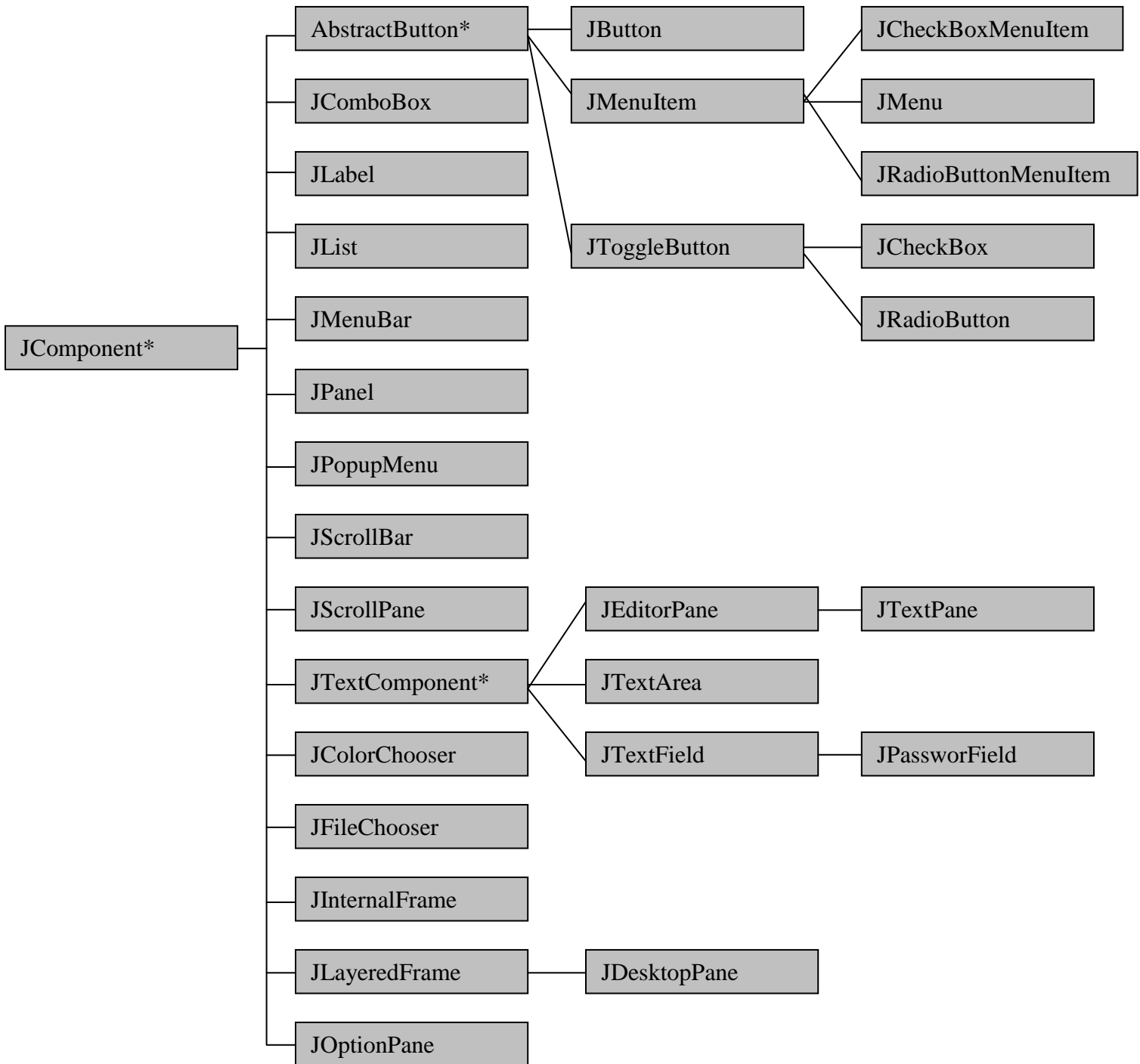
Swing components are derived from AWT. However, AWT components should not be used with Swing components. The Swing components are lightweight components. When heavy and light weight components are mixed, it is difficult to control the ordering of the components.

All swing components use the Model/View/Controller (MVC) architecture. This architecture allows the appearance and behavior of the components to be modified easily.

There are numerous packages that make up swing.

- javax.swing – Consists of components, adapters, models and interfaces
- javax.swing.border – Declares the Border interface for defining specific border styles
- javax.swing.colorchooser – Classes supporting the ColorChooser component
- javax.swing.event – Contains Swing specific events
- javax.swing.filechooser – Classes supporting the file chooser
- javax.swing.plaf - Supports the Pluggable Look and Feel (PLAF)
- javax.swing.table – Supports the Swing table component
- javax.swing.text – Supports the Swing document framework
- javax.swing.text.html - Supports an HTML 3.2 renderer and parser
- javax.swing.text.rtf – Support the rendering of RTF documents
- javax.swing.tree – Supports the Swing tree component
- javax.swing.undo – Helps implement undo/redo functionality
- javax.accessibility – Supports user accessibility features

JComponents



* - Abstract classes

Containers and Components

The Container class is an abstract class used to hold other components. Components are placed inside of a container. A Layout Manager is used in conjunction with a container to specify how components are displayed within the container.

```
public Component add(Component c);  
public void setLayout(LayoutManager mgr);
```

The **add** method will add a component to a container. Each container has a default layout manager that can be changed using the **setLayout** method. There are several layout managers available:

- FlowLayout
- GridLayout
- BorderLayout
- CardLayout
- GridBagLayout
- Box
- ScrollPane
- Absolute

The Component class is an abstract class forming the bases for many AWT classes. While never used directly, it possesses many methods inherited by derived classes

```
public Color getBackground();  
public void setBackground(Color c);  
public Color getForeground();  
public void setForeground(Color c);  
public Rectangle getBounds();  
public Dimension getSize();  
public Cursor getCursor();  
public synchronized void setCursor(Cursor cursor);  
public Font getFont();  
public void setFont(Font f);  
public FontMetric getFontMetrics(Font font);  
public void invalidate();  
public boolean isEnabled();  
public void setEnabled(boolean b);  
public boolean isShowing();  
public boolean isVisible();  
public void setVisible(boolean b);
```

Cursors

Cursors are an important GUI component. It is often desirable to be able to manipulate the cursor. The `Cursor` class's single integer constructor can be used to the cursor used for the container.

```
public Cursor(int type);
```

There are a number of predefined cursor available:

```
public static final int DEFAULT_CURSOR;  
public static final int CROSSHAIR_CURSOR;  
public static final int HAND_CURSOR;  
public static final int MOVE_CURSOR;  
public static final int TEXT_CURSOR;  
public static final int WAIT_CURSOR;  
public static final int N_RESIZE_CURSOR;  
public static final int S_RESIZE_CURSOR;  
public static final int E_RESIZE_CURSOR;
```

The `getPredefinedCursor` method retrieves the cursor and the `setCursor` method sets the actual cursor.

```
public class Demo extends JFrame {  
  
    public static void main(String args[]) {  
        ...  
    }  
  
    public Demo(String frameTitle) {  
        super(frameTitle);  
        ...  
        bottomPanel.setCursor(Cursor.getPredefinedCursor(  
                                Cursor.CROSSHAIR_CURSOR));  
        ...  
    }  
}
```

JPanel

A JPanel is a commonly used container. JPanels can be nested inside of other panels. By controlling the layout manager used by each panel, greater control can be exercised over the appearance of an application.

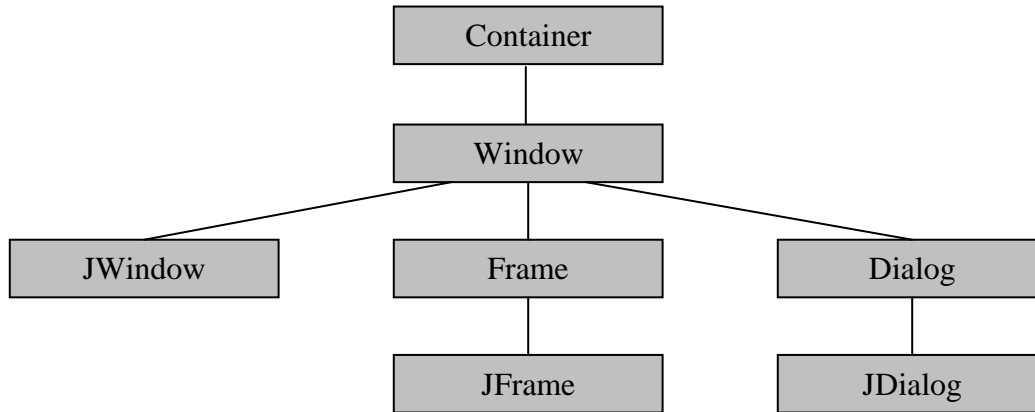
Double buffering is automatically enabled for the JPanel. Its state is controlled through the constructor. Double buffer is a technique where all of the drawing is performed first in memory and then is copied to the screen. Double buffering promotes faster display of graphics.

The default layout manager for JPanel is FlowLayout.

Section 3: Windows, JFrames and Dialogs

Windows, Frames and Dialogs

These classes provide a common means of providing a GUI interface for applications. Window is best used for splash screen and similar interfaces. The Frame class provides the capabilities most usually associated with a window. The Dialog class provides dialog box like capabilities.



Window

The window class possesses a number of methods that are used to manipulate containers.

```
void pack();  
void dispose();  
void getFocusOwner();  
void toFront();  
void toBack();
```

The **pack** method sets the size of the window such that it is at the minimum size possible that will display all of its contained components.

The **dispose** method hides the window and gets rid of any resources used by the window. It does not dispose of the component. Only native resources are gone. To display the window again, simply use the **setVisible** method. It is not necessary to recreate the window.

getFocusOwner returns the contained component that currently has focus.

The **toFront** and **toBack** methods place the window in front of or behind other windows.

In particular, the **dispose** method is used in conjunction with the **exit** method to terminate a windows application. These statements are usually placed in the windowClosing event that will be discussed shortly.

```
public void windowClosing(WindowEvent event) {  
    dispose();  
    System.exit(0);  
}
```

JFrame

JFrame is the foundation of a window in a Swing application.

```
public JFrame();  
public JFrame(String title);           // Caption of window
```

The JFrame class supports:

- Sizable border
- Menus
- Title
- Toolbar

To add children to a JFrame, add them to the content pane returned by the `getContentPane()` method. The default layout manager is BorderLayout.

The `setTitle` method will set the frame's title.

```
public String getTitle();  
public synchronized void setTitle(String title);
```

The `setMenuBar` method will specify the menu bar and the method `setIconImage` will specify the icon used for the window.

```
public Image getIconImage();  
public synchronized void setIconImage(Icon image);
```

The `setResizable` method controls whether the frame can be resized.

```
public boolean isResizable();  
public synchronized void setResizable(boolean b);
```

The `setVisible`, `hide` and `dispose` methods work the same way as with a window.

JFrame Dynamics

A JFrame will remain open by default unless:

- The window closing event is handled explicitly
- The application terminates
- The **setDefaultCloseOperation** is set to close the window

The **setDefaultCloseOperation** method specifies how the frame behaves when the user attempts to close it.

```
public void setDefaultCloseOperation(int operation)
```

There are four options for the **setDefaultCloseOperation** method:

- **DO_NOTHING_ON_CLOSE** – Nothing happens
- **HIDE_ON_CLOSE** – (Default) The window will be hidden
- **DISPOSE_ON_CLOSE** – When the user closes the window the frame will be disposed

```
setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
```

JRootPane

All Swing containers possess a JRootPane. This pane is not normally used directly by the programmer but has other panes that are used.

The JRootPane is a container that consists of two objects:

- A glass pane
- A layered pane

The layered pane consists of two objects:

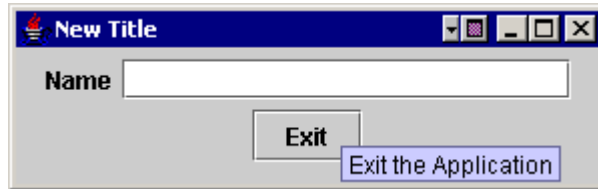
- An optional menu bar
- A content pane

The glass pane always appears in front of the layered pane and allows popup menus and tool tip text to work properly. This pane is not normally manipulated directly

ToolTip Text

A tool tip is a context sensitive string that indicates the function of an object. It appears when the mouse momentarily rests over an object. Tool tip text is set using the **setToolTipText** method of JComponent.

```
 JButton buttonExit = new JButton("Exit");  
 buttonExit.setToolTipText("Exit the Application");
```



Icon

Icons are used with many Swing components to provide a visual image. Objects that act as icons will implement the Icon interface. The ImageIcon class is an implementation of Icon and creates an Icon from the Image class. Its constructor uses one of several values:

- String – String to be displayed
- Filename – Path to the location of the image
- URL – URL path to the location of the image

In this example, the file carrot.gif is located in the same directory as the java file.

```
Icon iconCarrot = new ImageIcon("carrot.gif");  
btnExit.setIcon(iconCarrot);
```

Creating Images Using the ImageIcon Class

Images can be created using the ImageIcon constructor with either a URL or with a string containing the path to the image file. The preferred technique is to use the URL as this provides a better means of determining whether the image is available or not and to handle errors when it is not available. In addition, it makes it easier to group the application's resources together.

When the ImageIcon constructor is executed using either a URL or a file name, the current process is blocked until the data is loaded or until the data location proves to be invalid. If the data is invalid, but not null, an ImageIcon is still created but it has no size and will paint nothing. The **setImageObserver** method when used when additional information regarding the loading of the image is needed.

URL Technique

The getResource method is used to return a URL object representing the image. (See the resource section of more details on getResource). If the image is not available the method will return null. The return value should be tested.

If the image is available the two argument ImageIcon constructor is used. The first argument is the URL. The second argument provides assistive technologies that can aid visually impaired users.

```
String imageLocation = "images\\cross.gif";
URL imageURL = getClass().getResource(imageLocation);
if(imageURL != null) {
    ImageIcon imageIcon = new ImageIcon(imageURL, "Alternative Text");
} else {
    // Could not find the file
}
```

File Path Technique

The second technique uses a two argument ImageIcon constructor where the first argument is a string containing the path to the image file.

```
ImageIcon imageIcon = new ImageIcon("cross.gif", " Alternative Text");
```

JDIALOG

The JDIALOG class provides a means to display dialog boxes. Dialog boxes can be modal or non-modal.

From the user's perspective, a modal dialog box is a dialog that he must complete before control is returned back to the main window. The file save dialog box is an example of a modal dialog box. A non-modal dialog box is one where the user can move between the dialog box and the main window without closing the dialog box. The find and replace dialog found in many applications is an example of a non-modal dialog box.

The JDIALOG class has several overloaded constructors

```
public JDIALOG(JFrame parent);  
public JDIALOG (JFrame parent, boolean modal);  
public JDIALOG (JFrame parent, String title);  
public JDIALOG (JFrame parent, String title, boolean modal);
```

The parent is the window that the dialog originates from. The string argument specifies the caption of the dialog box.

There are user and developer aspects to modal dialog boxes. The user must close the dialog before continuing. As a result, the program will wait until control is returned from the dialog before another other code is executed. With a non-modal dialog box, the dialog box is displayed and control is immediately returned to the calling method.

The JDIALOG class possesses several useful methods:

```
public String getTitle();  
public boolean isModal();  
public boolean isResizable();  
public void setModal(boolean b);  
public synchronized void setResizable(boolean b);  
public synchronized void setTitle(String title);  
public void setVisible();
```

Dialog Example

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

public class DialogDemo extends JFrame implements ActionListener {

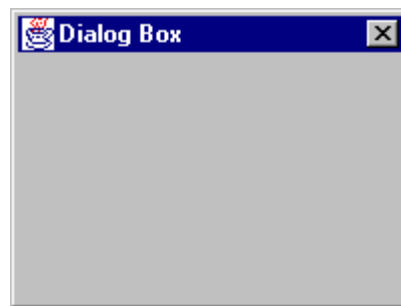
    static JPanel bottomPanel = new JPanel();
    static JButton btnDialog = new JButton("Show Dialog");

    public static void main(String args[]) {

        DialogDemo app = new DialogDemo("Dialog Example");
        app.setSize(300,100);
        app.setVisible(true);
    }

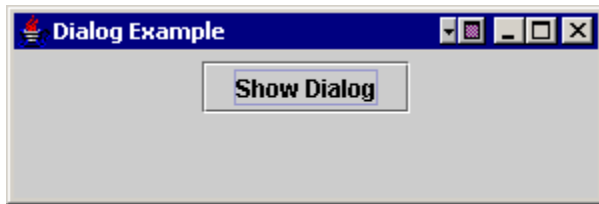
    public DialogDemo(String frameTitle) {
        super(frameTitle);
        btnDialog.addActionListener(this);
        bottomPanel.add(btnDialog);
        Container c = getContentPane();
        c.add(bottomPanel);
    }

    public void actionPerformed(ActionEvent e) {
        JDialog dlg = new JDialog(this, "Dialog Box");
        dlg.setModal(true);
        dlg.setSize(200,150);
        dlg.setVisible(true);
    }
}
```

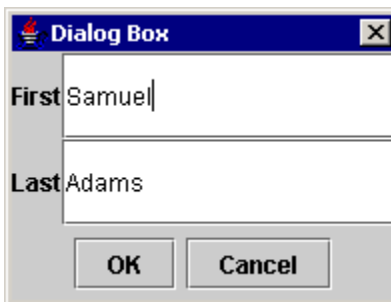


JDialog Based Class Demo

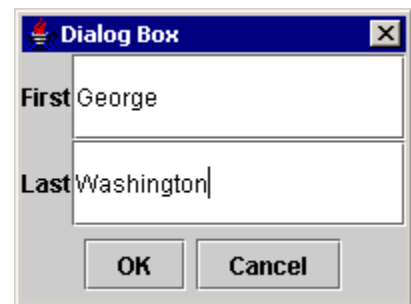
A more sophisticated version of a dialog box is shown here. The dialog box is derived from JDialog and has two JTextFields and two JButtons. The demonstration illustrates how to move information from the dialog box back to the application. The user will input a first name and a last name into the dialog box. If the user selects the OK button, these names are returned to the application. If the user presses the cancel button, no information is returned.



When the Show Dialog button is first pressed, the EmployeeDialog box appears:



The user can change the name at this time.



If the user presses the OK button, the new name is returned back to main application using two mutator methods.

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

public class DialogDemo extends JFrame implements ActionListener {

    private JPanel bottomPanel = new JPanel();
    private JButton btnDialog = new JButton("Show Dialog");
    private String firstName;
    private String lastName;

    public static void main(String args[]) {

        DialogDemo app = new DialogDemo("Dialog Example");
        app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        app.setSize(300,100);
        app.show();
    }

    public DialogDemo(String frameTitle) {
        super(frameTitle);
        btnDialog.addActionListener(this);
        bottomPanel.add(btnDialog);
        Container c = getContentPane();
        c.add(bottomPanel);
    }

    public void actionPerformed(ActionEvent e) {
        EmployeeDialog dlg =
            new EmployeeDialog(this, "Dialog Box", "Samuel", "Adams");
        dlg.setModal(true);
        dlg.setSize(200,150);
        dlg.setVisible(true);
    }

    public void setFirstName(String newFirstName) {
        firstName = newFirstName;
        System.out.println(firstName);
    }

    public void setLastName(String newLastName) {
        lastName = newLastName;
        System.out.println(lastName);
    }
}
```

```
class EmployeeDialog extends JDialog implements ActionListener {
    JButton btnOK;
    JButton btnCancel;
    JLabel lblFirstName;
    JLabel lblLastName;
    JTextField txtFirstName;
    JTextField txtLastName;

    DialogDemo owner;

    public EmployeeDialog (    DialogDemo owner,
                            String title,
                            String firstName,
                            String lastName) {

        super(owner, title);

        this.owner = owner;
        btnOK = new JButton("OK");
        btnOK.addActionListener(this);

        btnCancel = new JButton("Cancel");
        btnCancel.addActionListener(this);

        lblFirstName = new JLabel("First");
        lblLastName = new JLabel("Last");
        txtFirstName = new JTextField(20);
        txtLastName = new JTextField(20);

        txtFirstName.setText(firstName);
        txtLastName.setText(lastName);

        JPanel center = new JPanel();
        center.setLayout(new BorderLayout());

        JPanel south = new JPanel();
        south.setLayout(new FlowLayout());
        south.add(btnOK);
        south.add(btnCancel);

        JPanel labels = new JPanel();
        labels.setLayout(new GridLayout(2,1));
        labels.add(lblFirstName);
        labels.add(lblLastName);

        JPanel textboxes = new JPanel();
        textboxes.setLayout(new GridLayout(2,1));
        textboxes.add(txtFirstName);
```

```
        textboxes.add(txtLastName);

        Container c = getContentPane();
        c.setLayout(new BorderLayout());

        c.add(center, "Center");
        c.add(south,"South");

        center.add(labels,"West");
        center.add(textboxes,"Center");
    }

    public void actionPerformed(ActionEvent e) {
        if(e.getSource() == btnOK) {
            owner.setFirstName(txtFirstName.getText());
            owner.setLastName(txtLastName.getText());
            dispose();
        } else {
            dispose();
        }
    }
}
```

JFileChooser

The JFileChooser class provides a system dependent technique for displaying open/save file dialog boxes.

```
public JFileChooser ();
```

To display the dialog box use either:

- **showOpenDialog** – Displays the open dialog box
- **showSaveDialog** – Displays the open dialog box

Both of these methods use the dialog parent reference as its argument. There are several useful methods associated with this class:

- The **setCurrentDirectory** method is used to set the directory to be used when the dialog box appears
- The **setSelectedFile** method is used to set the file to be selected when the dialog box appears
- The **getSelectedFile** method returns the File object that was selected
- The File class' **getName** method returns the name of the file that was selected

JFileChooser Example

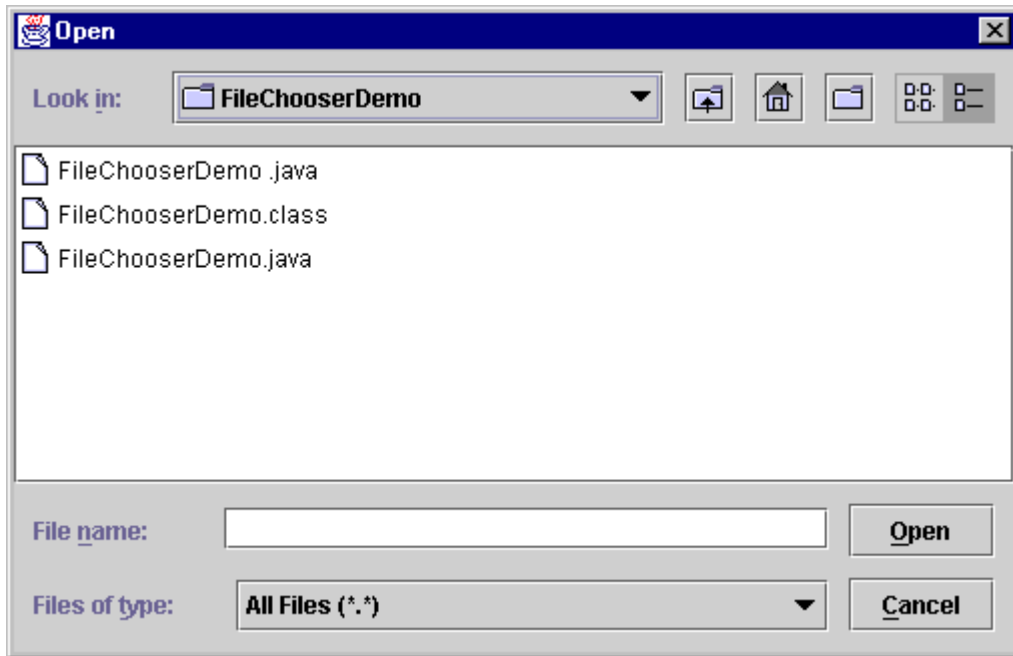
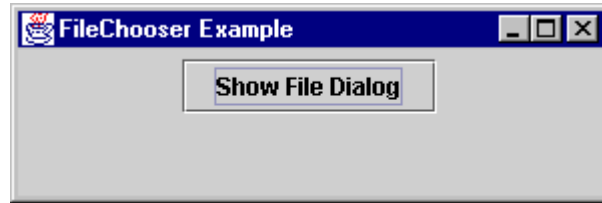
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class FileChooserDemo extends JFrame implements ActionListener{
    static JPanel bottomPanel = new JPanel();

    public static void main(String args[]) {
        FileChooserDemo app = new FileChooserDemo("FileChooser Example");
        app.setSize(300,100);
        app.setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        JFileChooser dialog = new JFileChooser("Test");
        dialog.showOpenDialog(this);
        if (dialog.getSelectedFile() != null)
            ;// Use file name
        else
            ;// No filename returned
    }

    public FileChooserDemo(String frameTitle) {
        super(frameTitle);
        JButton btnExit = new JButton("Show File Dialog");
        btnExit.addActionListener(this);
        bottomPanel.add(btnExit);
        Container c = getContentPane();
        c.add(bottomPanel);
    }
}
```



Summary

- The JFC provides a large set of objects that are used to provide a GUI interface
- Components are the building blocks of this interface
- Containers are objects that hold components
- There are numerous components available in Java
- Events are intercepted using various listener interfaces
- Several specialized windows are supported including dialog boxes and the file chooser