Java GUI programming

Mārtiņš Leitass

Agenda

- JFC and Swing overview
- GUI composition:
  - Create a Container Using Swing
  - Create Swing Components
  - Apply Layout Managers
- GUI event handling:
  - Key events
  - Mouse Events
- View-Model decomposition:
  - JTable

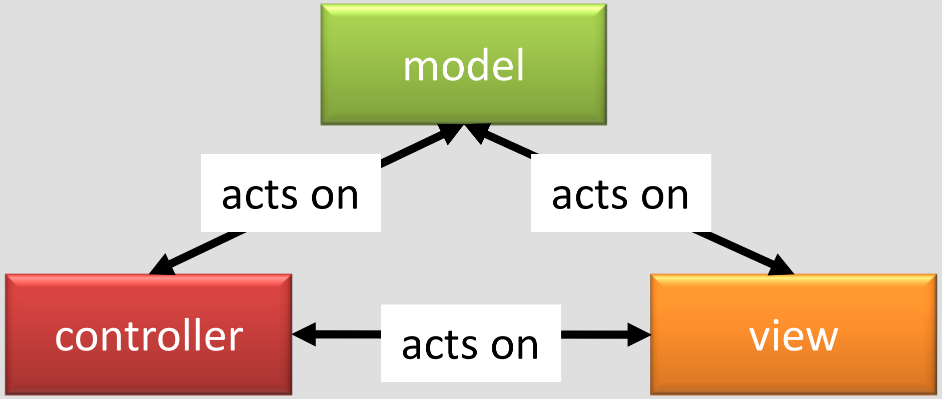
What is Java Swing?

- Part of the Java Foundation Classes (JFC)
- Provides a rich set of GUI components
- Used to create a Java program with a graphical user interface (GUI)
- table controls, list controls, tree controls, buttons, and labels, and so on…

What features are available?

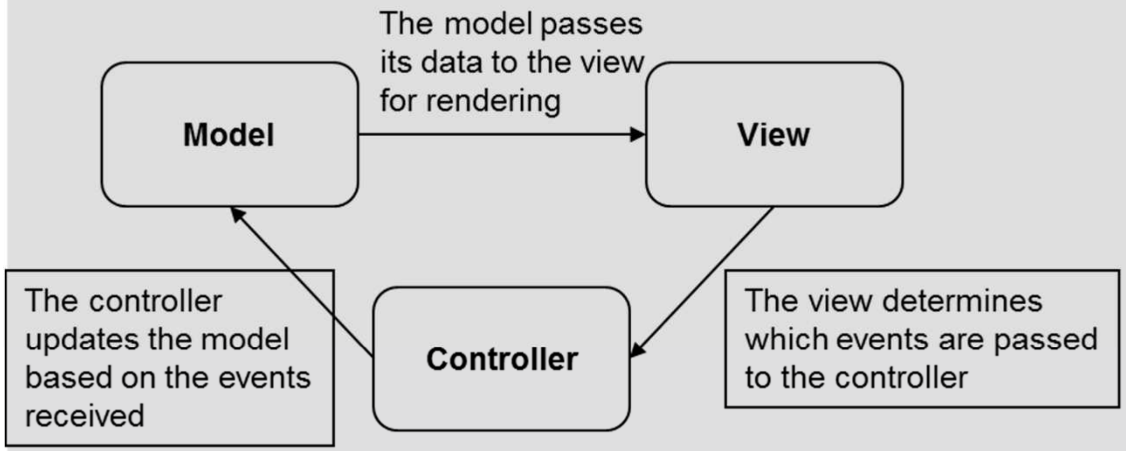
- GUI components like button, checkbox, and so on…
- Java 2D API: images, figures, animation
- Pluggable look and feel: use samples or create your own
- Data Transfer: cut, copy, paste, drag & drop
- Internationalization: supports different input language, right to left reading
- Accessibility API: for people with disabilities
- Undo Framework API: supports unlimited numbers of actions to undo and redo
- Flexible Deployment: run within a browser as an applet or Java Web Start

MVC: Model-View-Controller



- The **model** that stores the data that defines the component
- The **view** that creates the visual representation of the component from the data in the model
- The **controller** that deals with user interaction with the component and modifies the model and/or the view in response to a user action as necessary

Swing Architecture



Model

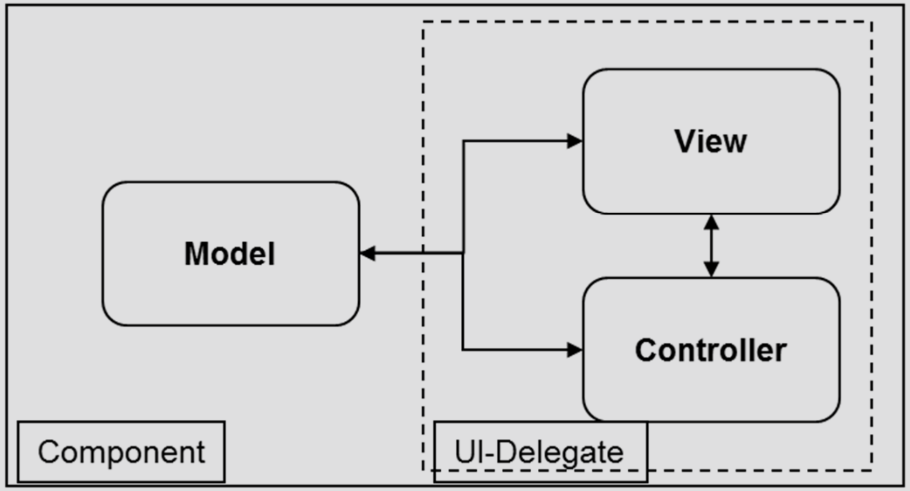
View

Controller

The model passes its data to the view for rendering

The controller updates the model based on the events received

The view determines which events are passed to the controller



Model

View

Controller

Component

UI-Delegate

With Swing, the view and the controller are combined into a UI-Delegate object

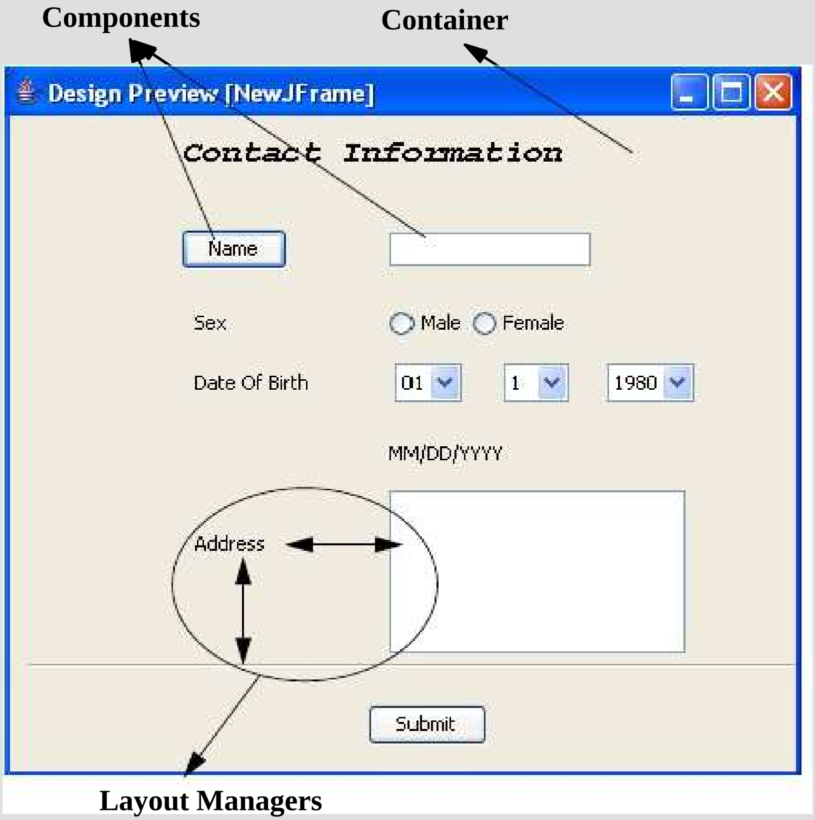
Agenda

- JFC and Swing overview
- GUI composition:
  - Create a Container Using Swing
  - Create Swing Components
  - Apply Layout Managers
- GUI event handling:
  - Key events
  - Mouse Events
- View-Model decomposition:
  - JTable

Examining the Composition of a Java Technology GUI

- A Swing API-based GUI is composed of the following elements:
  - Containers – Are on top of the GUI containment hierarchy.
  - Components – Contain all the GUI components that are derived from the JComponent class.
  - Layout Managers – Are responsible for laying out components in a container.

Examining the Composition of a Java Technology GUI



Swing Containers

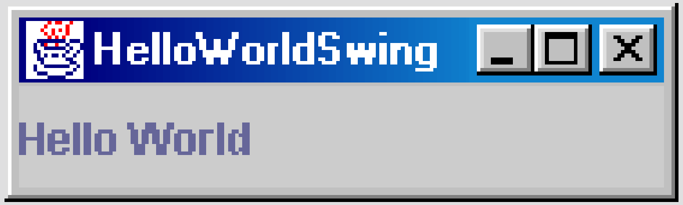
- There five Swing container classes that delegate their contents to a JRootPane instance:
  - Four heavyweight containers:
    - JFrame
    - JDialog
    - JWindow
    - JApplet.
  - One lightweight container:
    - JInternalFrame
- Because a container's contents are actually stored in its content pane, you never add() a component to a Swing container directly. Instead, you add the component to the container's content pane. You do that by calling a method named getContentPane(), using a statement similar to this:
  - aContainer.getContentPane().add(aComponent)

JFrame

- A frame implemented as an instance of the JFrame class, is a window that has decorations such as a border, a title and buttons for closing and iconifying the window.
    - The decorations on a frame are platform dependent.
- Applications with a GUI typically use at least one frame.

Example

```
import javax.swing.*;

public class HelloWorldSwing {

  public static void main(String[] args) {
    JFrame frame = new JFrame("HelloWorldSwing");
    final JLabel label = new JLabel("Hello World");
    frame.getContentPane().add(label);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);
  }
}
```



pack() causes a window to be sized to fit the preferred size and layouts of its sub-components

Most common Example

```
import javax.swing.*;

public class HelloWorldFrame extends JFrame {
  public HelloWorldFrame() {
    super(“HelloWorldSwing”);
    final JLabel label = new JLabel("Hello World");
    getContentPane().add(label);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    pack();
    setVisible(true);
  }
  public static void main(String[] args) {
    HelloWorldFrame frame = new HelloWorldFrame();
  }
}
```

In this example a custom frame is created

JDialog

- Every dialog is dependent on a frame
  - Destroying a frame destroys all its dependent dialogs.
  - When the frame is iconified, its dependent dialogs disappear from the screen.
  - When the frame is deiconified, its dependent dialogs return to the screen.
- A dialog can be modal. When a modal dialog is visible it blocks user input to all other windows in the program.
- To create custom dialogs, use the JDialog class directly (as in the previous examples).
- Swing provides several standard dialogs
  - JProgressBar, JFileChooser, JColorChooser, ...
- The JOptionPane class can be used to create simple modal dialogs
  - icons, title, text and buttons can be customized.

Containers typically used in GUI

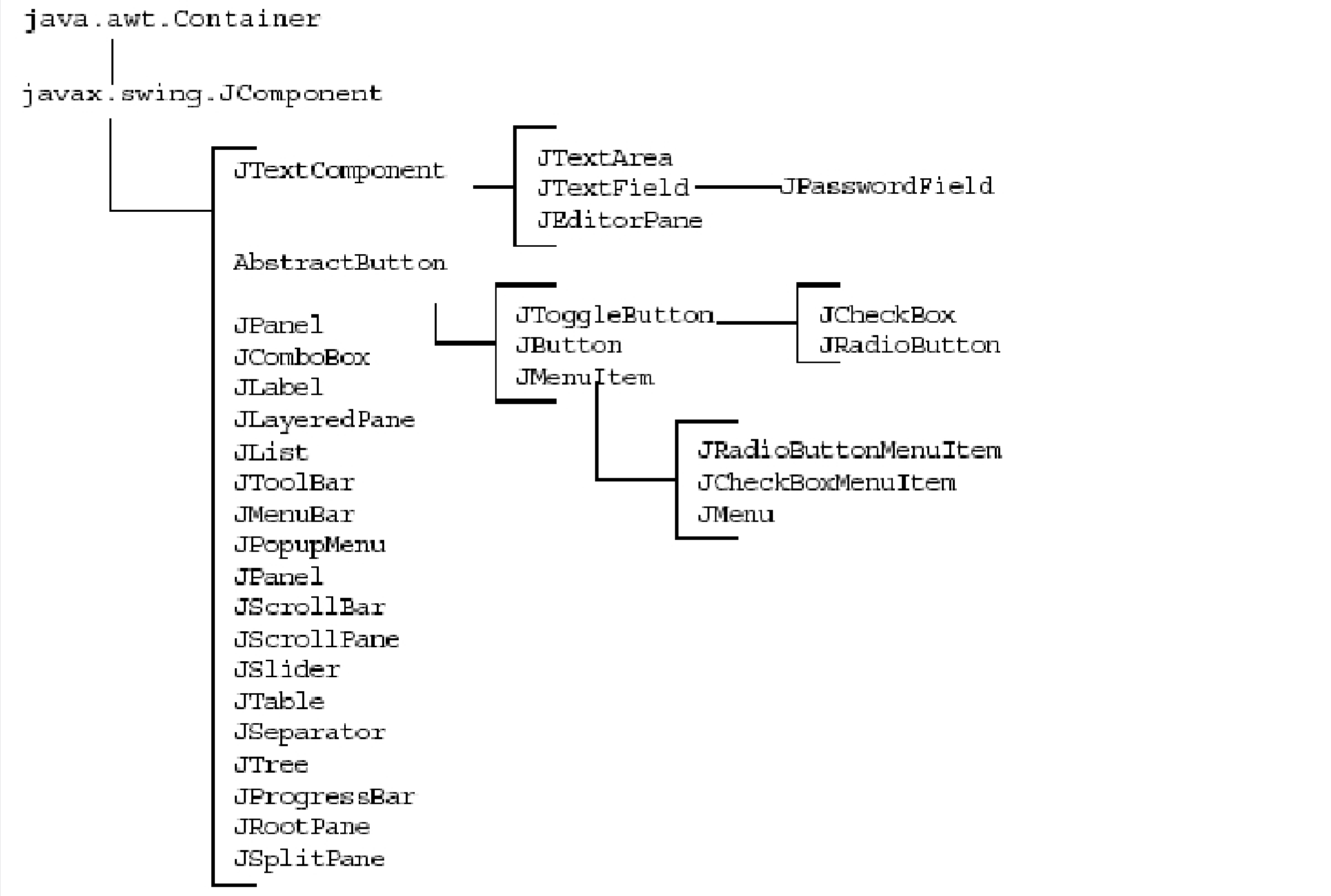
- JPanel
  - a simple container with no fancy additions
  - typically used with a layout
- JSplitPane
  - manages two panes that are separated horizontally or vertically by a divider that can be repositioned by the user
  - setLeftComponent() or setTopComponent()
  - setRightComponent() or setBottomComponent()
- JTabbedPane
  - manages multiple panes that completely overlap each other
  - tabs can be positioned to the top, bottom, left side, or right side of the container
- JDesktopPane and JInternalFrame
  - JDesktopPane is basically a container for one or more JInternalFrames
- JScrollPane and JViewport
  - A JScrollPane consists of JScrollBars, a JViewport, and the wiring between them
- JTextPane and JEditorPane
  - JTextPane and JEditorPane sound like general containers (because they are named "pane"), but in reality these are highly specialized containers that can display text and provide basic editing capabilities

Swing Components

Swing components can be broadly classified as:

- Buttons
- Text components
- Uneditable information display components
- Menus
- Formatted display components
- Other basic controls

Swing Component Hierarchy



- java.awt.Container
  - javax.swing.JComponent
    - JTextComponent
      - JTextArea
      - JTextField
        - JPasswordField
      - JEditorPane
    - AbstractButton
      - JToggleButton
        - JCheckBox
        - JRadioButton
      - JButton
      - JMenuItem
        - JRadioButtonMenuItem
        - JCheckBoxMenuItem
        - JMenu
    - JPanel
    - JComboBox
    - JLabel
    - JLayeredPane
    - JList
    - JToolBar
    - JMenuBar
    - JPopupMenu
    - JPanel
    - JScrollBar
    - JScrollPane
    - JSlider
    - JTable
    - JSeparator
    - JTree
    - JProgressBar
    - JRootPane
    - JSplitPane

JComponent

- JComponent is the base class for all Swing components except top-level containers.
    - JLabel, JButton, JList, JPanel, JTable, ...
- To use a component that inherits from JComponent, it must be placed in a containment hierarchy who's base is a top-level container.
- All the Swing components share some common properties because they all extend JComponent.
- Each component defines more specific properties.

JComponent (cont)

- The JComponent class provides the following (partial list):
  - Pluggable Look & Feel
  - Keystroke handling
  - Tooltip support
  - Accessibility
  - An infrastructure for painting
  - Support for borders.
- All descendents of JComponent are also Containers
  - A JButton can contain text, icons etc.

Common Component Properties

Property	**Methods**
Border	Border getBorder() void setBorder(Border b)
Background And foreground color	void setBackground(Color bg) void setForeground(Color bg)
Font	void setFont(Font f)
Opaque	void setOpaque(boolean isOpaque)
Maximum and minimum size	void setMaximumSize(Dimension d) void setMinimumSize(Dimension d)
Alignment	void setAlignmentX(float ax) void setAlignmentY(float ay)
Preferred size	void setPreferredSize(Dimension ps)

Component-Specific Properties

The following shows properties specific to JComboBox.

Properties	Methods
Maximum row count	void setMaximumRowCount(int count)
Model	void setModel(ComboBoxModel cbm)
Selected index	int getSelectedIndex()
Selected Item	Object getSelectedItem()
Item count	int getItemCount()
Renderer	void setRenderer(ListCellRenderer ar)
Editable	void setEditable(boolean flag)

Layout Managers

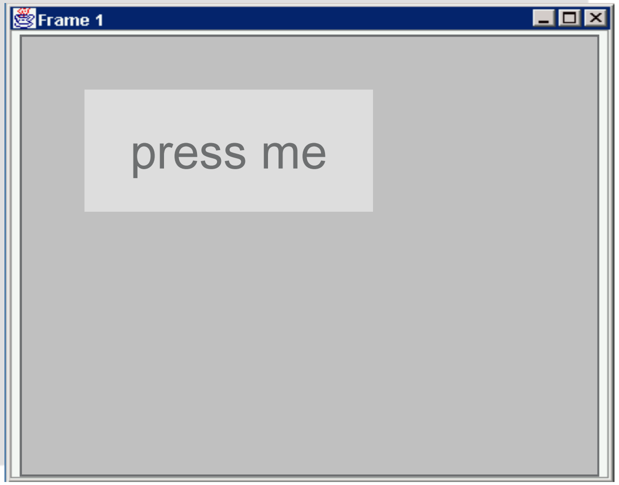
- Components (buttons, text areas, graphics, etc.) are added to Containers such as a Jframe, JPanel, or a JWindow
- A layout manager is a set of Java classes that you use to arrange the components of an interface
- Window and Frame containers use BorderLayout as their default
- layout manager

```
aComponent.setLayout(new LayoutManager());
```

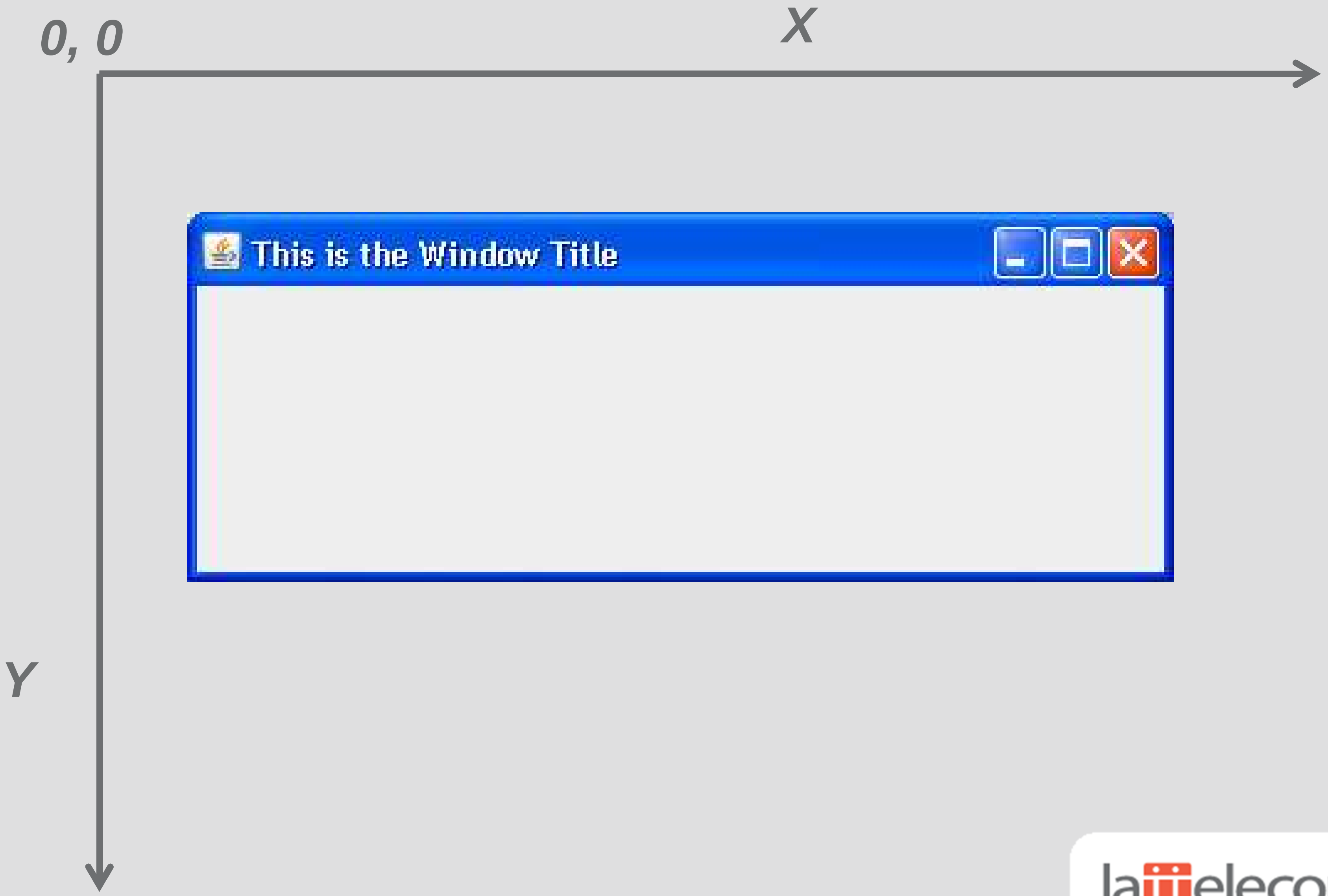
- Absolute vs Relative Placement
  - absolute size and placement can’t react interactively upon user actions

Code: null layout

```
JFrame f = new JFrame(“title”);
JPanel p = new JPanel( );
JButton b = new JButton(“press me”);

b.setBounds(new Rectangle(10,10,100,50));
p.setLayout(null);       // x,y layout
p.add(b);
f.setContentPane(p);
```

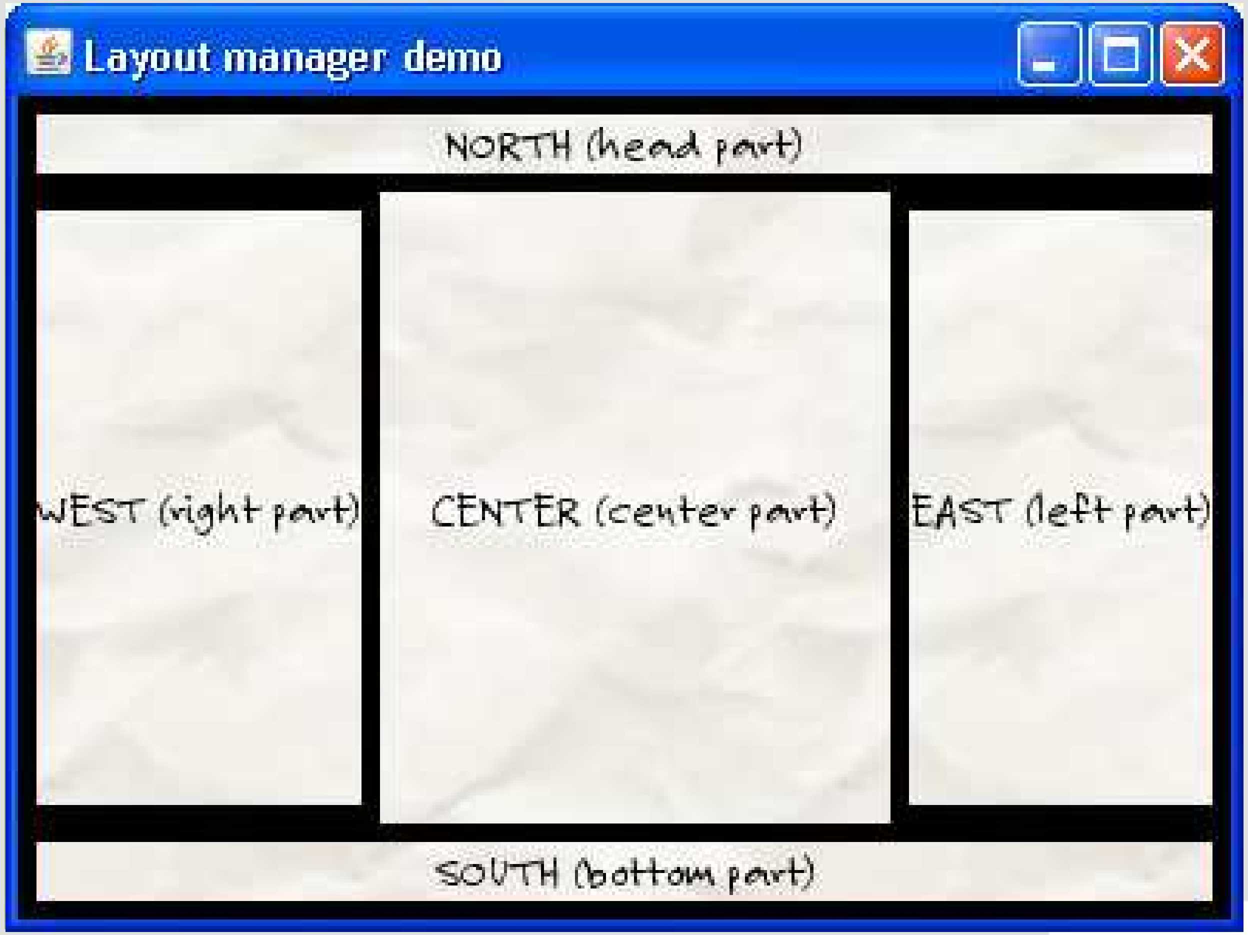


Screen x,y axis



BorderLayout Manager

- When a component is placed into one of these five regions, it will immediately expand to fill that area, observing any constraints for that area



BorderLayoutExample

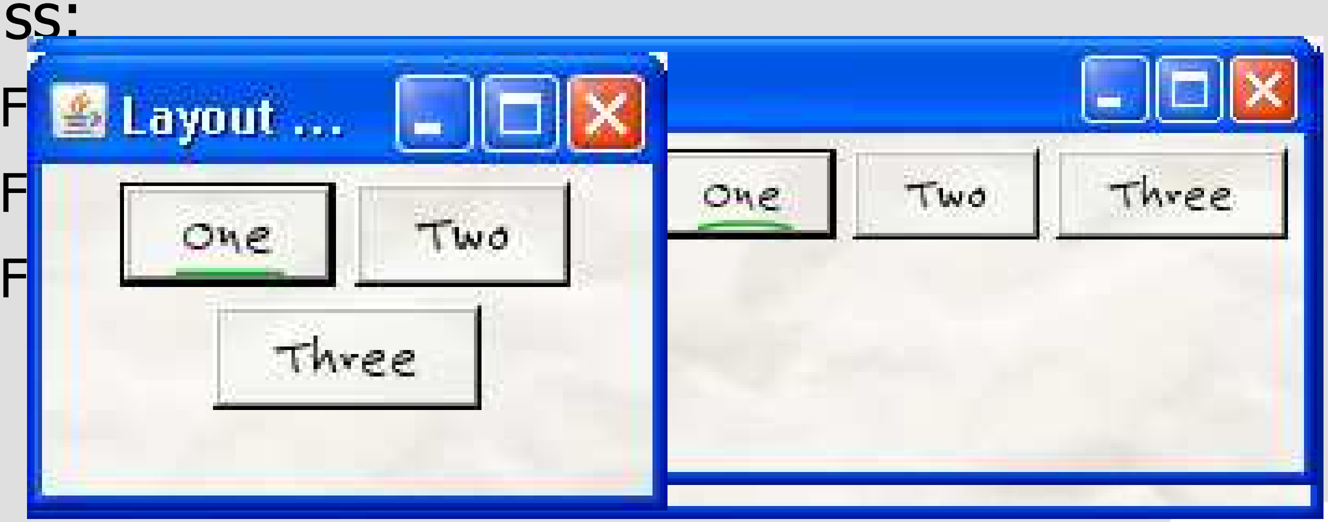
```
import java.awt.*;
import javax.swing.*;

public class BorderExample {
   private JFrame f;
   private JButton bn, bs, bw, be, bc;

   public BorderExample() {
      f = new JFrame("Border Layout");
      bn = new JButton("Button 1");
      bc = new JButton("Button 2");
      bw = new JButton("Button 3");
      bs = new JButton("Button 4");
      be = new JButton("Button 5");
   }

```

BorderLayoutExample

```
    public void launchFrame() {
        f.add(bn, BorderLayout.NORTH);
        f.add(bs, BorderLayout.SOUTH);
        f.add(bw, BorderLayout.WEST);
        f.add(be, BorderLayout.EAST);
        f.add(bc, BorderLayout.CENTER);
        f.setSize(400,200);
        f.setVisible(true);
    }

    public static void main(String args[]) {
        BorderExample guiWindow2 = new BorderExample();
        guiWindow2.launchFrame();
    }

}
```

FlowLayout Manager

- JPanel’s default layout manager is the FlowLayout
- Every component placed in the FlowLayout will maintain its suggested size and will *flow* from left to right
- The constructor FlowLayout (int index) may be used with one of the following constants in the FlowLayout class:
  - F
  - F
  - F



FlowLayoutExample

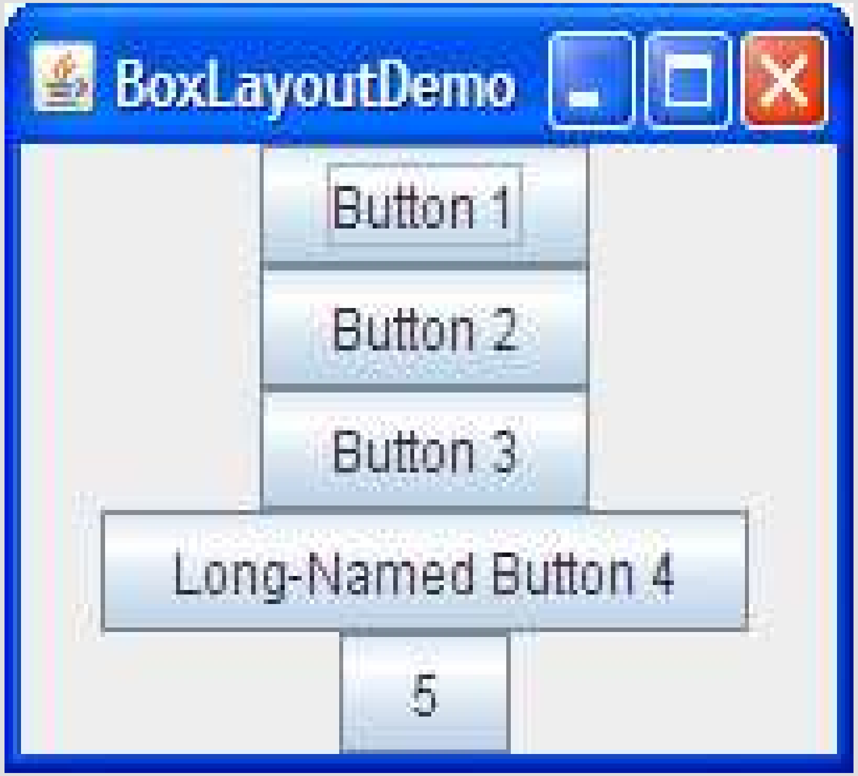
```
import javax.swing.*;
import java.awt.*;

public class LayoutExample {
        private JFrame f;
        private JButton b1;
        private JButton b2;
        private JButton b3;
        private JButton b4;
        private JButton b5;

        public LayoutExample() {
                f = new JFrame("GUI example");
                b1 = new JButton("Button 1");
                b2 = new JButton("Button 2");
                b3 = new JButton("Button 3");
                b4 = new JButton("Button 4");
                b5 = new JButton("Button 5");
        }
```

FlowLayoutExample

```

    public void launchFrame() {
        f.setLayout(new FlowLayout());
        f.add(b1);
        f.add(b2);
        f.add(b3);
        f.add(b4);
        f.add(b5);
        f.pack();
        f.setVisible(true);
    }

    public static void main(String args[]) {
        LayoutExample guiWindow = new LayoutExample();
        guiWindow.launchFrame();
    }

} // end of LayoutExample class
```

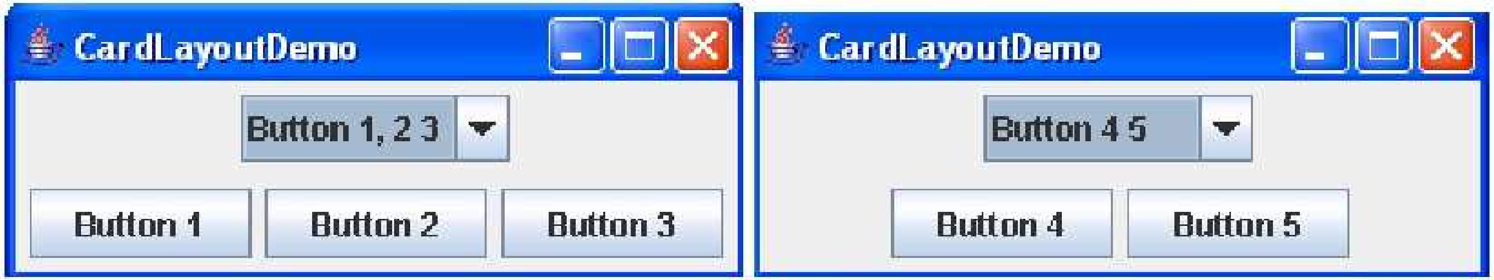
The BoxLayoutManager

The BoxLayout manager adds components from left to right, and from top to bottom in a single row of column.



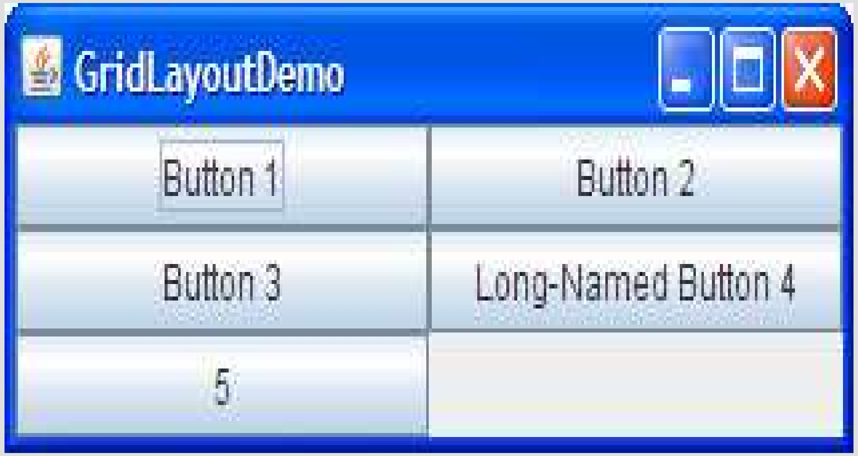
The CardLayoutManager

The CardLayout manager places the components in different cards. Cards are usually controlled by a combo box.



The GridLayoutManager

The GridLayout manager places components in rows and columns in the form of a grid.



GridLayoutExample

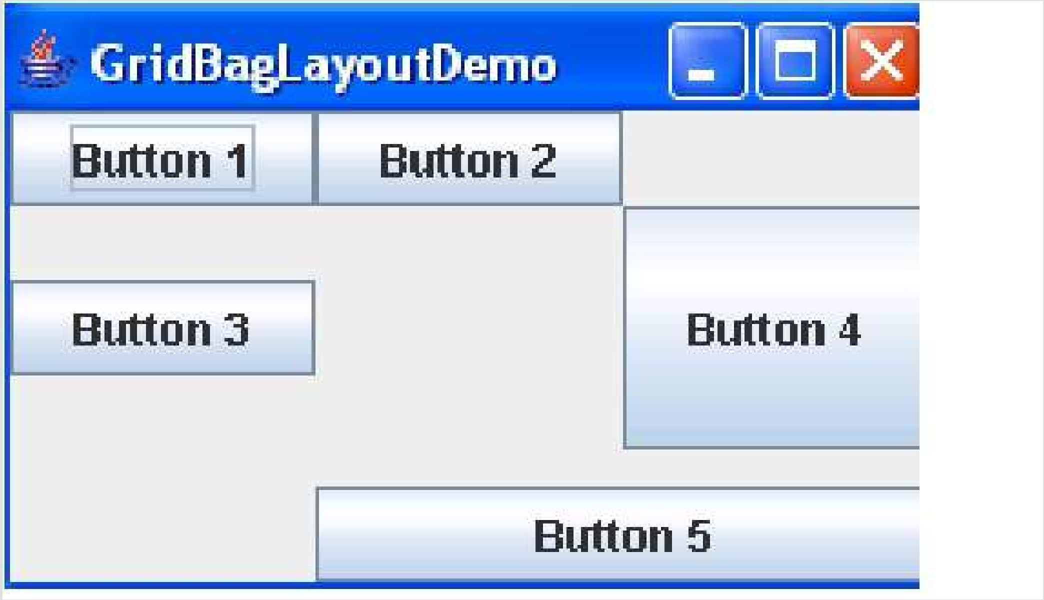
```
import java.awt.*;
import javax.swing.*;

public class GridExample {
    private JFrame f;
    private JButton b1, b2, b3, b4, b5;

    public GridExample() {
        f = new JFrame("Grid Example");
        b1 = new JButton("Button 1");
        b2 = new JButton("Button 2");
        b3 = new JButton("Button 3");
        b4 = new JButton("Button 4");
        b5 = new JButton("Button 5");
    }

```

GridLayoutExample

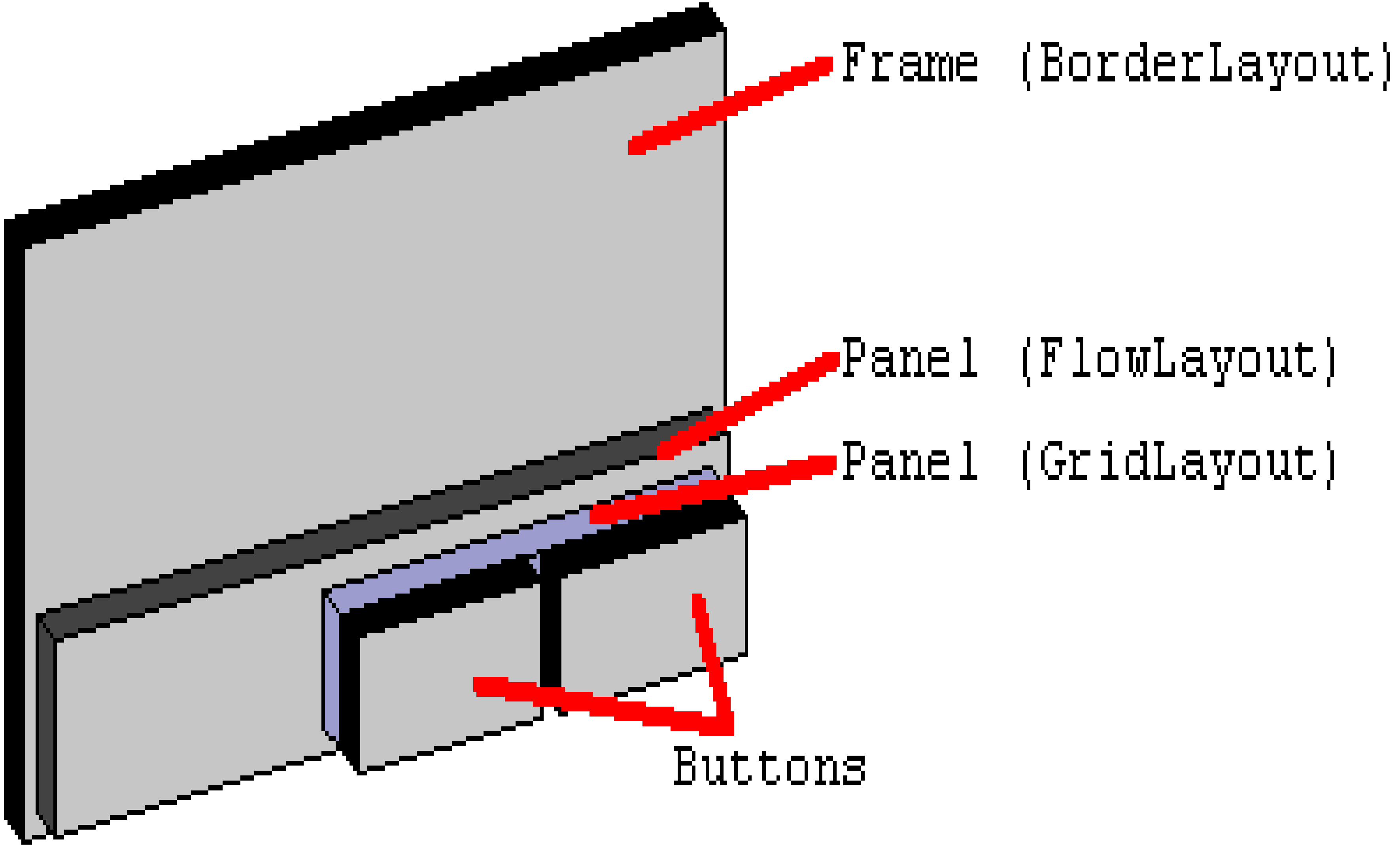
```
public void launchFrame() {
        f.setLayout (new GridLayout(3,2));

        f.add(b1);
        f.add(b2);
        f.add(b3);
        f.add(b4);
        f.add(b5);

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

    public static void main(String args[]) {
        GridExample grid = new GridExample();
        grid.launchFrame();
    }
}
```

The GridBagLayoutManager

The GridBagLayout manager arranges components in rows and columns, similar to a grid layout, but provides a wide variety of options for resizing and positioning the components.

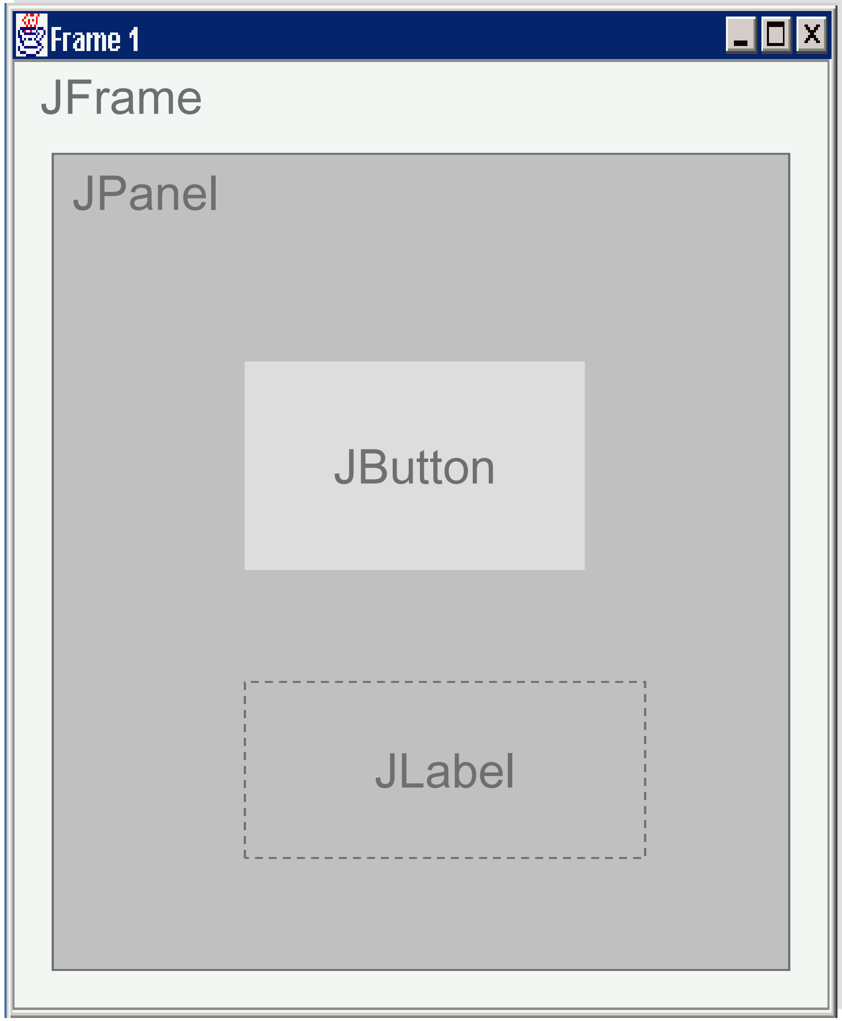


Putting it all together

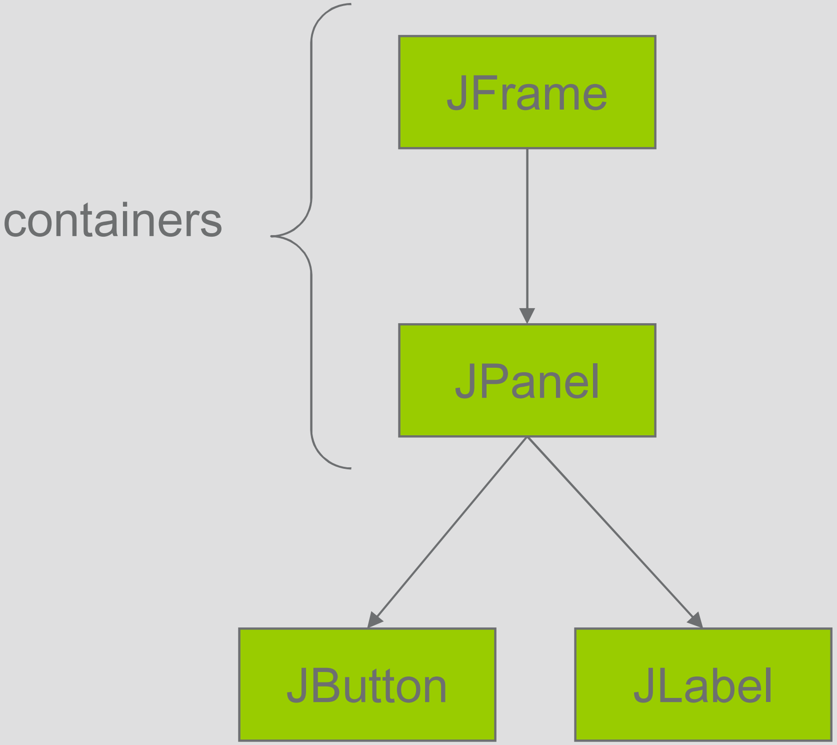


Anatomy of an Application GUI

GUI



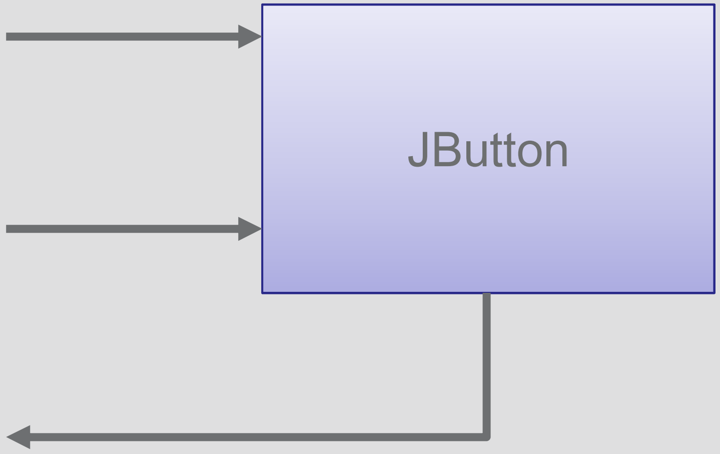
Internal structure



GUI Component API

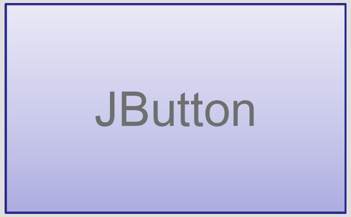
- Java: GUI component = class

- Properties
  -
- Methods
  -
- Events
  -



Using a GUI Component

1. Create it
   - Instantiate object: b = new JButton(“press me”);
2. Configure it
   - Properties: b.text = “press me”; [avoided in java]
   - Methods: b.setText(“press me”);
3. Add it
   - panel.add(b);
4. Listen to it
   - Events: Listeners



Using a GUI Component 2

1. Create it
2. Configure it
3. Add children (if container)
4. Add to parent (if not JFrame)
5. Listen to it

order important

GUI Construction

- Programmatic
- GUI builder tool

Programmatic Construction

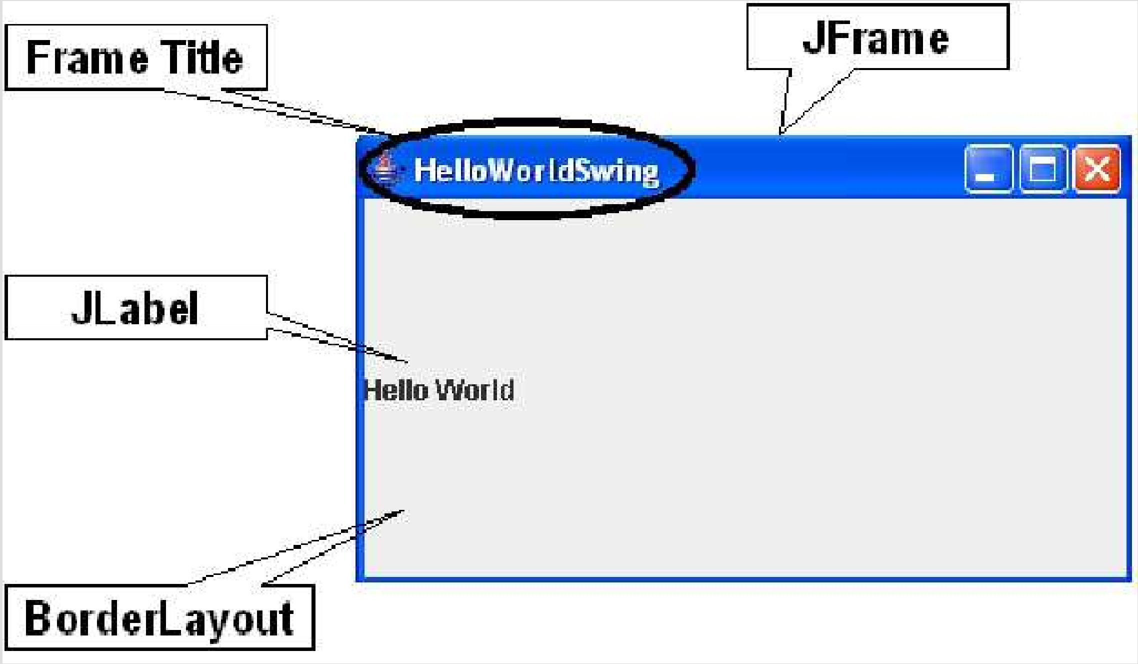
```
import javax.swing.*;
public class HelloWorldSwing {
    private static void createAndShowGUI() {
        JFrame frame = new JFrame("HelloWorldSwing");
      //Set up the window.
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel label = new JLabel("Hello World");
      // Add Label
        frame.add(label);
        frame.setSize(300,200);
      // Display Window
        frame.setVisible(true);}
```

Programmatic Construction

```
public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new  Runnable() {
            //Schedule for the event-dispatching thread:
            //creating,showing this app's GUI.
           public void run() {createAndShowGUI();}
        });
    }
}
```

Programmatic Construction

The output generated from the program



Key Methods

Methods for setting up the JFrame and adding JLabel:

- setDefaultCloseOperationJFrame.EXIT_ON_CLOSE) –Creates the program to exit when the close button is clicked.
- setVisible(true)– Makes the Jframe visible.
- add(label)– JLabelis added to the content pane not to the Jframe directly.

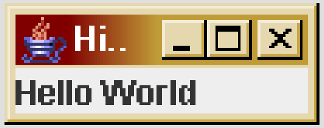
Swing and threads

- A thread is a lightweight process
- Most Swing components are not thread-safe
- Solution is to make sure all code that creates and modifies Swing components executes in the same 'event-dispatching' thread
- Start a Swing application using the following code..

Swing and Threads - starting up

```
public static void main(String[] args) {
SwingUtilities.invokeLater(new Runnable()
 {
    public void run()
     {
     createAndShowGUI(); // << method to start it
           }
       });
   }
```

createAndShowGUI

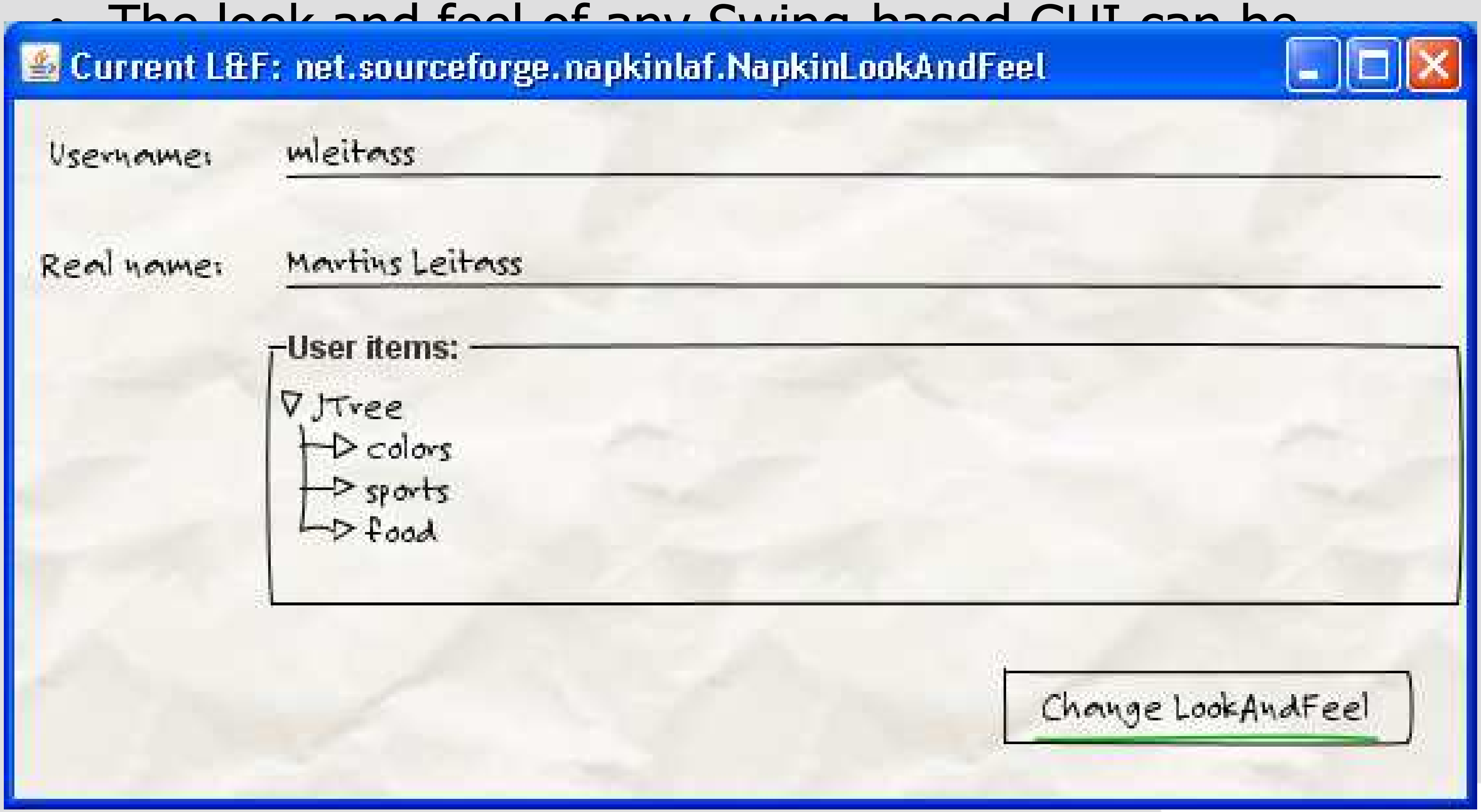


Hello World

```
private static void createAndShowGUI() {
  //Create and set up the window.
  JFrame frame = new JFrame("Hi..");
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  //Add a label.
  JLabel label = new JLabel("Hello World");
  frame.getContentPane().add(label);
  //Display the window.
  frame.pack();
  frame.setVisible(true);
}
```

Look and Feel

- Swing supports a pluggable look and feel
- The look and feel of any Swing-based GUI can be



Current L&F: net.sourceforge.napkinlaf.NapkinLookAndFeel

Username: mleitass

Real name: Martins Leitass

User items:

- JTree
  - colors
  - sports
  - food

Finding installed LaF’s

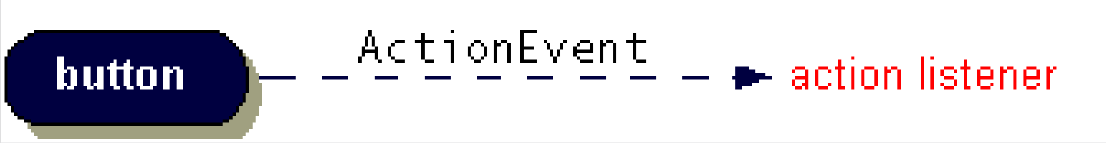
```
Object a[]=
UIManager.getInstalledLookAndFeels();
for (int i=0; i<a.length; i++)
  System.out.println(a[i]);
```

Agenda

- JFC and Swing overview
- GUI composition:
  - Create a Container Using Swing
  - Create Swing Components
  - Apply Layout Managers
- GUI event handling:
  - Key events
  - Mouse Events
- View-Model decomposition:
  - JTable

Events Handling

- Every time a user types a character or pushes a mouse button, an ***event*** occurs.
- Any object can be notified of an event by registering as an ***event listener*** on the appropriate ***event source***.
- Multiple listeners can register to be notified of events of a particular type from a particular source.



Event Handling

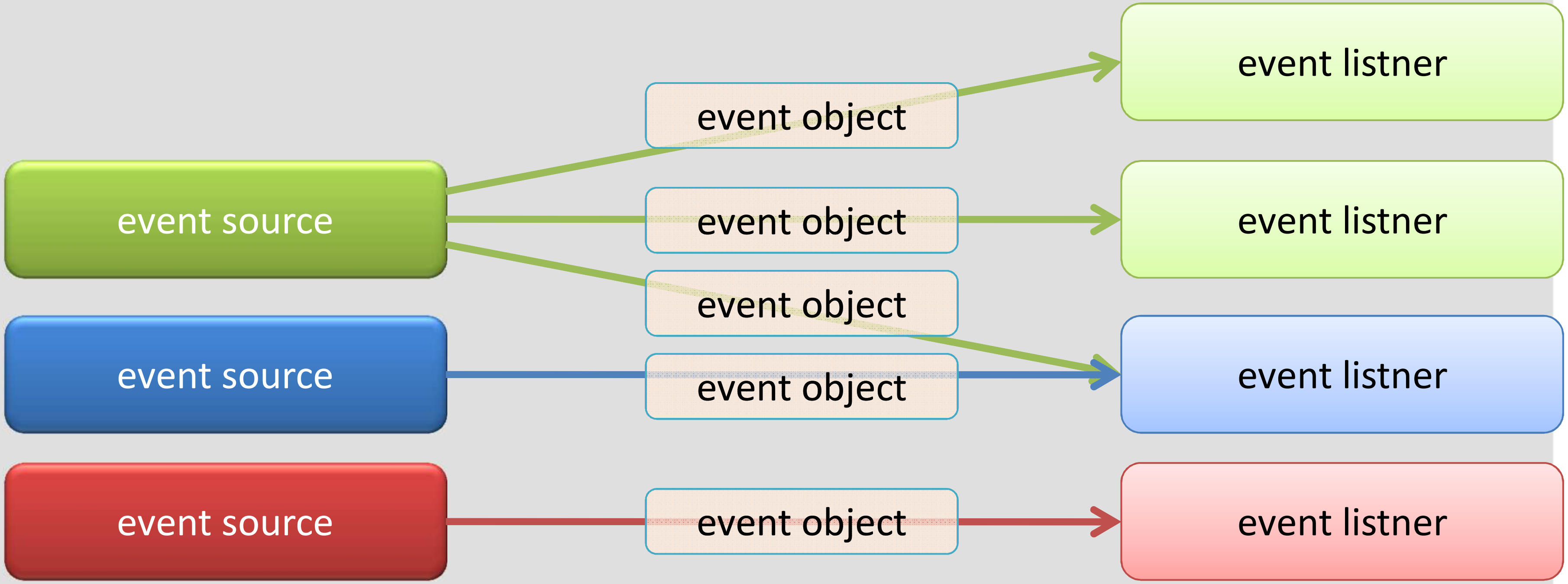
- *Event handling* is the way in which Java addresses how a GUI reacts when receiving user input.
- Java uses *event listeners* for event handling
  - Any object can be notified of the event
  - Implement the appropriate interface and be registered as an *event listener* on the appropriate *event source*
- Every event handler requires three pieces of code:
  - In the declaration for the event handler class, one line of code specifies that the class either implements a listener interface or extends a class that implements a listener interface

    -
      ```
      public class MyClass implements ActionListener {
      ```
  - Another line of code registers an instance of the event handler class as a listener on one or more components

    -
      ```
      someComponent.addActionListener(instanceOfMyClass);
      ```
  - The event handler class has code that implements the methods in the listener interface

    -
      ```
      public void actionPerformed(ActionEvent e) {
        ...//code that reacts to the action...
      }
      ```

Event handling



Delegation Model

- Client objects (handlers) register with a GUI component that they want to observe.
- GUI components trigger only the handlers for the type of event that has occurred.
- Most components can trigger more than one type of event.
- The delegation model distributes the work among multiple classes.

A Listener Example

```
import java.awt.*;
import javax.swing.*;
public class TestButton {
   private JFrame f;
   private JButton b;

   public TestButton() {
      f = new JFrame("Test");
      b = new JButton("Press Me!");
      b.setActionCommand("ButtonPressed");
   }

   public void launchFrame() {
      b.addActionListener(new ButtonHandler());
      f.add(b,BorderLayout.CENTER);
      f.pack();
      f.setVisible(true);
   }
```

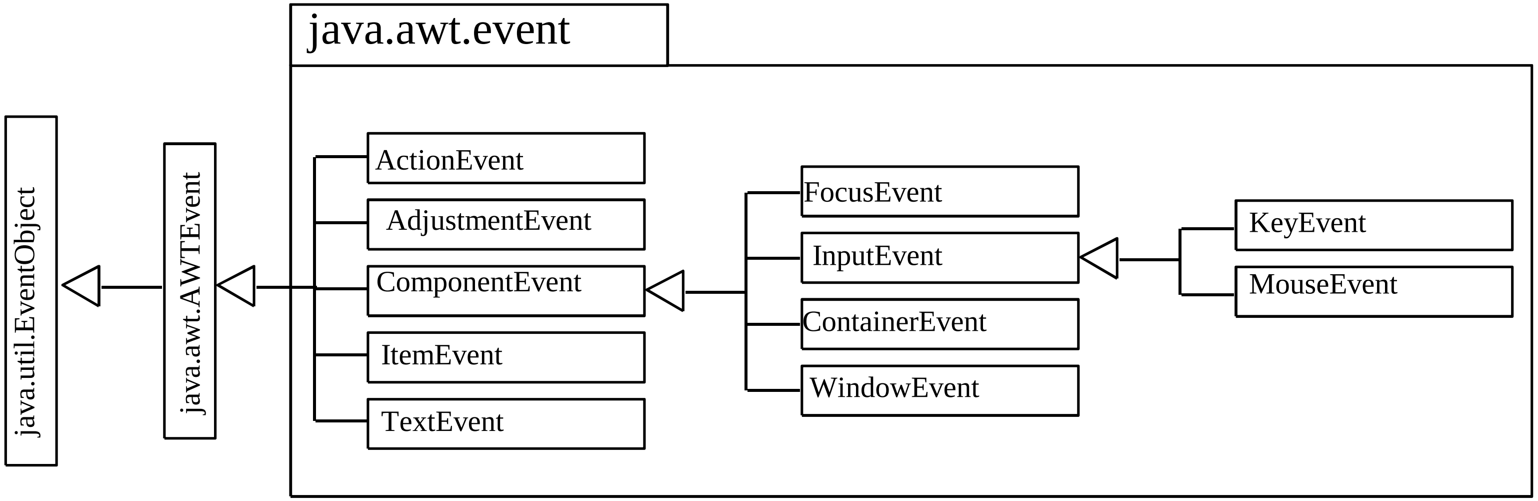
A Listener Example

```
    public static void main(String args[]) {
        TestButton guiApp = new TestButton();
        guiApp.launchFrame();
    }
}
```

Code for the event listener looks like the following:

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

public class ButtonHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Action occurred");
        System.out.println("Button’s command is: "
                                        + e.getActionCommand());
    }
}
```

Event Categories



Event handling: Interfaces

- ActionListener
  - User clicks a button, presses Enter while typing in a text field, or chooses a menu item
- WindowListener
  - User closes, minimizes or maximizes a frame (main window)
- MouseListener
  - User presses a mouse button while the cursor is over a component
- MouseMotionListener
  - User moves the mouse over a component
- ComponentListener
  - Component becomes visible
- FocusListener
  - Component gets the keyboard focus
- ListSelectionListener
  - Table or list selection changes
- PropertyChangeListener
  - Any property in a component changes such as the text on a label

Action Listener

- responds to the user's indication that some implementation-dependent action should occur
- The ActionListener interface is used in many separate situations:
  - When an item is selected from a list box with a double click
  - When an item is selected from a list in combo box
  - When a menu item is selected
  - When the ENTER key is clicked in a text field
  - When a certain amount of time has elapsed for a Timer component
  - When a button pressed
- Methods:
  - actionPerformed(actionEvent)
    - Called just after the user informs the listened-to component that an action should occur

The MouseListener Interface

- A mouse event is Java input event that occurs when a user clicks the mouse button or the cursor enters or leaves the area of component
- `mouseClicked(MouseEvent)`
  - Called just after the user clicks the listened-to component.
- `mouseEntered(MouseEvent)`
  - Called just after the cursor enters the bounds of the listened-to component.
- `mouseExited(MouseEvent)`
  - Called just after the cursor exits the bounds of the listened-to component.
- `mousePressed(MouseEvent)`
  - Called just after the user presses a mouse button while the cursor is over the listened-to component.
- `mouseReleased(MouseEvent)`
  - Called just after the user releases a mouse button after a mouse press over the listened-to component.

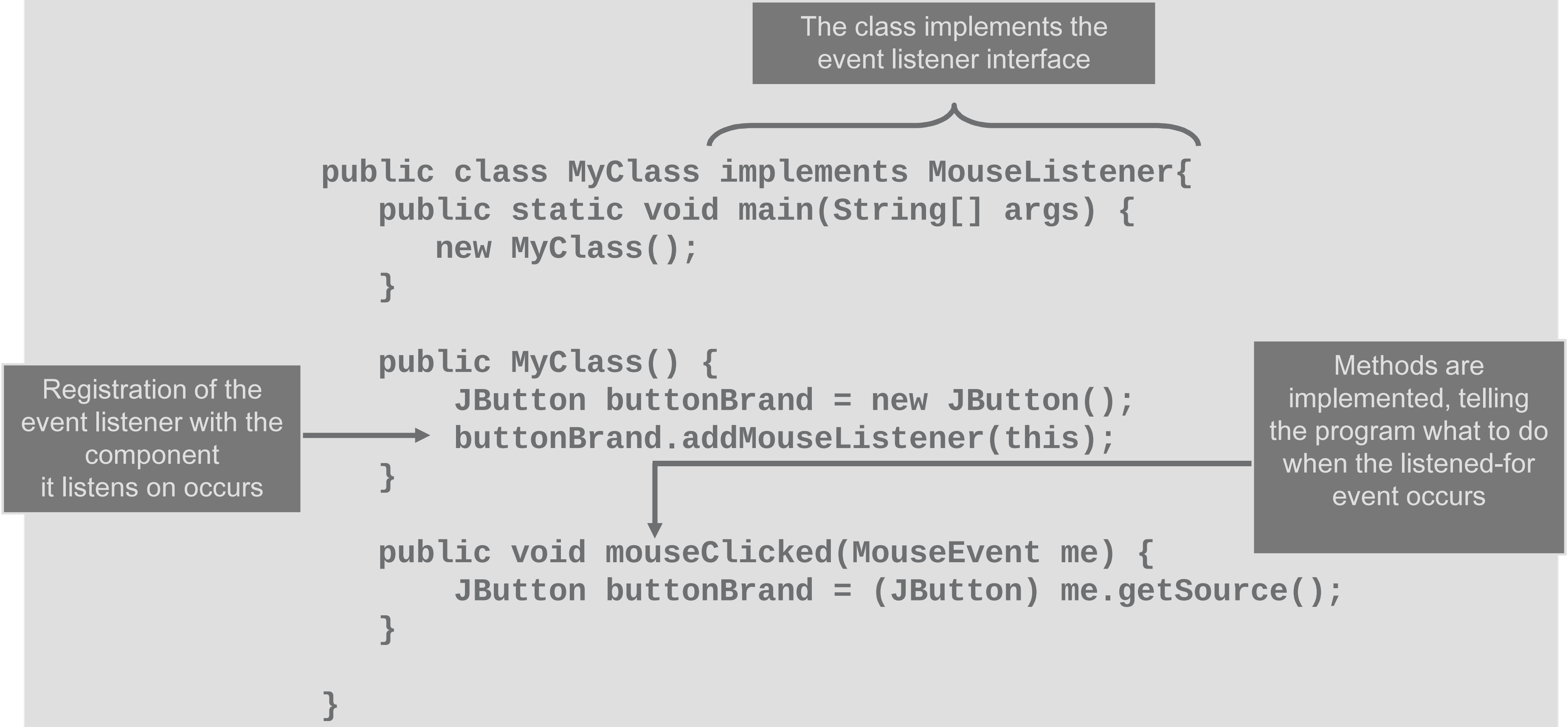
The KeyListener Interface

- key events are fired by the component with the keyboard focus when the user presses or releases keyboard keys
- `keyTyped(KeyEvent)`
  - Called just after the user types a Unicode character into the listened-to component.
- `keyPressed(KeyEvent)`
  - Called just after the user presses a key while the listened-to component has the focus.
- `keyReleased(KeyEvent)`
  - Called just after the user releases a key while the listened-to component has the focus.

The WindowListener interface

- `windowOpened(WindowEvent e);`
  - is called after the window has been opened
- `windowClosing(WindowEvent e);`
  - is called when the user has issued a window manager command to close the window. Note that the window will close only if its hide or dispose method is called
- `windowClosed(WindowEvent e);`
  - is called after the window has closed
- `windowIconified(WindowEvent e);`
  - is called after the window has been iconified
- `windowDeiconified(WindowEvent e);`
  - is called after the window has been deiconified
- `windowActivated(WindowEvent e);`
  - is called after the window has become active. Only a frame or dialog can be active
- `windowDeactivated(WindowEvent e);`
  - is called after the window has become deactivated

Event Handling Implementation



The class implements the event listener interface

```
public class MyClass implements MouseListener{
   public static void main(String[] args) {
      new MyClass();
   }

   public MyClass() {
       JButton buttonBrand = new JButton();
       buttonBrand.addMouseListener(this);
   }

   public void mouseClicked(MouseEvent me) {
       JButton buttonBrand = (JButton) me.getSource();
   }

}
```

Registration of the event listener with the component it listens on occurs

Methods are implemented, telling the program what to do when the listened-for event occurs

Adapter Classes

- Several of the AWT listener interfaces come with a companion adapter class that implements all the methods in the interface to do nothing
- FocusAdapter
  - implements FocusListener
- MouseMotionAdapter
  - implements MouseMotionListener
- KeyAdapter
  - implements KeyListener
- WindowAdapter
  - implements WindowListener, WindowStateListener, WindowFocusListener
- MouseAdapter
  - implements MouseListener

Semantic and Low-Level Events

- The AWT makes a useful distinction between low-level and semantic events:
  - A semantic event is one that expresses what the user is doing
  - Low-level events are those events that make semantic events possible

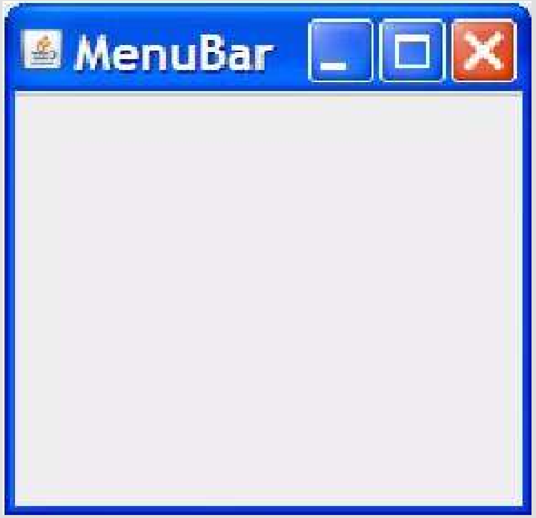
- Most commonly used semantic event classes:
  - `ActionEvent` (for a button click, a menu selection, selecting a list item, or ENTER typed in a text field)
  - `AdjustmentEvent` (the user adjusted a scrollbar)
  - `ItemEvent` (the user made a selection from a set of checkbox or list items)
- Five low-level event classes are commonly used:
  - `KeyEvent` (a key was pressed or released)
  - `MouseEvent` (the mouse button was pressed, released, moved, or dragged)
  - `MouseWheelEvent` (the mouse wheel was rotated)
  - `FocusEvent` (a component got focus, or lost focus). See page 321 for more information about the focus concept.
  - `WindowEvent` (the window state changed)

How to Create a Menu

1. Create a JMenuBar object, and set it into a menu container, such as a JFrame.
2. Create one or more JMenu objects, and add them to the menu bar object.
3. Create one or more JMenuItem objects, and add them to the menu object.

Creating a JMenuBar

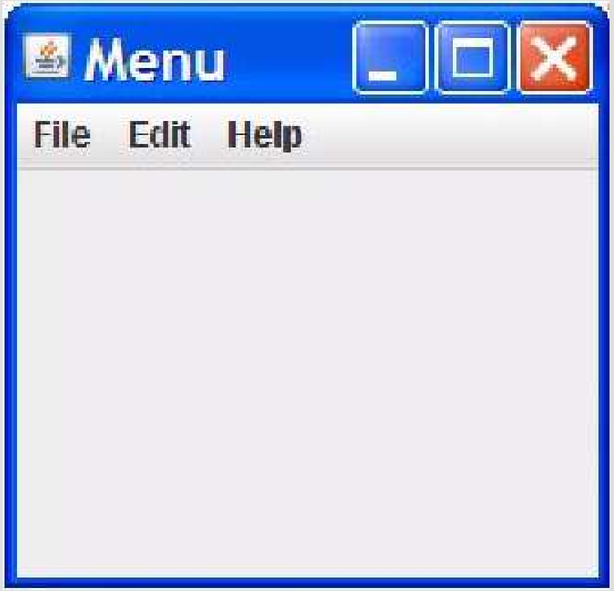
```
f  =  new  JFrame("MenuBar");
mb  =  new  JMenuBar();
f.setJMenuBar(mb);
```



Creating a JMenu

```
f  =  new  JFrame("Menu");
mb  =  new  JMenuBar();
m1  =  new  JMenu("File");
m2  =  new  JMenu("Edit");
m3  =  new  JMenu("Help");
mb.add(m1);
mb.add(m2);
mb.add(m3);
f.setJMenuBar(mb);
```

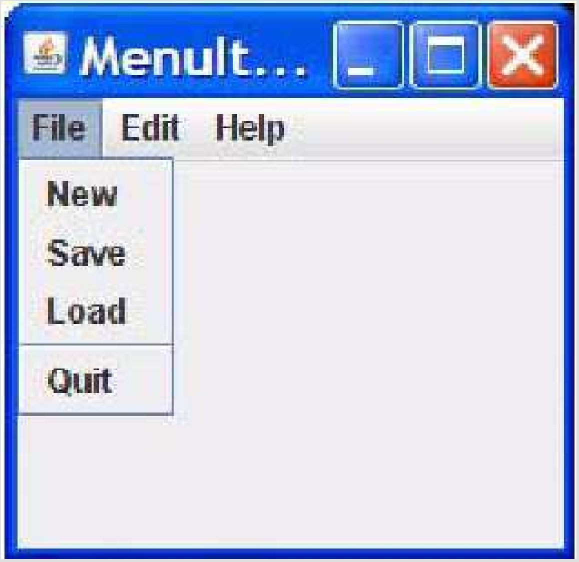
Creating a JMenu



Creating a JMenuItem

```
mi1 = new JMenuItem("New");
mi2 = new JMenuItem("Save");
mi3 = new JMenuItem("Load");
mi4 = new JMenuItem("Quit");
mi1.addActionListener(this);
mi2.addActionListener(this);
mi3.addActionListener(this);
mi4.addActionListener(this);
m1.add(mi1);
m1.add(mi2);
m1.add(mi3);
m1.addSeparator();
m1.add(mi4);
```

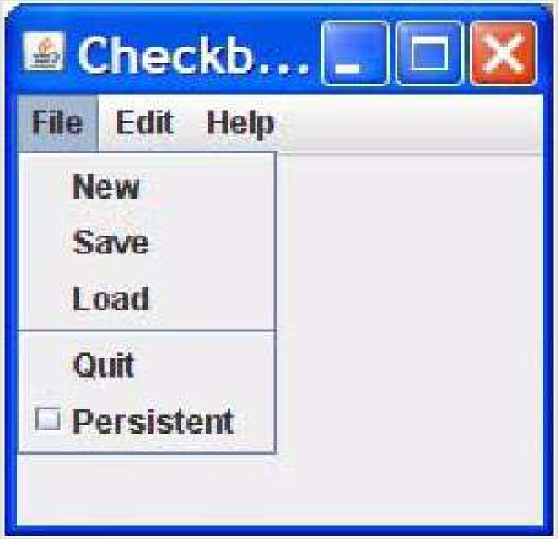
Creating a JMenuItem



Creating a JCheckBoxMenuItem

```
f = new JFrame("CheckboxMenuItem");
mb = new JMenuBar();
m1 = new JMenu("File");
m2 = new JMenu("Edit");
m3 = new JMenu("Help");
mb.add(m1);
mb.add(m2);
mb.add(m3);
f.setJMenuBar(mb);
...........
mi5 = new JCheckBoxMenuItem("Persistent");
mi5.addItemListener(this);
m1.add(mi5);
```

Creating a JCheckBoxMenuItem



Agenda

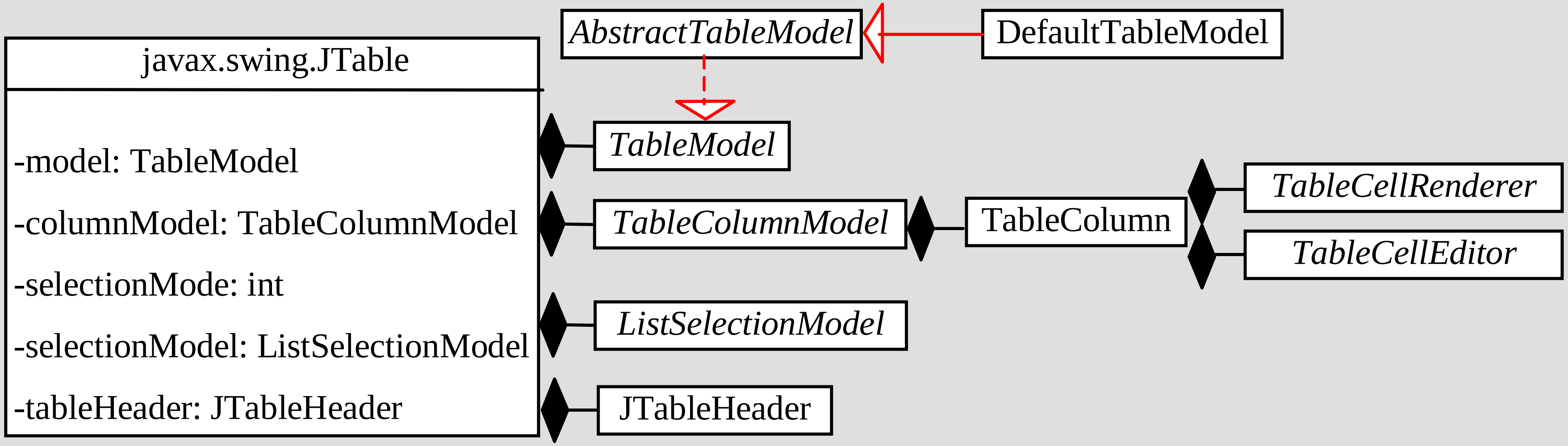
- JFC and Swing overview
- GUI composition:
  - Create a Container Using Swing
  - Create Swing Components
  - Apply Layout Managers
- GUI event handling:
  - Key events
  - Mouse Events
- View-Model decomposition:
  - JTable

Class `JTable`

- `JTable` is a user-interface component that presents data in a two-dimensional table format
- The `JTable` has many features that make it possible to customize its rendering and editing but provides defaults for these features.
- A `JTable` consists of:
  - Rows of data
  - Columns of data
  - Column headers
  - An editor, if you want cells to be editable
  - A `TableModel`, usually a subclass of `AbstractTableModel`, which stores the table’s data
  - A `TableColumnModel`, usually `DefaultTableColumnModel`, which controls the behavior of the table’s columns and gives access to the `TableColumns`
  - A `ListSelectionModel`, usually `DefaultListSelectionModel`, which keeps track of the `JTable`’s currently selected row(s)
  - A `TableCellRenderer`, usually an instance of `DefaultCellRenderer`
  - Multiple `TableColumns`, which store graphical information about each column
  - A `JTableHeader` which displays column headers

JTable and Its Supporting Models

NOTE: All the supporting interfaces and classes for JTable are grouped in the javax.swing.table package.



Class `JTable`

- Steps in creating and using `JTable`
  - Create a `JTable` (there are 7 different constructors)
  - Create a `JScrollPane` that can be used to scroll around the `JTable`
  - Place the `JTable` within a container
  - Control whether grid lines should be drawn via `setShowGrid()`
  - Specify a default value for a cell via `setValueAt()`
  - Get the value for a cell via `getValueAt()`
  - Make individual cells selectable via `setCellSelectionEnabled()`
  - Find out which cells are selected via the `JTable`’s `ListSelectionModel` and the `TableColumnModel`’s `ListSelectionModel`
  - Add new rows and columns via the `JTable`’s `TableModel`

Class `AbstractTableModel`

- `AbstractTableModel` is an abstract class that implements most of the `TableModel` interface

- The `TableModel` methods that are not implemented are `getRowCount()`, `getColumnCount()`, and `getValueAt()`

- Steps in creating and using `AbstractTableModel`
  - Create an `AbstractTableModel` subclass
  - Implement the `getRowCount()`, `getColumnCount()`, and `getValueAt()` methods
  - Instantiate an instance of the subclass
  - Create a `JTable` using the subclass via `new JTable( model )`

Class `AbstractTableModel`

- To set up a table with 10 rows and 10 columns of numbers:

```
TableModel dataModel = new AbstractTableModel() {
  public int    getColumnCount() { return 10; }
  public int    getRowCount()    { return 10;}
  public Object getValueAt(int row, int col) {
  return new Integer(row*col); }
  };
JTable table = new JTable(dataModel);
JScrollPane scrollpane = new JScrollPane(table);
```

Class `DefaultTableModel`

- `DefaultTableModel` is the JFC's default subclass of the abstract `AbstractTableModel` class

- If a `JTable` is created and no `TableModel` is specified, the `JTable` creates an instance of `DefaultJTableModel` and uses it to hold the table's data

- If you have complex data, you may prefer to extend the `AbstractTableModel` yourself

- Steps in creating and using `DefaultTableModel`
  - Create a `DefaultTableModel` (there are 6 different constructors)

    ```
    DefaultTableModel( Vector data, Vector columnIDs)
    ```

  - Create a `JTable` using the `DefaultTableModel` via `new JTable(model)`

Class `DefaultTableModel`

- Steps in creating and using `DefaultTableModel`
  - Define a `TableModelListener` to receive `TableModelEvents` when the model changes, or when one or more cell's contents change
  - Add a row to the `DefaultTableModel` via `addRow()`
  - Add a column to the `DefaultTableModel` via `addColumn()`
  - Get the current value of a cell in a `DefaultTableModel` via `getValueAt()`
  - Move one or more rows via `moveRow()`
  - Load a new set of data into a `DefaultTableModel` via `setDataVector()`
  - Get the number of rows or columns in a `DefaultTableModel` via `getRowCount()` and `getColumnCount()`

Class TableColumn

- A TableColumn contains the graphical attributes for a single column of data in a JTable's model

- It stores information about the column header, the column height and width, and how cells in the column should be drawn and edited

- Steps in creating and using TableColumn
  - TableColumns are created automatically when columns are added to the table model. They are accessed via the table column model via getColumn()
  - Specify the TableCellEditor to use when editing the TableColumn's cells

```
JCheckBox cbox = new JCheckBox()
DefaultCellEditor editor = new DefaultCellEditor(cbox)
tableColumn.setCellEditor(editor)
```

  - Change the column header via setHeaderValue()

Class `DefaultTableColumnModel`

- `DefaultTableColumnModel` is the JFC's default implementation of the `TableColumnModel` interface

- This class is used to keep track of information about table columns. It gives access to `TableColumns` and keeps track of general characteristics of columns, like column margins and widths. It also contains a `ListSelectionModel` that it uses to keep track of which columns are currently selected

- Steps in creating and using `DefaultTableColumnModel`
  - You will usually let the `JTable` create it
  - Specify the selection mode for the `DefaultTableColumnModel` via `setSelectionMode()`

More information

- http://docs.oracle.com/javase/tutorial/uiswing/TOC.html