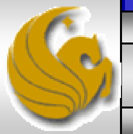
COP 4610L: Applications in the Enterprise
Spring 2005

GUI Components: Part 1

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CSB 242, 823-2790
http://www.cs.ucf.edu/courses/cop4610L/spr2005

School of Computer Science
University of Central Florida



GUI and Event-Driven Programming

- Most users of software will prefer a graphical user-interface (**GUI**) -based program over a console-based program any day of the week.
- A GUI gives an application a distinctive "look" and "feel".
- Providing different applications with consistent, intuitive user interface components allows users to be somewhat familiar with an application, so that they can learn it more quickly and use it more productively.
- Studies have found that users find GUIs easier to manipulate and more forgiving when misused.
- The GUI ease of functionality comes at a programming price – GUI-based programs are more complex in their structure than console-based programs.

The Trade-off Between Ease of Use and Software Complexity

High

Complexity

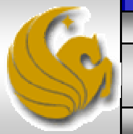
Low

Total Software Complexity

Difficulty of Use

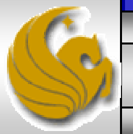
Popularity of GUIs

- Despite the complexity of GUI programming, its dominance in real-world software development makes it imperative that GUI programming be considered even in an introductory course such as this one.
- Do not confuse GUI-based programming with applets. Although some of the features of the first few GUIs that we look at will be similar to those you used in your first applet program, notice that we are developing application programs here not applets.
  - The execution of a GUI-based application also begins in its method main(). However, method main() is normally responsible only for creating an *instance* of the GUI.
  - After creating the GUI, the flow of control will shift from the main() method to an event-dispatching loop that will repeatedly check for user interactions with the GUI.



Components of the GUI

- GUI's are built from GUI components. These are sometimes called controls or widgets (short for *windows gadgets*) in languages other than Java.
- A GUI component is an object with which the user interacts via the mouse, keyboard, or some other input device (voice recognition, light pen, etc.).
- Many applications that you use on a daily basis use windows or dialog boxes (also called dialogs) to interact with the user.
- Java's `JOptionPane` class (package `javax.swing`) provides prepackaged dialog boxes for both input and output.
  - These dialogs are displayed by invoking static JOptionPane methods.
- The simple example on the next page illustrates this concept.

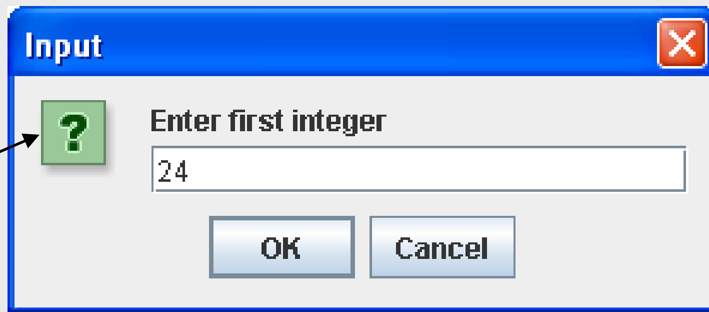


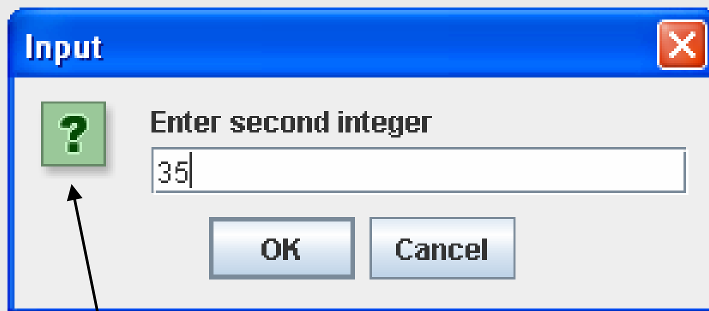
Example GUI illustrating The JOptionPane class

```
// A simple integer addition program that uses JOptionPane for input and output.
import javax.swing.JOptionPane; // program uses JOptionPane class

public class Addition
{
   public static void main( String args[] )
   {
      // obtain user input from JOptionPane input dialogs
      String firstNumber =
         JOptionPane.showInputDialog( "Enter first integer" );
      String secondNumber =
          JOptionPane.showInputDialog( "Enter second integer" );

      // convert String inputs to int values for use in a calculation
      int number1 = Integer.parseInt( firstNumber );
      int number2 = Integer.parseInt( secondNumber );

      int sum = number1 + number2; // add numbers

      // display result in a JOptionPane message dialog
      JOptionPane.showMessageDialog( null, "The sum is " + sum,
         "Sum of Two Integers", JOptionPane.INFORMATION_MESSAGE );
   } // end method main
} // end class Addition
```

This parameter, called the Message Dialog Constant, indicates the type of information that the box is displaying to the user and will cause the appropriate icon to appear in the dialog box (see next page).

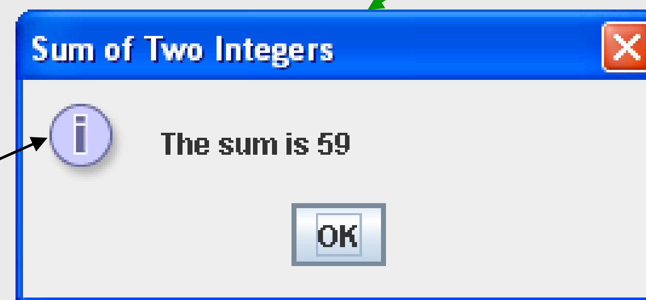
Output from execution of the Addition Example





User enters their integers in the dialog box and clicks OK after entering each.

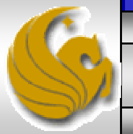
Result is displayed in a third dialog box.



The “?” icons appear because of the input dialog, the “I” icon appears because of a specific parameter.

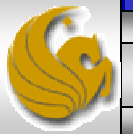
Overview of Swing Components

- Java GUI-based programming typically makes use of the `swing` API. The `swing` API provides over 40 different types of graphical components and 250 classes to support the use of these components.
  - JFrame: Represents a titled, bordered window.
  - JTextArea: Represents an editable multi-line text entry component.
  - JLabel: Displays a single line of uneditable text or icons.
  - JTextField: Enables the user to enter data from the keyboard. Can also be used to display editable or uneditable text.
  - JButton: Triggers and event when clicked with the mouse.
  - JCheckBox: Specifies an option that can be selected or not selected.
  - JComboBox: Provides a drop-down list of items from which the user can make a selection by clicking an item or possibly by typing into the box.
  - JList: Provides a list of items from which the user can make a selection by clicking on any item in the list. Multiple elements can be selected.
  - JPanel: Provides an area in which components can be placed and organized. Can also be used as a drawing area for graphics.



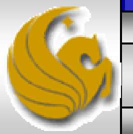
Displaying Text and Images in a Window

- Most windows that you will create will be an instance of class `JFrame` or a subclass of `JFrame`.
- `JFrame` provides the basic attributes and behaviors of a window that you expect – a title bar at the top of the window, and buttons to minimize, maximize, and close the window.
- Since an application's GUI is typically specific to the application, most of the examples in this section of the notes will consist of two classes – a subclass of `JFrame` that illustrates a GUI concept and an application class in which main creates and displays the application's primary window.



Labeling GUI Components

- A typical GUI consists of many components. In a large GUI, it can be difficult to identify the purpose of every component unless the GUI designer provides text instructions or information stating the purpose of each component.
- Such text is known as a label and is created with class `JLabel` (which is a subclass of `JComponent`).
- A `JLabel` displays a single line of read-only (noneditable) text, an image, or both text and an image.
- The sample code on the next page illustrates some of the features of the `JLabel` class.



```java
// Demonstrating the JLabel class.
import java.awt.FlowLayout; // specifies how components are arranged
import javax.swing.JFrame; // provides basic window features
import javax.swing.JLabel; // displays text and images
import javax.swing.SwingConstants; // common constants used with Swing
import javax.swing.Icon; // interface used to manipulate images
import javax.swing.ImageIcon; // loads images

public class LabelFrame extends JFrame
{
   private JLabel label1; // JLabel with just text
   private JLabel label2; // JLabel constructed with text and icon
   private JLabel label3; // JLabel with added text and icon

   // LabelFrame constructor adds JLabels to JFrame
   public LabelFrame()
   {
      super( "Testing JLabel" );
      setLayout( new FlowLayout() ); // set frame layout

      // JLabel constructor with a string argument
      label1 = new JLabel( "Label with text" );
      label1.setToolTipText( "This is label #1" );
      add( label1 ); // add label #1 to JFrame
```

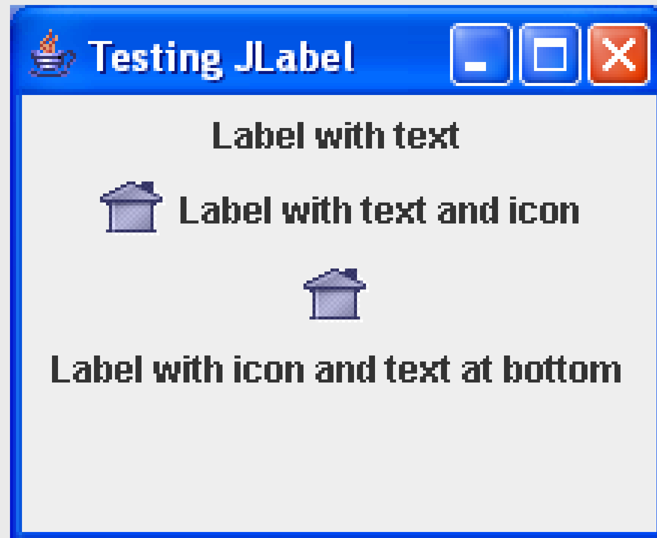
Example GUI illustrating
The JLabel class

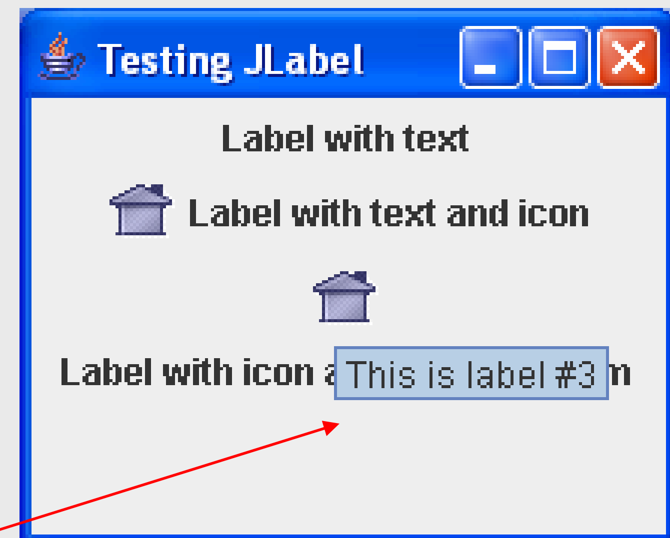
```
    // JLabel constructor with string, Icon and alignment arguments
    Icon home = new ImageIcon( getClass().getResource( "home.gif" ) );
    label2 = new JLabel( "Label with text and icon", home,
      SwingConstants.LEFT );
    label2.setToolTipText( "This is label #2" );
    add( label2 ); // add label #2 to JFrame

    label3 = new JLabel(); // JLabel constructor no arguments
    label3.setText( "Label with icon and text at bottom" );
    label3.setIcon( home ); // add icon to JLabel
    label3.setHorizontalTextPosition( SwingConstants.CENTER );
    label3.setVerticalTextPosition( SwingConstants.BOTTOM );
    label3.setToolTipText( "This is label #3" );
    add( label3 ); // add label3 to JFrame
  } // end LabelFrame constructor
} // end class LabelFrame
```

LabelFrame class continues

```
// Driver class for Testing LabelFrame.
import javax.swing.JFrame;
public class LabelTest {
   public static void main( String args[] )   {
      LabelFrame labelFrame = new LabelFrame(); // create LabelFrame
      labelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE
);
      labelFrame.setSize( 220, 180 ); // set frame size
      labelFrame.setVisible( true ); // display frame
   } // end main
} // end class LabelTest
```

Output from execution of the LabelTest Example

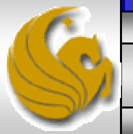




Moving cursor over the label will display ToolTipText if GUI designer provided it.

GUI Programming

- Besides having a different look and feel from console-based programs, GUI-based programs follow a different program execution paradigm – *event-driven programming*.
- A console-based program begins and ends in its `main()` method. To solve the problem, the main method statements are executed in order. After the last statement in `main()` executes, the program terminates.
- The execution of a GUI-based program also begins in its `main()` method. Normally the main method is responsible only for creating an instance of the GUI. After creating the GUI, the flow of control passes to an *event-dispatching loop* that repeatedly checks for user interactions with the GUI through *action events*. When an event occurs, an *action performer* method is invoked for each *listener* of that event. The performer then processes the interaction. After the performer handles the event, control is given again to the event-dispatching loop to watch for future interactions. This process continues until an action performer signals that the program has completed its task.



Console-based Execution

Console program

```
Method main() {
statement_1;
statement_2;
...
statement_m;
}
```

Console programs begin and end in main() method.

GUI-based Execution

GUI Program

```
main() {
   GUI gui = new GUI();
}

GUI Constructor() {
   constructor1;
   constructor2;
   ...
   constructorn;
}

Action Performer() {
  action1;
  action2;
  ...
  actionk;
}
```

GUI program being in method main(). The method creates a new instance of the GUI by invoking the GUI constructor. On completion, the event dispatching loop is started.

The constructor configures the components of the GUI. Part of the configuration is registering the listener-performer for user interactions.

The action performer implements the task of the GUI. After it completes, the event-dispatching loop is restarted.

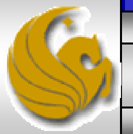
Event Dispatching Loop

```
do
   if an event occurs then
        signal its action listeners
until program ends
```

The event dispatching loop watches for user interactions with the GUI. When a user event occurs, the listener-performers for that event are notified.

GUI Program Structure

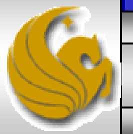
- GUI-based programs typically have at least three methods.
  - One method is the class method main() that defines an instance of the GUI.
  - The creation of that object is accomplished by invoking a constructor that creates and initializes the components of the GUI. The constructor also registers any event listeners that handle any program-specific responses to user interactions.
  - The third method is an action performer instance method that processes the events of interest. For many GUIs there is a separate listener-performer object for each of the major components of the GUI.
  - An action performer is always a **public** instance method with name `actionPerformed()`.



GUI Program Structure (cont.)

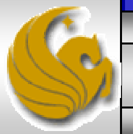
- GUI-based programs also have instance variables for representing the graphical components and the values necessary for its task.

- Thus, a GUI is a true object. Once constructed, a GUI has attributes and behaviors.
  - The attributes are the graphical component instance variables.
  - The behaviors are the actions taken by the GUI when events occur.



Specifying A GUI Layout

- When building a GUI, each GUI component must be attached to a container, such as a window created with a `JFrame`. Typically, you must also decide where to position each GUI component within the container. This is known as specifying the layout of the GUI components.
- Java provides several layout managers that can help you position components if you don't wish for a truly custom layout.
- Many IDEs provide GUI design tools in which you specify the exact size and location of each component in a visual manner using the mouse. The IDE then generates the GUI code automatically.



GridLayout Manager (cont.)

- As with all layout managers, you can call the setLayout() method of the container (or panel) and pass in a layout manager object. This is done as follows:

```
//Just use new in the method call because we don't need
//a reference to the layout manager.
Container canvas = getContentPane();
canvas.setLayout(new GridLayout(4,2));
```

- Or you could create an object and pass the object to the setLayout() method as follows:

```
//Create a layout manager object and pass it to the
//setLayout() method.
Container canvas = getContentPane();
GridLayout grid = new GridLayout(4,2);
canvas.setLayout(grid);
```

GridLayout Manager (cont.)

- The following program illustrates how you can change the layout dimension if necessary and repaint the window.
- This is accomplished by calling the JFrame's validate() method to layout current components and repaint() calls paint().
- Program `GridDemo.java` fills a frame's container with twelve buttons, and ten of the button labels are the names of cities. Only two buttons are active. When the "Show Florida Cities" button is clicked, the layout then shows the buttons with Florida cities. When the "Show Maryland Cities" button is clicked, the layout changes to show only the cities in Maryland. Each of these two dialogs also has one active button, "Show All Cities". This button toggles back to the four by three view of all the city buttons.

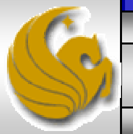
Example GUI illustrating the GridLayout Manager

```
//File:  GridDemo.java
//This program sets a 4x3 grid layout and then changes it
//to a different grid layout based on the user's choice.

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

public class GridDemo extends JFrame   {
        //Set up an array of 13 buttons.
        JButton buttonArray [] = new JButton [13];
        //Set up a String array for the button labels.
        String buttonText[] =  { "Orlando", "New York","Rock Creek", "Miami",
                "Bethesda", "Santa Fe", "Baltimore", "Oxon Hill", "Chicago",
                "Sarasota", "Show Florida Cities", "Show Maryland Cities", "Show All
                Cities" };
        Container canvas = getContentPane();
        public GridDemo()  {
                //Here's where we make our buttons and set their text.
                for(int i = 0; i<buttonArray.length; ++i)
                        buttonArray[i] = new JButton( buttonText[i] );
                addAllTheCities();
                buttonArray[10].setBackground(Color.cyan);
                buttonArray[11].setBackground(Color.magenta);
                buttonArray[12].setBackground(Color.green);
```

```
//Just going to show the Florida cities.
buttonArray[10].addActionListener( new ActionListener()   {
        public void actionPerformed( ActionEvent ae)  {
                addFloridaCities();
                //validate() causes a container to lay out its
                //components again after the components it
                //contains have been added to or modified.
                canvas.validate();
                //repaint() forces a call to paint() so that the
                //window is repainted.
                canvas.repaint();
        }
} );

//Just going to show the Maryland cities.
buttonArray[11].addActionListener( new ActionListener()   {
        public void actionPerformed( ActionEvent ae)  {
                addMarylandCities();
                //validate() causes a container to lay out its
                //components again after the components it
                //contains have been added to or modified.
                canvas.validate();
                //repaint() forces a call to paint() so that the
                //window is repainted.
                canvas.repaint();
        }
} );
```

```
        //Now show all the cities.
        buttonArray[12].addActionListener( new ActionListener()  {
                public void actionPerformed( ActionEvent ae)  {
                        addAllTheCities();
                        canvas.validate();
                        canvas.repaint();
                }
        } );
        this.setSize(500,150);
        this.setTitle("Grid Layout Demonstration Program ");
        this.show();
}

//Sets the container's canvas to 4x3 and adds all cities.
public void addAllTheCities()  {
        canvas.removeAll();
        canvas.setLayout(new GridLayout(4, 3));
        for(int i = 0; i < 12; ++i)  {
            canvas.add(buttonArray[i]);
        }
}
```

```
    //Sets the container's canvas to 2x2 and adds Florida cities.
    public void addFloridaCities()  {
            canvas.removeAll();
            canvas.setLayout(new GridLayout(2, 2));
            canvas.add(buttonArray[0]);
            canvas.add(buttonArray[3]);
            canvas.add(buttonArray[9]);
            canvas.add(buttonArray[12]);
    }
    //Sets the container's canvas to 3x2 and adds Maryland cities.
    public void addMarylandCities()     {
            canvas.removeAll();
            canvas.setLayout(new GridLayout(3, 2));
            canvas.add(buttonArray[2]);
            canvas.add(buttonArray[4]);
            canvas.add(buttonArray[6]);
            canvas.add(buttonArray[7]);
            canvas.add(buttonArray[12]);
    }


    public static void main( String args[] )
    {
            GridDemo app = new GridDemo();
            app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Output from execution of the GridDemo Example

Grid Layout Demonstration Program

Orlando	New York	Rock Creek
Miami	Bethesda	Santa Fe
Baltimore	Oxon Hill	Chicago
Sarasota	Show Florida Cities	Show Maryland Cities

Initial window

Grid Layout Demonstration Program

Orlando	Miami
Sarasota	Show All Cities

Resized grid after clicking “Show Florida Cities” button

Grid Layout Demonstration Program

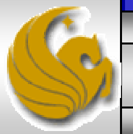
Rock Creek	Bethesda
Baltimore	Oxon Hill
Show All Cities	

Resized grid after clicking “Show Maryland Cities” button

More on Event Handling

- The previous example illustrates the basic concepts in event handling.
- Before an application can respond to an event for a particular GUI component, you must perform several coding steps:

1. Create a class than represents the event handler.
2. Implement an appropriate interface, known as an event-listener interface, in the class from Step 1.
3. Indicate than an object of the class from Steps 1 and 2 should be notified when the event occurs. This is known as registering the event handler.



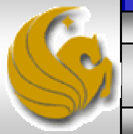
Using A Nested Class to Implement an Event Handler

- The examples so far have all utilized only top-level classes, i.e., the classes were not nested inside another class.
- Java allows for nested classes (a class declared inside another class) to be either static or non-static.
- Non-static nested classes are called inner classes and are frequently used for event handling.
  - An inner class is allowed to directly access its top-level class's variables and methods, even if they are private.
- Before an object of an inner class can be created, there must first be an object of the top-level class that contains the inner class. This is required because an inner class object implicitly has a reference to an object of its top-level class.
  - A nested class that is static does not require an object of its top-level class and has no implicit reference to an object in the top-level class.

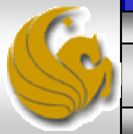
Example GUI illustrating Nested Classes

```
// Demonstrating the JTextField class and nested classes
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JPasswordField;
import javax.swing.JOptionPane;

public class TextFieldFrame extends JFrame
{
   private JTextField textField1; // text field with set size
   private JTextField textField2; // text field constructed with text
   private JTextField textField3; // text field with text and size
   private JPasswordField passwordField; // password field with text

   // TextFieldFrame constructor adds JTextFields to JFrame
   public TextFieldFrame()
   {
      super( "Testing JTextField and JPasswordField" );
      setLayout( new FlowLayout() ); // set frame layout

      // construct textfield with 10 columns
      textField1 = new JTextField( 10 );
      add( textField1 ); // add textField1 to JFrame
```

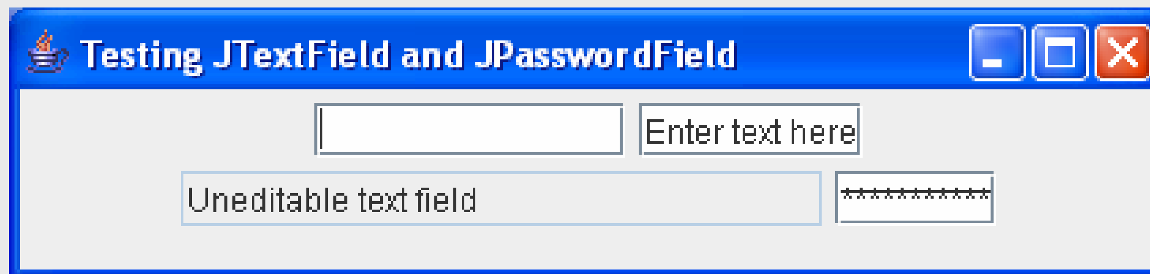
```
   // construct textfield with default text
   textField2 = new JTextField( "Enter text here" );
   add( textField2 ); // add textField2 to JFrame

   // construct textfield with default text and 21 columns
   textField3 = new JTextField( "Uneditable text field", 21 );
   textField3.setEditable( false ); // disable editing
   add( textField3 ); // add textField3 to JFrame

   // construct passwordfield with default text
   passwordField = new JPasswordField( "Hidden text" );
   add( passwordField ); // add passwordField to JFrame

   // register event handlers
   TextFieldHandler handler = new TextFieldHandler();
   textField1.addActionListener( handler );
   textField2.addActionListener( handler );
   textField3.addActionListener( handler );
   passwordField.addActionListener( handler );
} // end TextFieldFrame constructor
```

```java
// private inner class for event handling
private class TextFieldHandler implements ActionListener   {
  // process textfield events

  public void actionPerformed( ActionEvent event )     {
    String string = ""; // declare string to display
    // user pressed Enter in JTextField textField1
    if ( event.getSource() == textField1 )
      string = String.format( "textField1: %s",
        event.getActionCommand() );
    // user pressed Enter in JTextField textField2
    else if ( event.getSource() == textField2 )
      string = String.format( "textField2: %s",
        event.getActionCommand() );
    // user pressed Enter in JTextField textField3
    else if ( event.getSource() == textField3 )
      string = String.format( "textField3: %s",
        event.getActionCommand() );
    // user pressed Enter in JTextField passwordField
    else if ( event.getSource() == passwordField )
      string = String.format( "passwordField: %s",
        new String( passwordField.getPassword() ) );
    // display JTextField content
    JOptionPane.showMessageDialog( null, string );
  } // end method actionPerformed
} // end private inner class TextFieldHandler
} // end class TextFieldFrame
```

```java
// Driver class for testing TextFieldFrame.
import javax.swing.JFrame;

public class TextFieldTest
{
  public static void main( String args[] )
  {
    TextFieldFrame textFieldFrame = new TextFieldFrame();
    textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    textFieldFrame.setSize( 425, 100 ); // set frame size
    textFieldFrame.setVisible( true ); // display frame
  } // end main
} // end class TextFieldTest
```



Initial window when executing TextFieldTest