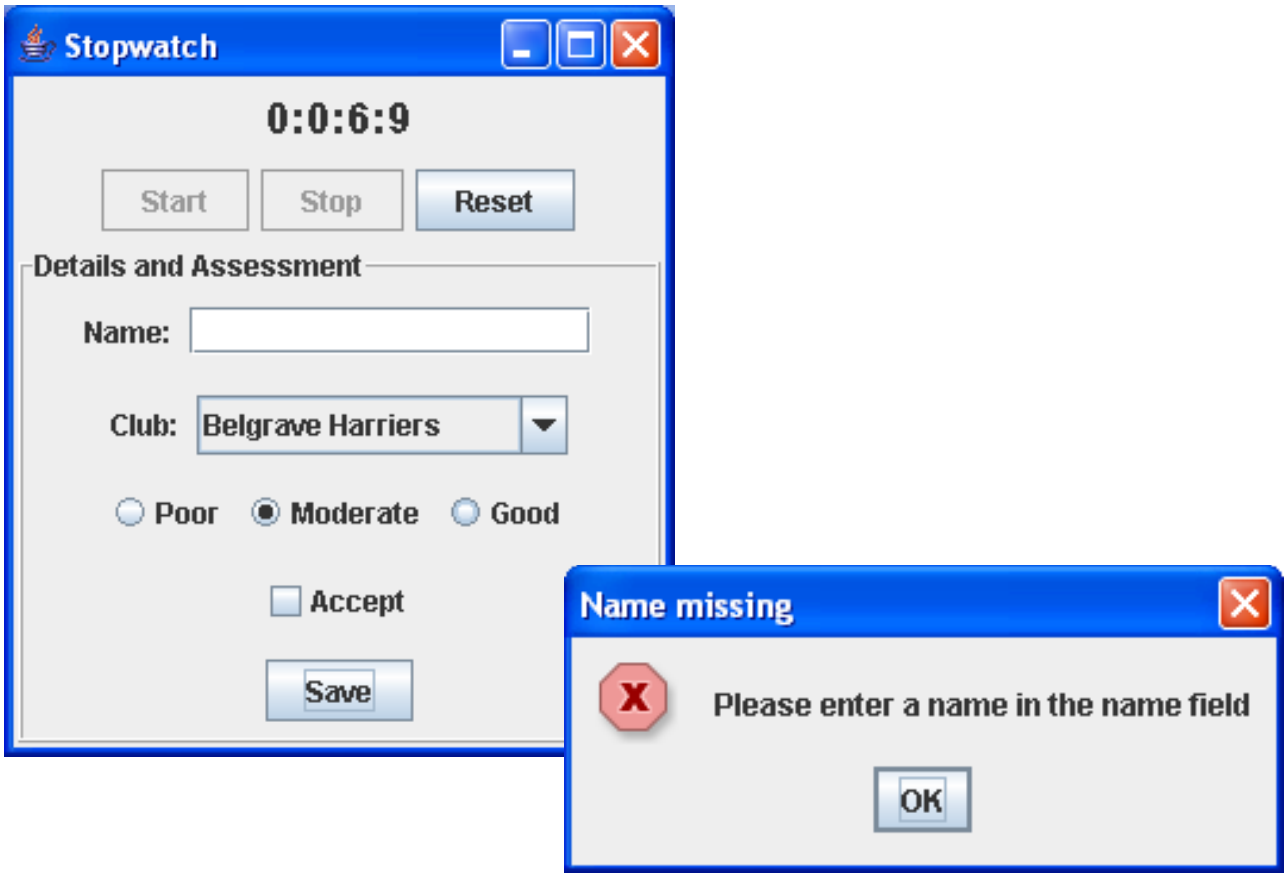
Lecture 9
Modal Dialogs, Borders, Resource management, Advanced use of text components

TableSwatch or ListSwatch error dialog



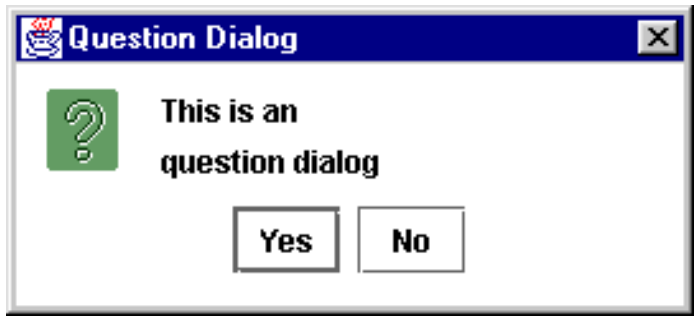
- Dialog box is popped up if the user presses Save without having entered a name

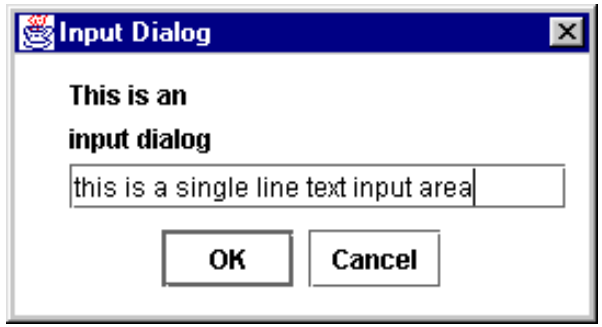
ListSwatchController code change:

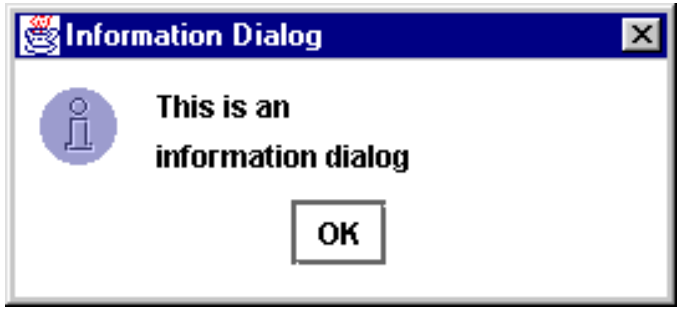
```
. . .
private final static String ENTER_NAME =
          "Please enter a name in the name field";
private final static String NAME_MISSING =
          "Name missing";
. . .
  public class SaveActionListener implements ActionListener {

    public void actionPerformed(ActionEvent event) {
        if (theState == STOPPED) {
          if (theView.getName().equals("")) {
              JOptionPane.showMessageDialog(null,
                 ENTER_NAME, NAME_MISSING,
                JOptionPane.ERROR_MESSAGE);
          } else {
          . . .
```

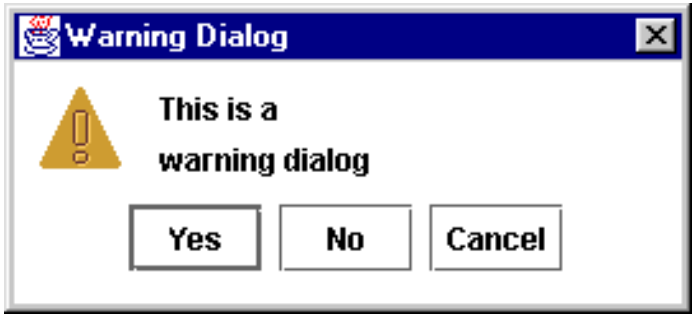
The above code change, before logging the time and data, asks *theView* if there has been anything entered in the Name field. If not, it pops up an error dialog and doesn't proceed with the logging operation.

Modal Dialogs









Modal dialogs

- When a modal dialog box pops up, the rest of the application cannot be interacted with until the user pops it down again
- The class *JOptionPane* provides static (classwide) methods for a number of standard modal dialogs as illustrated on the previous slide.
- The *error* and *information* dialogs return a void value
- The *question* and *warning* dialogs return a manifest int value depending on which button has been pressed
- The *input* dialog returns the contents of the input text field or null if the Cancel button was pressed

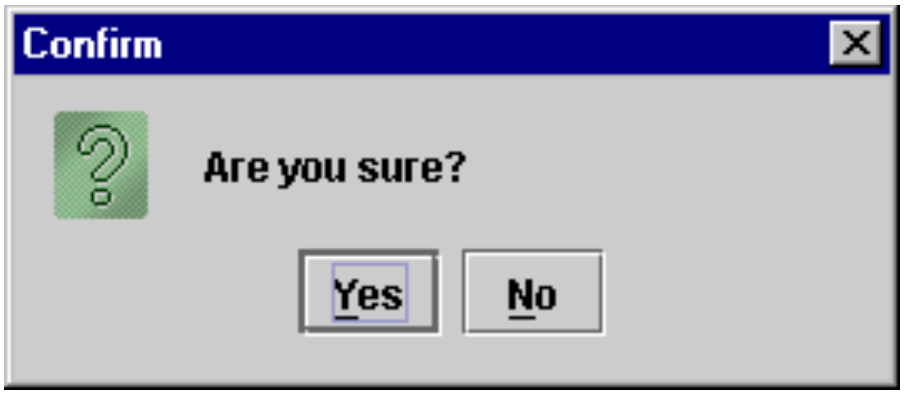
JOptionPane confirm Dialog

```
public static int showConfirmDialog(
   Component parentComponent, Object message, String title,
   int optionType)
```

Brings up a dialog where the number of choices is determined by the optionType parameter

```
. . .
int result = JOptionPane.showConfirmDialog(this,
     "Are you sure?", "Confirm",
     JOptionPane.YES_NO_OPTION);
if (result == JOptionPane.YES_OPTION) {
     //do some stuff
} else {
     //do some different stuff
}
. . . .
```

JOptionPane confirm Dialog cont'd



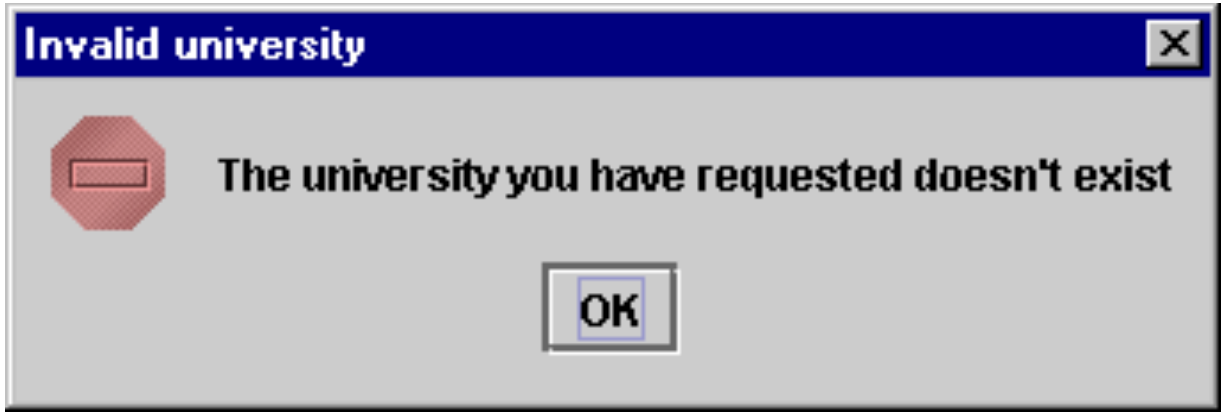
JOptionPane error dialog

```
static void showMessageDialog(Component parentComponent,
  Object message, String title,
  int messageType)
```

Brings up a dialog that displays a message using a default icon determined by the *messageType* parameter

```
. . . .
JOptionPane.showMessageDialog(this,
  "The university you have requested doesn't exist",
  "Invalid university",
  JOptionPane.ERROR_MESSAGE);
. . . .
```

JOptionPane error dialog –cont'd



Borders

- There are various classes that are used for borders, for example
  - *LineBorder, BevelBorder, EtchedBorder*
  - *etc*
- All these classes implement the interface *Border*
- Methods of the class *BorderFactory* can be used to generate instances of these different classes. The return type from these methods is *Border*
- A border can be put around a JComponent using the method

```
public void setBorder(Border border)
```

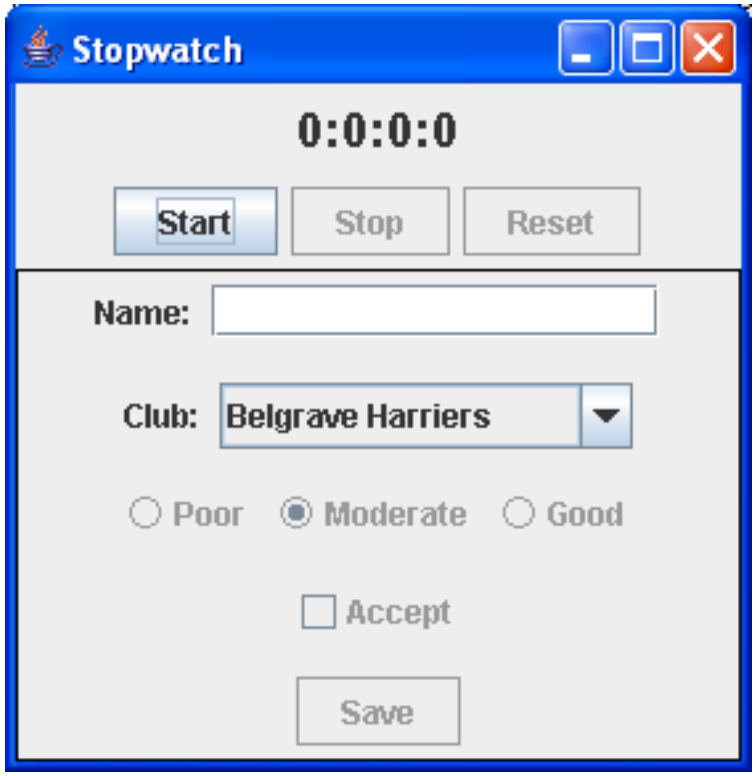
Creating borders using BorderFactory

static `Border createLineBorder(Color color)`
Creates a line border with the specified color.

static `Border createEtchedBorder(`**int** `type, Color highlight, Color shadow)`
Creates a border with an "etched" look using the specified highlighting and shading colors.

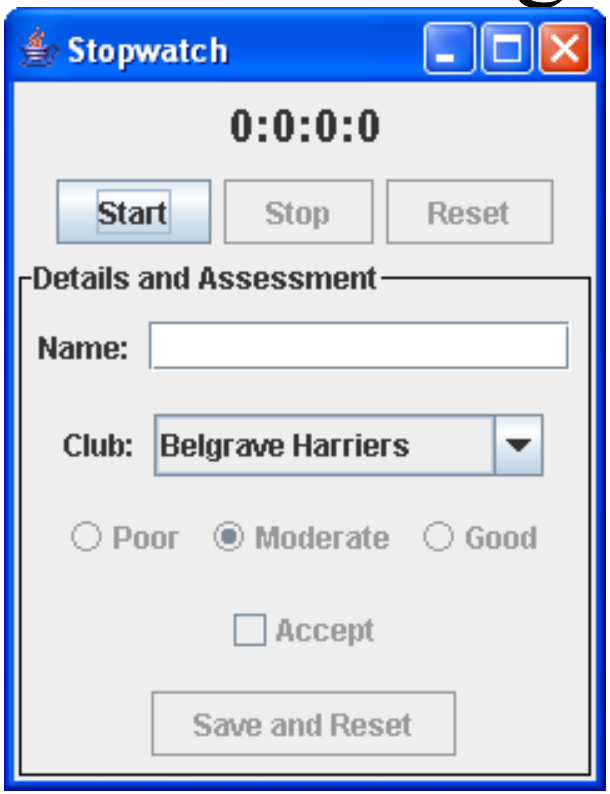
static `TitledBorder createTitledBorder(Border border, String title)`
Adds a title to an existing border, specifying the text of the title, using the default positioning (sitting on the top line) and default justification (leading) and using the default font and text color determined by the current look and feel.

ListStopwatch - using line border




```
dataPanel.setBorder(BorderFactory.createLineBorder(
                                    Color.black));
```

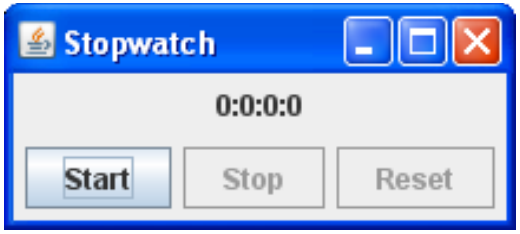
ListSwatch - using titled border

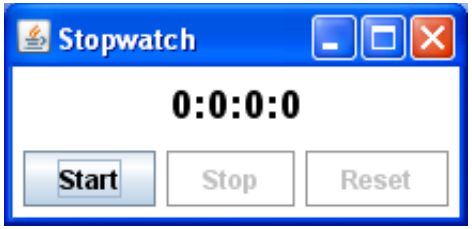


Code to be added for constructor for ListStopwatchUI:

```
dataPanel.setBorder(
     BorderFactory.createTitledBorder(
       BorderFactory.createLineBorder(
            Color.black),"Details and assessment"));
```

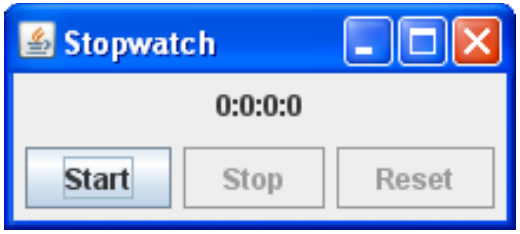
Resource Management

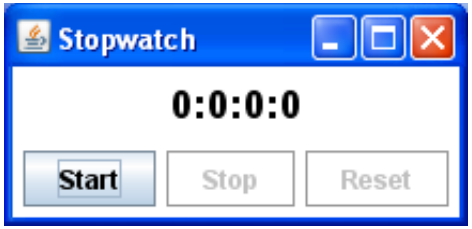




- The left-hand image shows the appearance of the *Stopwatch* artefact as would probably be produced by the implementation described. It is using the default Swing look and feel known as *Metal*.
- The right-hand image shows a revised apperance produced by the developer overriding some of the Metal resource specifications. This is accomplished by informing the UIManager object of which resources are to be set and what values they are to be set to.
- To achieve this an additional private method called *setResources()* is added to the *StopwatchView* class (and all the other view classes as developed in later lectures) and is called as part of its constructor

Resource Management





- The left-hand image shows the appearance of the *Stopwatch* artefact as would probably be produced by the implementation described. It is using the default Swing look and feel known as *Metal*.
- The right-hand image shows a revised apperance produced by the developer overriding some of the Metal resource specifications. This is accomplished by informing the UIManager object of which resources are to be set and what values they are to be set to.
- To achieve this an additional private method called *setResources()* is added to the *StopwatchView* class (and all the other view classes as developed in later lectures) and is called as part of its constructor

StopwatchView - setResources() - 1

```
private void setResources() {

    ColorUIResource defaultBackground = new
      ColorUIResource( Color.white);
    ColorUIResource defaultForeground = new
      ColorUIResource( Color.black);
    ColorUIResource disabledColor = new
      ColorUIResource( Color.lightGray);

    FontUIResource  smallFont = new FontUIResource(
             new Font( "Dialog", Font.BOLD, 12));
    FontUIResource  bigFont   = new FontUIResource(
                new Font( "Dialog", Font.BOLD, 18));
```

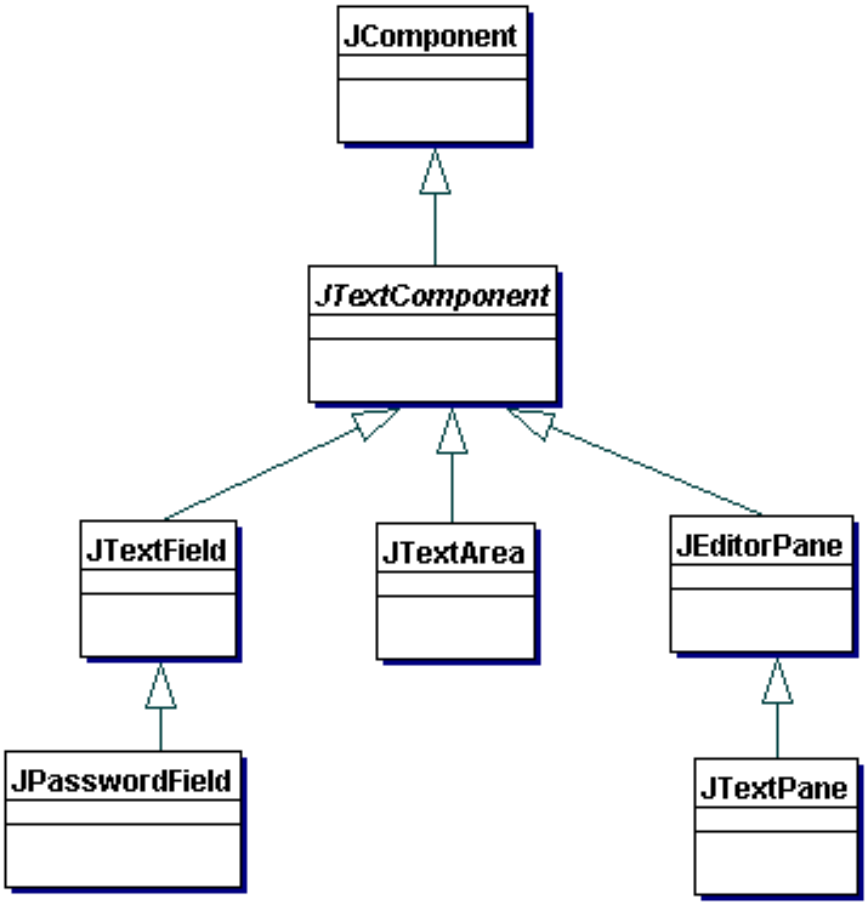
- The method declares five resources, three of the ColorUIResource class and two of the FontUIResource class. These classes are supplied by the javax.swing.plaf package and their constructors take an appropriate object from the AWT Color and Font classes.

StopwatchView - setResources() - 2

```
    UIManager.put( "Button.background", defaultBackground);
    UIManager.put( "Button.foreground", defaultForeground);
    UIManager.put( "Button.disabledText", disabledColor);
    UIManager.put( "Button.font", smallFont);
    UIManager.put( "Label.background", defaultBackground);
    UIManager.put( "Label.foreground", defaultForeground);
    UIManager.put( "Label.font", bigFont);
    UIManager.put( "Panel.background", defaultBackground);
    UIManager.put( "Panel.foreground", defaultForeground);
}//end setResources()
```

- The Swing UIManager *put()* method takes two arguments, the first is the name of the resource and the second is the value of the resource. A call of *put()* will place the name/ value pair in the user resources list.
- When widgets are constructed, resources are looked for first in the user resources list, then in the look and feel list (which can be changed) and finally in the system resources list.
- The nine user resources established by this method change the appearance of the *Stopwatch* artefact, as previously illustrated.

The JTextComponent hierarchy



JTextComponent - some methods

`Document getDocument()`

Fetches the model associated with the editor.

`Caret getCaret()`

Fetches the caret that allows text-oriented navigation over the view.

`String getSelectedText()`

Returns the selected text contained in this TextComponent.

`void setText(String t)`

Sets the text of this TextComponent to the specified text

`String getText()`

Returns the text contained in this TextComponent

JTextField method

void `addActionListener(ActionListener l)`

Adds the specified action listener to receive action events from this textfield. (Action events are generated by pressing the <ENTER> key)

Some JTextArea methods

```
void insert(String toInsert,int offset)
```

Inserts the String at the specified offset from the start of the JTextArea

```
void setLineWrap(boolean isWrap)
```

Gives the JTextArea the facility for continuing on the next line when text has filled the line.

MVC operation

- All `JTextComponents` have a three layer MVC architecture. Fortunately this complexity is (almost) fully encapsulated for simple usages. The model is an instance of a class implementing the interface `Document`
- The `Document` model will fire a `DocumentEvent` to any registered `DocumentListeners` as the text in the component changes. The identity of the Document can be obtained with the `JTextComponent`'s `getDocument()` method.

Document interface

```
void addDocumentListener(DocumentListener listener)
```

Registers the given observer to begin receiving notifications when changes are made to the document.

```
int getLength()
```

Returns number of characters of content currently in the document.

```
void remove(int offs, int len)
```

Removes a portion of the content of the document starting at the *index* offs for *len* number of characters.

- plus many other method specifications

DocumentListener interface

void changedUpdate(DocumentEvent e)

Gives notification that an attribute or set of attributes changed.

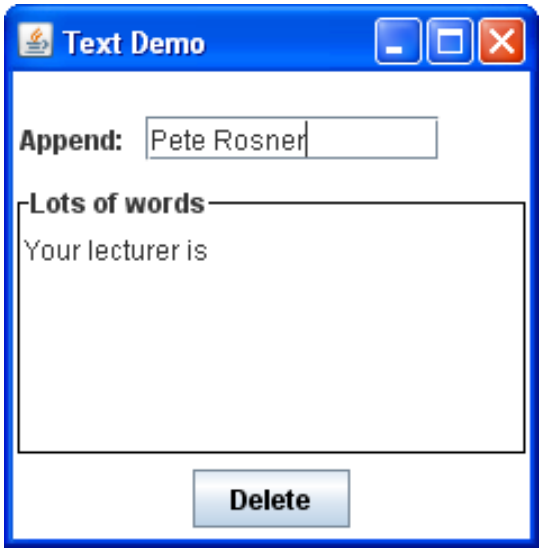
void insertUpdate(DocumentEvent e)

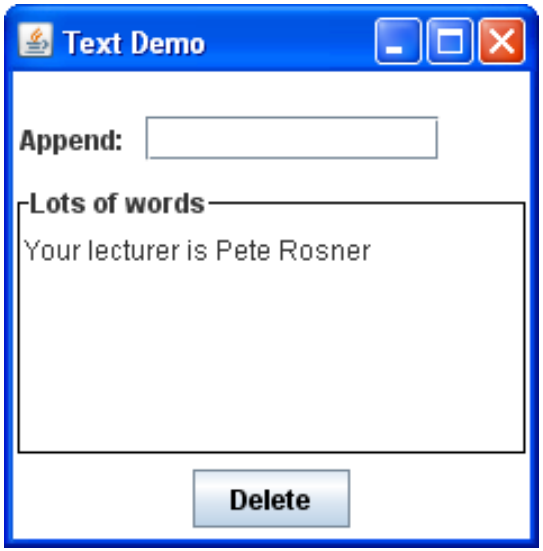
Gives notification that there was an insert into the document.

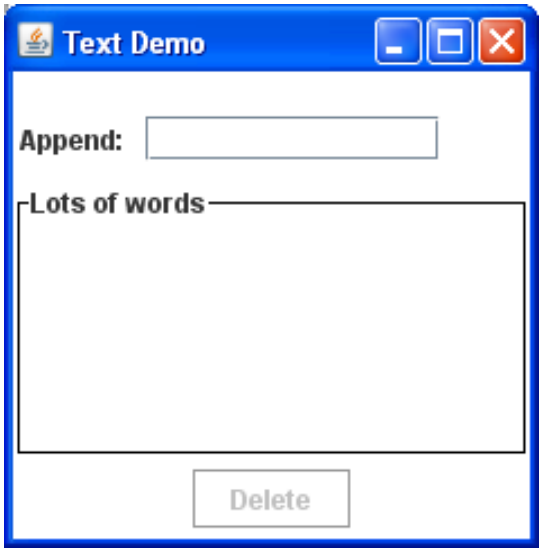
void removeUpdate(DocumentEvent e)

Gives notification that a portion of the document has been removed.

TextDemo

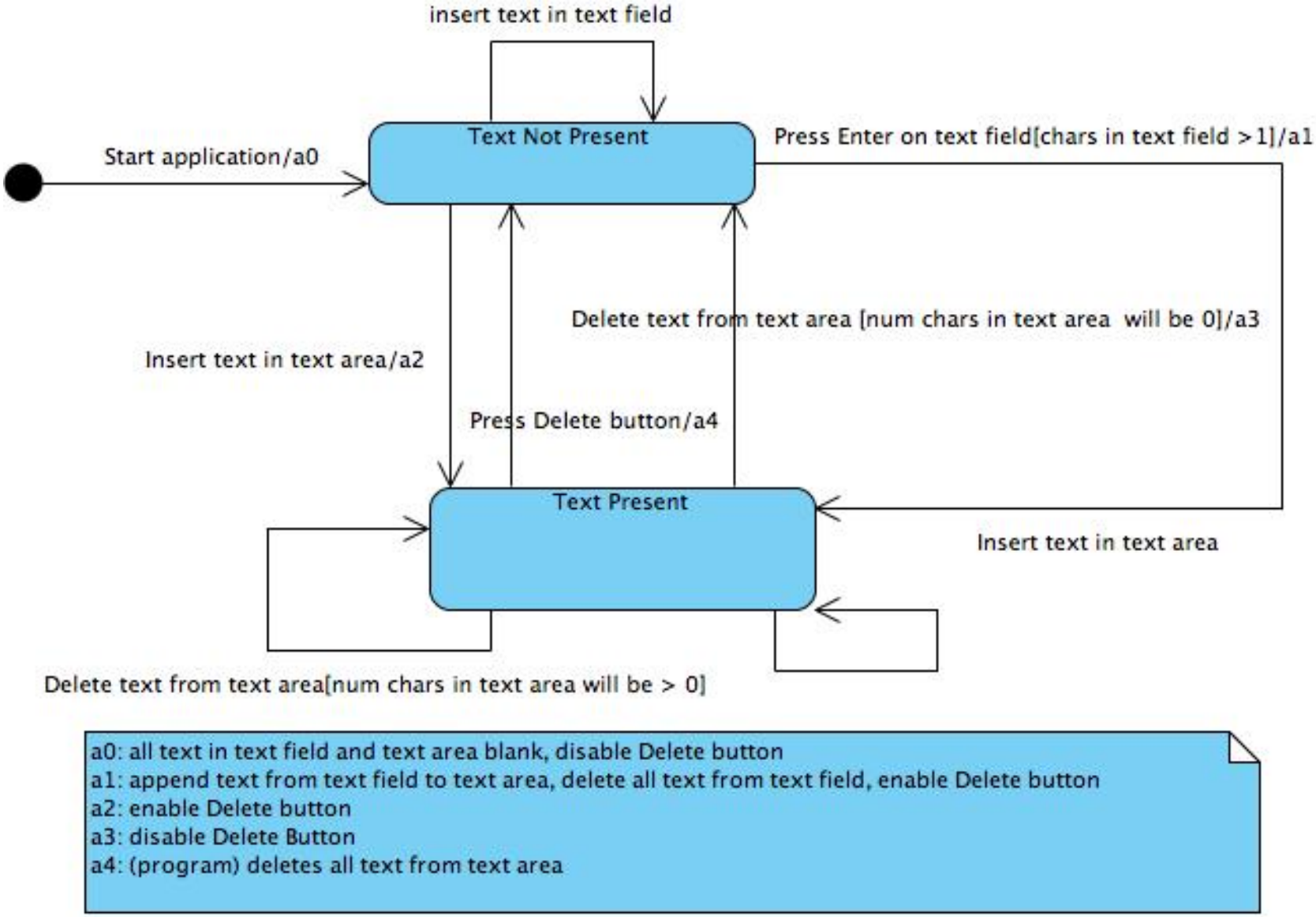




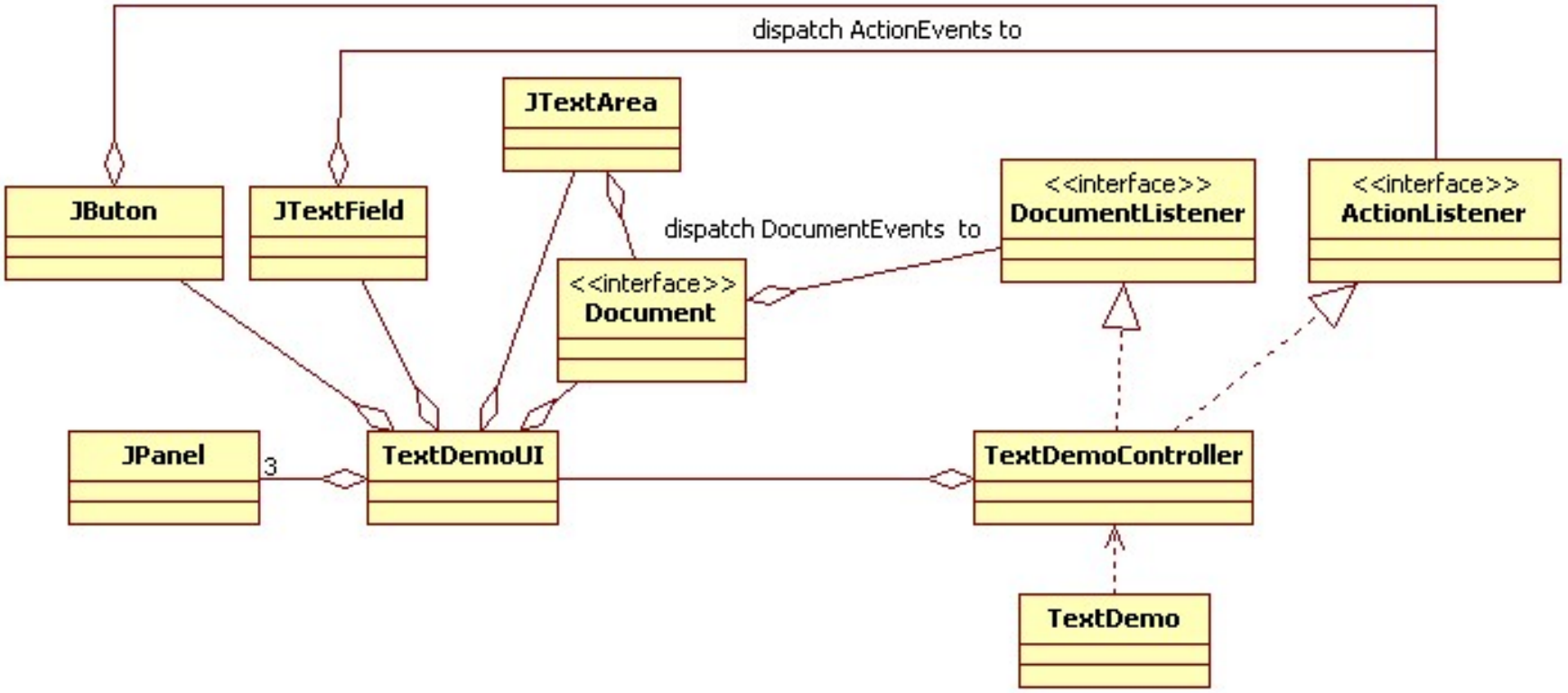


Press Enter key on field **Press Delete**

State Diagram



TextDemo class diagram



TextDemoUI - header

```
public class TextDemoUI extends JPanel {
  private JTextArea lotsOfWords = null;
  private JTextField appendField = null;
  private JLabel appendLabel = null;
  private JButton deleteButton = null;
  private Document doc = null; //Document Model of lotsOfWords JTextArea
```

TextDemoUI - constructor

```
public TextDemoUI(DocumentListener sendDocumentEventsHere,
       ActionListener sendActionEventsHere) {
   appendField = new JTextField(15);
   appendField.setActionCommand("textField");
   appendField.addActionListener(sendActionEventsHere);
   lotsOfWords = new JTextArea(6,20);
   lotsOfWords.setLineWrap(true);
   doc = lotsOfWords.getDocument();
   doc.addDocumentListener(sendDocumentEventsHere);
   . . .

   deleteButton = new JButton("Delete");
   deleteButton.addActionListener(sendActionEventsHere);
   deleteButton.setActionCommand("delete");
   deleteButton.setEnabled(false);
   . . .
}
```

ActionListener (supplied as constructor parameter) will listen for ActionEvents (Enter key pressed) from *appendField*

Document fetched from JTextArea instance

DocumentListener (supplied as constructor parameter) will listen for DocumentEvents originating from the JTextArea

ActionListener (supplied as constructor parameter) will listen for ActionEvents from *deleteButton*

textPresent(), setEnableDelete(), setDisableDelete()

- checks length of JTextArea's associated document for whether text is present

```
public boolean textPresent() {
    return doc.getLength() != 0;
}
```

- enables the Deletion

```
public void setEnableDelete(){
    deleteButton.setEnabled(true);
}
```

- disables the Deletion

```
public void setDisableDelete(){
    deleteButton.setEnabled(false);
}
```

deleteText()

```
public void deleteText() {
      try {
        doc.remove(0,doc.getLength());
      } catch (BadLocationException e) {
        throw new RuntimeException("Delete failed");
      }
  }
```

- Deletes the text in JTextArea *lotsOfWords* by removing all the text in the associated Document
- Needs to be placed in try/catch loop

copyText()

```
public void copyText() {
    lotsOfWords.insert(appendField.getText(),doc.getLength());
    appendField.setText("");
}
```

- Takes the content of the JTextField *appendField* and places it after the text already in the JTextArea *lotsOfWords*
- It finds out the postion of the end of the text in JTextArea *lotsOfWords* by querying the associated Document.

TextDemoController header and constructor

```
public class TextDemoController implements ActionListener,
                                            DocumentListener {

  private TextDemoUI ui = null;

  private static int TEXT_NOT_PRESENT = 0;
  private static int TEXT_PRESENT = 1;
  private int theState = TEXT_NOT_PRESENT;

  public TextDemoController(){
    ui = new TextDemoUI(this,this);
  }
```

- By implementing ActionListener and DocumentListener this class acts as a listener for both ActionEvents and DocumentEvents
- The constructor supplies the TextDemoController object itself as both of the parameters (since it is both of the listeners) to the constructor for TextDemoUI.

TextDemoController actionPerformed()

```
public void actionPerformed(ActionEvent event) {
    String command = event.getActionCommand();
    if (command.equals("delete")) {
      ui.deleteText();
    } else if (command.equals("textField")){
      ui.copyText();
    }
 }
```

- This will be called every time the user either clicks on the Delete button or else presses Enter on the keyboard whilst the cursor focus is on the JTextField.
- For the former the *deleteText()* method of TextDemoUI is called
- For the latter the *copyText()* method of TextDemoUI is called

Methods implementing DocumentListener interface

```
public void insertUpdate( DocumentEvent event) {
   if (theState == TEXT_NOT_PRESENT) {
      ui.setEnableDelete();
      theState = TEXT_PRESENT;
   }
}

public void changedUpdate(DocumentEvent event) {
     //not relevant so this is dummy
}
public void removeUpdate (DocumentEvent event) {
    if (!ui.textPresent()) {
       ui.setDisableDelete();
       theState = TEXT_NOT_PRESENT;
     }
}
```

insertUpdate() or *removeUpdate()* is called every time text is changed in the TextArea *lotsOfWords* either directly by the user or by the program