

Actions and JTextComponents

- You may have noticed that JTextComponents have a set of actions associated with them...so?
- You can get these actions and use them!
- It's often convenient to load the array of actions into a Hashtable so your program can retrieve an action by name

```
private void createActionTable(JTextComponent textComponent) {
    actions = new Hashtable();
    Action[] actionsArray = textComponent.getActions();
    for (int i = 0; i < actionsArray.length; i++) {
        Action a = actionsArray[i];
        actions.put(a.getValue(Action.NAME), a);
    }
}
```

- A method for retrieving an action by its name from the hashtable:

```
private Action getActionByName(String name) {
    return (Action) (actions.get(name));
}
```

More an Actions & JTextComponents

- Let's look at how a “Cut” menu item is created and associated to the action of removing text from the text component:

```
JButton button = new  
    JButton(getActionByName(DefaultEditorKit.cutAction));
```

- This code gets the action by name using the method described previously and adds the action to the menu.
- That's all you need to do. The menu and the action take care of everything else.
- Note that the cutAction comes from DefaultEditorKit

Action implications

- For performance and efficiency reasons, text components share actions. The Action object returned by `getActionByName(DefaultEditorKit.cutAction)` is shared by multiple components. This has two important ramifications:
 - Generally speaking, you shouldn't modify Action objects you get from editor kits. If you do, the changes affects all text components in your program.
 - Action objects can operate on other text components in the program, perhaps more than you intended. For example, an uneditable `JTextArea` shares actions with an editable `JTextPane`.
 - If you don't want to share, consider instantiating the Action object yourself. `DefaultEditorKit` defines a number of useful Action subclasses.

Keymaps

- The set of operations that an editor can perform is fixed and determined by the editor itself.
- Every text component has one or more keymaps-- each of which is an instance of the Keymap class.
- A keymap contains a collection of name-value pairs where the name is a KeyStroke and the value is an Action.
- Each pair binds the keystroke to the action such that when the user types the keystroke, the action occurs.
- By default, a text component has one keymap named `JTextComponent.DEFAULT_KEYMAP`.
- The mapping of keystrokes to actions depends on the look-and-feel in use, so it is not built into the editors. Instead, each text component's look-and-feel implementation provides a set of bindings that map keystrokes to the action names declared by the editor.

More on Keymaps

- JTextComponent takes the bindings for a particular component and the actions from its editor and maps the keystrokes from one to the actions in the other, using the action name to connect them, thus creating a Keymap.
- Keymaps are hierarchical. The default keymap is at the top of the hierarchy and provides fallback mappings that can be overridden
- After the default keymap is created, the text component's look and feel class will usually add mappings of its own.
- You can enhance or modify the default keymap:
 - Add a custom keymap to the text component with JTextComponent's addKeymap method.
 - Add key bindings to the default keymap with Keymap's addActionForKeyStroke method.
 - Remove key bindings from the default keymap with Keymap's removeKeyStrokeBinding method.
- The default Keymap is shared among text components, so use this with caution.

Using Keymaps

- To add new key bindings to a JTextComponent

`CTRL-B` for moving the caret backward one character

- The following code adds the CTRL-B key binding to the default keymap.

```
//Get the current, default map
Keymap keymap = textPane.addKeymap("MyEmacsBindings",
                                   textPane.getKeymap());

//Ctrl-b to go backward one character
Action action = getActionByName(StyledEditorKit.backwardAction);
KeyStroke key = KeyStroke.getKeyStroke(KeyEvent.VK_B, Event.CTRL_MASK);
keymap.addActionForKeyStroke(key, action);
```

- The code first adds a keymap to the components hierarchy. The `addKeymap` method creates the keymap for you with the name and parent provided in the method call.
- Next, the code gets the backward action from the editor kit and gets a `KeyStroke` object representing the CTRL-B key sequence.
- Finally, the code adds the action and keystroke pair to the keymap, thereby binding the key to the action.

DefaultEditorKit Actions

- Each name is a static string value in DefaultEditorKit
- static String backwardAction
 - moving the caret logically backward one position.
- static String beepAction
 - to create a beep.
- static String beginAction
 - moving the caret to the beginning of the document.
- static String beginLineAction
 - moving the caret to the beginning of a line.
- static String beginParagraphAction
 - moving the caret to the beginning of a paragraph.
- static String beginWordAction
 - moving the caret to the beginning of a word.

DefaultEditorKit actions continued

- static String copyAction
 - to copy the selected region and place the contents into the system clipboard.
- static String cutAction
 - to cut the selected region and place the contents into the system clipboard.
- static String defaultKeyTypedAction
 - that is executed by default if a key typed event is received and there is no keymap entry.
- static String deleteNextCharAction
 - to delete the character of content that follows the current caret position.
- static String deletePrevCharAction
 - to delete the character of content that precedes the current caret position.

DefaultEditorKit actions continued

- static String downAction
 - moving the caret logically downward one position.
- static String endAction
 - moving the caret to the end of the document.
- static String endLineAction
 - moving the caret to the end of a line.
- static String EndOfLineStringProperty
 - When reading a document if a CRLF is encountered a property with this name is added and the value will be "\r\n".
- static String endParagraphAction
 - moving the caret to the end of a paragraph.
- static String endWordAction
 - moving the caret to the end of a word.
- static String forwardAction
 - moving the caret logically forward one position.

DefaultEditorKit actions continued

- `Static String insertBreakAction`
 - to place a line/paragraph break into the document.
- `static String insertContentAction`
 - to place content into the associated document.
- `static String insertTabAction`
 - to place a tab character into the document.
- `static String nextWordAction`
 - moving the caret to the beginning of the next word.
- `static String pageDownAction`
 - to page down vertically.
- `static String pageUpAction`
 - to page up vertically.
- `static String pasteAction`
 - to paste the contents of the system clipboard into the selected region, or before the caret if nothing is selected.

DefaultEditorKit actions continued

- static String previousWordAction
 - moving the caret to the beginning of the previous word.
- static String readOnlyAction
 - to set the editor into read-only mode.
- static String selectAllAction
 - selecting the entire document
- static String selectionBackwardAction
 - extending the selection by moving the caret logically backward one position.
- static String selectionBeginAction
 - moving the caret to the beginning of the document.
- static String selectionBeginLineAction
 - moving the caret to the beginning of a line, extending the selection.
- static String selectionBeginParagraphAction
 - moving the caret to the beginning of a paragraph, extending the selection.

DefaultEditorKit actions continued

- static String selectionBeginWordAction
 - moving the caret to the beginning of a word, extending the selection.
- static String selectionDownAction
 - moving the caret logically downward one position, extending the selection.
- static String selectionEndAction
 - moving the caret to the end of the document.
- static String selectionEndLineAction
 - moving the caret to the end of a line, extending the selection.
- static String selectionEndParagraphAction
 - moving the caret to the end of a paragraph, extending the selection.

DefaultEditorKit actions continued

- static String selectionEndWordAction
 - moving the caret to the end of a word, extending the selection.
- static String selectionForwardAction
 - extending the selection by moving the caret logically forward one position.
- static String selectionNextWordAction
 - moving the selection to the beginning of the next word, extending the selection.
- static String selectionPreviousWordAction
 - moving the selection to the beginning of the previous word, extending the selection.
- static String selectionUpAction
 - moving the caret logically upward one position, extending the selection.
- static String selectLineAction
 - selecting a line around the caret.

DefaultEditorKit actions continued

- static String selectParagraphAction
 - selecting a paragraph around the caret.
- static String selectWordAction
 - selecting a word around the caret.
- static String upAction
 - moving the caret logically upward one position.
- static String writableAction
 - to set the editor into writeable mode

Other Text components/interfaces

- Caret interface
 - DefaultCaret class implements
 - Mouse and focus handling
 - Grabs focus when mouse clicks in text component
 - Manages “Dot” and “Mark” locations for highlighting/selection
- Highlighter interface
 - DefaultHighlighter class implements
 - You can change highlight color by
 - Subclassing DefaultHighlighter
 - Use the setSelectedTextColor() and setSelectionColor() methods of JTextComponent.

JTextArea



- Multiple lines of text
 - No graphics
 - No fonts
- Similar to AWT
 - Supports internationalization
 - Supports (but isn't built in) horizontal/vertical scroll bars
 - `JTextArea textArea = new JTextArea(20, 30)`
 - `JScrollPane p = new JScrollPane(textArea)`
- Support ability to save/load text to and from a stream
- Constructors

```
JTextArea()
```

```
JTextArea(String text)
```

```
JTextArea(int rows, int columns)
```

```
JTextArea(String text, int rows, int columns)
```

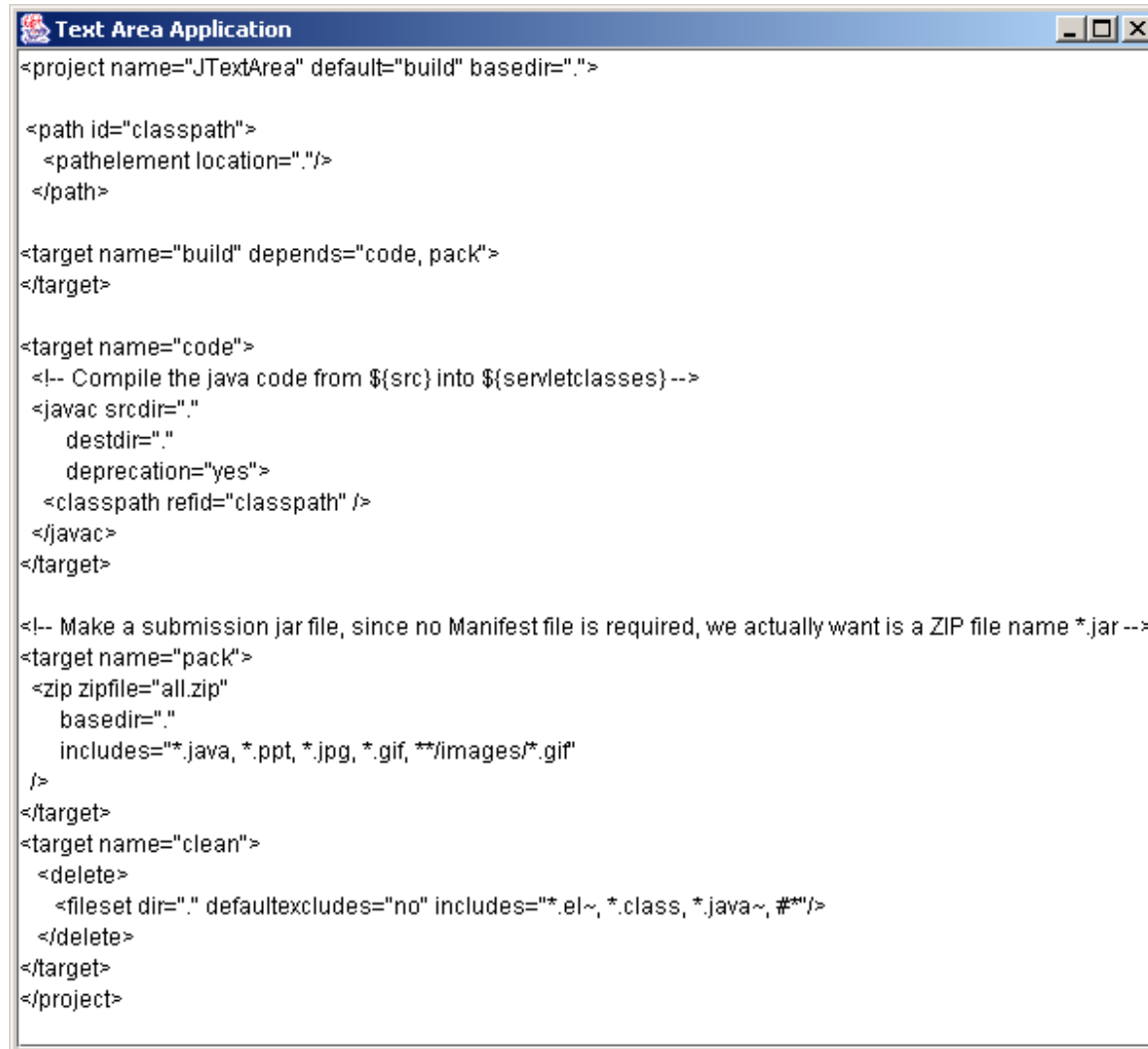
```
JTextArea(Document doc)
```

```
JTextArea(Document doc, String text, int rows, int columns)
```

JTextArea methods

- `public void setLineWrap (boolean wrap)`
 - Sets the ability to line wrap within the JTextArea
- `public void setWrapStyleWord(boolean word)`
 - Sets the ability to line wrap at word boundaries
- `public void setRows(int rows)`
 - Set the number of rows in the JTextArea
- `public void setColumns(int columns)`
 - Set the number of columns in the JTextArea
- `public void setTabSize(int columns)`
 - unlike JTextField, JTextArea will expand tabs
 - By default, tab stops are every 8 columns.
 - You can change the tab size with this.
- `public void setEditable(boolean canEdit)`
 - Enable/disable editing of this field

Code example MyTextArea.java



```
<project name="JTextArea" default="build" basedir=".">

  <path id="classpath">
    <pathelement location="."/>
  </path>

  <target name="build" depends="code, pack">
  </target>

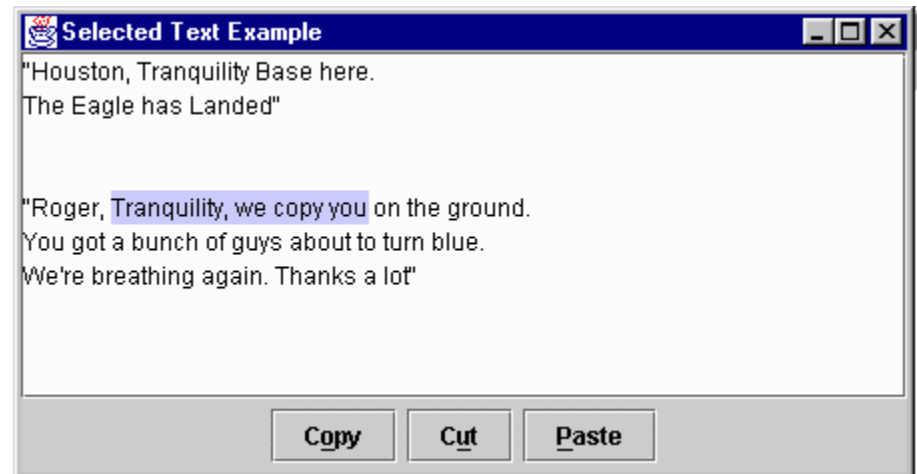
  <target name="code">
    <!-- Compile the java code from ${src} into ${servletclasses} -->
    <javac srcdir="."
      destdir="."
      deprecation="yes">
      <classpath refid="classpath" />
    </javac>
  </target>

  <!-- Make a submission jar file, since no Manifest file is required, we actually want is a ZIP file name *.jar -->
  <target name="pack">
    <zip zipfile="all.zip"
      basedir="."
      includes="*.java, *.ppt, *.jpg, *.gif, **/images/*.gif"
    />
  </target>

  <target name="clean">
    <delete>
      <fileset dir="." defaultexcludes="no" includes="*.el~, *.class, *.java~, #**"/>
    </delete>
  </target>
</project>
```

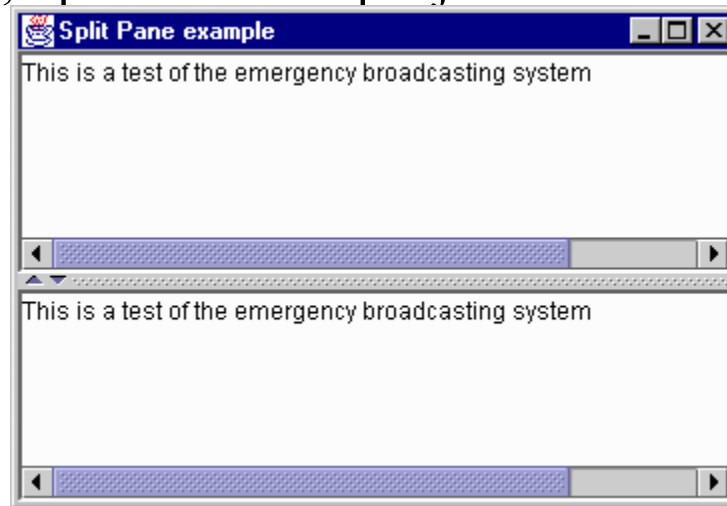
System Clipboard

- Convenience methods supplied by JTextComponent
 - public void copy()
 - public void cut()
 - public void paste()
- Movement of text to and from system clipboard
- Built in highlighting
 - Mouse selectable
 - Two clicks selects word
 - Three clicks selects line
- Code Example,
SelectedTextExample.java



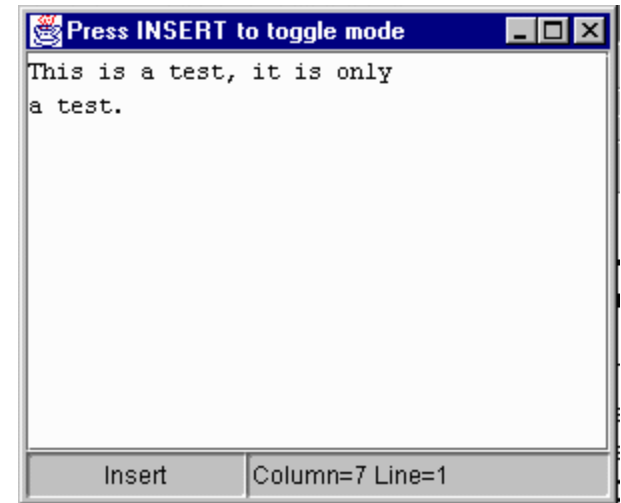
Multiple Views of a Document

- Since the view of a document is separate from the document itself, you can get multiple views of a single document
 - `secondTextArea.setDocument(firstTextArea.getDocument())`
- The document isn't aware that it is connected to multiple views
- It has no direct reference to its text component
 - changes in the document will cause the model to send the same `DocumentEvent` to the views of both components
- Code Example, `SplitPaneExample.java`

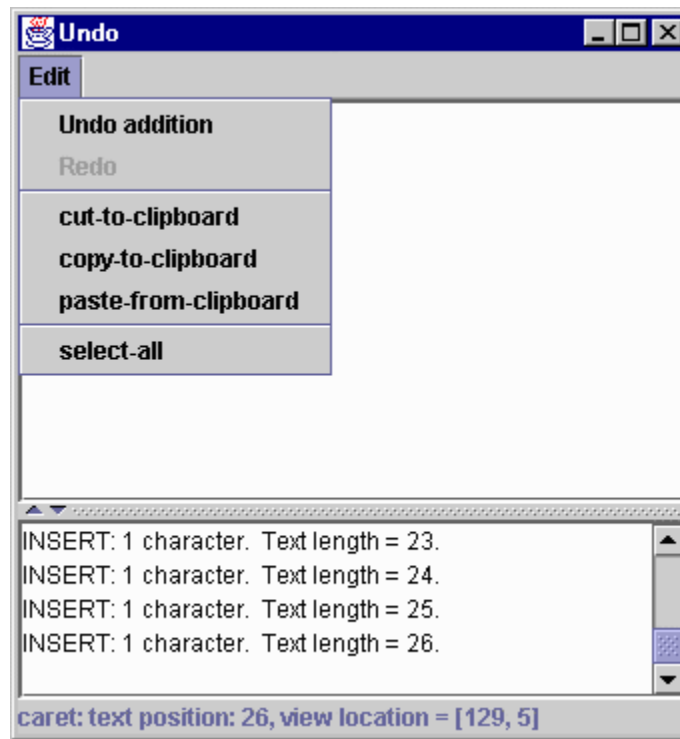


Code Example, OverwriteMode.java

- Construct a JTextArea with a customized insertString method. When in overwrite mode, insertString should replace existing characters instead of inserting them. In order to find out what mode we are in we need to be able to communicate with our custom caret.
- Attach a KeyListener to toggle modes. This is accomplished using OverwriteCaret's setMode method:
- Attach a CaretListener to track the caret's position and display it in a status bar:
- Create a Caret with an insert/overwrite flag and a customized paint method. OverwriteCaret is constructed from DefaultCaret. It paints a thicker cursor when in overwrite mode:



Undo.java



Undo and Redo

- Implementing undo/redo has two parts:
 - Remembering the undoable edits that occur.
 - Implementing the undo and redo commands and providing a user interface for them.
- A document notifies interested listeners whenever an undoable edit occurs on its content. An important step in implementing undo and redo is register an undoable edit listener on the document of the text component.

```
protected UndoManager undo = new UndoManager();  
lpd.addUndoableEditListener(new MyUndoableEditListener());
```

- The undoable edit listener used in this example adds the edit to the undo manager's list:

```
protected class MyUndoableEditListener implements UndoableEditListener {  
public void undoableEditHappened(UndoableEditEvent e) {  
    //Remember the edit and update the menus  
    undo.addEdit(e.getEdit());  
    undoAction.update();  
    redoAction.update(); } }
```

More on Undo and Redo

- The first step in this part of implementing undo and redo is to create the actions to put in the Edit menu.

```
JMenu menu = new JMenu("Edit");  
  
//Undo and redo are actions of our own creation  
undoAction = new UndoAction();  
menu.add(undoAction);  
  
redoAction = new RedoAction();  
menu.add(redoAction);  
...
```

- The undo and redo actions are implemented by custom `AbstractAction` subclasses: `UndoAction` and `RedoAction` respectively.

More on Undo and Redo

- When the user invokes the Undo command, UndoAction's actionPerformed method, shown here, gets called:

```
public void actionPerformed(ActionEvent e) {
    try {
        undo.undo();
    } catch (CannotUndoException ex) {
        System.out.println("Unable to undo: " + ex);
        ex.printStackTrace();
    }
    update();
    redoAction.update();
}
```

- Similarly, when the user invokes the Redo command, the actionPerformed method in RedoAction gets called:

```
public void actionPerformed(ActionEvent e) {
    try { undo.redo();
    } catch (CannotRedoException ex) {
        System.out.println("Unable to redo: " + ex);
        ex.printStackTrace(); }
    update();
    undoAction.update(); }
```

JScrollPane and Text Wrapping

- By default, any component derived from JTextComponent would wrap text if the horizontal space allocated to it were not enough.
- When a component is mounted in a JScrollPane it can elect to be treated two different ways
 - The component can have its width set to the width of the JScrollPane's viewport
 - The component can retain its preferred width.
- JTextComponent chooses the first of these two possibilities
- This is because the JTextComponents implement the Scrollable interface, and their getScrollableTracksViewportWidth() method always returns true (except for JEditorPane, which sometimes returns false if it is less than its minimum size)
- You can subclass and override this method
 - But if viewport is larger than the width of your panel, you get gray, dead-space where the TextComponent no longer covers.

DragTextArea.java

- Enables default Drag and Drop Support
- Note that Jlist has setDragEnabled() as well

