8

Input Fields

There are two fundamental ways for users to provide input to a program: they can type on a keyboard, or they can select something (a button, a menu item, etc.) using a mouse. When you want a user to provide input to your program, you can display a list of choices to choose from or allow the user to interact with your program by typing with the keyboard. Presenting choices to the user is covered in Chapter 9, Pick Me. As far as keyboard input goes, the java.awt package provides two options. The TextField class is a single line input field, while the TextArea class is a multiline one. Both TextField and TextArea are subclasses of the class TextComponent, which contains all the common functionality of the two. TextComponent is a subclass of Component, which is a subclass of Object. So you inherit all of these methods when you work with either TextField or TextArea.

8.1 Text Component

By themselves, the TextField and TextArea classes are fairly robust. However, in order to reduce duplication between the classes, they both inherit a number of methods from the TextComponent class. The constructor for TextComponent is package private, so you cannot create an instance of it yourself. Some of the activities shared by TextField and TextArea through the TextComponent methods include setting the text, getting the text, selecting the text, and making it read-only.
8.1.1 TextComponent Methods

Contents

Both TextField and TextArea contain a set of characters whose content determines the current value of the TextComponent. The following methods are usually called in response to an external event.

public String getText ()

The getText() method returns the current contents of the TextComponent as a String object.

public void setText (String text)

The setText() method sets the content of the TextComponent to text. If the TextComponent is a TextArea, you can embed newline characters (\n) in the text so that it will appear on multiple lines.

Text selection

Users can select text in TextComponents by pressing a mouse button at a starting point and dragging the cursor across the text. The selected text is displayed in reverse video. Only one block of text can be selected at any given time within a single TextComponent. Once selected, this block could be used to provide the user with some text-related operation such as cut and paste (on a PopupMenu).

Depending on the platform, you might or might not be able to get selected text when a TextComponent does not have the input focus. In general, the component with selected text must have input focus in order for you to retrieve any information about the selection. However, in some environments, the text remains selected when the component no longer has the input focus.

public int getSelectionStart ()

The getSelectionStart() method returns the initial position of any selected text. The position can be considered the number of characters preceding the first selected character. If there is no selected text, getSelectionStart() returns the current cursor position. If the start of the selection is at beginning of the text, the return value is 0.

public int getSelectionEnd ()

The getSelectionEnd() method returns the ending cursor position of any selected text—that is, the number of characters preceding the end of the selection. If there is no selected text, getSelectionEnd() returns the current cursor position.
public String getSelectedText ()

The getSelectedText() method returns the currently selected text of the TextComponent as a String. If nothing is selected, getSelectedText() returns an empty String, not null.

public void setSelectionStart (int position) ★

The setSelectionStart() method changes the beginning of the current selection to position. If position is after getSelectionEnd(), the cursor position moves to getSelectionEnd(), and nothing is selected.

public void setSelectionEnd (int position) ★

The setSelectionEnd() method changes the end of the current selection to position. If position is before getSelectionStart(), the cursor position moves to position, and nothing is selected.

public void select (int selectionStart, int selectionEnd)

The select() method selects the text in the TextComponent from selectionStart to selectionEnd. If selectionStart is after selectionEnd, the cursor position moves to selectionEnd. Some platforms allow you to use select() to ensure that a particular position is visible on the screen.

public void selectAll ()

The selectAll() method selects all the text in the TextComponent. It basically does a select() call with a selectionStart position of 0 and a selectionEnd position of the length of the contents.

Carets

Introduced in Java 1.1 is the ability to set and get the current insertion position within the text object.

public int getCaretPosition () ★

The getCaretPosition() method returns the current text insertion position (often called the “cursor”) of the TextComponent. You can use this position to paste text from the clipboard with the java.awt.datatransfer package described in Chapter 16, Data Transfer.

public void setCaretPosition (int position) ★

The setCaretPosition() method moves the current text insertion location of the TextComponent to position. If the TextComponent does not have a peer yet, setCaretPosition() throws the IllegalComponentStateException runtime exception. If position < 0, this method throws the run-time exception IllegalArgumentException. If position is too big, the text insertion point is positioned at the end.
Prior to Java version 1.1, the insertion location was usually set by calling 
\texttt{select(position, position)}.

\textbf{Read-only text}

By default, a \texttt{TextComponent} is editable. If a user types while the component has 
input focus, its contents will change. A \texttt{TextComponent} can also be used in an output-only (read-only) mode.

\texttt{public void setEditable(boolean state)}

The \texttt{setEditable()} method allows you to change the current editable state of 
the \texttt{TextComponent} to \texttt{state}. \texttt{true} means the component is editable; \texttt{false} 
means read-only.

\texttt{public boolean isEditable()}

The \texttt{isEditable()} method tells you if the \texttt{TextComponent} is editable (\texttt{true}) or 
read-only (\texttt{false}).

The following listing is an applet that toggles the editable status for a \texttt{TextArea} 
and sets a label to show the current status. As you can see in Figure 8-1, platforms 
can change the display characteristics of the \texttt{TextComponent} to reflect whether the 
component is editable. (Windows 95 darkens the background. Motif and Windows 
NT do nothing.)

\begin{verbatim}
import java.awt.*;
import java.applet.*;
public class readonly extends Applet {
    TextArea area;
    Label label;
    public void init () {
        setLayout (new BorderLayout (10, 10));
        add ("South", new Button ("toggleState"));
        add ("Center", area = new TextArea ("Help Me", 5, 10));
        add ("North", label = new Label ("Editable", Label.CENTER));
    }
    public boolean action (Event e, Object o) {
        if (e.target instanceof Button) {
            if ("toggleState".equals(o)) {
                area.setEditable (!area.isEditable ());
                label.setText ((area.isEditable () ? "Editable" : "Read-only"));
            } 
            return true;
        } 
        return false;
    }
}
\end{verbatim}
**Miscellaneous methods**

```java
public synchronized void removeNotify()
```

The `removeNotify()` method destroys the peer of the `TextComponent` and removes it from the screen. Prior to the `TextComponent` peer's destruction, the current state is saved so that a subsequent call to `addNotify()` will put it back. (`TextArea` and `TextField` each have their own `addNotify()` methods.) These methods deal with the peer object, which hides the native platform's implementation of the component. If you override this method for a specific `TextComponent`, put in the customizations for your new class first, and call `super.removeNotify()` last.

```java
protected String paramString()
```

When you call the `toString()` method of a `TextField` or `TextArea`, the default `toString()` method of `Component` is called. This in turn calls `paramString()`, which builds up the string to display. The `TextComponent` level potentially adds four items. The first is the current contents of the `TextComponent` (`getText()`). If the text is editable, `paramString()` adds the word `editable` to the string. The last two items included are the current selection range (`getSelectionStart()` and `getSelectionEnd()`).

### 8.1.2 TextComponent Events

With the 1.1 event model, you can register listeners for text events. A text event occurs when the component's content changes, either because the user typed something or because the program called a method like `setText()`. Listeners are...
registered with the addTextListener() method. When the content changes, the TextListener.textValueChanged() method is called through the protected method processTextEvent(). There is no equivalent to TextEvent in Java 1.0; you would have to direct keyboard changes and all programmatic changes to a common method yourself.

In addition to TextEvent listeners, Key, mouse, and focus listeners are registered through the Component methods addKeyListener(), addMouseListener(), addMouseMotionListener(), and addFocusListener(), respectively.

Listeners and 1.1 event handling

The addTextListener() method registers listener as an object interested in receiving notifications when a TextEvent passes through the EventQueue with this TEXTComponent as its target. The listener.textValueChanged() method is called when these events occur. Multiple listeners can be registered.

The following applet, text13, demonstrates how to use a TextListener to handle the events that occur when a TextField is changed. Whenever the user types into the TextField, a TextEvent is delivered to the textValueChanged() method, which prints a message on the Java console. The applet includes a button that, when pressed, modifies the text field tf by calling setText(). These changes also generate a TextEvent.

```java
// Java 1.1 only
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
class TextFieldSetter implements ActionListener {
    TextField tf;
    TextFieldSetter (TextField tf) {
        this.tf = tf;
    }
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("Set")) {
            tf.setText ("Hello");
        }
    }
}

class text13 extends Applet implements TextListener {
    TextField tf;
    int i=0;
    public void init () {
        Button b;
        tf = new TextField ("Help Text", 20);
        add (tf);
        tf.addTextListener (this);
        add (b = new Button ("Set"));
        b.addActionListener (new TextFieldSetter (tf));
    }
}```
public void textValueChanged(TextEvent e) {
    System.out.println (++i + " : " + e);
}

public void removeTextListener(TextListener listener) ★

The removeTextListener() method removes listener as an interested listener. If listener is not registered, nothing happens.

protected void processEvent(AWTEvent e) ★

The processEvent() method receives all AWTEvents with this TextComponent as its target. processEvent() then passes the events along to any listeners for processing. When you subclass TextComponent, overriding processEvent() allows you to process all events yourself, before sending them to any listeners. In a way, overriding processEvent() is like overriding handleEvent() using the 1.0 event model.

If you override processEvent(), remember to call super.processEvent(e) last to ensure that regular event processing can occur. If you want to process your own events, it’s a good idea to call enableEvents() (inherited from Component) to ensure that events are delivered even in the absence of registered listeners.

protected void processTextEvent(TextEvent e) ★

The processTextEvent() method receives all TextEvents with this TextComponent as its target. processTextEvent() then passes them along to any listeners for processing. When you subclass TextField or TextArea, overriding the processTextEvent() method allows you to process all text events yourself, before sending them to any listeners. There is no equivalent to processTextEvent() within the 1.0 event model.

If you override processTextEvent(), remember to call the method super.processTextEvent(e) last to ensure that regular event processing can occur. If you want to process your own events, it’s a good idea to call enableEvents() (inherited from Component) to ensure that events are delivered even in the absence of registered listeners.

8.2 TextField

TextField is the TextComponent for single-line input. Some constructors permit you to set the width of the TextField on the screen, but the current LayoutManager may change it. The text in the TextField is left justified, and the justification is not customizable. To change the font and size of text within the TextField, call setFont() as shown in Chapter 3, Fonts and Colors.
The width of the field does not limit the number of characters that the user can type into the field. It merely suggests how wide the field should be. To limit the number of characters, it is necessary to override the `keydown()` method for the `Component`. Section 8.4 contains an example showing how to do this.

### 8.2.1 TextField Methods

#### Constructors

- **public TextField ()**
  
  This constructor creates an empty `TextField`. The width of the `TextField` is zero columns, but it will be made wide enough to display just about one character, depending on the current font and size.

- **public TextField (int columns)**
  
  This constructor creates an empty `TextField`. The `TextField` width is `columns`. The `TextField` will try to be wide enough to display `columns` characters in the current font and size. As I mentioned previously, the layout manager may change the size.

- **public TextField (String text)**
  
  This constructor creates a `TextField` with `text` as its content. In Java 1.0 systems, the `TextField` is 0 columns wide (the `getColumns()` result), but the system will size it to fit the length of text. With Java 1.1, `getColumns()` actually returns `text.length`.

- **public TextField (String text, int columns)**
  
  This constructor creates a `TextField` with `text` as its content and a width of `columns`.

The following example uses all four constructors; the results are shown in Figure 8-2. With the third constructor, you see that the `TextField` is not quite wide enough for our text. The system uses an average width per character to try to determine how wide the field should be. If you want to be on the safe side, specify the field’s length explicitly, and add a few extra characters to ensure that there is enough room on the screen for the entire text.

```java
import java.awt.TextField;
public class texts extends java.applet.Applet {
    public void init () {
        add (new TextField ());  // A
        add (new TextField (15)); // B
        add (new TextField ("Empty String")); // C
        add (new TextField ("Empty String", 20)); // D
    }
}
```
Figure 8–2: Using the TextField constructors

**Sizing**

```java
public int getColumns ()
```

The `getColumns()` method returns the number of columns set with the constructor or a later call to `setColumns()`. This could be different from the displayed width of the `TextField`, depending upon the current `LayoutManager`.

```java
public void setColumns (int columns)
```

The `setColumns()` method changes the preferred number of columns for the `TextField` to display to `columns`. Because the current `LayoutManager` will do what it wants, the new setting may be completely ignored. If `columns < 0`, `setColumns()` throws the run-time exception `IllegalArgumentException`.

```java
public Dimension getPreferredSize (int columns)
```  
`preferredSize()` is the Java 1.0 name for this method.

```java
public Dimension getPreferredSize ()
```

The `getPreferredSize()` method returns the `Dimension` (width and height) for the preferred size of the `TextField` with a width of `columns`. The `columns` specified may be different from the number of columns designated in the constructor.

```java
public Dimension getPreferredSize ()
```

The `getPreferredSize()` method returns the `Dimension` (width and height) for the preferred size of the `TextField`. Without the `columns` parameter, this `getPreferredSize()` uses the constructor’s number of columns (or the value from a subsequent call to `setColumns()`) to calculate the `TextField`'s preferred size.
preferredSize() is the Java 1.0 name for this method.

```java
public Dimension getMinimumSize (int columns) ★
public Dimension minimumSize (int columns) ☆
```

The `getMinimumSize()` method returns the minimum Dimension (width and height) for the size of a TextField with a width of `columns`. The `columns` specified may be different from the columns designated in the constructor.

`minimumSize()` is the Java 1.0 name for this method.

```java
public Dimension getMinimumSize () ★
public Dimension minimumSize ()
```

The `getMinimumSize()` method returns the minimum Dimension (width and height) for the size of the TextField. Without the `columns` parameter, this `getMinimumSize()` uses the constructor’s number of columns (or the value from a subsequent call to `setColumns()`) to calculate the TextField’s minimum size.

`minimumSize()` is the Java 1.0 name for this method.

### Echoing character

It is possible to change the character echoed back to the user when he or she types. This is extremely useful for implementing password entry fields.

```java
public char getEchoChar ()
```

The `getEchoChar()` method returns the currently echoed character. If the TextField is echoing normally, `getEchoChar()` returns zero.

```java
public void setEchoChar (char c) ★
public void setEchoCharacter (char c) ☆
```

The `setEchoChar()` method changes the character that is displayed to the user to `c` for every character in the TextField. It is possible to change the echo character on the fly so that existing characters will be replaced. A `c` of zero, `(char)0`, effectively turns off any change and makes the TextField behave normally.

`setEchoCharacter()` is the Java 1.0 name for this method.

```java
public boolean echoCharIsSet ()
```

The `echoCharIsSet()` method returns `true` if the echo character is set to a nonzero value. If the TextField is displaying input normally, this method returns `false`. 

---

8.2 TEXTFIELD

10 July 2002 22:21
**Miscellaneous methods**

```java
public synchronized void addNotify ()
```

The `addNotify()` method creates the `TextField` peer. If you override this method, first call `super.addNotify()`, then add your customizations for the new class. Then you will be able to do everything you need with the information about the newly created peer.

```java
protected String paramString ()
```

When you call the `toString()` method of `TextField`, the default `toString()` method of `Component` is called. This in turn calls `paramString()`, which builds up the string to display. The `TextField` level can add only one item. If the echo character is nonzero, the current echo character is added (the method `getEchoChar()`). Using `new TextField ("Empty String", 20)`, the results displayed could be:

```java
java.awt.TextField[0,0,0x0,invalid,text="Empty String",editable,selection=0-0]
```

### 8.2.2 TextField Events

With the 1.0 event model, `TextField` components can generate `KEY_PRESS` and `KEY_ACTION` (which calls `keyDown()`), `KEY_RELEASE` and `KEY_ACTION_RELEASE` (which calls `keyUp()`), and `ACTION_EVENT` (which calls `action()`).

With the 1.1 event model, you register an `ActionListener` with the method `addActionListener()`. Then when the user presses Return within the `TextField` the `ActionListener.actionPerformed()` method is called through the protected `TextField.processActionEvent()` method. Key, mouse, and focus listeners are registered through the three `Component` methods of `addKeyListener()`, `addMouseListener()`, and `addFocusListener()`, respectively.

**Action**

```java
public boolean action (Event e, Object o)
```

The `action()` method for a `TextField` is called when the input focus is in the `TextField` and the user presses the Return key. `e` is the `Event` instance for the specific event, while `o` is a `String` representing the current contents (the `getText()` method).

**Keyboard**

```java
public boolean keyDown (Event e, int key)
```

The `keyDown()` method is called whenever the user presses a key. `keyDown()` may be called many times in succession if the key remains pressed. `e` is the `Event` instance for the specific event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) for `keyDown()` could
be either `Event.KEY_PRESS` for a regular key or `Event.KEY_ACTION` for an action-oriented key (i.e., an arrow or function key). Some of the things you can do through this method are validate input, convert each character to uppercase, and limit the number or type of characters entered. The technique is simple: you just need to remember that the user’s keystroke is actually displayed by the `TextField` peer, which receives the event after the `TextField` itself. Therefore, a `TextField` subclass can modify the character displayed by modifying the `key` field (`e.key`) of the `Event` and returning `false`, which passes the `Event` on down the chain; remember that returning `false` indicates that the `Event` has not been completely processed. The following method uses this technique to convert all input to uppercase.

```java
public boolean keyDown (Event e, int key) {
    e.key = Character.toUpperCase (char(key));
    return false;
}
```

If `keyDown()` returns `true`, it indicates that the `Event` has been completely processed. In this case, the `Event` never propagates to the peer, and the keystroke is never displayed.

```java
public boolean keyUp (Event e, int key)
```

The `keyUp()` method is called whenever the user releases a key. `e` is the `Event` instance for the specific event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) for `keyUp()` could be either `Event.KEY_RELEASE` for a regular key or `Event.KEY_ACTION_RELEASE` for an action-oriented key (i.e., an arrow or function key). Among other things, `keyUp()` may be used to determine how long the key has been pressed.

**Mouse**

Ordinarily, the `TextField` component does not trigger any mouse events.

**NOTE** Mouse events are not generated for `TextField` with JDK 1.0.2. Your run-time environment may behave differently. See Appendix C for more information about platform dependencies.

**Focus**

The `TextField` component does not reliably generate focus events.
NOTE

The `GOT_FOCUS` and `LOST_FOCUS` events can be generated by `TextField` objects, but these events are not reliable across platforms. With Java 1.0, they are generated on most UNIX platforms but not on Windows NT/95 platforms. They are generated on all platforms under Java 1.1. See Appendix C for more information about platform dependencies.

```java
public boolean gotFocus (Event e, Object o)

The `gotFocus()` method is triggered when the `TextField` gets the input focus. `e` is the `Event` instance for the specific event, while `o` is a `String` representation of the current contents (`getText()`).

```java
public boolean lostFocus (Event e, Object o)

The `lostFocus()` method is triggered when the input focus leaves the `TextField`. `e` is the `Event` instance for the specific event, while `o` is a `String` representation of the current contents (`getText()`).

Listeners and 1.1 event handling

With the 1.1 event model, you register event listeners that are told when an event occurs. You can register text event listeners by calling the method `TextComponent.addTextListener()`.

```
public void addActionListener(ActionListener listener)

The `addActionListener()` method registers `listener` as an object interested in receiving notifications when an `ActionEvent` passes through the `EventQueue` with this `TextField` as its target. The `listener.actionPerformed()` method is called when these events occur. Multiple listeners can be registered. The following code demonstrates how to use an `ActionListener` to reverse the text in the `TextField`.

```java
// Java 1.1 only
import java.awt.*;
import java.awt.event.*;
class MyAL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("The current text is: " + e.getActionCommand());
        if (e.getSource() instanceof TextField) {
            TextField tf = (TextField)e.getSource();
            StringBuffer sb = new StringBuffer(e.getActionCommand());
            tf.setText(sb.reverse().toString());
        }
    }
}
```
public class text11 extends Applet {
    public void init () {
        TextField tf = new TextField("Help Text", 20);
        add(tf);
        tf.addActionListener (new MyAL());
    }
}

public void removeActionListener(ActionListener listener) ★
The removeActionListener() method removes listener as a interested listener. If listener is not registered, nothing happens.

protected void processEvent(AWTEvent e) ★
The processEvent() method receives all AWTEvents with this TextField as its target. processEvent() then passes them along to any listeners for processing. When you subclass TextField, overriding processEvent() allows you to process all events yourself, before sending them to any listeners. In a way, overriding processEvent() is like overriding handleEvent() using the 1.0 event model.

If you override processEvent(), remember to call super.processEvent(e) last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call enableEvents() (inherited from Component) to ensure that events are delivered even in the absence of registered listeners.

protected void processActionEvent(ActionEvent e) ★
The processActionEvent() method receives all ActionEvents with this TextField as its target. processActionEvent() then passes them along to any listeners for processing. When you subclass TextField, overriding the method processActionEvent() allows you to process all action events yourself, before sending them to any listeners. In a way, overriding processActionEvent() is like overriding action() using the 1.0 event model.

If you override the processActionEvent() method, remember to call super.processActionEvent(e) last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call enableEvents() (inherited from Component) to ensure that events are delivered even in the absence of registered listeners.

The following applet is equivalent to the previous example, except that it overrides processActionEvent() to receive events, eliminating the need for an ActionListener. The constructor calls enableEvents() to make sure that events are delivered, even if no listeners are registered.

// Java 1.1 only
import java.applet.*;
import java.awt.*;
```java
import java.awt.event.*;

class MyTextField extends TextField {
    public MyTextField (String s, int len) {
        super (s, len);
        enableEvents (AWTEvent.ACTION_EVENT_MASK);
    }
    protected void processActionEvent(ActionEvent e) {
        System.out.println ("The current text is: 
                          " + e.getActionCommand());
        TextField tf = (TextField)e.getSource();
        StringBuffer sb = new StringBuffer (e.getActionCommand());
        tf.setText (sb.reverse().toString());
        super.processActionEvent(e)
    }
}

public class text12 extends Applet {
    public void init () {
        TextField tf = new MyTextField ("Help Text", 20);
        add (tf);
    }
}
```

8.3 TextArea

TextArea is the TextComponent for multiline input. Some constructors permit you to set the rows and columns of the TextArea on the screen. However, the Layout-Manager may change your settings. As with TextField, the only way to limit the number of characters that a user can enter is to override the keyDown() method. The text in a TextArea appears left justified, and the justification is not customiz-able.

In Java 1.1, you can control the appearance of a TextArea scrollbar; earlier versions gave you no control over the scrollbars. When visible, the vertical scrollbar is on the right of the TextArea, and the horizontal scrollbar is on the bottom. You can remove either scrollbar with the help of several new TextArea constants; you can’t move them to another side. When the horizontal scrollbar is not present, the text wraps automatically when the user reaches the right side of the TextArea. Prior to Java 1.1, there was no way to enable word wrap.

8.3.1 TextArea Variables

Constants

The constants for TextArea are new to Java 1.1; they allow you to control the visibility and word wrap policy of a TextArea scrollbar. There is no way to listen for the events when a user scrolls a TextArea.
**public static final int SCROLLBARS_BOTH ★**
The SCROLLBARS_BOTH mode is the default for TextArea. It shows both scrollbars all the time and does no word wrap.

**public static final int SCROLLBARS_HORIZONTAL_ONLY ★**
The SCROLLBARS_HORIZONTAL_ONLY mode displays a scrollbar along the bottom of the TextArea. When this scrollbar is present, word wrap is disabled.

**public static final int SCROLLBARS_NONE ★**
The SCROLLBARS_NONE mode displays no scrollbars around the TextArea and enables word wrap. If the text is too long, the TextArea displays the lines surrounding the cursor. You can use the cursor to move up and down within the TextArea, but you cannot use a scrollbar to navigate. Because this mode has no horizontal scrollbar, word wrap is enabled.

**public static final int SCROLLBARS_VERTICAL_ONLY ★**
The SCROLLBARS_VERTICAL_ONLY mode displays a scrollbar along the right edge of the TextArea. If the text is too long to display, you can scroll within the area. Because this mode has no horizontal scrollbar, word wrap is enabled.

### 8.3.2 TextArea Methods

**Constructors**

**public TextArea ()**
This constructor creates an empty TextArea with both scrollbars. The TextArea is 0 rows high and 0 columns wide. Depending upon the platform, the TextArea could be really small (and useless) or rather large. It is a good idea to use one of the other constructors to control the size of the TextArea.

**public TextArea (int rows, int columns)**
This constructor creates an empty TextArea with both scrollbars. The TextArea is rows high and columns wide.

**public TextArea (String text)**
This constructor creates a TextArea with an initial content of text and both scrollbars. The TextArea is 0 rows high and 0 columns wide. Depending upon the platform, the TextArea could be really small (and useless) or rather large. It is a good idea to use one of the other constructors to control the size of the TextArea.

**public TextArea (String text, int rows, int columns)**
This constructor creates a TextArea with an initial content of text. The TextArea is rows high and columns wide and has both scrollbars.

The following example uses the first four constructors. The results are shown in Figure 8-3. With the size-less constructors, notice that Windows 95 creates a rather
large TextArea. UNIX systems create a much smaller area. Depending upon the
LayoutManager, the TextAreas could be resized automatically.

```java
import java.awt.TextArea;
public class textas extends java.applet.Applet {
    public void init () {
        add (new TextArea ()); // A
        add (new TextArea (3, 10)); // B
        add (new TextArea (*Empty Area*)); // C
        add (new TextArea (*Empty Area*, 3, 10)); // D
    }
}
```

Figure 8–3: TextArea constructor

public TextArea (String text, int rows, int columns, int scrollbarPolicy) ★

The final constructor creates a TextArea with an initial content of text. The
TextArea is rows high and columns wide. The initial scrollbar display policy is
designated by the scrollbarPolicy parameter and is one of the TextArea con-
stants in the previous example. This constructor is the only way provided to
change the scrollbar visibility; there is no setScrollbarVisibility() method.
Figure 8-4 displays the different settings.
Setting text

The text-setting methods are usually called in response to an external event. When you handle the insertion position, you must translate it from the visual row and column to a one-dimensional position. It is easier to position the insertion point based upon the beginning, end, or current selection (getSelectionStart() and getSelectionEnd()).

```java
public void insert (String string, int position)
public void insertText (String string, int position)
```

The `insert()` method inserts string at position into the TextArea. If position is beyond the end of the TextArea, string is appended to the end of the TextArea.

`insertText()` is the Java 1.0 name for this method.

```java
public void append (String string)
public void appendText (String string)
```

The `append()` method inserts string at the end of the TextArea.

`appendText()` is the Java 1.0 name for this method.

```java
public void replaceRange (String string, int startPosition, int endPosition)
public void replaceText (String string, int startPosition, int endPosition)
```

The `replaceRange()` method replaces the text in the current TextArea from `startPosition` to `endPosition` with string. If `endPosition` is before `startPosition`, it may or may not work as expected. (For instance, on a Windows 95 platform, it works fine when the TextArea is displayed on the screen. However, when the TextArea is not showing, unexpected results happen. Other platforms may vary.) If `startPosition` is 0 and `endPosition` is the length of the contents, this method functions the same as `TextComponent.setText()`.
replaceText() is the Java 1.0 name for this method.

**Sizing**

```java
public int getRows ()
```

The `getRows()` method returns the number of rows set by the constructor or a subsequent call to `setRows()`. This could be different from the displayed height of the `TextArea`.

```java
public void setRows (int rows) ★
```

The `setRows()` method changes the preferred number of rows to display for the `TextField` to `rows`. Because the current `LayoutManager` will do what it wants, the new setting may be ignored. If `rows < 0`, `setRows()` throws the run-time exception `IllegalArgumentException`.

```java
public int getColumns ()
```

The `getColumns()` method returns the number of columns set by the constructor or a subsequent call to `setColumns()`. This could be different from the displayed width of the `TextArea`.

```java
public void setColumns (int columns) ★
```

The `setColumns()` method changes the preferred number of columns to display for the `TextArea` to `columns`. Because the current `LayoutManager` will do what it wants, the new setting may be ignored. If `columns < 0`, `setColumns()` throws the run-time exception `IllegalArgumentException`.

```java
public Dimension getPreferredSize (int rows, int columns)
```

```java
public Dimension preferredSize (int rows, int columns) ★
```

The `getPreferredSize()` method returns the `Dimension` (width and height) for the preferred size of the `TextArea` with a preferred height of `rows` and width of `columns`. The `rows` and `columns` specified may be different from the current settings.

`preferredSize()` is the Java 1.0 name for this method.

```java
public Dimension getPreferredSize (int rows, int columns)
```

```java
public Dimension preferredSize () ★
```

The `getPreferredSize()` method returns the `Dimension` (width and height) for the preferred size of the `TextArea`. Without the `rows` and `columns` parameters, this `getPreferredSize()` uses the constructor’s number of rows and columns to calculate the `TextArea`’s preferred size.

`preferredSize()` is the Java 1.0 name for this method.
public Dimension getMinimumSize (int rows, int columns) ★
public Dimension minimumSize (int rows, int columns) ✧
The getMinimumSize() method returns the minimum Dimension (width and height) for the size of the TextArea with a height of rows and width of columns. The rows and columns specified may be different from the current settings.

minimumSize() is the Java 1.0 name for this method.

public Dimension getMinimumSize () ★
public Dimension minimumSize () ✧
The getMinimumSize() method returns the minimum Dimension (width and height) for the size of the TextArea. Without the rows and columns parameters, this getMinimumSize() uses the current settings for rows and columns to calculate the TextArea’s minimum size.

minimumSize() is the Java 1.0 name for this method.

Miscellaneous methods

public synchronized void addNotify ()
The addNotify() method creates the TextArea peer. If you override this method, call super.addNotify() first, then add your customizations for the new class. You will then be able to do everything you need with the information about the newly created peer.

public int getScrollbarVisibility () ★
The getScrollbarVisibility() method retrieves the scrollbar visibility setting, which is set by the constructor. There is no setScrollbarVisibility() method to change the setting. The return value is one of the TextArea constants: SCROLLBARS_BOTH, SCROLLBARS_HORIZONTAL_ONLY, SCROLLBARS_NONE, or SCROLLBARS_VERTICAL_ONLY.

protected String paramString ()
When you call the toString() method of TextArea, the default toString() method of Component is called. This in turn calls paramString(), which builds up the string to display. The TextArea level adds the number of rows and columns for the TextArea, and Java 1.1 adds the scrollbar visibility policy. Using new TextArea("Empty Area", 3, 10), the results displayed could be:

java.awt.TextArea[text0,0,0,0x0,invalid,text="Empty Area", editable,selection=0-0, rows=3,columns=10, scrollbarVisibility=both]
8.3.3 **TextArea Events**

With the 1.0 event model, the TextArea component can generate `KEY_PRESS` and `KEY_ACTION` (which calls `keyDown()` along with `KEY_RELEASE` and `KEY_ACTION_RELEASE` (which called `keyUp()`). There is no `ACTION_EVENT` generated for `TextArea`.

**NOTE** The `GOT_FOCUS` and `LOST_FOCUS` events can be generated by this component but not reliably across platforms. Currently, they are generated on most UNIX platforms but not on Microsoft Windows NT/95 under Java 1.0. These events are generated under Java 1.1.

Similarly, the mouse events are not generated with JDK 1.0.2. See Appendix C for more information about platform dependencies.

With the Java 1.1 event model, there are no listeners specific to `TextArea`. You can register key, mouse, and focus listeners through the `Component` methods of `addKeyListener()`, `addMouseListener()`, and `addFocusListener()`, respectively. To register listeners for text events, call `TextComponent.addTextListener()`.

**Action**

The `TextArea` component has no way to trigger the action event, since carriage return is a valid character. You would need to put something like a `Button` on the screen to cause an action for a `TextArea`. The following `Rot13` program demonstrates this technique. The user enters text in the `TextArea` and selects the Rotate Me button to rotate the text. If the user selects Rotate Me again, it rotates again, back to the original position. Without the button, there would be no way to trigger the event. Figure 8-5 shows this example in action.

```java
import java.awt.*;

public class Rot13 extends Frame {
    TextArea ta;
    Component rotate, done;
    public Rot13 () {
        super ("Rot-13 Example");
        add ("North", new Label ("Enter Text to Rotate:"));
        ta = (TextArea)(add ("Center", new TextArea (5, 40)));
        Panel p = new Panel ();
        rotate = p.add (new Button ("Rotate Me"));
        done = p.add (new Button ("Done"));
        add ("South", p);
    }
    public static void main (String args[]) {
        Rot13 rot = new Rot13();
        rot.pack();
    }
}```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

public class TextArea323 {
    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                rot.show();
            }
        });

        public boolean handleEvent (Event e) {
            if (e.id == Event.WINDOW_DESTROY) {
                hide();
                dispose();
                System.exit (0);
                return true;
            }
            return super.handleEvent (e);
        }

        public boolean action (Event e, Object o) {
            if (e.target == rotate) {
                ta.setText (rot13Text (ta.getText()));
                return true;
            } else if (e.target == done) {
                hide();
                dispose();
                System.exit (0);
            }
            return false;
        }

        String rot13Text (String s) {
            int len = s.length();
            StringBuffer returnString = new StringBuffer (len);
            char c;
            for (int i=0;i<len;i++) {
                c = s.charAt (i);
                if (((c >= 'A') && (c <= 'M')) ||
                    ((c >= 'a') && (c <= 'm')))
                    c += 13;
                else if (((c >= 'N') && (c <= 'Z')) ||
                    ((c >= 'n') && (c <= 'z')))
                    c -= 13;
                returnString.append (c);
            }
            return returnString.toString();
        }
    }
}

8.3 TextArea

Ordinarily, the TextArea component generates all the key events.

public boolean keyDown (Event e, int key)

The keyDown() method is called whenever the user presses a key. keyDown() may be called many times in succession if the key remains pressed. e is the Event instance for the specific event, while key is the integer representation of the character pressed. The identifier for the event (e.id) for keyDown() could be either Event.KEY_PRESS for a regular key or Event.KEY_ACTION for an action-oriented key (i.e., an arrow or function key). Some of the things you can do through this method are validate input, convert each character to
uppercase, and limit the number or type of characters entered. The technique is simple: you just need to remember that the user’s keystroke is actually displayed by the TextArea peer, which receives the event after the TextArea itself. Therefore, a TextArea subclass can modify the character displayed by modifying the key field (e.key) of the Event and returning false, which passes the Event on down the chain; remember that returning false indicates that the Event has not been completely processed. The following method uses this technique to convert all alphabetic characters to the opposite case:

```java
public boolean keyDown (Event e, int key) {
    if (Character.isUpperCase ((char)key)) {
        e.key = Character.toLowerCase ((char)key);
    } else if (Character.isLowerCase ((char)key)) {
        e.key = Character.toUpperCase ((char)key);
    }
    return false;
}
```

If keyDown() returns true, it indicates that the Event has been completely processed. In this case, the Event never propagates to the peer, and the keystroke is never displayed.

```java
public boolean keyUp (Event e, int key)
```

The keyUp() method is called whenever the user releases a key. e is the Event instance for the specific event, while key is the integer representation of the character pressed. The identifier for the event (e.id) for keyUp() could be either Event.KEY_RELEASE for a regular key, or Event.KEY_ACTION_RELEASE for an action-oriented key (i.e., an arrow or function key).

**Mouse**

Ordinarily, the TextArea component does not trigger any mouse events.
NOTE Mouse events are not generated for TextArea with JDK 1.0.2. See Appendix C for more information about platform dependencies.

Focus

The TextArea component does not reliably generate focus events.

NOTE The GOT_FOCUS and LOST_FOCUS events can be generated by this component but not reliably across platforms. With the JDK, they are generated on most UNIX platforms but not on Microsoft Windows NT/95 under JDK 1.0. These events are generated with JDK 1.1. See Appendix C for more information about platform dependencies.

public boolean gotFocus (Event e, Object o)

The gotFocus() method is triggered when the TextArea gets the input focus. e is the Event instance for the specific event, while o is a String representation of the current contents (getText()).

public boolean lostFocus (Event e, Object o)

The lostFocus() method is triggered when the input focus leaves the TextArea. e is the Event instance for the specific event, while o is a String representation of the current contents (getText()).

Listeners and 1.1 event handling

There are no listeners specific to the TextArea class. You can register Key, mouse, and focus listeners through the Component methods of addKeyListener(), addMouseListener(), and addFocusListener(), respectively. Also, you register listeners for text events by calling TextComponent.addTextListener().

8.4 Extending TextField

To extend what you learned so far, Example 8-1 creates a sub-class of TextField that limits the number of characters a user can type into it. Other than the six constructors, all the work is in the keyDown() method. The entire class follows.
Example 8–1: The SizedTextField Class Limits the Number of Characters a User can Type

```java
import java.awt.*;
public class SizedTextField extends TextField {
    private int size; // size = 0 is unlimited
    public SizedTextField () {
        super (" ");
        this.size = 0;
    }
    public SizedTextField (int columns) {
        super (columns);
        this.size = 0;
    }
    public SizedTextField (int columns, int size) {
        super (columns);
        this.size = Math.max (0, size);
    }
    public SizedTextField (String text) {
        super (text);
        this.size = 0;
    }
    public SizedTextField (String text, int columns) {
        super (text, columns);
        this.size = 0;
    }
    public SizedTextField (String text, int columns, int size) {
        super (text, columns);
        this.size = Math.max (0, size);
    }
    public boolean keyDown (Event e, int key) {
        if ((e.id == Event.KEY_PRESS) && (this.size > 0) &&
            (((TextField)(e.target)).getText ().length () >= this.size)) {
            // Check for backspace / delete / tab—let these pass through
            if ((key == 127) || (key == 8) || (key == 9)) {
                return false;
            }
            return true;
        }
        return false;
    }
    protected String paramString () {
        String str = super.paramString ();
        if (size != 0) {
            str += ",size=" + size;
        }
        return str;
    }
}
```

Most of the SizedTextField class consists of constructors; you really don’t need to provide an equivalent to all the superclass’s constructors, but it’s not a bad idea.
The `keyDown()` method looks at what the user types before it reaches the screen and acts accordingly. It checks the length of the `TextField` and compares it to the maximum length. It then does another check to see if the user typed a Backspace, Delete, or Tab, all of which we want to allow: if the field has gotten too long, we want to allow the user to shorten it. We also want to allow tab under all circumstances, so that focus traversal works properly. The rest of the logic is simple:

- If the user typed Backspace, Delete, or Tab, return `false` to propagate the event.
- If the field is too long, return `true` to prevent the event from reaching the peer. This effectively ignores the character.