Java Control Fundamentals

This article continues our exploration of the Abstract Window Toolkit (AWT). It examines the standard controls defined by Java. Controls are components that allow a user to interact with your application in various ways—for example; a commonly used control is the push button.

The AWT supports the following types of controls:-

1. Labels
2. Push buttons
3. Check boxes
4. Choice lists
5. Lists
6. Scroll bars
7. Text editing
   These controls are subclasses of Component.

Adding and Removing Controls

To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling add( ), which is defined by Container. The add( ) method has several forms.

Component add(Component compObj)

Here, compObj is an instance of the control that you want to add. A reference to compObj is returned.

Once a control has been added, it will automatically be visible whenever its parent window is displayed.

Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call remove( ). This method is also defined by Container. It has this general form:

void remove(Component obj)

Here, obj is a reference to the control you want to remove. You can remove all controls by calling removeAll( ).

Responding to Controls

Except for labels, which are passive controls, all controls generate events when they are accessed by the user. For example, when the user clicks on a push button, an event is sent that identifies the push button. In general, your program simply implements the appropriate interface and then registers an event listener for each control that you need to monitor. Once a listener has been installed, events are automatically sent to it.

Labels

The easiest control to use is a label. A label is an object of type Label, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user.

Label defines the following constructors:

1. Label( )
2. Label(String str)
3. Label(String str, int how)
   The first version creates a blank label.

   The second version creates a label that contains the string specified by str. This string is left-justified.

   The third version creates a label that contains the string specified by str using the alignment specified by how. The value of how must be one of these three constants: Label.LEFT, Label.RIGHT, or Label.CENTER.

   You can set or change the text in a label by using the setText( ) method. You can obtain the current label by calling getText( ).

   These methods are shown here:

1. void setText(String str)
2. String getText( )

For setText( ), str specifies the new label. For getText( ), the current label is returned.

You can set the alignment of the string within the label by calling setAlignment( ).

To obtain the current alignment, call getAlignment( ).

The methods are as follows:

1. void setAlignment(int how)
2. int getAlignment( )
   Here, how must be one of the alignment constants shown earlier.

The following example creates three labels and adds them to an applet:-

Code:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/
public class LabelDemo extends Applet
{
        public void init()
        {
                Label one = new Label("One");
                Label two = new Label("Two");
                Label three = new Label("Three");

                // add labels to applet window

                add(one);
                add(two);
                add(three);
        }
}
```

Output:-

Using Buttons

The most widely used control is the push button. A push button is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type Button. Button defines these two constructors:

1. Button( )
2. Button(String str)

The first version creates an empty button.

The second creates a button that contains str as a label.

After a button has been created, you can set its label by calling setLabel( ). You can retrieve its label by calling getLabel( ).

These methods are as follows:

1. void setLabel(String str)
2. String getLabel( )
   Here, str becomes the new label for the button.

Handling Buttons

Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component. Each listener implements the ActionListener interface. That interface defines the actionPerformed( ) method, which is called when an event occurs. An ActionEvent object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the string that is the label of the button.

Example:

Here is an example that creates three buttons labeled “Yes,” “No,” and “Undecided.”

Each time one is pressed, a message is displayed that reports which button has been pressed. In this version, the label of the button is used to determine which button has been pressed. The label is obtained by calling the getActionCommand( ) method on the

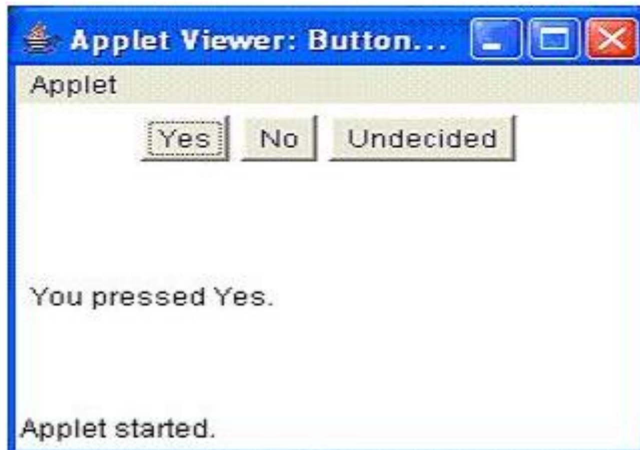
ActionEvent object passed to actionPerformed( ).

Code:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/

public class ButtonDemo extends Applet implements ActionListener
{
        String msg = "";
        Button yes, no, maybe;
```

```
	public void init()
	{
		yes = new Button("Yes");
		no = new Button("No");
		maybe = new Button("Undecided");
		add(yes);
		add(no);
		add(maybe);
		yes.addActionListener(this);
		no.addActionListener(this);
		maybe.addActionListener(this);
	}

	public void actionPerformed(ActionEvent ae)
	{
		String str = ae.getActionCommand();
		if(str.equals("Yes"))
		{
			msg = "You pressed Yes.";
		}
		else if(str.equals("No"))
		{
			msg = "You pressed No.";
		}
		else
		{
			msg = "You pressed Undecided.";
		}
		repaint();
	}

	public void paint(Graphics g)
	{
		g.drawString(msg, 6, 100);
	}
}
```

Sample output is as follows:-



Check Boxes

A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the Checkbox class.

Checkbox supports these constructors:

1. Checkbox( )
2. Checkbox(String str)
3. Checkbox(String str, boolean on)
4. Checkbox(String str, boolean on, CheckboxGroup cbGroup)
5. Checkbox(String str, CheckboxGroup cbGroup, boolean on)

The first form creates a check box whose label is initially blank. The state of the check box is unchecked.

The second form creates a check box whose label is specified by str. The state of the check box is unchecked.

The third form allows you to set the initial state of the check box. If on is true, the

check box is initially checked otherwise, it is cleared.

The fourth and fifth forms create a check box whose label is specified by str and whose group is specified by cbGroup. If this check box is not part of a group, then cbGroup must be null. (Check box groups are described soon in this article) The value of on determines the initial state of the check box.

To retrieve the current state of a check box, call getState( ). To set its state, call setState(). You can obtain the current label associated with a check box by calling getLabel( ). To set the label, call setLabel( ).

These methods are as follows:

1. boolean getState( )
2. void setState(boolean on)
3. String getLabel( )
4. void setLabel(String str)
   Here, if on is true, the box is checked. If it is false, the box is cleared. The string passed in str becomes the new label associated with the invoking check box.

Handling Check Boxes

Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the ItemListener interface. That interface defines the itemStateChanged( ) method. An ItemEvent object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or deselection).

Example:

The following program creates four check boxes. The initial state of the first box is checked. The status of each check box is displayed. Each time you change the state of a check box, the status display is updated.

Code:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CheckboxDemo" width=250 height=200>
</applet>
*/
public class CheckboxDemo extends Applet implements ItemListener
{
        String msg = "";
        Checkbox Win98, winNT, solaris, mac;

        public void init()
        {
                Win98 = new Checkbox("Windows 98/XP", null, true);
                winNT = new Checkbox("Windows NT/2000");
                solaris = new Checkbox("Solaris");
                mac = new Checkbox("MacOS");
                add(Win98);
                add(winNT);
                add(solaris);
                add(mac);
                Win98.addItemListener(this);
                winNT.addItemListener(this);
                solaris.addItemListener(this);
                mac.addItemListener(this);
        }

        public void itemStateChanged(ItemEvent ie)
        {
                repaint();
        }

        // Display current state of the check boxes.
        public void paint(Graphics g)
        {
                msg = "Current state: ";
```

```
                g.drawString(msg, 6, 80);
                msg = " Windows 98/XP: " + Win98.getState();
                g.drawString(msg, 6, 100);
                msg = " Windows NT/2000: " + winNT.getState();
                g.drawString(msg, 6, 120);
                msg = " Solaris: " + solaris.getState();
                g.drawString(msg, 6, 140);
                msg = " MacOS: " + mac.getState();
                g.drawString(msg, 6, 160);
        }
}
```

Sample output is as follows:-

CheckboxGroup

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called radio buttons, because they act like the station selector on a car radio—only one station can be selected at any one time. To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and

then specify that group when you construct the check boxes. Check box groups are objects of type CheckboxGroup. Only the default constructor is defined, which creates an empty group. You can determine which check box in a group is currently selected by calling getSelectedCheckbox( ). You can set a check box by calling setSelectedCheckbox( ).

These methods are as follows:

1. Checkbox getSelectedCheckbox( )
2. void setSelectedCheckbox(Checkbox which)

Here, which is the check box that you want to be selected. The previously selected check box will be turned off.

Example:

Here is a program that uses check boxes that are part of a group:

Code:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CBGroup" width=250 height=200>
</applet>
*/
public class CBGroup extends Applet implements ItemListener
{
        String msg = "";
        Checkbox Win98, winNT, solaris, mac;
        CheckboxGroup cbg;
        public void init()
        {
                cbg = new CheckboxGroup();
                Win98 = new Checkbox("Windows 98/XP", cbg, true);
                winNT = new Checkbox("Windows NT/2000", cbg, false);
                solaris = new Checkbox("Solaris", cbg, false);
```

```
                mac = new Checkbox("MacOS", cbg, false);
                add(Win98);
                add(winNT);
                add(solaris);
                add(mac);
                Win98.addItemListener(this);
                solaris.addItemListener(this);
                mac.addItemListener(this);
        }

        public void itemStateChanged(ItemEvent ie)
        {
                repaint();
        }
        // Display current state of the check boxes.
        public void paint(Graphics g)
        {
                msg = "Current selection: ";
                msg += cbg.getSelectedCheckbox().getLabel();
                g.drawString(msg, 6, 100);
        }
}
```

Sample output is as shown:-

Choice Controls

The Choice class is used to create a pop-up list of items from which the user may choose. Thus, a Choice control is a form of menu. When inactive, a Choice component takes up only enough space to show the currently selected item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made. Each item in the list is a string that appears as a left-justified label in the order it is added to the Choice object. Choice only defines the default constructor, which creates an empty list.

To add a selection to the list, call add( ). It has this general form:

void add(String name)

Here, name is the name of the item being added. Items are added to the list in the order in which calls to add( ) occur. To determine which item is currently selected, you may call either getSelectedItem( ) or getSelectedIndex( ).

These methods are shown here:

1. String getSelectedItem( )
2. int getSelectedIndex( )

The getSelectedItem( ) method returns a string containing the name of the item. getSelectedIndex( ) returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected. To obtain the number of items in the list, call getItemCount( ). You can set the currently selected item using the select( ) method with either a zero-based integer index or a string that will match a name in the list.

These methods are shown here:

1. int getItemCount( )
2. void select(int index)
3. void select(String name)

Given an index, you can obtain the name associated with the item at that index by calling getItem( ), which has this general form:

String getItem(int index)

Here, index specifies the index of the desired item.

Handling Choice Lists

Each time a choice is selected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the ItemListener interface. That interface defines the itemStateChanged( ) method. An ItemEvent object is supplied as the argument to this method.

Example:-

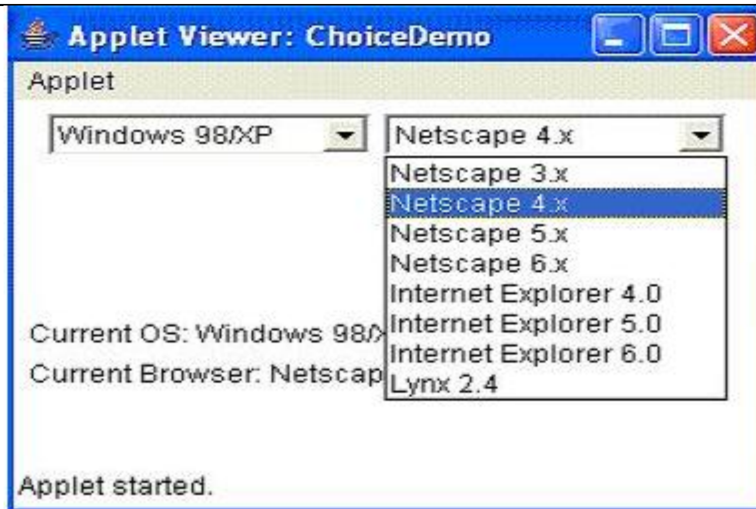
Here is an example that creates two Choice menus. One selects the operating system. The other selects the browser.

Code:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ChoiceDemo" width=300 height=180>
</applet>
*/
public class ChoiceDemo extends Applet implements ItemListener
{
        Choice os, browser;
        String msg = "";
        public void init()
        {
```

```
        os = new Choice();
        browser = new Choice();
        // add items to os list
        os.add("Windows 98/XP");
        os.add("Windows NT/2000");
        os.add("Solaris");
        os.add("MacOS");
        // add items to browser list
        browser.add("Netscape 3.x");
        browser.add("Netscape 4.x");
        browser.add("Netscape 5.x");
        browser.add("Netscape 6.x");
        browser.add("Internet Explorer 4.0");
        browser.add("Internet Explorer 5.0");
        browser.add("Internet Explorer 6.0");
        browser.add("Lynx 2.4");
        browser.select("Netscape 4.x");
        // add choice lists to window
        add(os);
        add(browser);
        // register to receive item events
        os.addItemListener(this);
        browser.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
{
        repaint();
}
// Display current selections.
public void paint(Graphics g)
{
        msg = "Current OS: ";
        msg += os.getSelectedItem();
        g.drawString(msg, 6, 120);
        msg = "Current Browser: ";
        msg += browser.getSelectedItem();
        g.drawString(msg, 6, 140);
```

```
        }
}
```



Lists

The List class provides a compact, multiple-choice, scrolling selection list. Unlike the Choice object, which shows only the single selected item in the menu, a List object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections.

List provides these constructors:

1. List( )
2. List(int numRows)
3. List(int numRows, boolean multipleSelect)

The first version creates a List control that allows only one item to be selected at any one time.

In the second form, the value of numRows specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed).

In the third form, if multipleSelect is true, then the user may select two or more items at a time. If it is false, then only one item may be selected.

To add a selection to the list, call add( ). It has the following two forms:

1. void add(String name)
2. void add(String name, int index)

Here, name is the name of the item added to the list.

The first form adds items to the end of the list.

The second form adds the item at the index specified by index. Indexing begins at zero. You can specify –1 to add the item to the end of the list.

For lists that allow only single selection, you can determine which item is currently selected by calling either getSelectedItem( ) or getSelectedIndex( ).

These methods are shown here:

1. String getSelectedItem( )
2. int getSelectedIndex( )
   The getSelectedItem( ) method returns a string containing the name of the item. If more than one item is selected or if no selection has yet been made, null is returned. getSelectedIndex( ) returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned. For lists that allow multiple selection, you must use either getSelectedItems( ) or getSelectedIndexes( ), shown here, to determine the current selections:

1. String[ ] getSelectedItems( )
2. int[ ] getSelectedIndexes( )
   getSelectedItems( ) returns an array containing the names of the currently selected items. getSelectedIndexes( ) returns an array containing the indexes of the currently selected items.

To obtain the number of items in the list, call getItemCount( ). You can set the currently selected item by using the select( ) method with a zero-based integer index.

These methods are shown here:

1. int getItemCount( )
2. void select(int index)
   Given an index, you can obtain the name associated with the item at that index by calling getItem( ), which has this general form:

String getItem(int index)

Here, index specifies the index of the desired item.

Handling Lists

To process list events, you will need to implement the ActionListener interface. Each time a List item is double-clicked, an ActionEvent object is generated. Its getActionCommand( ) method can be used to retrieve the name of the newly selected item. Also, each time an item is selected or deselected with a single click, an ItemEvent object is generated. Its getStateChange( ) method can be used to determine whether a selection or deselection triggered this event. getItemSelectable( ) returns a reference to the object that triggered this event.

Example:

Here is an example that converts the Choice controls in the preceding section into

List components, one multiple choice and the other single choice:

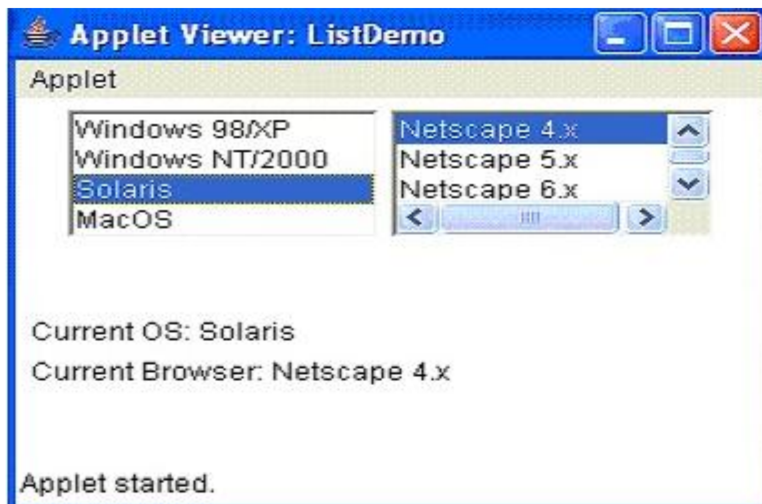
Code:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ListDemo" width=300 height=180>
</applet>
*/
public class ListDemo extends Applet implements ActionListener
{
        List os, browser;
        String msg = "";

        public void init()
```

```
{
        os = new List(4, true);
        browser = new List(4, false);
        // add items to os list
        os.add("Windows 98/XP");
        os.add("Windows NT/2000");
        os.add("Solaris");
        os.add("MacOS");
        // add items to browser list
        browser.add("Netscape 3.x");
        browser.add("Netscape 4.x");
        browser.add("Netscape 5.x");
        browser.add("Netscape 6.x");
        browser.add("Internet Explorer 4.0");
        browser.add("Internet Explorer 5.0") ;
        browser.add("Internet Explorer 6.0");
        browser.add("Lynx 2.4");
        browser.select(1);
        // add lists to window
        add(os);
        add(browser);
        // register to receive action events
        os.addActionListener(this);
        browser.addActionListener(this);
}
public void actionPerformed(ActionEvent ae)
{
        repaint();
}

// Display current selections.
public void paint(Graphics g)
{
        int idx[];
        msg = "Current OS: ";
        idx = os.getSelectedIndexes();
        for(int i=0; i<idx.length; i++)
```

```
                msg += os.getItem(idx[i]) + " ";
                g.drawString(msg, 6, 120);
                msg = "Current Browser: ";
                msg += browser.getSelectedItem();
                g.drawString(msg, 6, 140);
        }
}
```

Sample output is shown below:-



Scroll Bars

Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the slider box (or thumb) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. This action translates into some form of page up and page down.

Scroll bars are encapsulated by the Scrollbar class.

Scrollbar defines the following constructors:

1. Scrollbar( )
2. Scrollbar(int style)
3. Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
   The first form creates a vertical scroll bar.

   The second and third forms allow you to specify the orientation of the scroll bar. If style is Scrollbar.VERTICAL, a vertical scroll bar is created. If style is Scrollbar.HORIZONTAL, the scroll bar is horizontal.

   In the third form of the constructor, the initial value of the scroll bar is passed in initialValue. The number of units represented by the height of the thumb is passed in thumbSize. The minimum and maximum values for the scroll bar are specified by min and max.

   If you construct a scroll bar by using one of the first two constructors, then you need to set its parameters by using setValues( ), shown here, before it can be used:

   void setValues(int initialValue, int thumbSize, int min, int max)

   The parameters have the same meaning as they have in the third constructor just described.

   To obtain the current value of the scroll bar, call getValue( ). It returns the current setting. To set the current value, call setValue( ).

   These methods are as follows:

1. int getValue( )
2. void setValue(int newValue)
   Here, newValue specifies the new value for the scroll bar. When you set a value, the slider box inside the scroll bar will be positioned to reflect the new value.

   You can also retrieve the minimum and maximum values via getMinimum( ) and getMaximum( ), shown here:

1. int getMinimum( )
2. int getMaximum( )
   They return the requested quantity.

   By default, 1 is the increment added to or subtracted from the scroll bar each time it

is scrolled up or down one line. You can change this increment by calling setUnitIncrement( ). By default, page-up and page-down increments are 10. You can change this value by calling setBlockIncrement( ).

These methods are shown here:

1. void setUnitIncrement(int newIncr)
2. void setBlockIncrement(int newIncr)

Handling Scroll Bars

To process scroll bar events, you need to implement the AdjustmentListener interface. Each time a user interacts with a scroll bar, an AdjustmentEvent object is generated. Its getAdjustmentType( ) method can be used to determine the type of the adjustment.

The types of adjustment events are as follows:

BLOCK_DECREMENT--A page-down event has been generated.
BLOCK_INCREMENT--A page-up event has been generated.
TRACK--An absolute tracking event has been generated.
UNIT_DECREMENT--The line-down button in a scroll bar has been pressed.
UNIT_INCREMENT--The line-up button in a scroll bar has been pressed.

Example:

The following example creates both a vertical and a horizontal scroll bar. The current settings of the scroll bars are displayed. If you drag the mouse while inside the window, the coordinates of each drag event are used to update the scroll bars. An asterisk is displayed at the current drag position.

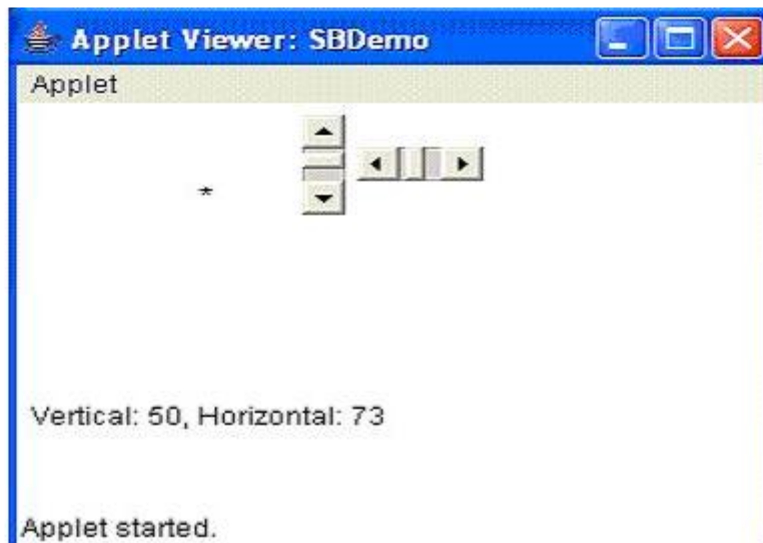
Code:

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

```
/*
<applet code="SBDemo" width=300 height=200>
</applet>
*/
public class SBDemo extends Applet implements AdjustmentListener,
MouseMotionListener
{
        String msg = "";
        Scrollbar vertSB, horzSB;
        public void init()
        {
                int width = Integer.parseInt(getParameter("width"));
                int height = Integer.parseInt(getParameter("height"));
                vertSB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, height);
                horzSB = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, width);
                add(vertSB);
                add(horzSB);
                // register to receive adjustment events
                vertSB.addAdjustmentListener(this);
                horzSB.addAdjustmentListener(this);
                addMouseMotionListener(this);
        }

        public void adjustmentValueChanged(AdjustmentEvent ae)
        {
                repaint();
        }

        // Update scroll bars to reflect mouse dragging.

        public void mouseDragged(MouseEvent me)
        {
                int x = me.getX();
                int y = me.getY();
                vertSB.setValue(y);
                horzSB.setValue(x);
                repaint();
```

```
        }

        // Necessary for MouseMotionListener
        public void mouseMoved(MouseEvent me)
        {
        }
        // Display current value of scroll bars.
        public void paint(Graphics g)
        {
                msg = "Vertical: " + vertSB.getValue();
                msg += ", Horizontal: " + horzSB.getValue();
                g.drawString(msg, 6, 160);
                // show current mouse drag position
                g.drawString("*", horzSB.getValue(),
                vertSB.getValue());
        }
}
```

Sample output is as shown below:-



TextField

The TextField class implements a single-line text-entry area, usually called an edit control. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections.

TextField is a subclass of TextComponent.

TextField defines the following constructors:

1. TextField( )
2. TextField(int numChars)
3. TextField(String str)
4. TextField(String str, int numChars)
   The first version creates a default text field.

The second form creates a text field that is numChars characters wide.

The third form initializes the text field with the string contained in str.

The fourth form initializes a text field and sets its width.

TextField (and its superclass TextComponent) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call getText( ). To set the text , call setText( ).

These methods are as follows:

1. String getText( )
2. void setText(String str)
   Here, str is the new string.

The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using select( ). A program can obtain the currently selected text by calling getSelectedText( ).

These methods are shown here:

1. String getSelectedText( )
2. void select(int startIndex, int endIndex)

getSelectedText( ) returns the selected text. The select( ) method selects the characters beginning at startIndex and ending at endIndex–1.

You can control whether the contents of a text field may be modified by the user by calling setEditable( ). You can determine editability by calling isEditable( ).

These methods are shown here:

1. boolean isEditable( )
2. void setEditable(boolean canEdit)
   isEditable( ) returns true if the text may be changed and false if not. In setEditable( ), if canEdit is true, the text may be changed. If it is false, the text cannot be altered.

There may be times when you will want the user to enter text that is not displayed, such as a password. You can disable the echoing of the characters as they are typed by calling setEchoChar( ). This method specifies a single character that the TextField will display when characters are entered (thus, the actual characters typed will not be shown). You can check a text field to see if it is in this mode with the echoCharIsSet( ) method. You can retrieve the echo character by calling the getEchoChar( ) method.

These methods are as follows:

1. void setEchoChar(char ch)
2. boolean echoCharIsSet( )
3. char getEchoChar( )
   Here, ch specifies the character to be echoed.

Handling a TextField

Since text fields perform their own editing functions, your program generally will not respond to individual key events that occur within a text field. However, you may want to respond when the user presses ENTER. When this occurs, an action event is generated.

Example:

Here is an example that creates the classic user name and password screen:

Code:

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

```
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
public class TextFieldDemo extends Appletimplements ActionListener
{
        TextField name, pass;
        public void init()
        {
                Label namep = new Label("Name: ", Label.RIGHT);
                Label passp = new Label("Password: ", Label.RIGHT);
                name = new TextField(12);
                pass = new TextField(8);
                pass.setEchoChar('?');
                add(namep);
                add(name);
                add(passp);
                add(pass);
                // register to receive action events
                name.addActionListener(this);
                pass.addActionListener(this);
        }
        // User pressed Enter.
        public void actionPerformed(ActionEvent ae)
        {
                repaint();
        }
        public void paint(Graphics g)
        {
                g.drawString("Name: " + name.getText(), 6, 60);
                g.drawString("Selected text in name: "+
name.getSelectedText(), 6, 80);
                g.drawString("Password: " + pass.getText(), 6, 100);
        }
}
```

Sample output is as shown below:-

TextArea

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called TextArea.
Following are the constructors for TextArea:

1. TextArea( )
2. TextArea(int numLines, int numChars)
3. TextArea(String str)
4. TextArea(String str, int numLines, int numChars)
5. TextArea(String str, int numLines, int numChars, int sBars)
   Here, numLines specifies the height, in lines, of the text area, and numChars specifies its width, in characters. Initial text can be specified by str.

   In the fifth form you can specify the scroll bars that you want the control to have. sBars must be one of these values:

- SCROLLBARS_BOTH
- SCROLLBARS_NONE
- SCROLLBARS_HORIZONTAL_ONLY
- SCROLLBARS_VERTICAL_ONLY

TextArea is a subclass of TextComponent. Therefore, it supports the getText( ), setText( ), getSelectedText( ), select( ), isEditable( ), and setEditable( ) methods.

TextArea adds the following methods:

1. void append(String str)
2. void insert(String str, int index)
3. void replaceRange(String str, int startIndex, int endIndex)

The append( ) method appends the string specified by str to the end of the current text. insert( ) inserts the string passed in str at the specified index. To replace text, call replaceRange( ). It replaces the characters from startIndex to endIndex–1, with the replacement text passed in str.

Text areas are almost self-contained controls. Your program incurs virtually no management overhead. Text areas only generate got-focus and lost-focus events.

Normally, your program simply obtains the current text when it is needed.

Example:

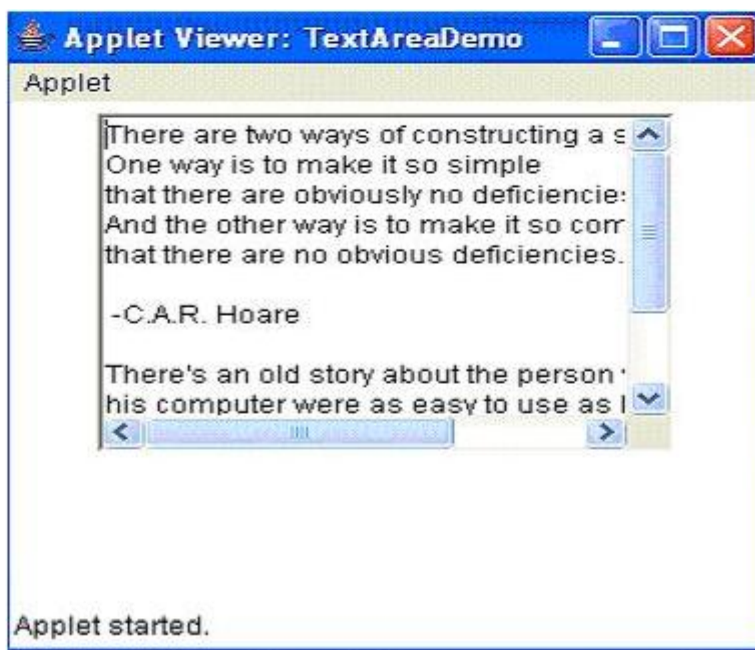
The following program creates a TextArea control:

Code:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/
public class TextAreaDemo extends Applet
{
        public void init()
        {
                String val = "There are two ways of constructing " +"a
software design.\n" +
"One way is to make it so simple\n" + "that there are obviously no
deficiencies.\n" +
"And the other way is to make it so complicated\n" +
"that there are no obvious deficiencies.\n\n" + " -C.A.R. Hoare\n\n" +
"There's an old story about the person who wished\n" +
"his computer were as easy to use as his telephone.\n" +
"That wish has come true,\n" +
"since I no longer know how to use my telephone.\n\n" +
```

```
" -Bjarne Stroustrup, AT&T, (inventor of C++)";
                TextArea text = new TextArea(val, 10, 30);
                add(text);
        }
}
```

Sample output is shown below:-



Source: http://www.go4expert.com/articles/java-control-fundamentals-t21961/