In this chapter:
- Choice
- Lists
- Checkbox
- CheckboxGroup
- ItemSelectable

Pick Me

Three AWT components let you present a list of choices to users: Choice, List, and Checkbox. All three components implement the ItemSelectable interface (Java 1.1). These components are comparable to selection mechanisms in modern GUIs so most readers will be able to learn them easily, but I’ll point out some special enhancements that they provide.

Choice and List are similar; both offer a list of choices for the user to select. Choice provides a pull-down list that offers one selection at a time, whereas List is a scrollable list that allows a user to make one or multiple selections. From a design standpoint, which you choose depends at least partially on screen real estate; if you want the user to select from a large group of alternatives, Choice requires the least space, List requires somewhat more, while Checkbox requires the most. Choice is the only component in this group that does not allow multiple selections. A List allows multiple or single selection; because each Checkbox is a separate component, checkboxes inherently allow multiple selection. In order to create a list of mutually exclusive checkboxes, in which only one box can be selected at a time (commonly known as radio buttons), you can put several checkboxes together into a CheckboxGroup, which is discussed at the end of this chapter.

9.1 Choice

The Choice component provides pop-up/pull-down lists. It is the equivalent of Motif’s OptionMenu or Windows MFC’s ComboBox. (Java 1.1 departs from the MFC world.) With the Choice component, you can provide a short list of choices to the user, while taking up the space of a single item on the screen. When the component is selected, the complete list of available choices appears on the
screen. After the user has selected an option, the list is removed from the screen and the selected item is displayed. Selecting any item automatically deselects the previous selection.

### 9.1.1 Component Methods

#### Constructors

```java
public Choice ()
```

There is only one constructor for `Choice`. When you call it, a new instance of `Choice` is created. The component is initially empty, with no items to select. Once you add some items using `addItem()` (version 1.0) or `add()` (version 1.1) and display the `Choice` on the screen, it will look something like the leftmost component in Figure 9-1. The center component shows what a `Choice` looks like when it is selected, while the one on the right shows what a `Choice` looks like before any items have been added to it.

![Figure 9–1: How Choices are displayed](image)

#### Items

```java
public int getItemCount ()
```

`.getItemCount()` method returns the number of selectable items in the `Choice` object. In Figure 9-1, `.getItemCount()` would return 6.

```java
public int countItems ()
```

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

```java
public String getItem (int index)
```

The `.getItem()` method returns the text for the item at position `index` in the `Choice`. If `index` is invalid—either `index < 0` or `index >= getItemCount()`—the `.getItem()` method throws the `ArrayIndexOutOfBoundsException` run-time exception.
public synchronized void add (String item) ★

public synchronized void addItem (String item)✩

add() adds item to the list of available choices. If item is already an option in the Choice, this method adds it again. If item is null, add() throws the run-time exception NullPointerException. The first item added to a Choice becomes the initial (default) selection.

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

public synchronized void insert (String item, int index) ★

insert() adds item to the list of available choices at position index. An index of 0 adds the item at the beginning. An index larger than the number of choices adds the item at the end. If item is null, insert() throws the run-time exception NullPointerException. If index is negative, insert() throws the run-time exception IllegalArgumentException.

public synchronized void remove (String item) ★

remove() removes item from the list of available choices. If item is present in Choice multiple times, a call to remove() removes the first instance. If item is null, remove() throws the run-time exception NullPointerException. If item is not found in the Choice, remove() throws the IllegalArgumentException run-time exception.

public synchronized void remove (int position) ★

remove() removes the item at position from the list of available choices. If position is invalid—either position < 0 or position >= getItemCount()—remove() throws the run-time exception ArrayIndexOutOfBoundsException.

public synchronized void removeAll () ★

The removeAll() method removes every option from the Choice. This allows you to refresh the list from scratch, rather than creating a new Choice and repopulating it.

Selection

The Choice has one item selected at a time. Initially, it is the first item that was added to the Choice.

public String getSelectedItem ()

The getSelectedItem() method returns the currently selected item as a String. The text returned is the parameter used in the addItem() or add() call that put the option in the Choice. If Choice is empty, getSelectedItem() returns null.
public Object[] getSelectedObjects () ★

The getSelectedObjects() method returns the currently selected item as an Object array, instead of a String. The array will either be a one-element array, or null if there are no items. This method is required by the ItemSelectable interface and allows you to use the same method to look at the items selected by a Choice, List, or Checkbox.

public int getSelectedIndex ()

The getSelectedIndex() method returns the position of the currently selected item. The Choice list uses zero-based indexing, so the position of the first item is zero. The position of the last item is the value of countItems()-1. If the list is empty, this method returns -1.

public synchronized void select (int position)

This version of the select() method makes the item at position the selected item in the Choice. If position is too big, select() throws the run-time exception IllegalArgumentException. If position is negative, nothing happens.

public void select (String string)

This version of select() makes the item with the label string the selected item. If string is in the Choice multiple times, this method selects the first. If string is not in the Choice, nothing happens.

Miscellaneous methods

public synchronized void addNotify ()

The addNotify() method creates the Choice’s 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.

protected String paramString ()

When you call the toString() method of a Choice, the default toString() method of Component gets called. This in turn calls paramString() which builds up the string to display. At the Choice level, paramString() appends the currently selected item (the result of getSelectedItem()) to the output. Using the first Choice instance in Figure 9-1, the results would be:

java.awt.Choice[139,5,92x27,current=Dialog]
9.1.2 Choice Events

The primary event for a Choice occurs when the user selects an item in the list. With the 1.0 event model, selecting an item generates an ACTION_EVENT, which triggers a call to the action() method. Once the Choice has the input focus, the user can change the selection by using the arrow or keyboard keys. The arrow keys scroll through the list of choices, triggering the KEY_ACTION, ACTION_EVENT, and KEY_ACTION_RELEASE event sequence, which in turn invokes the keyDown(), action(), and keyUp() methods, respectively. If the mouse is used to choose an item, no mouse events are triggered as you scroll over each item, and an ACTION_EVENT occurs only when a specific choice is selected.

With the 1.1 event model, you register ItemListener with addItemListener(). Then when the user selects the Choice, the ItemListener.itemStateChanged() method is called through the protected Choice.processItemEvent() method. Key, mouse, and focus listeners are registered through the Component methods of addKeyListener(), addMouseListener(), and addFocusListener(), respectively.

Action

public boolean action (Event e, Object o)

The action() method for a choice signifies that the user selected an item. e is the Event instance for the specific event, while o is the String from the call to addItem() or add() that represents the current selection. Here’s a trivial implementation of the method:

```java
public boolean action (Event e, Object o) {
    if (e.target instanceof Choice) {
        System.out.println ("Choice is now set to " + o);
    }
    return false;
}
```

Keyboard

The keyboard events for a Choice can be generated once the Choice has the input focus. In addition to the KEY_ACTION and KEY_ACTION_RELEASE events you get with the arrow keys, an ACTION_EVENT is generated over each entry.

public boolean keyDown (Event e, int key)

The keyDown() method is called whenever the user presses a key and the Choice has the input focus. 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., arrow or function key). If you check the current selection in this method through the method getSelectedItem() or getSelectedIndex(), you will be given the previously selected item because the Choice's selection has not changed yet. keyDown() is not called when the Choice is changed by using the mouse.

\textit{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 KEY_RELEASE for a regular key or KEY_ACTION_RELEASE for an action-oriented key (i.e., arrow or function key).

\textbf{Mouse}

Ordinarily, the Choice component does not trigger any mouse events.

\textbf{Focus}

Ordinarily, the Choice component does not trigger any focus events.

\textbf{Listeners and 1.1 event handling}

With the 1.1 event model, you register listeners for different event types; the listeners are told when the event happens. These methods register listeners, and let the Choice component inspect its own events.

\textit{public void addItemListener(ItemListener listener)} ★

The addItemListener() method registers listener as an object interested in being notified when an ItemEvent passes through the EventQueue with this Choice as its target. The listener.itemStateChanged() method is called when an event occurs. Multiple listeners can be registered.

\textit{public void removeItemListener(ItemListener listener)} ★

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

\textit{protected void processEvent(AWTEvent e)} ★

The processEvent() method receives all AWTEvents with this Choice as its target. processEvent() then passes them along to any listeners for processing. When you subclass Choice, 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.

```java
protected void processItemEvent(ItemEvent e) ★

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

If you override `processItemEvent()`, remember to call the method `super.processItemEvent(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 simple applet below demonstrates how a component can receive its own events by overriding `processItemEvent()`, while still allowing other objects to register as listeners. `MyChoice11` is a subclass of `Choice` that processes its own item events. `choice11` is an applet that uses the `MyChoice11` component and registers itself as a listener for item events.

```java
// Java 1.1 only
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
class MyChoice11 extends Choice {
    super ();
    enableEvents (AWTEvent.ITEM_EVENT_MASK);
}
protected void processItemEvent(ItemEvent e) {
    ItemSelectable ie = e.getItemSelectable();
    System.out.println("Item Selected: "+ ie.getSelectedObjects()[0]);
    // If you do not call super.processItemEvent()
    // no listener will be notified
    super.processItemEvent (e);
}
}
public class choice11 extends Applet implements ItemListener {
    Choice c;
    public void init () {
        String []fonts;
        fonts = Toolkit.getDefaultToolkit().getFontList();
```
c = new MyChoice11();
for (int i = 0; i < fonts.length; i++) {
    c.add (fonts[i]);
}
add (c);
c.addItemListener (this);
}

public void itemStateChanged (ItemEvent e) {
    ItemSelectable ie = e.getItemSelectable();
    System.out.println ("State Change: " + ie.getSelectedObjects()[0]);
}

A few things are worth noticing. MyChoice11 calls enableEvents() in its constructor to make sure that item events are delivered, even if nobody registers as a listener: MyChoice11 needs to make sure that it receives events, even in the absence of listeners. Its processItemEvent() method ends by calling the superclass’s processItemEvent() method, with the original item event. This call ensures that normal item event processing occurs; super.processItemEvent() is responsible for distributing the event to any registered listeners. The alternative would be to implement the whole registration and event distribution mechanism inside MyChoice11, which is precisely what object-oriented programming is supposed to avoid, or being absolutely sure that you will only use MyChoice11 in situations in which there won’t be any listeners, drastically limiting the usefulness of this class.

choice11 doesn’t contain many surprises. It implements ItemListener, the listener interface for item events; provides the required itemStateChanged() method, which is called whenever an item event occurs; and calls MyChoice11’s method addItemListener() to register as a listener for item events. (MyChoice11 inherits this method from the Choice class.)

## 9.2 Lists

Like the Choice component, the List provides a way to present your user with a fixed sequence of choices to select. However, with List, several items can be displayed at a time on the screen. A List can also allow multiple selection, so that more than one choice can be selected.

Normally, a scrollbar is associated with the List to enable the user to move to the items that do not fit on the screen. On some platforms, the List may not display the scrollbar if there is enough room to display all choices. A List can be resized by the LayoutManager according to the space available. Figure 9-2 shows two lists, one of which has no items to display.
9.2.1 List Methods

Constructors

public List ()
This constructor creates an empty List with four visible lines. You must rely on the current LayoutManager to resize the List or override the preferredSize() (version 1.0) or getPreferredSize() (version 1.1) method to affect the size of the displayed List. A List created with this constructor is in single-selection mode, so the user can select only one item at a time.

public List (int rows)
This constructor creates a List that has rows visible lines. This is just a request; the LayoutManager is free to adjust the height of the List to some other amount based upon available space. A List created with this constructor is in single-selection mode, so the user will be able to select only one item at a time.

public List (int rows, boolean multipleSelections)
The final constructor for List creates a List that has rows visible lines. This is just a request; the LayoutManager is free to adjust the height of the List to some other amount based upon available space. If multipleSelections is true, this List permits multiple items to be selected. If false, this is a single-selection list.

Figure 9–2: Two lists; the list on the right is empty

Content control

public int getItemCount () ★
public int countItems () ☆
The getItemCount() method returns the length of the list. The length of the list is the number of items in the list, not the number of visible rows.
countItems() is the Java 1.0 name for this method.

public String getItem (int index)

The getItem() method returns the String representation for the item at position index. The String is the parameter passed to the addItem() or add() method.

public String[] getItems () ★

The getItems() method returns a String array that contains all the elements in the List. This method does not care if an item is selected or not.

public synchronized void add (String item) ★

The add() method adds item as the last entry in the List. If item already exists in the list, this method adds it again.

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

public synchronized void add (String item, int index) ★

This version of the add() method has an additional parameter, index, which specifies where to add item to the List. If index < 0 or index >= getItemCount(), item is added to the end of the List. The position count is zero based, so if index is 0, it will be added as the first item.

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

public synchronized void replaceItem (String newItem, int index)

The replaceItem() method replaces the contents at position index with newItem. If the item at index has been selected, newItem will not be selected.

public synchronized void removeAll () ★

The removeAll() method clears out all the items in the list.

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

NOTE

Early versions (Java 1.0) of the clear() method did not work reliably across platforms. You were better off calling the method listVar.delItems(0, listVar.countItems()-1), where listVar is your List instance.

public synchronized void remove (String item) ★

The remove() method removes item from the list of available choices. If item appears in the List several times, only the first instance is removed. If item is
null, remove() throws the run-time exception NullPointerException. If item is not found in the List, remove() throws the IllegalArgumentException run-time exception.

```java
public synchronized void remove (int position) ★
```

The `remove()` method removes the entry at position from the List. If position is invalid—either position < 0 or position >= getItemCount()—remove() throws the ArrayIndexOutOfBoundsException run-time exception with a message indicating that position was invalid.

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

```java
public synchronized void delItems (int start, int end) ☆
```

The `delItems()` method removes entries from position start to position end from the List. If either parameter is invalid—either start < 0 or end >= getItemCount()—delItems() throws the ArrayIndexOutOfBoundsException run-time exception with a message indicating which position was invalid. If start is greater than end, nothing happens.

### Selection and positioning

```java
public synchronized int getSelectedIndex ()
```

The `getSelectedIndex()` method returns the position of the selected item. If nothing is selected in the List, `getSelectedIndex()` returns -1. The value -1 is also returned if the List is in multiselect mode and multiple items are selected. For multiselection lists, use `getSelectedIndexes()` instead.

```java
public synchronized int[] getSelectedIndexes ()
```

The `getSelectedIndexes()` method returns an integer array of the selected items. If nothing is selected, the array will be empty.

```java
public synchronized String getSelectedItem ()
```

The `getSelectedItem()` method returns the label of the selected item. The label is the string used in the add() or addItem() call. If nothing is selected in the List, `getSelectedItem()` returns null. The return value is also null if List is in multiselect mode and multiple items are selected. For multiselection lists, use `getSelectedItems()` instead.

```java
public synchronized String[] getSelectedItems ()
```

The `getSelectedItems()` method returns a String array of the selected items. If nothing is selected, the array is empty.
public synchronized Object[] getSelectedObjects ()

The getSelectedObjects() method returns the results of the method getSelectedItems() as an object array instead of a String array, to conform to the ItemSelectable interface. If nothing is selected, the returned array is empty.

public synchronized void select (int index)

The select() method selects the item at position index, which is zero based. If the List is in single-selection mode, any other selected item is deselected. If the List is in multiple-selection mode, calling this method has no effect on the other selections. The item at position index is made visible.

NOTE A negative index seems to select everything within the List. This seems more like an irregularity than a feature to rely upon.

public synchronized void deselect (int index)

The deselect() method deselects the item at position index, which is zero based. deselect() does not reposition the visible elements.

public boolean isIndexSelected (int index) ★

public boolean isSelected (int index) ☆

The isIndexSelected() method checks whether index is currently selected. If it is, isIndexSelected() returns true; otherwise, it returns false.

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

public boolean isMultipleMode () ★

public boolean allowsMultipleSelections () ☆

The isMultipleMode() method returns the current state of the List. If the List is in multiselection mode, isMultipleMode() returns true; otherwise, it returns false.

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

public void setMultipleMode (boolean value) ★

public void setMultipleSelections (boolean value) ☆

The setMultipleMode() method allows you to change the current state of a List from one selection mode to the other. The currently selected items change when this happens. If value is true and the List is going from single-to multiple-selection mode, the selected item gets deselected. If value is false and the List is going from multiple to single, the last item physically selected remains selected (the last item clicked on in the list, not the item with the highest index). If there was no selected item, the first item in the list becomes
selected, or the last item that was deselected becomes selected. If staying
within the same mode, setMultipleMode() has no effect on the selected items.

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

public void makeVisible (int index)
The makeVisible() method ensures that the item at position index is dis-
dplayed on the screen. This is useful if you want to make sure a certain entry is
displayed when another action happens on the screen.

public int getVisibleIndex ()
The getVisibleIndex() method returns the last index from a call to the
method makeVisible(). If makeVisible() was never called, -1 is returned.

Sizing

public int getRows ()
The getRows() method returns the number of rows passed to the constructor
of the List. It does not return the number of visible rows. To get a rough idea
of the number of visible rows, compare the getSize() of the component with
the results of getPreferredSize(getRows()).

public Dimension getPreferredSize (int rows) ★
public Dimension preferredSize (int rows) ☆
The getPreferredSize() method returns the preferable Dimension (width
and height) for the size of a List with a height of rows. The rows specified
may be different from the rows designated in the constructor.

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

public Dimension getPreferredSize () ★
public Dimension preferredSize () ☆
The getPreferredSize() method returns the Dimension (width and height)
for the preferred size of the List. Without the rows parameter, this version of
getPreferredSize() uses the constructor’s number of rows to calculate the
List’s preferred size.

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

public Dimension getMinimumSize (int rows) ★
public Dimension minimumSize (int rows) ☆
The getMinimumSize() method returns the minimum Dimension (width and
height) for the size of a List with a height of rows. The rows specified may be
different from the rows designated in the constructor. For a List, getMinimum-
Size() and getPreferredSize() should return the same dimensions.

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 List. Without the rows parameter, this getMinimumSize() uses the constructor’s number of rows to calculate the List’s minimum size.

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

Miscellaneous methods

public synchronized void addNotify ()

The addNotify() method creates the List 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 synchronized void removeNotify ()

The removeNotify() method destroys the peer of the List and removes it from the screen. Prior to the List peer’s destruction, the last selected entry is saved. If you override this method for a specific List, issue the particular commands that you need for your new object, then call super.removeNotify() last.

protected String paramString ()

When you call the toString() method of List, the default toString() method of Component is called. This in turn calls paramString(), which builds up the string to display. At the List level, the currently selected item (getSelectedltem()) is appended to the output. Using Figure 9-2 as an example, the results would be the following:

java.awt.List[0,34,107x54,selected=null]

9.2.2 List Events

The primary event for a List occurs when the user selects an item in the list. With the 1.0 event model, double-clicking a selection causes an ACTION_EVENT and triggers the action() method, while single-clicking causes a LIST_SELECT or LIST_DESELECT event. Once the List has the input focus, it is possible to change the selection by using the arrow or keyboard keys. The arrow keys scroll through the list of choices, triggering the KEY_ACTION, LIST_SELECT, LIST_DESELECT, and KEY_ACTION_RELEASE events, and thus the keyDown(), handleEvent(), and keyUp() methods (no specific method gets called for LIST_SELECT and LIST_DESELECT). action() is called only when the user double-clicks on an item with the mouse. If the mouse is used to scroll through the list, no mouse events are triggered; ACTION_EVENT is generated only when the user double-clicks on an item.
With the 1.1 event model, you register an ItemListener with addItemListener() or an ActionListener with the addActionListener() method. When the user selects the List, either the ItemListener.itemStateChanged() method or the ActionListener.actionPerformed() method is called through the protected List.processItemEvent() method or List.processActionEvent() method. Key, mouse, and focus listeners are registered through the three Component methods of addKeyListener(), addMouseListener(), and addFocusListener(), respectively.

### Action

**public boolean action (Event e, Object o)**

The action() method for a List is called when the user double-clicks on any item in the List. e is the Event instance for the specific event, while o is the label for the item selected, from the add() or addItem() call. If List is in multiple-selection mode, you might not wish to catch this event because it's not clear whether the user wanted to choose the item just selected or all of the items selected. You can solve this problem by putting a multi-selecting list next to a Button that the user presses when the selection process is finished. Capture the event generated by the Button. The following example shows how to set up and handle a list in this manner, with the display shown in Figure 9-3. In this example, I just print out the selections to prove that I captured them.

```java
import java.awt.*;
import java.applet.*;
public class list3 extends Applet {
    List l;
    public void init () {
        String fonts[];
        fonts = Toolkit.getDefaultToolkit().getFontList();
        l = new List(4, true);
        for (int i = 0; i < fonts.length; i++) {
            l.addItem (fonts[i]);
        }
        setLayout (new BorderLayout (10, 10));
        add ("North", new Label ("Pick Font Set"));
        add ("Center", l);
        add ("South", new Button ("Submit"));
        resize (preferredSize());
        validate();
    }
    public boolean action (Event e, Object o) {
        if (e.target instanceof Button) {
            String chosen[] = l.getSelectedItems();
            for (int i=0; i<chosen.length; i++)
                System.out.println (chosen[i]);
        }
        return false;
    }
}
```
Ordinarily, `List` generates all the `KEY` events once it has the input focus. But the way it handles keyboard input differs slightly depending upon the selection mode of the list. Furthermore, each platform offers slightly different behavior, so code that depends on keyboard events in `List` is not portable. One strategy is to take advantage of the keyboard events when they are available but allow for another way of managing the list in case they are not.

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

The `keyDown()` method is called whenever the user presses a key while the `List` has the input focus. `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 `KEY_PRESS` for a regular key or `KEY_ACTION` for an action-oriented key (i.e., arrow or function key). If you check the current selection in this method through `getSelectedItem()` or `getSelectedIndex()`, you will actually be told the previously selected item because the `List`’s selection has not changed yet. `keyDown()` is not called when the user selects items with the mouse.

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

The `keyUp()` method is called whenever the user releases a key while the `List` has the input focus. `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 `KEY_RELEASE` for a regular key or `KEY_ACTION_RELEASE` for an action-oriented key (i.e., arrow or function key).

**Mouse**

Ordinarily, the `List` component does not trigger any mouse events. Double-
clicking the mouse over any element in the list generates an ACTION_EVENT. Single-clicking could result in either a LIST_SELECT or LIST_DESELECT, depending on the mode of the List and the current state of the item chosen. When the user changes the selection with the mouse, the ACTION_EVENT is posted only when an item is double-clicked.

**List**

There is a special pair of events for lists: LIST_SELECT and LIST_DESELECT. No special method is called when these events are triggered. However, you can catch them in the handleEvent() method. If the List is in single-selection mode, a LIST_SELECT event is generated whenever the user selects one of the items in the List. In multiple-selection mode, you will get a LIST_SELECT event when an element gets selected and a LIST_DESELECT event when it is deselected. The following code shows how to use this event type.

```java
public boolean handleEvent (Event e) {
    if (e.id == Event.LIST_SELECT) {
        System.out.println ("Selected item: " + e.arg);
        return true;
    } else {
        return super.handleEvent (e);
    }
}
```

**Focus**

Normally, the List component does not reliably trigger any focus events.

**Listeners and 1.1 event handling**

With the 1.1 event model, you register listeners, and they are told when the event happens.

```java
public void addItemListener(ItemListener listener) ★
```

The addItemListener() method registers listener as an object interested in being notified when an ItemEvent passes through the EventQueue with this List as its target. The listener.itemStateChanged() method is called when these events occur. Multiple listeners can be registered.

```java
public void removeItemListener(ItemListener listener) ★
```

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

The addActionListener() method registers listener as an object interested in being notified when an ActionEvent passes through the EventQueue with this List as its target. The listener.actionPerformed() method is called when these events occur. Multiple listeners can be registered.

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 List as its target. processEvent() then passes them along to any listeners for processing. When you subclass List, overriding processEvent() allows you to process all events yourself, before sending them to any listeners. In a way, overriding the method 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 processItemEvent(ItemEvent e) ★

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

If you override processItemEvent(), remember to call the method super.processItemEvent(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 List as its target. processActionEvent() then passes them along to any listeners for processing. When you subclass List, overriding 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 `processActionEvent()`, remember to call the method `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.

### 9.3 Checkbox

The Checkbox is a general purpose way to record a true or false state. When several checkboxes are associated in a `CheckboxGroup` (Section 9.4), only one can be selected at a time; selecting each Checkbox causes the previous selection to become deselected. The `CheckboxGroup` is Java’s way of offering the interface element known as radio buttons or a radio box. When you create a Checkbox, you decide whether to place it into a `CheckboxGroup` by setting the proper argument in its constructor.

Every Checkbox has both a label and a state, although the label could be empty. You can change the label based on the state of the Checkbox. Figure 9-4 shows what several Checkbox components might look like. The two on the left are independent, while the five on the right are in a `CheckboxGroup`. Note that the appearance of a Checkbox varies quite a bit from platform to platform. However, the appearance of a `CheckboxGroup` is always different from the appearance of an ungrouped Checkbox, and the appearance of a checked Checkbox is different from an unchecked Checkbox.

![Figure 9-4: Two separate checkboxes and a CheckboxGroup](image-url)
9.3 Checkbox Methods

Constructors

public Checkbox ()

This constructor for Checkbox creates a new instance with no label or grouping. The initial state of the item is false. A checkbox doesn’t necessarily need a label; however, a checkbox without a label might be confusing, unless it is being used as a column in a table or a spreadsheet.

public Checkbox (String label)

The second constructor creates a new Checkbox with a label of label and no grouping. The initial state of the item is false. If you want a simple yes/no choice and plan to make no the default, use this constructor. If the Checkbox will be in a group or you want its initial value to be true, use the next constructor.

public Checkbox (String label, boolean state) ★

This constructor allows you to specify the Checkbox’s initial state. With it you create a Checkbox with a label of label and an initial state of state.

public Checkbox (String label, boolean state, CheckboxGroup group) ★

public Checkbox (String label, CheckboxGroup group, boolean state)

The final constructor for Checkbox is the most flexible. With this constructor you create a Checkbox with a label of label, a CheckboxGroup of group, and an initial state of state. If group is null, the Checkbox is independent.

In Java 1.0, you created an independent Checkbox with an initial value of true by using null as the group:

    Checkbox cb = new Checkbox ("Help", null, true)

The shape of the Checkbox reflects whether it’s in a CheckboxGroup or independent. On Microsoft Windows, grouped checkboxes are represented as circles. On a UNIX system, they are diamonds. On both systems, independent checkboxes are squares.

Label

public String getLabel ()

The getLabel() method retrieves the current label on the Checkbox and returns it as a String object.
```java
public synchronized void setLabel (String label)
    The setLabel() method changes the label of the Checkbox to label. If the new label is a different size than the old one, you have to validate() the container after the change to ensure the entire label will be seen.

State
A state of true means the Checkbox is selected. A state of false means that the Checkbox is not selected.

public boolean getState ()
    The getState() method retrieves the current state of the Checkbox and returns it as a boolean.

public void setState (boolean state)
    The setState() method changes the state of the Checkbox to state. If the Checkbox is in a CheckboxGroup and state is true, the other items in the group become false.

ItemSelectable method

public Objects[] getSelectedObjects () ★
    The getSelectedObjects() method returns the Checkbox label as a one-element Object array if it is currently selected, or null if the Checkbox is not selected. Because this method is part of the ItemSelectable interface, you can use it to look at the selected items in a Choice, List, or Checkbox.

CheckboxGroup
This section lists methods that you issue to Checkbox to affect its relationship to a CheckboxGroup. Methods provided by the CheckboxGroup itself can be found later in this chapter.

public CheckboxGroup getCheckboxGroup ()
    The getCheckboxGroup() method returns the current CheckboxGroup for the Checkbox. If the Checkbox is not in a group, this method returns null.

public void setCheckboxGroup (CheckboxGroup group)
    The setCheckboxGroup() method allows you to insert a Checkbox into a different CheckboxGroup. To make the Checkbox independent, pass a group argument of null. The method sets every Checkbox in the original CheckboxGroup to false (cb.getCheckboxGroup().setCurrent(null)), then the Checkbox is added to the new group without changing any values in the new group.
```
Checkbox components take on a different shape when they are in a CheckboxGroup. If the checkbox was originally not in a CheckboxGroup, the shape of the checkbox does not change automatically when you put it in one with setCheckboxGroup(). (This also holds when you remove a Checkbox from a CheckboxGroup and make it independent or vice versa.) In order for the Checkbox to look right once added to group, you need to destroy and create (removeNotify() and addNotify(), respectively) the Checkbox peer to correct the shape. Also, it is possible to get multiple true Checkbox components in group this way, since the new CheckboxGroup’s current selection does not get adjusted. To avoid this problem, make sure it is grouped properly the first time, or be sure to clear the selections with a call to getCheckboxGroup().setCurrent(null).

**Miscellaneous methods**

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

The `addNotify()` method will create the Checkbox peer in the appropriate shape. 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.

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

When you call the `toString()` method of Checkbox, the default `toString()` method of Component is called. This in turn calls `paramString()` which builds up the string to display. At the Checkbox level, the label (if non-null) and the state of the item are appended. Assuming the Dialog Checkbox in Figure 9-4 was selected, the results would be:

```
java.awt.Checkbox[85,34,344x32,label=Dialog,state=true]
```

### 9.3.2 Checkbox Events

The primary event for a Checkbox occurs when the user selects it. With the 1.0 event model, this generates an `ACTION_EVENT` and triggers the `action()` method. Once the Checkbox has the input focus, the various keyboard events can be generated, but they do not serve any useful purpose because the Checkbox doesn’t change. The sole key of value for a Checkbox is the spacebar. This may generate the `ACTION_EVENT` after `KEY_PRESS` and `KEY_RELEASE`; thus the sequence of method calls would be `keyDown()`, `keyUp()`, and then `action()`.

With the version 1.1 event model, you register an ItemListener with the method `addItemListener()`. Then when the user selects the Checkbox, the method `ItemListener.itemStateChanged()` is called through the protected
Checkbox.processItemEvent() method. Key, mouse, and focus listeners are registered through the Component methods of addKeyListener(), addMouseListener(), and addFocusListener(), respectively.

Action

public boolean action (Event e, Object o)

The action() method for a Checkbox is called when the user selects it. e is the Event instance for the specific event, while o is the opposite of the old state of the toggle. If the Checkbox was true when it was selected, o will be false. Likewise, if it was false, o will be true. This incantation sounds unnecessarily complex, and for a single Checkbox, it is: o is just the new state of the Checkbox.

The following code uses action() with a single Checkbox.

```java
public boolean action (Event e, Object o) {
    if (e.target instanceof Checkbox) {
        System.out.println ("Checkbox is now " + o);
    }
    return false;
}
```

On the other hand, if the Checkbox is in a CheckboxGroup, o is still the opposite of the old state of the toggle, which may or may not be the new state of the Checkbox. If the Checkbox is initially false, o will be true, and the Checkbox’s new state will be true. However, if the Checkbox is initially true, selecting the Checkbox doesn’t change anything because one Checkbox in the group must always be true. In this case, o is false (the opposite of the old state), though the Checkbox’s state remains true.

Therefore, if you’re working with a CheckboxGroup and need to do something once when the selection changes, perform your action only when o is true. To find out which Checkbox was actually chosen, you need to call the getLabel() method for the target of event e. (It would be nice if o gave us the label of the Checkbox that was selected, but it doesn’t.) An example of this follows:

```java
public boolean action (Event e, Object o) {
    if (e.target instanceof Checkbox) {
        System.out.println (((Checkbox)(e.target)).getLabel() + " was selected.");
        if (new Boolean (o.toString()).booleanValue()) {
            System.out.println ("New option chosen");
        } else {
            System.out.println ("Use re-selected option");
        }
    }
    return false;
}
```
One other unfortunate twist of CheckboxGroup: it would be nice if there was some easy way to find out about checkboxes that change state without selection—for example, if you could find out which Checkbox was deselected when a new Checkbox was selected. Unfortunately, you can’t, except by keeping track of the state of all your checkboxes at all times. When a Checkbox state becomes false because another Checkbox was selected, no additional event is generated, in either Java 1.0 or 1.1.

**Keyboard**

Checkboxes are able to capture keyboard-related events once the Checkbox has the input focus, which happens when it is selected. If you can find a use for this, you can use keyDown() and keyUp(). For most interface designs I can think of, action() is sufficient. A possible use for keyboard events is to jump to other Checkbox options in a CheckboxGroup, but I think that is more apt to confuse users than help.

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

The keyDown() method is called whenever the user presses a key while the Checkbox has the input focus. 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 KEY_PRESS for a regular key or KEY_ACTION for an action-oriented key (i.e., arrow or function key). There is no visible indication that the user has pressed a key over the checkbox.

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

The keyUp() method is called whenever the user releases a key while the Checkbox has the input focus. 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 KEY_RELEASE for a regular key or KEY_ACTION_RELEASE for an action-oriented key (i.e., arrow or function key). keyUp() may be used to determine how long key has been pressed.

**Mouse**

Ordinarily, the Checkbox component does not reliably trigger any mouse events.

**Focus**

Ordinarily, the Checkbox component does not reliably trigger any focus events.
Listeners and 1.1 event handling

With the 1.1 event model, you register listeners, and they are told when the event happens.

```java
public void addItemListener(ItemListener listener)
```

The `addItemListener()` method registers `listener` as an object interested in being notified when an `ItemEvent` passes through the `EventQueue` with this Checkbox as its target. Then, the `listener.itemStateChanged()` method will be called. Multiple listeners can be registered.

```java
public void removeItemListener(ItemListener listener)
```

The `removeItemListener()` method removes `listener` as a interested listener. If `listener` is not registered, nothing happens.

```java
protected void processEvent(AWTEvent e)
```

The `processEvent()` method receives every `AWTEvent` with this Checkbox as its target. `processEvent()` then passes it along to any listeners for processing. When you subclass `Checkbox`, 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.

```java
protected void processItemEvent(ItemEvent e)
```

The `processItemEvent()` method receives every `ItemEvent` with this Checkbox as its target. `processItemEvent()` then passes it along to any listeners for processing. When you subclass `Checkbox`, overriding `processItemEvent()` allows you to process all events yourself, before sending them to any listeners. In a way, overriding `processItemEvent()` is like overriding `action()` using the 1.0 event model.

If you override `processItemEvent()`, remember to call the method `super.processItemEvent(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.
9.4 CheckboxGroup

The CheckboxGroup lets multiple checkboxes work together to provide a mutually exclusion choice (at most one Checkbox can be selected at a time). Because the CheckboxGroup is neither a Component nor a Container, you should normally put all the Checkbox components associated with a CheckboxGroup in their own Panel (or other Container). The LayoutManager of the Panel should be GridLayout (0, 1) if you want them in one column. Figure 9-5 shows both a good way and bad way of positioning a set of Checkbox items in a CheckboxGroup. The image on the left is preferred because the user can sense that the items are grouped; the image on the right suggests three levels of different checkboxes and can therefore surprise the user when checkboxes are deselected.

![Figure 9–5: Straightforward and confusing layouts of Checkbox components](image)

9.4.1 CheckboxGroup Methods

Constructors

- `public CheckboxGroup ()`

  This constructor creates an instance of CheckboxGroup.

Miscellaneous methods

- `public int getSelectedCheckbox () ★`
- `public Checkbox getCurrent () ☆`

  The `getSelectedCheckbox()` method returns the Checkbox within the CheckboxGroup whose value is true. If no item is selected, null is returned.

  `getCurrent()` is the Java 1.0 name for this method.
public synchronized void setSelectedCheckbox (Checkbox checkbox) ★

The setSelectedCheckbox() method makes checkbox the currently selected Checkbox within the CheckboxGroup. If checkbox is null, the method deselects all the items in the CheckboxGroup. If checkbox is not within the CheckboxGroup, nothing happens.

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

public String toString () ★

The toString() method of CheckboxGroup creates a String representation of the current choice (as returned by getSelectedCheckbox()). Using the “straightforward” layout in Figure 9-5 as an example, the results would be:

java.awt.CheckboxGroup[current=java.awt.Checkbox[0,31,85x21, label=Helvetica,state=true]]

If there is no currently selected item, the results within the square brackets would be current=null.

9.5 ItemSelectable

In Java 1.1, the classes Checkbox, Choice, List, and CheckboxMenuItem (covered in the next chapter) share a common interface that defines a method for getting the currently selected item or items. This means that you can use the same methods to retrieve the selection from any of these classes. More important, it means that you can write code that doesn’t know what kind of selectable item it’s working with. For example, you could write a method that returns the selectable component from some user interface. This method might have the signature:

public ItemSelectable getChooser();

After you call this method, you can read selections from the user interface without knowing exactly what you’re dealing with.

9.5.1 Methods

public Object[] getSelectedObjects () ★

The getSelectedObjects() method returns the currently selected item or items as an Object array. The return value is null if there is nothing selected.