Would You Like to Choose from the Menu?

In Chapter 6, *Containers*, I mentioned that a Frame can have a menu. Indeed, to offer a menu in the AWT, you have to attach it to a Frame. With versions 1.0.2 and 1.1, Java does not support menu bars within an applet or any other container. We hope that future versions of Java will allow menus to be used with other containers. Java 1.1 goes partway toward solving this problem by introducing a PopupMenu that lets you attach context menus to any Component. Java 1.1 also adds MenuShortcut events, which represent keyboard accelerator events for menus.

Implementing a menu in a Frame involves connections among a number of different objects: MenuBar, Menu, MenuItem, and the optional CheckboxMenuItem. Several of these classes implement the MenuContainer interface. Once you’ve created a few menus, you’ll probably find the process quite natural, but it’s hard to describe until you see what all the objects are. So this chapter describes most of the menu classes first and then shows an example demonstrating their use.

All the components covered in previous chapters were subclasses of Component. Most of the objects in this chapter subclass MenuComponent, which encapsulates the common functionality of menu objects. The MenuComponent class hierarchy is shown in Figure 10-1.

To display a Menu, you must first put it in a MenuBar, which you add to a Frame. (Pop-up menus are different in that they don’t need a Frame.) A Menu can contain MenuItem as well as other menus that form submenus. CheckboxMenuItem is a specialized MenuItem that (as you might guess) the user can toggle like a Checkbox. One way to visualize how all these things work together is to imagine a set of curtains. The different MenuItem components are the fabrics and panels that make up the curtains. The Menus are the curtains. They get hung from the MenuBar, which is
like a curtain rod. Then you place the MenuBar curtain rod into the Frame (the window, in our metaphor), curtains and all.

It might puzzle you that a Menu is a subclass of MenuItem, not the other way around. This is because a Menu can appear on a Menu just like another MenuItem, which would not be possible if the hierarchy was the other way around. Figure 10-2 points out the different pieces involved in the creation of a menu: the MenuBar and various kinds of menu items, including a submenu.

Figure 10–1: MenuComponent class hierarchy

Figure 10–2: The pieces that make up a Menu
10.1 MenuComponent

MenuComponent is an abstract class that is the parent of all menu-related objects. You will never create an instance of the object. Nor are you likely to subclass it yourself—to make the subclass work, you’d have to provide your own peer on every platform where you want the application to run.

10.1.1 MenuComponent Methods

Constructor

`public MenuComponent () — cannot be called directly`

Since MenuComponent is an abstract class, you cannot create an instance of the object. This method is called when you create an instance of one of its children.

Fonts

`public Font getFont ()`

The `getFont()` method retrieves the font associated with the `MenuComponent` from `setFont()`. If the current object’s font has not been set, the parent menu’s font is retrieved. If there is no parent and the current object’s font has not been set, `getFont()` returns `null`.

`public void setFont (Font f)`

The `setFont()` method allows you to change the font of the particular menu-related component to `f`. When a `MenuComponent` is first created, the initial font is `null`, so the parent menu’s font is used.

---

**NOTE**  Some platforms do not support changing the fonts of menu items. Where supported, it can make some pretty ugly menus.

Names

The name serves as an alternative, nonlocalized reference identifier for menu components. If your event handlers compare menu label strings to an expected value and labels are localized for a new environment, the approach fails.

`public String getName ()`

The `getName()` method retrieves the name of the menu component. Every instance of a subclass of `MenuComponent` is named when it is created.
**public void setName (String name)**
The `setName()` method changes the current name of the component to `name`.

**Peers**

**public MenuComponentPeer getPeer ()** ✦
The `getPeer()` method returns a reference to the `MenuComponent` peer as a `MenuComponentPeer`.

**public synchronized void removeNotify ()**
The `removeNotify()` method destroys the peer of the `MenuComponent` and removes it from the screen. `addNotify()` will be specific to the subclass.

**Events**

Event handling is slightly different between versions. If using the 1.0 event model, use `postEvent()`. Otherwise, use `dispatchEvent()` to post an event to this `MenuComponent` or `processEvent()` to receive and handle an event. Remember not to mix versions within your programs.

**public boolean postEvent (Event e)** ✦
The `postEvent()` method posts `Event e` to the `MenuComponent`. The event is delivered to the `Frame` at the top of the object hierarchy that contains the selected `MenuComponent`. The only way to capture this event before it gets handed to the `Frame` is to override this method. There are no helper functions as there are for `Components`. Find out which `MenuComponent` triggered the event by checking `e.arg`, which contains its label, or `((MenuItem)e.target).getName()` for the nonlocalized name of the target.

```java
public boolean postEvent (Event e) {
    // Use getName() vs. e.arg for localization possibility
    if ("About".equals (((MenuItem)e.target).getName()))
        playLaughingSound(); // Help request
    return super.postEvent (e);
}
```

If you override this method, in order for this `Event` to propagate to the `Frame` that contains the `MenuComponent`, you must call the original `postEvent()` method (`super.postEvent(e)`).

The actual value returned by `postEvent()` is irrelevant.

**public final void dispatchEvent(AWTEvent e)** ★
The `dispatchEvent()` method allows you to post new AWT events to this menu component's listeners. `dispatchEvent()` tells the `MenuComponent` to deal with the `AWTEvent e` by calling its `processEvent()` method. This method is similar
to Java 1.0’s postEvent() method. Events delivered in this way bypass the system’s event queue. It’s not clear why you would want to bypass the event queue, except possibly to deliver some kind of high priority event.

protected void processEvent(AWTEvent e)

The processEvent() method receives all AWTEvents with a subclass of MenuComponent as its target. processEvent() then passes them along for processing. When you subclass a child class, overriding processEvent() allows you to process all events without having to provide listeners. However, remember to call super.processEvent(e) last to ensure regular functionality is still executed. This is like overriding postEvent() using the 1.0 event model.

Miscellaneous methods

public MenuContainer getParent()

The getParent() method returns the parent MenuContainer for the MenuComponent. MenuContainer is an interface that is implemented by Component (in 1.1 only), Frame, Menu, and MenuBar. This means that getParent() could return any one of the four.

protected String paramString()

The paramString() method of MenuComponent helps build up the string to display when toString() is called for a subclass. At the MenuComponent level, the current name of the object is appended to the output.

public String toString()—can be called by user for subclass

The toString() method at the MenuComponent level cannot be called directly. This toString() method is called when you call a subclass’s toString() and the specifics of the subclass is added between the brackets ([ and ]). At this level, the results would be:

java.awt.MenuComponent[aname1]

10.2 MenuContainer

MenuContainer is an interface implemented by the three menu containers: Frame, Menu, and MenuBar; Java 1.1 adds a fourth, Component. You should never need to worry about the interface since it does all its work behind the scenes for you. You will notice that the interface does not define an add() method. Each type of MenuContainer defines its own add() method to add menus to itself.
10.2.1 MenuContainer Methods

public abstract Font getFont()

The getFont() method should provide an object’s font. MenuItem implements this method, so all of its subclasses inherit it. MenuBar implements it, too, while Frame gets the method from Component.

public abstract boolean postEvent(Event e) ★

The postEvent() method should post Event e to the object. MenuComponent implements this method, so all of its subclasses inherit it. (Frame gets the method from Component.)

public abstract void remove(MenuComponent component)

The remove() method should remove the MenuComponent component from the object. If component was not contained within the object, nothing should happen.

10.3 MenuShortcut

MenuShortcut is a class used to represent a keyboard shortcut for a MenuItem. When these events occur, an action event is generated that triggers the menu component. When a shortcut is associated with a MenuItem, the MenuItem automatically displays a visual clue, which indicates that a keyboard accelerator is available.

10.3.1 MenuShortcut Methods

Constructors

public MenuShortcut(int key) ★

The first MenuShortcut constructor creates a MenuShortcut with key as its designated hot key. The key parameter can be any of the virtual key codes from the KeyEvent class (e.g., VK_A, VK_B, etc.). These constants are listed in Table 4-4. To use the shortcut, the user must combine the given key with a platform-specific modifier key. On Windows and Motif platforms, the modifier is the Control key; on the Macintosh, it is the Command key. For example, if the shortcut key is F1 (VK_F1) and you’re using Windows, you would press Ctrl+F1 to execute the shortcut. To find out the platform’s modifier key, call the Toolkit.getMenuShortcutKeyMask() method.

public MenuShortcut(int key, boolean useShiftModifier) ★

This MenuShortcut constructor creates a MenuShortcut with key as its designated hot key. If useShiftModifier is true, the Shift key must be depressed for this shortcut to trigger the action event (in addition to the shortcut key).
The key parameter represents the integer value of a **KEY_PRESS** event, so in addition to ASCII values, possible values include the various **Event** keyboard constants (listed in Table 4-2) like **Event.F1**, **Event.HOME**, and **Event.PAUSE**. For example, if key is the ASCII value for A and **useShiftModifier** is true, the shortcut key is **Shift+Ctrl+A** on a Windows/Motif platform.

**Miscellaneous methods**

**public int getKey()** ★
The **getKey()** method retrieves the virtual key code for the key that triggered this **MenuShortcut**. The virtual key codes are the **VK** constants defined by the **KeyEvent** class (see Table 4-4).

**public boolean usesShiftModifier()** ★
The **usesShiftModifier()** method returns **true** if this **MenuShortcut** requires the Shift key be pressed, **false** otherwise.

**public boolean equals(MenuShortcut s)** ★
The **equals()** method overrides **Object**'s **equals()** method to define equality for menu shortcuts. Two **MenuShortcut** objects are equal if their **key** and **useShiftModifier** values are equal.

**protected String paramString()** ★
The **paramString()** method of **MenuShortcut** helps build up a string describing the shortcut; it appends the shortcut key and a shift modifier indicator to the string under construction. Oddly, this method is not currently used, nor can you call it; **MenuShortcut** has its own **toString()** method that does the job itself.

**public String toString()** ★
The **toString()** method of **MenuShortcut** builds a **String** to display the contents of the **MenuShortcut**.

### 10.4 MenuItem

A **MenuItem** is the basic item that goes on a **Menu**. Menus themselves are menu items, allowing submenus to be nested inside of menus. **MenuItem** is a subclass of **MenuComponent**.
10.4.1 MenuItem Methods

Constructors

`public MenuItem ()` ★

The first `MenuItem` constructor creates a `MenuItem` with an empty label and no
keyboard shortcut. To set the label at later time, use `setLabel()`.

`public MenuItem (String label)`

This `MenuItem` constructor creates a `MenuItem` with a label of `label` and no key-
board shortcut. A label of "–" represents a separator.

`public MenuItem (String label, MenuShortcut shortcut)` ★

The final `MenuItem` constructor creates a `MenuItem` with a label of `label` and a
`MenuShortcut` of `shortcut`. Pressing the shortcut key is the same as selecting
the menu item.

Menu labels

Each `MenuItem` has a label. This is the text that is displayed on the menu.

**NOTE** Prior to Java 1.1, there was no portable way to associate a hot key
with a `MenuItem`. However, in Java 1.0, if you precede a character with
an & on a Windows platform, it will appear underlined, and that key
will act as the menu’s mnemonic key (a different type of shortcut
from `MenuShortcut`). Unfortunately, on a Motif platform, the user
will see the &. Because the & is part of the label, even if it is not dis-
played, you must include it explicitly whenever you compare the
label to a string.

`public String getLabel ()`

The `getLabel()` method retrieves the label associated with the `MenuItem`.

`public void setLabel (String label)`

The `setLabel()` method changes the label of the `MenuItem` to `label`.

Shortcuts

`public MenuShortcut getMenuShortcut ()` ★

The `getMenuShortcut()` method retrieves the shortcut associated with this
`MenuItem`.

`public void setShortcut (MenuShortcut shortcut)` ★

The `setShortcut()` method allows you to change the shortcut associated with
a `MenuItem` to `shortcut` after the `MenuItem` has been created.
public void deleteMenuShortcut () ★
The deleteMenuShortcut() method removes any associated MenuShortcut from the MenuItem. If there was no shortcut, nothing happens.

Enabling

public boolean isEnabled ()
The isEnabled() method checks to see if the MenuItem is currently enabled. An enabled MenuItem can be selected by the user. A disabled MenuItem, by convention, appears grayed out on the Menu. Initially, each MenuItem is enabled.

public synchronized void setEnabled(boolean b) ★
public void enable (boolean condition) ☆
The setEnabled() method either enables or disables the MenuItem based on the value of condition. If condition is true, the MenuItem is enabled. If condition is false, it is disabled. When enabled, the user can select it, generating ACTION_EVENT or notifying the ActionListener. When disabled, the peer does not generate an ACTION_EVENT if the user tries to select the MenuItem. A disabled MenuItem is usually grayed out to signify its state. The way that disabling is signified is platform specific.

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

public synchronized void enable () ☆
The enable() method enables the MenuItem. In Java 1.1, it is better to use setEnabled().

public synchronized void disable () ☆
The disable() method disables the component so that the user cannot select it. In Java 1.1, it is better to use setEnabled().

Miscellaneous methods

public synchronized void addNotify ()
The addNotify() method creates the MenuItem peer.

public String paramString ()
The paramString() method of MenuItem should be protected like other paramString() methods. However, it is public so you have access to it. When you call the toString() method of a MenuItem, the default toString() method of MenuComponent is called. This in turn calls paramString() which builds up the string to display. At the MenuItem level, the current label of the object and the shortcut (if present) is appended to the output. If the constructor for the MenuItem was new MenuItem("File"), the results of toString() would be:
10.4.2 MenuItem Events

Event handling

With 1.0 event handing, a MenuItem generates an ACTION_EVENT when it is selected. The argument to action() will be the label of the MenuItem. But the target of the ACTION_EVENT is the Frame containing the menu. You cannot subclass MenuItem and catch the Event within it with action(), but you can with postEvent(). No other events are generated for MenuItem instances.

```java
public boolean action (Event e, Object o)—overridden by user, called by system
```

The action() method for a MenuItem signifies that the user selected it. e is the Event instance for the specific event, while o is the label of the MenuItem.

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 String getActionCommand() ★
```

The getActionCommand() method retrieves the command associated with this MenuItem. By default, it is the label. However, the default can be changed by using the setActionCommand() method (described next). The command acts like the second parameter to the action() method in the 1.0 event model.

```java
public void setActionCommand(String command) ★
```

The setActionCommand() method changes the command associated with a MenuItem. When an ActionEvent happens, the command is part of the event. By default, this would be the label of the MenuItem. However, you can change the action command by calling this method. Using action commands is a good idea, particularly if you expect your code to run in a multilingual environment.

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

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

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

The removeActionListener() method removes listener as an interested listener. If listener is not registered, nothing happens.
protected final void enableEvents(long eventsToEnable)

Using the enableEvents() method is usually not necessary. When you register an action listener, the MenuItem listens for action events. However, if you wish to listen for events when listeners are not registered, you must enable the events explicitly by calling this method. The settings for the eventsToEnable parameter are found in the AWTEvent class; you can use any of the EVENT_MASK constants like COMPONENT_EVENT_MASK, MOUSE_EVENT_MASK, and WINDOW_EVENT_MASK ORed together for the events you care about. For instance, to listen for action events, call:

   enableEvents (AWTEvent.ACTION_EVENT_MASK);

protected final void disableEvents(long eventsToDisable)

Using the disableEvents() method is usually not necessary. When you remove an action listener, the MenuItem stops listening for action events if there are no more listeners. However, if you need to, you can disable events explicitly by calling disableEvents(). The settings for the eventsToDisable parameter are found in the AWTEvent class; you can use any of the EVENT_MASK constants such as FOCUS_EVENT_MASK, MOUSE_MOTION_EVENT_MASK, and ACTION_EVENT_MASK ORed together for the events you no longer care about.

protected void processEvent(AWTEvent e)

The processEvent() method receives all AWTEvents with this MenuItem as its target. processEvent() then passes them along to any listeners for processing. When you subclass MenuItem, overriding processEvent() allows you to process all events yourself, before sending them to any listeners. In a way, overriding processEvent() is like overriding postEvent() 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() to ensure that events are delivered even in the absence of registered listeners.

protected void processActionEvent(ItemEvent e)

The processActionEvent() method receives all ActionEvents with this MenuItem as its target. processActionEvent() then passes them along to any listeners for processing. When you subclass MenuItem, 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() to ensure that events are delivered even in the absence of registered listeners.
10.5 Menu

Menus are the pull-down objects that appear on the MenuBar of a Frame or within other menus. They contain Menus or CheckboxMenuItems for the user to select. The Menu class subclasses MenuItem (so it can appear on a Menu, too) and implements MenuContainer. Tear-off menus are menus that can be dragged, placed elsewhere on the screen, and remain on the screen when the input focus moves to something else. Java supports tear-off menus if the underlying platform does. Motif (UNIX) supports tear-off menus; Microsoft Windows platforms do not.

10.5.1 Menu Methods

Constructors

public Menu ()

The first constructor for Menu creates a menu that has no label and cannot be torn off. To set the label at a later time, use setLabel().

public Menu (String label)

This constructor for Menu creates a Menu with label displayed on it. The Menu cannot be torn off.

public Menu (String label, boolean tearOff)

This constructor for Menu creates a Menu with label displayed on it. The handling of tearOff is platform dependent.

Figure 10-3 shows a tear-off menu for Windows NT/95 and Motif. Since Windows does not support tear-off menus, the Windows menu looks and acts like a regular menu.

![Menu Example](image)

Figure 10–3: Tear-off menu
Items

public int getItemCount() ★
public int countItems() ☆

The getItemCount() method returns the number of items within the Menu. Only top-level items are counted: if an item is a submenu, this method doesn’t include the items on it.

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

public MenuItem getItem (int index)

The getItem() method returns the MenuItem at position index. If index is invalid, getItem() throws the ArrayIndexOutOfBoundsException run-time exception.

public synchronized MenuItem add (MenuItem item)

The add() method puts item on the menu. The label assigned to item when it was created is displayed on the menu. If item is already in another menu, it is removed from that menu. If item is a Menu, it creates a submenu. (Remember that Menu subclasses MenuItem.)

public void add (String label)

This version of add() creates a MenuItem with label as the text and adds that to the menu. If label is the String “-”, a separator bar is added to the Menu.

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

The insert() method puts item on the menu at position index. The label assigned to item when it was created is displayed on the menu. Positions are zero based, and if index < 0, insert() throws the IllegalArgumentException run-time exception.

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

This version of insert() method creates a MenuItem with label as the text and adds that to the menu at position index. If label is the String “-”, a separator bar is added to the Menu. Positions are zero based, and if index < 0, this method throws the IllegalArgumentException run-time exception.

public void addSeparator ()

The addSeparator() method creates a separator MenuItem and adds that to the menu. Separator menu items are strictly cosmetic and do not generate events when selected.

public void insertSeparator(int index) ★

The insertSeparator() method creates a separator MenuItem and adds that to the menu at position index. Separator menu items are strictly cosmetic and do not generate events when selected. Positions are zero based. If index < 0, insertSeparator() throws the IllegalArgumentException run-time exception.
public synchronized void remove (int index)

The remove() method removes the MenuItem at position index from the Menu. If index is invalid, remove() throws the ArrayIndexOutOfBoundsException run-time exception. index is zero based, so it can range from 0 to getItemCount()-1.

public synchronized void remove (MenuComponent component)

This version of remove() removes the menu item component from the Menu. If component is not in the Menu, nothing happens.

public synchronized void removeAll()

The removeAll() removes all MenuItems from the Menu.

Peers

public synchronized void addNotify()

The addNotify() method creates the Menu peer with all the MenuItems on it.

public synchronized void removeNotify()

The removeNotify() method destroys the peer of the MenuComponent and removes it from the screen. The peers of the items on the menu are also destroyed.

Miscellaneous methods

public boolean isTearOff()

The isTearOff() method returns true if this Menu is a tear-off menu, and false otherwise. Once a menu is created, there is no way to change the tear-off setting. This method can return true even on platforms that do not support tear-off menus.

public String paramString() ★

The paramString() method of Menu should be protected like other paramString() methods. However, it is public so you have access to it. When you call the toString() method of a Menu, the default toString() method of MenuComponent is called. This in turn calls paramString(), which builds up the string to display. At the Menu level, the setting for TearOff (from constructor) and whether or not it is the help menu (from MenuBar.setHelpMenu()) for the menu bar are added. If the constructor for the Menu was new Menu ("File"), the results of toString() would be:

java.awt.Menu [menu0, label=File, tearOff=false, isHelpMenu=false]
10.5.2 Menu Events

A Menu does not generate any event when it is selected. An event is generated when a MenuItem on the menu is selected, as long as it is not another Menu. You can capture all the events that happen on a Menu by overriding postEvent().

10.6 CheckboxMenuItem

The CheckboxMenuItem is a subclass of MenuItem that can be toggled. It is similar to a Checkbox but appears on a Menu. The appearance depends upon the platform. There may or may not be a visual indicator next to the choice. However, when the MenuItem is selected (true), a checkmark or some similar graphic will be displayed next to the label.

There is no way to put CheckboxMenuItem components into a CheckboxGroup to form a radio menu group.

An example of a CheckboxMenuItem is the Show Java Console menu item in Netscape Navigator.

10.6.1 CheckboxMenuItem Methods

Constructors

public CheckboxMenuItem (String label)

The first CheckboxMenuItem constructor creates a CheckboxMenuItem with no label displayed next to the check toggle. The initial value of the CheckboxMenuItem is false. To set the label at a later time, use setLabel().

public CheckboxMenuItem (String label)

The next CheckboxMenuItem constructor creates a CheckboxMenuItem with label displayed next to the check toggle. The initial value of the CheckboxMenuItem is false.

public CheckboxMenuItem (String label, boolean state)

The final CheckboxMenuItem constructor creates a CheckboxMenuItem with label displayed next to the check toggle. The initial value of the CheckboxMenuItem is state.

Selection

public boolean getState ()

The getState() method retrieves the current state of the CheckboxMenuItem.
**public void setState (boolean condition)**

The setState() method changes the current state of the CheckboxMenuItem to condition. When true, the CheckboxMenuItem will have the toggle checked.

**public Object[] getSelectedObjects () ★**

The getSelectedItems() method returns the currently selected item as an Object array. This method, which is required by the ItemSelectable interface, allows you to use the same methods to retrieve the selected items of any Checkbox, Choice, or List. The array has at most one element, which contains the label of the selected item; if no item is selected, getSelectedItems() returns null.

**Miscellaneous methods**

**public synchronized void addNotify ()**

The addNotify() method creates the CheckboxMenuItem peer.

**public String paramString ()**

The paramString() method of CheckboxMenuItem should be protected like other paramString() methods. However, it is public, so you have access to it. When you call the toString() method of a CheckboxMenuItem, the default toString() method of MenuComponent is called. This in turn calls paramString() which builds up the string to display. At the CheckboxMenuItem level, the current state of the object is appended to the output. If the constructor for the CheckboxMenuItem was new CheckboxMenuItem("File") the results would be:

```
java.awt.CheckboxMenuItem[label=File,state=false]
```

### 10.6.2 CheckboxMenuItem Events

**Event handling**

A CheckboxMenuItem generates an ACTION_EVENT when it is selected. The argument to action() is the label of the CheckboxMenuItem, like the method provided by MenuItem, not the state of the CheckboxMenuItem as used in Checkbox. The target of the ACTION_EVENT is the Frame containing the menu. You cannot subclass CheckboxMenuItem and handle the Event within the subclass unless you override postEvent().

**Listeners and 1.1 event handling**

With the Java 1.1 event model, you register listeners, which are told when the event happens.
public void addItemListener(ItemListener listener) ★

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

public void removeItemListener(ItemListener listener) ★

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

protected void processEvent(AWTEvent e) ★

The processEvent() method receives every AWTEvent with this CheckboxMenuItem as its target. processEvent() then passes it along to any listeners for processing. When you subclass CheckboxMenuItem, overriding processEvent() allows you to process all events yourself, before sending them to any listeners. In a way, overriding processEvent() is like overriding postEvent() 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() to ensure that events are delivered, even in the absence of registered listeners.

protected void processItemEvent(ItemEvent e) ★

The processItemEvent() method receives every ItemEvent with this CheckboxMenuItem as its target. processItemEvent() then passes it along to any listeners for processing. When you subclass CheckboxMenuItem, overriding processItemEvent() allows you to process all item 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() to ensure that events are delivered even in the absence of registered listeners.

10.7 MenuBar

The MenuBar is the component you add to the Frame that is displayed on the top line of the Frame; the MenuBar contains menus. A Frame can display only one MenuBar at a time. However, you can change the MenuBar based on the state of the program so that different menus can appear at different points. The MenuBar class extends MenuComponent and implements the MenuContainer interface.
A MenuBar can be used only as a child component of a Frame. An applet cannot have a MenuBar attached to it, unless you implement the whole thing yourself. Normally, you cannot modify the MenuBar of the applet holder (the browser), unless it is Java based. In other words, you cannot affect the menus of Netscape Navigator, but you can customize appletviewer and HotJava, as shown in the following code with the result shown in Figure 10-4. The getTopLevelParent() method was introduced in Section 6.4 with Window.

```java
import java.awt.*;
public class ChangeMenu extends java.applet.Applet {
    public void init () {
        Frame f = ComponentUtilities.getTopLevelParent(this);
        if (f != null) {
            MenuBar mb = f.getMenuBar();
            Menu m = new Menu("Cool");
            mb.add(m);
        }
    }
}
```

*Figure 10–4: Customizing appletviewer’s MenuBar*

**NOTE** When you add a MenuBar to a Frame, it takes up space that is part of the drawing area. You need to get the top insets to find out how much space is occupied by the MenuBar and be careful not to draw under it. If you do, the MenuBar will cover what you draw.

### 10.7.1 MenuBar Methods

#### Constructors

*public MenuBar()*

The MenuBar constructor creates an empty MenuBar. To add menus to the MenuBar, use the add() method.
Menus

public int getMenuCount () ★
public int countMenus () ✧

The getMenuCount () method returns the number of top-level menus within the MenuBar.

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

public Menu getMenu (int index)
The getMenu () method returns the Menu at position index. If index is invalid, getMenu () throws the run-time exception ArrayIndexOutOfBoundsException.

public synchronized Menu add (Menu m)
The add () method puts choice m on the MenuBar. The label used to create m is displayed on the MenuBar. If m is already in another MenuBar, it is removed from it. The order of items added determines the order displayed on the MenuBar, with one exception: if a menu is designated as a help menu by setHelpMenu (), it is placed at the right end of the menu bar. Only a Menu can be added to a MenuBar; you can’t add a MenuItem. In other words, a MenuItem has to lie under at least one menu.

public synchronized void remove (int index)
The remove () method removes the Menu at position index from the MenuBar. If index is invalid, remove () throws the ArrayIndexOutOfBoundsException run-time exception. index is zero based.

public synchronized void remove (MenuComponent component)
This version of remove () removes the menu component from the MenuBar. If component is not in MenuBar, nothing happens. The system calls this method when you add a new Menu to make sure it does not exist on another MenuBar.

Shortcuts

public MenuItem getShortcutMenuItem (MenuShortcut shortcut) ★
The getShortcutMenuItem () method retrieves the MenuItem associated with the MenuShortcut shortcut. If MenuShortcut does not exist for this Menu, the method returns null. getShortcutMenuItem () walks through the all submenus recursively to try to find shortcut.

public synchronized Enumeration shortcuts () ★
The shortcuts () method retrieves an Enumeration of all the MenuShortcut objects associated with this MenuBar.
public void deleteShortcut (MenuShortcut shortcut) ★

The deleteShortcut() method removes MenuShortcut from the associated MenuItem in the MenuBar. If the shortcut is not associated with any menu item, nothing happens.

Help menus

It is the convention on many platforms to display help menus as the last menu on the MenuBar. The MenuBar class lets you designate one of the menus as this special menu. The physical position of a help menu depends on the platform, but those giving special treatment to help menus place them on the right. A Menu designated as a help menu doesn’t have to bear the label “Help”; the label is up to you.

public Menu getHelpMenu ()

The getHelpMenu() method returns the Menu that has been designated as the help menu with setHelpMenu(). If the menu bar doesn’t have a help menu, getHelpMenu() returns null.

public synchronized void setHelpMenu (Menu m)

The setHelpMenu() method sets the menu bar’s help menu to m. This makes m the rightmost menu on the MenuBar, possibly right justified. If m is not already on the MenuBar, nothing happens.

Peers

public synchronized void addNotify ()

The addNotify() method creates the MenuBar peer with all the menus on it, and in turn their menu items.

public synchronized void removeNotify ()

The removeNotify() method destroys the peer of the MenuBar and removes it from the screen. The peers of the items on the MenuBar are also destroyed.

10.7.2 MenuBar Events

A MenuBar does not generate any events.

10.8 Putting It All Together

Now that you know about all the different menu classes, it is time to show an example. Example 10-1 contains the code to put up a functional MenuBar attached to a Frame, using the 1.0 event model. Figure 10-2 (earlier in the chapter) displays the resulting screen. The key parts to examine are how the menus are put together in the MenuTest constructor and how their actions are handled within action(). I
implement one real action in the example: the one that terminates the application when the user chooses Quit. Any other action just displays the label of the item and (if it was a CheckBoxMenuItem) the item’s state, to give you an idea of how you can use the information returned in the event.

*Example 10–1: MenuTest 1.0 Source Code*

```java
import java.awt.*;
public class MenuTest extends Frame {
    MenuTest () {
        super("MenuTest");
        MenuItem mi;
        Menu file = new Menu("File", true);
        file.add("Open");
        file.add(mi = new MenuItem("Close"));
        mi.disable();
        Menu extras = new Menu("Extras", false);
        extras.add(new CheckboxMenuItem("What"));
        extras.add("Yo");
        extras.add("Yo");
        file.add(extras);
        file.addSeparator();
        file.add("Quit");
        Menu help = new Menu("Help");
        help.add("About");
        MenuBar mb = new MenuBar();
        mb.add(file);
        mb.add(help);
        mb.setHelpMenu(help);
        setMenuBar(mb);
        resize(200, 200);
    }
    public boolean handleEvent (Event e) {
        if (e.id == Event.WINDOW_DESTROY) {
            System.exit(0);
        }
        return super.handleEvent (e);
    }
    public boolean action (Event e, Object o) {
        if (e.target instanceof MenuItem) {
            if ("Quit".equals (o)) {
                dispose();
                System.exit(1);
            } else {
                System.out.println("User selected " + o);
                if (e.target instanceof CheckboxMenuItem) {
                    CheckboxMenuItem cb = (CheckboxMenuItem)e.target;
                    System.out.println("The value is: " + cb.getState());
                }
            }
        }
        return true;
    }
    return false;
}
```
Example 10-1: MenuTest 1.0 Source Code (continued)

```java
public static void main(String[] args) {
    MenuTest f = new MenuTest();
    f.show();
}

The MenuTest constructor builds all the menus, creates a menu bar, adds the menus to the menu bar, and adds the menu bar to the Frame. To show what is possible, I’ve included a submenu, a separator bar, a disabled item, and a help menu.

The handleEvent() method exists to take care of WINDOW_DESTROY events, which are generated if the user uses a native command to exit from the window.

The action() method does the work; it received the action events generated whenever the user selects a menu. We ignore most of them, but a real application would need to do more work figuring out the user’s selection. As it is, action() is fairly simple. If the user selected a menu item, we check to see whether the item's label was “Quit”; if it was, we exit. If the user selected anything else, we print the selection and return true to indicate that we handled the event.

10.8.1 Using Java 1.1 Events

Example 10-2 uses the Java 1.1 event model but is otherwise very similar to Example 10-1. Take a close look at the differences and similarities. Although the code that builds the GUI is basically the same in both examples, the event handling is completely different. The helper class MyMenuItem is necessary to simplify event handling. In Java 1.1, every menu item can be an event source, so you have to register a listener for each item. Rather than calling addActionListener() explicitly for each item, we create a subclass of MenuItem that registers a listener automatically. The listener is specified in the constructor to MyMenuItem; in this example, the object that creates the menus (MenuTest12) always registers itself as the listener. An alternative would be to override processActionEvent() in MyMenuItem, but then we’d also need to write a subclass for CheckboxMenuItem.

Having said all that, the code is relatively simple. MenuTest12 implements ActionListener so it can receive action events from the menus. As I noted previously, it registers itself as the listener for every menu item when it builds the interface. The actionPerformed() method is called whenever the user selects a menu item; the logic of this method is virtually the same as it was in Example 10-1. Notice, though, that we use getActionCommand() to read the label of the menu item. (Note also that getActionCommand() doesn’t necessarily return the label; you can change the
command associated with the menu item by calling setActionCommand(). Similarly, we call the event’s getSource() method to get the menu item that actually generated the event; we need this to figure out whether the user selected a CheckboxMenuItem (which implements ItemSelectable).

We override processWindowEvent() so that we can receive WINDOW_CLOSING events without registering a listener. Window closings occur when the user uses the native display manager to close the application. If one of these events arrives, we shut down cleanly. To make sure that we receive window events even if there are no listeners, the MenuTest12 constructor calls enableEvents(WINDOW_EVENT_MASK).

Example 10–2: MenuTest12 Source Code, Using Java 1.1 Event Handling

```java
// Java 1.1 only
import java.awt.*;
import java.awt.event.*;
public class MenuTest12 extends Frame implements ActionListener {
    class MyMenuItem extends MenuItem {
        public MyMenuItem (String s, ActionListener al) {
            super (s);
            addActionListener (al);
        }
    }

    public MenuTest12 () {
        super ("MenuTest");
        MenuItem mi;
        Menu file = new Menu ("File", true);
        file.add (new MyMenuItem ("Open", this));
        mi = file.add (new MyMenuItem ("Close", this));
        mi.setEnabled (false);
        Menu extras = new Menu ("Extras", false);
        mi = extras.add (new CheckboxMenuItem ("What"));
        mi.addActionListener(this);
        mi = extras.add (new MyMenuItem ("Yo", this));
        mi.setActionCommand ("Yo1");
        mi = extras.add (new MyMenuItem ("Yo", this));
        mi.setActionCommand ("Yo2");
        file.add (extras);
        file.addSeparator();
        file.add (new MyMenuItem ("Quit", this));
        Menu help = new Menu("Help");
        help.add (new MyMenuItem ("About", this));
        MenuBar mb = new MenuBar();
        mb.add (file);
        mb.add (help);
        mb.setHelpMenu (help);
        mb.setMenuBar (mb);
        setSize (200, 200);
        enableEvents (AWTEvent.WINDOW_EVENT_MASK);
    }
}
```

// Cannot override processActionEvent since method of MenuItem
// Would have to subclass both MenuItem and CheckboxMenuItem
Example 10–2: MenuTest12 Source Code, Using Java 1.1 Event Handling (continued)

    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("Quit")) {
            System.exit(0);
        }
        System.out.println("User selected " + e.getActionCommand());
        if (e.getSource() instanceof ItemSelectable) {
            ItemSelectable is = (ItemSelectable)e.getSource();
            System.out.println("The value is: " +
                (is.getSelectedObjects().length != 0));
        }
    }

    protected void processWindowEvent(WindowEvent e) {
        if (e.getID() == WindowEvent.WINDOW_CLOSING) {
            // Notify others we are closing
            super.processWindowEvent(e);
            System.exit(0);
        } else {
            super.processWindowEvent(e);
        }
    }

    public static void main(String[] args) {
        MenuTest12 f = new MenuTest12();
        f.show();
    }

I took the opportunity when writing the 1.1 code to make one additional improvement to the program. By using action commands, you can easily differentiate between the two Yo menu items. Just call `setActionCommand()` to assign a different command to each item. (I used “Yo1” and “Yo2”.) You could also differentiate between the items by saving a reference to each menu item, calling `getSource()` in the event handler, and comparing the result to the saved references. However, if the `ActionListener` is another class, it would need access to those references. Using action commands is simpler and results in a cleaner event handler.

The intent of the `setActionCommand()` and `getActionCommand()` methods is more for internationalization support. For example, you could use `setActionCommand()` to associate the command Quit with a menu item, then set the item’s label to the appropriate text for the user’s locality.

### 10.9 PopupMenu

The `PopupMenu` class is new to Java 1.1; it allows you to associate context-sensitive menus with Java components. To associate a pop-up menu with a component, create the menu, and add it to the component using the `add(PopupMenu)` method, which all components inherit from the `Component` class.
In principle, any GUI object can have a pop-up menu. In practice, there are a few exceptions. If the component's peer has its own pop-up menu (i.e., a pop-up menu provided by the run-time platform), that pop-up menu effectively overrides the pop-up menu provided by Java. For example, under Windows NT/95, a TextArea has a pop-up menu provided by the Windows NT/95 platforms. Java can't override this menu; although you can add a pop-up menu to a TextArea, you can't display that menu under Windows NT/95 with the usual mouse sequence.

10.9.1 PopupMenu Methods

Constructors

public PopupMenu()★

The first PopupMenu constructor creates an untitled PopupMenu. Once created, the menu can be populated with menu items like any other menu.

public PopupMenu(String label)★

This constructor creates a PopupMenu with a title of label. The title appears only on platforms that support titles for context menus. Once created, the menu can be populated with menu items like any other menu.

Miscellaneous methods

public void show(Component origin, int x, int y)★

Call the show() method to display the PopupMenu. x and y specify the location at which the pop-up menu should appear; origin specifies the Component whose coordinate system is used to locate x and y. In most cases, you'll want the menu to appear at the point where the user clicked the mouse; to do this, set origin to the Component that received the mouse event, and set x and y to the location of the mouse click. It is easy to extract this information from an old-style (1.0) Event or a Java 1.1 MouseEvent. In Java 1.1, the platform-independent way to say "give me the mouse events that are supposed to trigger pop-up menus" is to call MouseEvent.isPopupTrigger(). If this method returns true, you should show the pop-up menu if one is associated with the event source. (Note that the mouse event could also be used for some other purpose.)

If the PopupMenu is not associated with a Component, show() throws the runtime exception NullPointerException. If origin is not the MenuContainer for the PopupMenu and origin is not within the Container that the pop-up menu belongs to, show() throws the run-time exception IllegalArgumentException. Finally, if the Container of origin does not exist or is not showing, show() throws a run-time exception.
public synchronized void addNotify () ★

The addNotify() method creates the PopupMenu peer with all the MenuItems on it.

Example 10-3 is a simple applet that raises a pop-up menu if the user clicks the appropriate mouse button anywhere within the applet. Although the program could use the 1.0 event model, under the 1.0 model, it is impossible to tell which mouse event is appropriate to display the pop-up menu.

Example 10–3: Using a PopupMenu

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

public class PopupTest extends Applet implements ActionListener {
    PopupMenu popup;
    public void init() {
        MenuItem mi;
        popup = new PopupMenu("Title Goes Here");
        popup.add(mi = new MenuItem ("Undo");
        mi.addActionListener (this);
        popup.addSeparator();
        popup.add(mi = new MenuItem("Cut").setEnabled(false);
        mi.addActionListener (this);
        popup.add(mi = new MenuItem("Copy").setEnabled(false);
        mi.addActionListener (this);
        popup.add(mi = new MenuItem("Paste"));
        mi.addActionListener (this);
        popup.add(mi = new MenuItem("Select All"));
        mi.addActionListener (this);
        add (popup);
        resize(200, 200);
        enableEvents (AWTEvent.MOUSE_EVENT_MASK);
    }
    protected void processMouseEvent (MouseEvent e) {
        if (e.isPopupTrigger())
            popup.show(e.getComponent(), e.getX(), e.getY());
        super.processMouseEvent (e);
    }
    public void actionPerformed(ActionEvent e) {
        System.out.println (e);
    }
}