Application Menu and Pop-up List Programming Topics

User Experience: Menus

2007-06-26

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Listings

Introduction to Application Menus and Pop-up Lists

This document describes how menus and popup lists work and how you can use them.

Organization of This Document

This programming topic contains the following articles:

- "How Menus Work" (page 9) describes Cocoa's classes for managing menus.
- "Enabling Menu Items" (page 15) describes how to enable and disable menu items.
- "Using Menu Item States" (page 13) describes how to check and uncheck a menu item by changing its state.
- "Removing a Menu" (page 19) describes how to programmatically remove a menu from the menu bar.
- "Setting a Menu Item's Key Equivalent" (page 21) describes how to assign a key equivalent to a menu item.
- "Managing Pop-Up Buttons and Pull-Down Lists" (page 23) describes how to work with pop-up buttons and pull-down lists.
- "Displaying a Contextual Menu" (page 25) describes how to display a contextual menu associated with a view.
- "Views in Menu Items" (page 27) describes how you can provide a menu item with a custom view.

How Menus Work

The classes `NSMenu` and `NSMenuItem` are the basis for all types of menus. An instance of `NSMenu` manages a collection of menu items and draws them one beneath another. An instance of `NSMenuItem` represents a menu item; it encapsulates all the information its `NSMenu` object needs to draw and manage it, but does no drawing or event-handling itself. Generally, you use Interface Builder to create and modify any type of menu. However, `NSMenu` and `NSMenuItem` provide you with methods to change your application's menus dynamically.

Menu Basics

Cocoa gives you a core set of classes that handle menus no matter where they appear. Menus commonly appear in various parts of the user interface:

- The application's menu bar. This is at the top of the screen.
- A pop-up menu. This can appear anywhere in a window.
- The status bar. This begins at the right side of the menu bar (to the left of Menu Extras and the menu bar clock) and grows to the left as items are added to it.
- Contextual menus. These appear when the user right-clicks or left-Control-clicks an item.
- The Dock menu. A menu for each dock icon appears when the user right-clicks or left-Control-clicks the icon, or when the user left-presses the mouse pointer on the icon.

The classes `NSMenu` and `NSMenuItem` are the basis for all types of menus. An instance of `NSMenu` manages a collection of menu items and draws them one beneath another. An instance of `NSMenuItem` represents a menu item; it encapsulates all the information its `NSMenu` object needs to draw and manage it, but does no drawing or event-handling itself.

Menu views are capable of having one attached menu view at any given time. An attached menu view displays the contents of a submenu and is typically positioned next to the menu item with which it is associated.

`NSMenuItem` lets you set the titles, actions, targets, tags, images, enabled states, and similar attributes of individual menu items, as well as to obtain the current values of these attributes. Whenever an attribute for a menu item changes, it notifies its associated `NSMenu` with the `itemChanged:` method.

You typically use Interface Builder to create and modify any type of menu, so often there is no need to write any code. However, `NSMenu` and `NSMenuItem` provide you with methods to change your application's menus dynamically, in particular to allow you to enable and disable existing menu items (see "Enabling Menu Items" (page 15)).

Application Menus

All of an application's menus in the menu bar are owned by one `NSMenu` instance that's created by the application when it starts up. You can retrieve this main menu with the `NSApplication` method `mainMenu`.

Application menus drop down from the menu bar when the user clicks in a menu's title, and submenus appear to the right or left of their menus, depending on the available screen space.

Pop-Up Buttons and Pull-Down Lists

Pop-up buttons are implemented by the `NSPopUpButton` class. You can choose from a pop-up list or a pull-down list, with the `setPullsDown:` method:

- A pop-up list lets the user choose one option among several and generally displays the option that was last selected.

  You should use a pop-up list to select items from a medium-sized set of options, approximately 5 to 12 items. Generally, smaller lists are better handled with a group of radio buttons; and larger lists, with a scrolling list. However, if space is at a premium a pop-up list may be appropriate for other list sizes. For example, a pop-up list displaying various zoom factors can easily fit next to a scroll bar at the bottom of a window.
- A pull-down list is generally used for selecting commands in a specific context.

An `NSPopUpButton` object contains an `NSPopUpButtonCell` object. The button contains the button's data, and the cell controls the button's appearance. Generally, you'll invoke methods on the `NSPopUpButton` object, although most of the work is handled by the `NSPopUpButtonCell` instance. Most of `NSPopUpButton`'s methods are convenience methods which simply invoke the same method on the cell.

To implement its menu, the button cell contains an `NSMenu` object, which in turn contains several `NSMenuItem` objects, one for each item in the menu. Avoid invoking methods on the `NSMenu` object directly, but instead invoke methods on the `NSPopUpButton` instance, which may need to do some housekeeping before invoking the appropriate methods on the menu. However, you can retrieve the menu with the `NSPopUpButton` method `menu`. The `NSPopUpButton` methods you use most often are the methods that tell you which item is selected.

Generally, you create an `NSPopUpButton` with Interface Builder. You can define the `NSPopUpButton` object's target and action, as well as the targets and actions of each item in the button's list, programmatically or through Interface Builder. For more details about how to use Interface Builder, see *Interface Builder User Guide*.

Contextual Menus

You can attach a contextual menu to any NSView object. When the user Control-clicks on that view, the menu appears. To assign a menu to a view, use `setMenu:`, which NSView inherits from NSResponder.

Your subclass can define a menu that's used for all instances by implementing the `defaultMenu` class method. To change the menu displayed based on the mouse event, override the `menuForEvent:` instance method. This allows the view clicked to display different menus based on the location of the mouse and of the view's state, or to change or enable individual menu items based on the commands available for the view or for that region of the view.

Using Menu Item States

Menu items can have a state: on (`NSOnState`), off (`NSOffState`), or mixed (`NSMixedState`). These are useful if the menu item represents a setting in your application. The menu item automatically displays its state, as follows. If the state is `NSOnState`, a checkmark appears beside it. If the state is `NSMixedState`, a dash appears beside it. If the state is `NSOffState`, nothing appears beside it. To set the menu item's state, use `setState:`.

The mixed state is useful if the setting is true for only some items in the application or the current selection. For example, if some of the selected characters in a word processor document are italic and others are not, the Italic menu item would have a dash beside it. If the menu item is in the mixed state, then choosing it should cycle through all three states. Going back to the mixed state should leave the selection as it was.

You can use states to implement a group of mutually exclusive menu items, much like a group of radio buttons. For example, a game could have three menu items to show the level of play: Beginner, Intermediate, and Advanced. To implement a such a group, create one action message that they all use. This action message changes the appropriate setting, and then reflects that change by unchecking the currently checked item and checking the newly selected item.

In an action method that responds to all commands in the group use `setState:` to uncheck the menu item that is currently marked:

```
[curItem setState:NSOffState];
```

Then mark the newly selected command:

```
[sender setState:NSOnState];
```

Enabling Menu Items

By default, every time a user event occurs, `NSMenu` automatically enables and disables each visible menu item. You can also force a menu to update using `NSMenu`'s `update` method.

There are two ways to enable menus:

- "Automatic Menu Enabling" (page 15): `NSMenu` updates every menu item whenever a user event occurs. A menu item is enabled if `NSMenu` can find an appropriate object that responds to the menu item's action. If you want a menu item to remain disabled even though an object responds to the menu item's action, define a `validateMenuItem:` method for that object.
- "Manual Menu Enabling" (page 17): You use `setEnabled:` to enable or disable every menu item individually.

To choose a system, use `NSMenu`'s `setAutoenablesItems:` with an argument of `YES` (for automatic menu enabling) or `NO` (for manual menu enabling). Automatic menu enabling is on by default.

Automatic Menu Enabling

When you use automatic menu enabling, `NSMenu` updates the status of every menu item whenever a user event occurs. To update the status of a menu item, `NSMenu` first determines the target of the item then determines whether the target implements `validateMenuItem:` or `validateUserInterfaceItem:` (in that order). In more detail:

- If the menu item's target is set, then `NSMenu` first checks to see if that object implements the item's action method. If it does not, then the item is disabled. If the target does implement the item's action method, `NSMenu` first checks to see if that object implements `validateMenuItem:` or `validateUserInterfaceItem:` method. If it does not, then the menu item is enabled. If it does, then the enabled status of the menu item is determined by the return value of the method.
- If the menu item's target is not set (that is, if it is `nil`—typically if the menu item is connected to First Responder) and the `NSMenu` object is not a contextual menu, then `NSMenu` uses the responder chain (described in About the Responder Chain) to determine the target. If there is no object in the responder chain that implements the item's action, the item is disabled.

  If there is an object in the responder chain that implements the item's action, `NSMenu` then checks to see if that object implements the `validateMenuItem:` or`validateUserInterfaceItem:` method. If it does not, then the menu item is enabled. If it does, then the enabled status of the menu item is determined by the return value of the method.
- If the menu item's target is not set and the `NSMenu` object *is* a contextual menu, `NSMenu` goes through the same steps as before but the search order for the responder chain is different:

  1. The responder chain for the window in which the view that triggered the context menu resides, starting with the view.

2. The window itself.
3. The window's delegate.
4. The `NSApplication` object.
5. The `NSApplication` object's delegate.

Implementing validation

As noted above, before it is displayed a menu item checks to see if its target implements `validateMenuItem:` or `validateUserInterfaceItem:`. If it does, then the enabled status of the menu item is determined by the return value of the method. You can therefore conditionally enable or disable a menu item by implementing either of these methods in the menu's target (see "Choosing validateMenuItem: or validateUserInterfaceItem:" (page 17) to determine which is the most appropriate). The implementation strategy is the same whichever you choose:

1. To decide whether or not an item should be enabled, you need to know what it will do if the user selects it. You typically first therefore check to see what *action* is associated with the item (you need to test for each of the actions you're interested in).

   Checking the action rather than, say, the title makes sure that: (a) your code works with different localizations and is robust against changes in the title due to changes in the user interface, and (b) you avoid the fragility of having to remember to use the same tag for each user interface element that invokes the same method on the target. (In the case of `validateUserInterfaceItem:`, you can only check the action or the tag.)

2. If the action is something you're interested in, then return a Boolean value appropriate for the current context.
3. If the action is not something you're interested in, then either:

   a. If your superclass implements the validation method (for example, `NSDocument` and `NSObjectController` implement `validateUserInterfaceItem:`, and `NSObjectController` implements `validateMenuItem:`), invoke super's implementation; otherwise

   b. Return a default value (typically `YES`).

The following example illustrates the implementation of `validateUserInterfaceItem:` in a subclass of `NSDocument`.

```
- (BOOL)validateUserInterfaceItem:(id <NSValidatedUserInterfaceItem>)anItem
{
    SEL theAction = [anItem action];

    if (theAction == @selector(copy:))
    {
        if ( /* there is a current selection and it is copyable */ )
        {
            return YES;
        }
```

```
        return NO;
    } else if (theAction == @selector(paste:))
    {
        if ( /* there is a something on the pasteboard we can use and
                the user interface is in a configuration in which it makes sense
 to paste */ )
        {
            return YES;
        }
        return NO;
    } else
        /* check for other relevant actions ... */
    }
    // subclass of NSDocument, so invoke super's implementation
    return [super validateUserInterfaceItem:anItem];
}
```

Choosing validateMenuItem: or validateUserInterfaceItem:

In general, you should use `validateUserInterfaceItem:` instead of `validateMenuItem:` since the former will also work for toolbar items which have the same target and action. If, however, there is additional work that you want to do that is specific to a menu item, use `validateMenuItem:`—for example, `validateMenuItem:` is also a good place to toggle titles or set state on menu items to make sure they're always correct.

Here is an example of using `validateUserInterfaceItem:` to override automatic enabling. If your application has a Copy menu item that sends the `copy:` action message to the first responder, that menu item is automatically enabled any time an object that responds to `copy:`, such as an `NSText` object, is the first responder of the key or main window. If you create a class whose instances might become the first responder, and which doesn't support copying of everything it allows the user to select, you should implement `validateUserInterfaceItem:` in that class. `validateUserInterfaceItem:` should then return `NO` if items that can't be copied are selected (or if no items are selected) and `YES` if all items in the selection can be copied. By implementing `validateUserInterfaceItem::`, you can have the Copy menu item disabled even though the target object does implement the `copy:` method. If a class never permits copying, then you simply omit an implementation of `copy:` in that class, and the Copy menu item is disabled automatically whenever an instance of that class is the first responder.

Manual Menu Enabling

When you use manual menu enabling, you use `setEnabled:` to enable or disable every menu item individually. None of the menu items, even those controlled by Application Kit classes like `NSTextView`, are updated automatically.

To turn on manual menu enabling, use `NSMenu`'s `setAutoenablesItems:` with an argument of `NO`.

Important: If you send a `setEnabled:` message when automatic updating is on, other objects might undo what you have done after another user event occurs. Hence you can never be sure that the menu item will remain the way you set it.

Removing a Menu

To remove a menu item from a menu, you send `removeItem:` or `removeItemAtIndex:` to the `NSMenu` object managing the menu item.

To remove an entire menu from the menu bar, you use the same technique. The menus in the menu bar are themselves items of another menu: the root menu, or main menu. To get the main menu, send `mainMenu` to `NSApp`, the global application instance. Then send `removeItem:` to the main menu; or find the index of the menu to be removed and send `removeItemAtIndex:` to the main menu. Listing 1 illustrates the latter procedure.

Listing 1 Removing a menu from the menu bar

```
- (IBAction)removeMenu:(id)sender {
    NSMenu* rootMenu = [NSApp mainMenu];
    // sender is an NSMenuItem
   [rootMenu removeItemAtIndex:[rootMenu indexOfItemWithSubmenu:[sender menu]]];
}
```

Setting a Menu Item's Key Equivalent

You can assign a keyboard equivalent to an NSMenuItem, so that when the user types a character the menu item sends its action. The keyboard equivalent is defined in two parts. First is the basic key equivalent, which must be a Unicode character that can be generated by a single key press without modifier keys (Shift excepted). It is also possible to use a sequence of Unicode characters so long as the user's key mapping is able to generate the sequence with a single key press. The basic key equivalent is set using `setKeyEquivalent:` and returned by `keyEquivalent`. The second part defines the modifier keys that must also be pressed. This is set using `setKeyEquivalentModifierMask:` and returned by `keyEquivalentModifierMask`. The modifier mask by default includes `NSCommandKeyMask`, and may also include the masks for the Shift, Option, or other modifier keys. Specifying keyboard equivalents in two parts allows you to define a modified keyboard equivalent without having to know which character is generated by the basic key plus the modifier. For example, you can define the keyboard equivalent Command-Option-f without having to know which character is generated by typing Option-f.

Note: To specify the Option key, use the constant `NSAlternateKeyMask`.

Certain methods in NSMenuItem can override assigned keyboard equivalents with those the user has specified in the defaults system. The `setUsesUserKeyEquivalents:` method turns this behavior on or off, and `usesUserKeyEquivalents` returns its status. To determine the user-defined key equivalent for an NSMenuItem object, invoke the `userKeyEquivalent` instance method. If user-defined key equivalents are active and an NSMenuItem object has a user-defined key equivalent, its `keyEquivalent` method returns the user-defined key equivalent and not the one set using `setKeyEquivalent:`.

Managing Pop-Up Buttons and Pull-Down Lists

Pop-up buttons and pull-down lists are both implemented by the `NSPopUpButton` class. You can choose from a pop-up list or a pull-down list, with the `setPullsDown:` method. A pop-up list lets the user choose one option among several and generally displays the option that was last selected. A pull-down list is generally used for selecting commands in a specific context.

Pop-Up Buttons

A pop-up button displays several options and generally displays the option that was last selected. The following illustration shows a pop-up button before and while its menu is displayed. (Note that "Language:" isn't part of the pop-up button.)

When the pop-up menu is displayed, the pop-up button's cell displays the list on top of the popup button. The currently selected choice appears in the same location as the popup button with the other items arranged above or below the current selection according to the order of the cell's menu. When the popup list is dismissed, the title of the popup button changes to match the title of the currently selected item.

Pull-Down Lists

You generally use a pull-down for selecting commands in a specific context. The following illustration shows a pull-down list before and while its menu is displayed.

You can think of a pull-down list as a compact form of menu. A pull-down list's display isn't affected by the user's actions, and a pull-down list usually displays the first item in the list. When the commands only make sense in the context of a particular display, you can use a pull-down list in that display to keep the related actions nearby and to keep them out of the way when that display isn't visible.

Pulldown lists typically display themselves adjacent to the popup button in the same way a submenu is displayed next to its parent item. Unlike popup lists, the title of a popup button displaying a pulldown list is not based on the currently selected item and thus remains fixed unless you change using the cell's `setTitle:` method.

While popup lists are displayed right over their button, the location of the pulldown list depends on the preferred edge, set through the cell's `setPreferredEdge:` method. By default, the list appears under the cell. When drawing the list, the cell does its best to honor the preferred edge if there is enough space on the user's screen. If there is not enough space, the cell positions the list somewhere else.

Note that in a pull-down list, the first item is stored at index 1, not index 0 as is the case with pop-up lists or ordinary menus. This is done so that the pull-down list's title can be stored at index 0 if necessary. Even when the title is stored at index 0, always change the buttons title with the `setTitle:` method.

Managing Pop-Up Button Items

In many cases, it is easiest to populate a pop-up button using Cocoa bindings (see *Cocoa Bindings Programming Topics*). You typically bind the `contentObjects` value of the button to the `arrangedObjects` of an array controller, and the `contentValues` value of the button to the same path but adding as the final path component the key of the attribute you want to use as the button title. For more details, see NSPopUpButton Bindings.

You can also modify a pop-up button programmatically. To add items to a pop-up button cell's list, use the methods `addItemWithTitle:`, `addItemsWithTitles:`, and `insertItemWithTitle:atIndex:`. These methods create or replace a menu item with the given title and assign it a default action and key equivalent. Once the item is added to the list, use `NSMenuItem` methods to modify the attributes of the item. To remove one or more items, use the `removeItemWithTitle:`, `removeItemAtIndex:`, or `removeAllItems` method.

Pop-Up Button Actions

There are two ways to assign an action method to a pop-up button. You can assign actions to each menu item in the pop-up. Or you can assign an action directly to the pop-up button, and it's invoked whenever the user selects any menu item that doesn't have its own action. The second way is especially useful with pop-up lists.

Here's a sample action for a pop-up menu. It's part of a controller object that sets an instance variable named `language` to a constant corresponding to the item the user has chosen. You use Interface Builder to create the `NSPopUpButton` object, configure it as a pop-up list, and add items to it—setting the tags of the menu items to equal the corresponding value in the `enum`. The code you write might look like this:

```
typedef enum _languageValue
{
     English = 1, French, German, Spanish, Swedish
} languageValue;

- (void)setLanguageFrom:(id)sender
{
    [self setLanguage:[[sender selectedItem] tag]];
}
```

Displaying a Contextual Menu

The Application Kit interprets right-mouse-down events and left-mouse-down events modified by the Control key as commands to display a contextual menu for the clicked view. Your view subclasses have several alternative approaches for displaying a contextual menu. If the view's menu is to remain unchanged regardless of context, you can do one of three simple procedures:

- **Configure in Interface Builder**: Add a standalone (rootless) menu to a nib file and customize it to suit, including the specification of targets and actions. Then connect it to your custom view's `menu` outlet, which is inherited from `NSView`.
- **Programmatically assign a generic menu**: Override the `defaultMenu` class method of `NSView` to create and return a menu that's common to all instances of your subclass. (See Listing 1 (page 25) for a sample implementation of this method.) This default menu is also accessible via the `NSResponder menu` method unless some other `NSMenu` object has been associated with the view.
- **Programmatically assign an instance-specific menu**: In the custom view's `initWithFrame:` or `awakeFromNib` methods, create the menu and associate it with the view by using the `setMenu:` method (`NSResponder`).

After you complete any of these procedures, the Application Kit displays the contextual menu whenever the user left-Control-clicks or right-clicks the view. Note that the Application Kit automatically also validates the menu items of contextual menus, unless you request it not to.

However, you might want the view's contextual menu to change based on where the mouse click occurs in the view or on the current state of the view; you may want to add, delete, enable, or disable menu items or change some item attributes to reflect the current context. There is more than one way to accomplish this, but a good approach is to override the `defaultMenu` and `menuForEvent:` methods of `NSView`. In the former method implementation, create and return an `NSMenu` object that is the "base" contextual menu, suitable for most contexts. Be sure to create menu items for the menu that have all necessary attributes (including action selector and possibly target object). Listing 1 shows how you might do this.

Listing 1 Returning the default menu

```
+ (NSMenu *)defaultMenu {
    NSMenu *theMenu = [[[NSMenu alloc] initWithTitle:@"Contextual Menu"]
autorelease];
   [theMenu insertItemWithTitle:@"Beep" action:@selector(beep:) keyEquivalent:@""
 atIndex:0];
   [theMenu insertItemWithTitle:@"Honk" action:@selector(honk:) keyEquivalent:@""
 atIndex:1];
    return theMenu;
}
```

For contextual-menu events, the Application Kit invokes the `menuForEvent:` method if your view subclass implements it. In your implementation of this method, test for a certain condition (event type, mouse location, view state, and so on) and if that condition holds modify and return the default menu. Otherwise, return the default menu unchanged. Listing 2 gives an example that tests for a mouse click in a certain area and returns a modified contextual menu if that test holds true.

Listing 2 Displaying a dynamically modified contextual menu

```
- (NSMenu *)menuForEvent:(NSEvent *)theEvent {
    NSPoint curLoc = [self convertPoint:[theEvent locationInWindow] fromView:nil];
     NSRect magic_square = NSMakeRect(0.0, 0.0, 10.0, 10.0);

    if ([self mouse:curLoc inRect:magic_square]) {
        NSMenu *theMenu = [[self class] defaultMenu];
        [theMenu insertItemWithTitle:@"Wail" action:@selector(wail:)
keyEquivalent:@"" atIndex:[theMenu numberOfItems]-1];
        return theMenu;
    }
    return [[self class] defaultMenu];
}
```

If you want your view to display a contextual menu in response to events other than right-mouse clicks and left-mouse-Control clicks, you can directly handle the event message in the appropriate `NSResponder` method. For example, if you want users to be able to left-click an image view to get a menu of export options, you would override the `mouseDown:` method. In your implementation of the method, create a menu and then invoke the `NSMenu` class method `popUpContextMenu:withEvent:forView:`, passing in the event object related to the mouse-down event and the view owning the contextual menu. Listing 3 illustrates this approach.

Listing 3 Displaying a contextual menu upon receiving a left-mouse event

```
- (void)mouseDown:(NSEvent *)theEvent {

    NSMenu *theMenu = [[[NSMenu alloc] initWithTitle:@"Contextual Menu"]
autorelease];
   [theMenu insertItemWithTitle:@"Beep" action:@selector(beep:) keyEquivalent:@""
 atIndex:0];
   [theMenu insertItemWithTitle:@"Honk" action:@selector(honk:) keyEquivalent:@""
 atIndex:1];

    [NSMenu popUpContextMenu:theMenu withEvent:theEvent forView:self];
}
```

Contextual menus, including any menu you pop up with `popUpContextMenu:withEvent:forView:`, automatically insert menu items from any contextual menu plug-ins that the user has installed into the menu. A contextual menu plug-in, which is `CFPlugIn` bundle installed in a `Library/Contextual Menu Items` directory at the appropriate level of the system, enables applications and other forms of software to extend the list of commands found on contextual menus such as the Finder's. The applications do not have to be running for their items to appear. If you are trying to programmatically display a menu, you might not want those items to appear. The preferred approach for programmatically displaying a non-contextual menu is to create an `NSPopUpButtonCell` object, set its menu, and then call send a `attachPopUpWithFrame:inView:` message to the pop-up button cell.

Note: For information on how to create a `CFPlugIn` plug-in, see *Plug-ins*. For information on the Carbon Menu Manager functions you must implement for a contextual menu plug-in, see *Menu Manager Reference*.

Views in Menu Items

In Mac OS X v10.5, the `NSMenuItem` class is enhanced to allow you to include views in the menu item. This article describes the feature.

In Mac OS X v10.5, you can set the view for a menu item using `NSMenuItem`'s `setView:` method (by default, a menu item has a `nil` view)—note, though, that menu item views are not supported in the Dock menu. The following code fragment illustrates how you can create a new menu and add it to your application's menu bar

```
NSMenuItem* menuBarItem = [[NSMenuItem alloc]
                        initWithTitle:@"Custom" action:NULL keyEquivalent:@""];
// title localization is omitted for compactness
NSMenu* newMenu = [[NSMenu alloc] initWithTitle:@"Custom"];
[menuBarItem setSubmenu:newMenu];
[newMenu release];
[[NSApp mainMenu] insertItem:menuBarItem atIndex:3];
[menuBarItem release];

/*
 Assume that myView1 and myView2 are existing view objects;
 for example, you may have created them in a NIB file.
 */
NSMenuItem* newItem;
newItem = [[NSMenuItem alloc]
               initWithTitle:@"Custom Item 1"
               action:@selector(menuItem1Action:)
               keyEquivalent:@""];
[newItem setView: myView1];
[newItem setTarget:self];
[newMenu addItem:newItem];
[newItem release];

newItem = [[NSMenuItem alloc]
               initWithTitle:@"Custom Item 2"
               action:@selector(menuItem2Action:)
               keyEquivalent:@""];
[newItem setView: myView2];
[newItem setTarget:self];
[newMenu addItem:newItem];
[newItem release];
```

A menu item with a view does not draw its title, state, font, or other standard drawing attributes, and assigns drawing responsibility entirely to the view. Keyboard equivalents and type-select continue to use the key equivalent and title as normal.

A view in a menu item can receive all mouse events as normal, but keyboard events are not supported. During "non-sticky" menu tracking (that is, manipulating menus with the mouse button held down), a view in a menu item receives `mouseDragged:` events.

You can add animation to a menu item view as you would any other view (you set a timer to call `setNeedsDisplay:` or `display`), but because menu tracking occurs in the `NSEventTrackingRunLoopMode`, you must add the timer to the run loop in that mode.

When the menu is opened, the view is added to a window; when the menu is closed the view is removed from the window. If you are using a custom view, you can therefore override `viewDidMoveToWindow` as a convenient place to start or stop animation, reset tracking rectangles and so on, but you should not attempt to move or otherwise modify the window

A menu item with a view sizes itself according to the view's frame, and the width of the other menu items. The menu item will always be at least as wide as its view, but it may be wider. If you want your view to auto-expand to fill the menu item, then make sure that its autoresizing mask has `NSViewWidthSizable` set. A menu item will resize itself as the view's frame changes, but resizing during menu tracking is not supported.

When a menu item is copied using `NSCopying`, any attached view is copied using archiving and unarchiving.

Document Revision History

This table describes the changes to *Application Menu and Pop-up List Programming Topics*.

Date	Notes
2007-06-26	Added article to describe use of views in menus in Mac OS X v10.5.
2006-12-05	Removed mention of deprecated classes; added articles on removing menus from menu bar and displaying a contextual menu.
2006-06-28	Revised the discussion of Enabling Menu Items.
2004-06-28	Reorganized introduction. Corrected typos.
2003-04-01	Clarified the search order for context menus in "Enabling Menu Items" (page 15).
2003-02-20	Corrected errors in the Java sample code in "How Pop-Up Lists Work".
2002-11-12	Revision history was added to existing topic.