Toolbar Programming Topics for Cocoa

User Experience: Controls

2009-02-04

Contents

Figures and Listings

Introduction to Toolbars

NSToolbar and NSToolbarItem provide you with a standard way to display a toolbar for a titled window below its title bar. These classes also provide users with a standard way to customize toolbars and save those customizations.

Organization of This Document

This programming topic describes how to use toolbars. It contains the following articles:

- "How Toolbars Work" (page 9) gives basic information on toolbars.
- "Toolbar Management Checklist" (page 13) describes what happens when you create a toolbar.
- "Adding and Removing Toolbar Items" (page 17) describes managing toolbar items, and the role of the NSToolbar delegate.
- "Setting a Toolbar Item's Representation" (page 27) describes the different ways a toolbar item can be represented and how to set them.
- "Validating Toolbar Items" (page 29) describes how to set whether a toolbar item is clickable or not.
- "Setting a Toolbar Item's Size" (page 31) describes how to set a toolbar item's minimum and maximum size.
- "Selectable Toolbar Items" (page 33) describes how to support selectable toolbar items.
- "Subclassing NSToolbarItem" (page 35) describes the methods to override when subclassing NSToolbarItem.
- "Techniques for Toolbar Management" (page 37) describes how to determine if a toolbar has items in the overflow menu and how to calculate toolbar height.

See Also

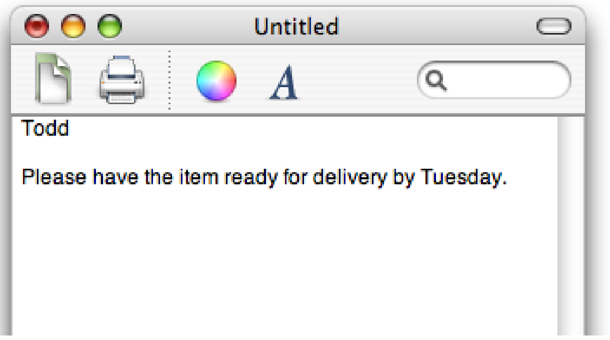
The following sample code is available through Apple Developer Connection:

- *iSpend* shows how to add basic toolbar support to an application.
- *ToolbarSample* implements an example toolbar including using custom views in a toolbar item.

How Toolbars Work

`NSToolbar` and `NSToolbarItem` classes provide you with a standard way to display a toolbar for a titled window below its title bar. These classes also provide users with a standard way to customize toolbars and save those customizations. This is what a toolbar looks like:

Figure 1 Example toolbar



Todd

Please have the item ready for delivery by Tuesday.

To create a toolbar, you must create a delegate that provides important information:

- A list of default toolbar identifiers. This list is used when reverting to default, and constructing the initial toolbar.

  The default set of toolbar items can also be specified using toolbar items found in the Interface Builder library.
- A list of allowed item identifiers. The allowed item list is used to construct the customization palette, if the toolbar is customizable.
- The toolbar item for a given item identifier.

When you create an NSToolbar, you give it an identifier. NSToolbar assumes all toolbars with the same identifier are the same, and automatically synchronizes changes. For instance, take a mail application. Your application's compose windows' toolbars would have the same identifier string. So, when you re-order items in one toolbar, the changes automatically propagate to any other compose windows currently open.

Most toolbars will contain simple clickable items that act like buttons. The simplest toolbar item is defined by its icon, label, palette label (used in the customize sheet), target, action, and tooltip. Most toolbars can be represented using these simple items. If you need something more complex in your toolbar, call `setView:` on a toolbar item to provide a custom view. For example, you could create a toolbar item that contains a pop-up menu or a text field and button.

Note: In Mac OS X v10.4 and earlier versions of the operating system, only toolbar items that are pop-up buttons can have key equivalents. In Mac OS X v10.5 and later, all visible controls in a toolbar can have key equivalents.

There are a couple of standard toolbar item identifiers that NSToolbar knows about. `NSToolbarSeparatorItemIdentifier` is the standard vertical line separator. `NSToolbarSpaceItemIdentifier` is a fixed width space. `NSToolbarFlexibleSpaceItemIdentifier` is a variable width space. Additionally, there are `NSToolbarShowColorsItemIdentifier`, `NSToolbarShowFontsItemIdentifier`, `NSToolbarPrintItemIdentifier`, and `NSToolbarCustomizeToolbarItemIdentifier`. These items are accessible only by identifier.

If you need to change the action sent by a standard item, write a `toolbarWillAddItem:` notification method.

When a user customizes the toolbar of an application's window, that customization is saved as a user preference. The customized toolbar is used thereafter when the application launches in place of the default set of toolbar items specified by the developer.

Toolbar configurations

There are kinds of toolbars, and there are individual toolbar objects. A kind of toolbar is represented by string called the toolbar identifier. When you create an NSToolbar you supply a toolbar identifier so your toolbar delegate will know that the new instance is to be of that kind. For example, consider a Mail application that has two kinds of windows, Message and Mailbox. Each needs its own kind of toolbar: one appropriate for viewing a message and one appropriate for listing the messages in a mailbox. To implement this user interface, the application has two toolbar identifiers, `"Message"`, and `"Mailbox"`. Each message window has its own distinct toolbar object, but all message window toolbar objects have the same identifier: `"Message"`. When you modify the toolbar in a window (either through the user interface or programmatically) you change the toolbar configuration of that *kind* of toolbar, not just of that instance. So, when you customize the toolbar in one message window, the new configuration automatically propagates to the toolbars in all other message windows currently open. If the toolbar is hidden in any message window, it will stay hidden, but when it is shown again, it looks like the others of its kind.

The toolbar's identifier is fixed at object creation, but you can change other attributes of it with `setAllowsUserCustomization:`, `setAutosavesConfiguration:`, and `setDisplayMode:`.

Toolbar Items

Each item in an NSToolbar is an instance of NSToolbarItem. The visible parts of a toolbar item are its content, its text label, and its menu form representation.

The toolbar item's content is either an NSImage or an NSView. An item whose content is an NSImage is called an image item. An item whose content is an NSView is called a view item. The search field shown in Figure 1 is a view item, the others are image items.

Print is a standard item provided by NSToolbar. The other items are custom items, supplied by this application. Blue text is a custom image item. Font Style and Font Size are custom view items.

The menu representation is used at two different times: when the toolbar is displayed using labels only and when the window is too small to show the complete toolbar and some items are displayed in an overflow menu.

NSWindow and the toolbar

NSWindow has a number of methods to support NSToolbar: The methods `toolbar` and `setToolbar:` are for attaching a toolbar to the window; the `toggleToolbarShown:` method is the action for the Hide Toolbar / Show Toolbar menu item; `runToolbarCustomizationPalette:` is the action method for the Customize Toolbar menu item.

Interface Builder predefines an application's View menu with Show Toolbar and Customize Toolbar menu items and connects these items to `toggleToolbarShown:` and `runToolbarCustomizationPalette:` through the First Responder. NSWindow validates both toolbar menu items and toggles the title of the former menu item between "Show Toolbar" and "Hide Toolbar" to correspond with the actual state of the toolbar.

Toolbar Management Checklist

Before you begin coding:

- If you have image-based toolbar items, find or create the images (in the proper size and aspect ratio) and add them to your project as a resource.
- If you have view-based toolbar items, create each view in Interface Builder, specify an outlet for a custom controller object, and connect the view to the outlet.

  If the view is an off-the-shelf palette object, just make an outlet connection.
- Specify or (for default toolbar items) identify the unique string identifiers that you intend to use for toolbars and toolbar items.
- Add to an application menu (usually named View) the menu items Show Toolbar and Customize Toolbar..., connect these to the First Responder icon in the nib file window, and select the actions `toggleToolbarShown:` and `runToolbarCustomizationPalette:`. (Note that `NSWindow` automatically changes "Show Toolbar" to "Hide Toolbar" as appropriate.)

For further information, see "Adding and Removing Toolbar Items" (page 17).

Note: You can create a toolbar in Interface Builder as described in "Creating a Toolbar in Interface Builder" (page 23). The toolbar can have a default set and allowed set of toolbar items. If you use Interface Builder for this purpose, you may ignore many of the programmatic steps described below. At runtime, you can combine toolbars and toolbar items unarchived from a nib file and those created programmatically.

What happens: The application launches or a document is created or opened, causing a nib file to be loaded and its object unarchived.

- A custom controller class in `awakeFromNib` or, for document-based applications, an `NSDocument` subclass in `windowControllerDidLoadNib:` completes the following steps for each toolbar it uses:

  1. It makes a new `NSToolbar` object using `initWithIdentifier:`.
  2. It sets attributes of the toolbar if the defaults won't do using `setAllowsUserCustomization:`, `setAutosavesConfiguration:`, and `setDisplayMode:`.
  3. It sets the toolbar's delegate (usually itself).
  4. It associates the toolbar with a window by sending the NSWindow object a `setToolbar:` message.

For further information see "Adding and Removing Toolbar Items" (page 17)

What happens: The `NSToolbar` object begins communicating with its delegate in order to populate the toolbar with toolbar items.

1. The window gets the allowed and default toolbar item identifiers:

- The toolbar object calls the delegate method `toolbarAllowedItemIdentifiers:` to get the total set of possible toolbar items.
- Unless it finds the default toolbar configuration in user preferences, the toolbar calls the delegate method `toolbarDefaultItemIdentifiers:` to get the default set.

  To have the default configuration saved to and read from user preferences, the `NSToolbar` object's `autosavesConfiguration` attribute must be set.
- If certain toolbar items should indicate a selected state, the delegate should implement `toolbarSelectableItemIdentifiers:` to return the identifiers of those toolbar items.

2. The window asks for each `NSToolbarItem` object (by identifier) to insert into the toolbar.

- To add each toolbar item to the toolbar, the `NSToolbar` object sends `toolbar:itemForItemIdentifier:willBeInsertedIntoToolbar:` to the delegate.

  If the `NSToolbarItem` object is image-based, get the image from the application bundle (for example, by using the `NSImage` class method `imageNamed:`) and send `setImage:` to the toolbar item. Also set the toolbar item's label, palette label, target, and action. You may also set a menu form representation.

  If the toolbar item is view-based, send `setView:` to the toolbar-item object, passing in the outlet to the view. Also set the toolbar item's label, palette label, its minimum size (`minSize`), and its maximum size (`maxSize`). (If you do not set a `minSize` and `maxSize`, the view does not appear because it is sized to zero in both dimensions.) You may also set a menu form representation.
- If the delegate wants to customize a toolbar item before it is added, it can also implement the `toolbarWillAddItem:` notification method.

For further information see "Adding and Removing Toolbar Items" (page 17), "Setting a Toolbar Item's Representation" (page 27), "Setting a Toolbar Item's Size" (page 31) and "Setting a Toolbar Item's Size" (page 31).

What happens: Users click toolbar items; the runtime context of the application changes.

- Declare and implement action methods for each of your custom toolbar items, usually in a custom controller class.

  When you create a toolbar item you can identify the selector of each of these methods through the `setAction:` method of `NSToolbarItem`. Also set the target by calling the `setTarget:`, usually passing in `self`.
- Validate toolbar items.

  If the toolbar item is image-based, the target of an action should implement `validateToolbarItem:` if it wants validation more specialized than the default. If the toolbar item is view-based, you should create a subclass of `NSToolbarItem` for the item and override the `validate` method.

For further information, see "Validating Toolbar Items" (page 29).

What happens: The user chooses the Customize Toolbar menu item.

- As the customization sheet opens, the toolbar object calls the delegate methods `toolbarAllowedItemIdentifiers:` and `toolbarDefaultItemIdentifiers:`. Then as the toolbar adds each toolbar item to the customization palette, it sends to the delegate if the item kind is custom image or custom view.
- When the user adds an item to the toolbar, the toolbar invokes the delegate method `toolbar:itemForItemIdentifier:willBeInsertedIntoToolbar:`; if a new instance of the toolbar item is needed, the toolbar sends `toolbarWillAddItem:` to the delegate just before it adds the item.
- Just after the user removes an item from the toolbar, the toolbar sends `toolbarDidRemoveItem:` to the delegate.
- When the user drags the default set to the toolbar, the toolbar reuses as many items already in the toolbar as possible, calling `toolbarDidRemoveItem:` for the items it needs to remove and calling `toolbarWillAddItem:` for the ones it needs to add.

Note that the toolbar does not call any delegate methods when the user closes the customization sheet.

Adding and Removing Toolbar Items

The delegate of an NSToolbar instance is responsible for providing the identifiers specifying the toolbar items that are allowed in a toolbar, as well as the default items visible in a new toolbar.

Allowed and default toolbar items

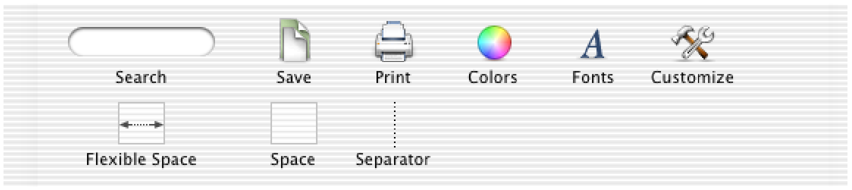
The required delegate method `toolbarAllowedItemIdentifiers:` returns an array of identifiers specifying the toolbar items that can be displayed by the specified NSToolbar instance. By default, a new toolbar instance does not assume any items are allowed, not even the separator item.

The example implementation shown in Listing 1 configures the toolbar to allow a selection of the standard Cocoa toolbar items as well as two application specific toolbar items. The resulting toolbar is shown in Figure 1.

Listing 1 Example toolbarAllowedItemIdentifiers: method implementation

```
- (NSArray *) toolbarAllowedItemIdentifiers: (NSToolbar *) toolbar {
    return [NSArray arrayWithObjects: SaveDocToolbarItemIdentifier,
        NSToolbarPrintItemIdentifier,
        NSToolbarShowColorsItemIdentifier,
        NSToolbarShowFontsItemIdentifier,
        NSToolbarCustomizeToolbarItemIdentifier,
        NSToolbarFlexibleSpaceItemIdentifier,
        NSToolbarSpaceItemIdentifier,
        NSToolbarSeparatorItemIdentifier, nil];
}
```

Figure 1 Example toolbar item configuration



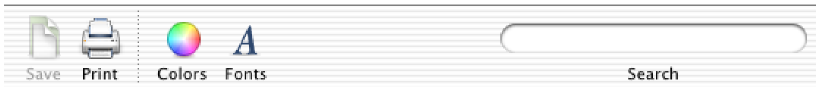
The default set of toolbar items is returned by implementing the required method `toolbarDefaultItemIdentifiers:`. The implementation is expected to return an array of identifiers containing the specified toolbar's default items. If during the toolbar's initialization no overriding values are found in the user defaults (by having called `setAutosavesConfiguration:` with `YES` as the argument) these items are used as the default set for the toolbar. In addition, if the user chooses to revert to the default toolbar items the set returned by `toolbarDefaultItemIdentifiers:` will be used.

The example code in Listing 2 causes the default items to be configured as shown in Figure 2.

Listing 2 Example toolbarDefaultItemIdentifiers: method implementation

```
- (NSArray *) toolbarDefaultItemIdentifiers: (NSToolbar *) toolbar {
    return [NSArray arrayWithObjects: SaveDocToolbarItemIdentifier,
        NSToolbarPrintItemIdentifier,
        NSToolbarSeparatorItemIdentifier,
        NSToolbarShowColorsItemIdentifier,
        NSToolbarShowFontsItemIdentifier,
        NSToolbarFlexibleSpaceItemIdentifier,
        NSToolbarSpaceItemIdentifier,
        SearchDocToolbarItemIdentifier, nil];
}
```

Figure 2 Example toolbar items configuration



Creating new toolbar items when requested

An implementation of the required method `toolbar:itemForItemIdentifier:willBeInsertedIntoToolbar:` returns an NSToolbarItem instance as specified by the toolbar item identifier. The item may not be immediately added to the toolbar, as this method is also called when the customization sheet is created.

The example code in Listing 3 creates a simple NSToolbarItem instance for the application's custom toolbar item, `SaveDocToolbarItemIdentifier`. This item appears as an icon in the toolbar using the default NSToolbarItem rendering.

Listing 3 A `toolbar:itemForItemIdentifier:willBeInsertedIntoToolbar:` method implementation to create a simple NSToolbarItem instance

```
- (NSToolbarItem *) toolbar:(NSToolbar *)toolbar
      itemForItemIdentifier:(NSString *)itemIdentifier
  willBeInsertedIntoToolbar:(BOOL)flag
{
   NSToolbarItem *toolbarItem = [[[NSToolbarItem alloc] initWithItemIdentifier:
 itemIdentifier] autorelease];

    if ([itemIdentifier isEqual: SaveDocToolbarItemIdentifier]) {
    // Set the text label to be displayed in the
    // toolbar and customization palette
    [toolbarItem setLabel:@"Save"];
    [toolbarItem setPaletteLabel:@"Save"];

    // Set up a reasonable tooltip, and image
    // you will likely want to localize many of the item's properties
    [toolbarItem setToolTip:@"Save Your Document"];
    [toolbarItem setImage:[NSImage imageNamed:@"SaveDocumentItemImage"]];

    // Tell the item what message to send when it is clicked
    [toolbarItem setTarget:self];
    [toolbarItem setAction:@selector(saveDocument:)];
    } else  {
    // itemIdentifier referred to a toolbar item that is not
    // provided or supported by us or Cocoa
    // Returning nil will inform the toolbar
    // that this kind of item is not supported
    toolbarItem = nil;
    }
    return toolbarItem;
}
```

Returning a toolbar item that creates a custom view is somewhat more complicated. In addition to specifying the information required for a simple toolbar item, the application must also explicitly set the view that should be used to draw the item, the minimum size that the view requires to display properly, and the maximum size that the custom toolbar item can be resized.

Note: As of Mac OS X v10.5, if you call `setView:` on an NSToolbarItem object without also calling `setMinSize:` or `setMaxSize:`, the toolbar item sets its minimum and maximum size equal to the view's frame.

One side effect of using a custom view for a toolbar item that must be considered is the case when the toolbar is being displayed as text-only. The default behavior is to display the item's label as disabled text, effectively disabling the toolbar item entirely. A toolbar item can instead specify an NSMenu instance that will be used when the toolbar is displayed in the text-only mode. This menu is also used when a toolbar item that uses a custom view is displayed in a toolbar's overflow menu.

The example code in Listing 4 implements the search toolbar item as shown in Figure 1 (page 17).

Listing 4 A `toolbar:itemForItemIdentifier:willBeInsertedIntoToolbar:` method implementation to create an view- based NSToolbarItem instance

```
- (NSToolbarItem *) toolbar:(NSToolbar *)toolbar
      itemForItemIdentifier:(NSString *)itemIdentifier
  willBeInsertedIntoToolbar:(BOOL)flag
{
    NSToolbarItem *toolbarItem = [[[NSToolbarItem alloc]
initWithItemIdentifier:itemIdentifier] autorelease];

    if ([itemIdentifier isEqual: SearchDocToolbarItemIdentifier]) {
    // Set up the standard properties
    [toolbarItem setLabel:@"Search"];
    [toolbarItem setPaletteLabel:@"Search"];
    [toolbarItem setToolTip:@"Search Your Document"];

    // Use a custom view, a rounded text field,
    // attached to searchFieldOutlet in InterfaceBuilder as
    // the custom view
    [toolbarItem setView:searchFieldOutlet];
   [toolbarItem setMinSize:NSMakeSize(100,NSHeight([searchFieldOutlet frame]))];
   [toolbarItem setMaxSize:NSMakeSize(400,NSHeight([searchFieldOutlet frame]))];

    // Create the custom menu
    NSMenu *submenu=[[[NSMenu alloc] init] autorelease];
   NSMenuItem *submenuItem=[[[NSMenuItem alloc] initWithTitle: @"Search Panel"
            action:@selector(searchUsingSearchPanel:)
            keyEquivalent: @""] autorelease];
    NSMenuItem *menuFormRep=[[[NSMenuItem alloc] init] autorelease];
    [submenu addItem: submenuItem];
    [submenuItem setTarget:self];
    [menuFormRep setSubmenu:submenu];
    [menuFormRep setTitle:[toolbarItem label]];
    [toolbarItem setMenuFormRepresentation:menuFormRep];
    } else {
    // itemIdentifier referred to a toolbar item that is not
    // not provided or supported by us or cocoa
    // Returning nil will inform the toolbar
    // this kind of item is not supported
    toolbarItem = nil;
    }
    return toolbarItem;
}
```

Note: Beginning in Mac OS X v10.5, if you always set a fresh view on the toolbar item in `toolbar:itemForItemIdentifier:willBeInsertedIntoToolbar:`, then `NSToolbar` does not copy that view. In previous versions of the operating system, `NSToolbar` would always copy the view when displaying it in the customization palette, using a keyed archiver.

Programmatically inserting and removing toolbar items

While under most circumstances the user will add and remove items from a toolbar using the customization sheet or the context menu, it is possible to insert and remove items programmatically.

The method `insertItemWithItemIdentifier:atIndex:` inserts the item specified by the item identifier at the specified index.

The method `removeItemAtIndex:` removes the toolbar item at the specified index.

Being notified when items are added or removed from the toolbar

An `NSToolbar` delegate can optionally be notified when toolbar items have been added or removed from the toolbar.

The optional delegate method `toolbarWillAddItem:` provides notification that a toolbar item is about to be added to the toolbar. The identifier for the toolbar item is contained in the notification's userInfo dictionary under the key `@"item"`. The typical use for this method is to set the target for a standard toolbar item, such as Print, and perhaps to alter other attributes of the item, such as its tool tip.

The example code in Listing 5 sets the target and action for the standard toolbar Print item when the delegate is notified that the item will be added to the toolbar.

Listing 5 An implementation of the toolbarWillAddItem: method to configure an NSToolbarItem before it is added to the toolbar

```
- (void) toolbarWillAddItem:(NSNotification *)notification {
    NSToolbarItem *addedItem = [[notification userInfo] objectForKey: @"item"];
    if ([[addedItem itemIdentifier] isEqual: NSToolbarPrintItemIdentifier]) {
    [addedItem setToolTip:@"Print Your Document"];
    [addedItem setTarget:self];
    [addedItem setAction:@selector(myCustomPrintAction:)];
    }
}
```

The optional delegate method `toolbarDidRemoveItem:` provides notification that the toolbar has removed a toolbar item from the toolbar. The identifier for the toolbar item is contained in the notification's userInfo dictionary under the key `@"item"`. You could use this opportunity to inform the window controller that the data associated with this item is no longer valid.

The example in Listing 6 invalidates the application's search string when the search item is removed from the toolbar.

Listing 6 An implementation of the `toolbarDidRemoveItem:` method that invalidates a value when a toolbar item is removed from the toolbar.

```
- (void)toolbarDidRemoveItem:(NSNotification *)notification {
   NSToolbarItem *removedItem = [[notification userInfo] objectForKey: @"item"];
    if ([removedItem itemIdentifier] isEqual:SearchDocToolbarItemIdentifier) {
    [self invalidateSearchString];
    }
}
```

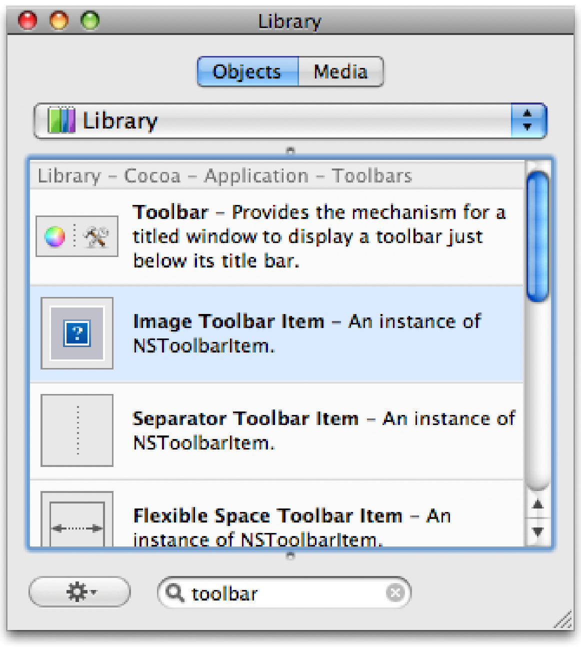
Objects other than the toolbar's delegate can track the adding and removing toolbar items by registering for `NSToolbarWillAddItemNotification` and `NSToolbarDidRemoveItemNotification` notifications explicitly.

Creating a Toolbar in Interface Builder

Beginning with Mac OS X v10.5 (Leopard), you can create a toolbar in Interface Builder. You can do almost anything with toolbars in Interface Builder that you can do programmatically. At runtime, you may combine toolbars and toolbar items that are unarchived from a nib file and those that are programmatically created.

Adding a Toolbar to a Window

Open your application's nib file and search the Interface Builder library for "toolbar". A toolbar object and a number of toolbar items are listed in the library.



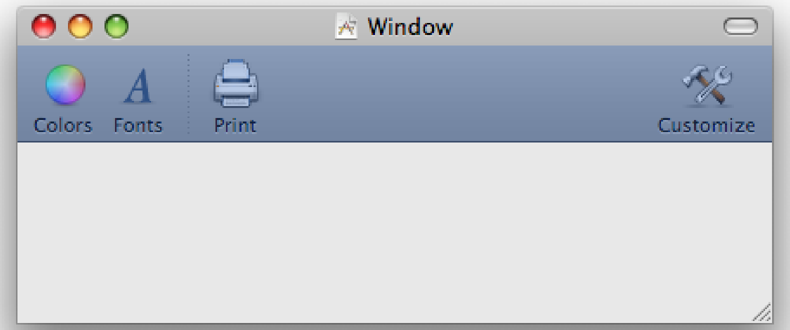
Toolbar – Provides the mechanism for a titled window to display a toolbar just below its title bar.

Image Toolbar Item – An instance of NSToolbarItem.

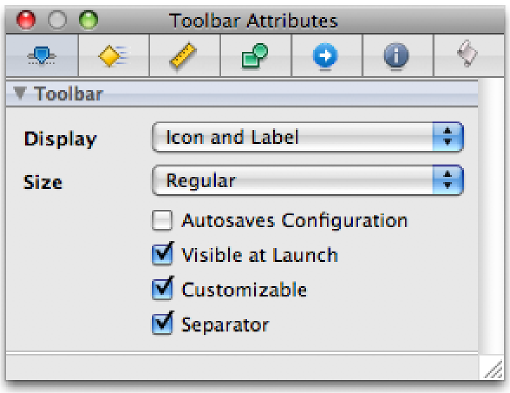
Separator Toolbar Item – An instance of NSToolbarItem.

Flexible Space Toolbar Item – An instance of NSToolbarItem.

Drag the toolbar object onto a window. Interface Builder automatically positions it below the window bar and above the window's content view. It displays the "default" default set of toolbar items.



Select the toolbar (if it isn't already selected) and display the Attributes pane for the toolbar in the inspector (Command-1). Set the desired attributes of the toolbar; make sure Customizable is selected if you want users to add items from the allowed set.


Toolbar Attributes

Toolbar

Display: Icon and Label

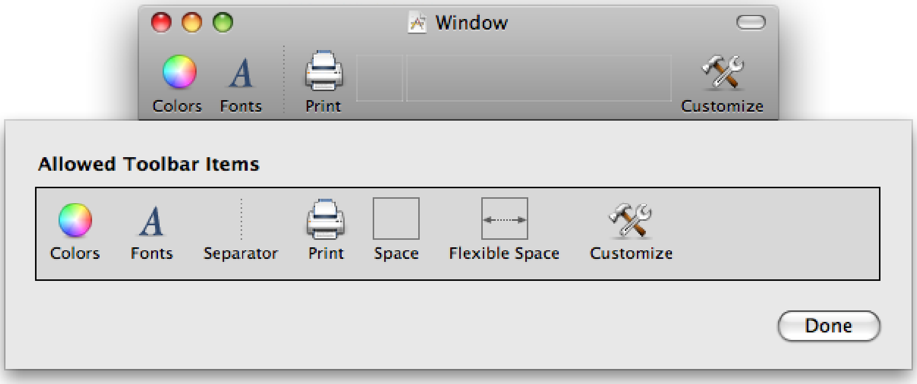
Size: Regular

- [ ] Autosaves Configuration
- [x] Visible at Launch
- [x] Customizable
- [x] Separator

Note that the Attributes pane for toolbars does not include a field for the toolbar identifier. Remember that the purpose of the identifier is to differentiate toolbars assigned to different kind of windows, for example, between the toolbars for document windows and the toolbar for a utility window. Because you would store these different toolbars in different nib files, there is little need for toolbar identifiers.

Adding and Configuring Toolbar Items

Next you add and possibly remove toolbar items to both the allowed set of items and the default set. A toolbar item can be one of the standard ones (for example, Colors) or can be a custom toolbar item. Start by clicking the toolbar if it is already selected or double-clicking it if it is not selected. This displays a view containing the allowed toolbar items.



To add a toolbar item, drag it from the library onto the row of allowed toolbar items. If you want the same item to be in the default set, drag it from the allowed toolbar items onto the toolbar. Position it where you want it to appear. To remove an item, drag it away from the window and release the mouse button; this applies to toolbar items in both the default set and the allowed set.

Custom toolbars are of two sorts: custom image and custom view. To add a custom image toolbar item, drag the Image toolbar item from the library and drop it on the row of allowed items. Complete the following steps to configure the custom toolbar item:

1. Add the image you want to use for the toolbar item to the project (Project > Add to Project).
2. Select the custom toolbar item and display the attributes for it (Command-1).
3. Enter the name of the image file (minus extension) in the Image Name field.
4. Enter labels and a tag number for the toolbar item and select the Autovalidates option.

   You can use a toolbar item's tag as a way to access it programmatically from its toolbar.

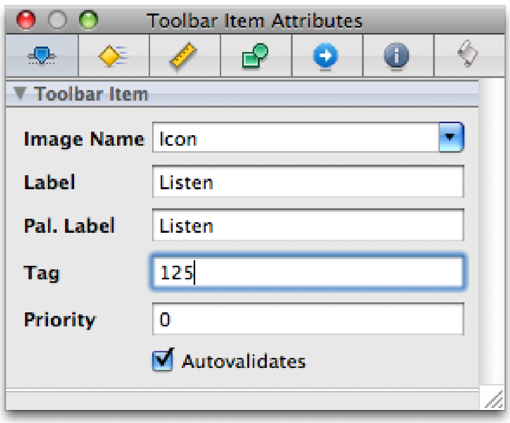
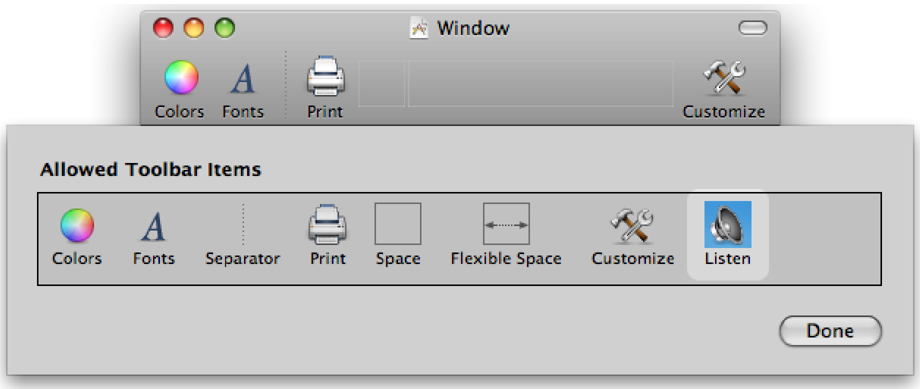


Image Name: Icon
Label: Listen
Pal. Label: Listen
Tag: 125
Priority: 0
[✓] Autovalidates

For the Priority option, see the description of the `setVisibilityPriority:` method.

You custom toolbar item will look similar to the one in this example.



If you want this toolbar item in the default set, drag it to the toolbar.

The procedure for adding a custom view item is very similar to that for a custom image item. ("Custom" in this context means any object from the Interface Builder library as well as instances of a custom `NSView` subclass.) Just drag any view object from the library onto the Allowed Toolbar Items area. Click the item once and press Command-1 to display the Attributes pane for the object as a toolbar item; click again to edit the attributes of the item as itself. You should modify the size of the custom view item in the Size pane of the inspector, not directly. If you drag a Custom View object into the allowed-items set, click it twice and set the name of the custom `NSView` class in the Identity pane of the inspector (Command-6).

Setting the Target, Action, and Toolbar Delegate

After you complete the default and allowed sets of toolbar items, make sure the appropriate ones are set up to invoke a target method in a target object. For example, for a custom toolbar item you would complete the following steps:

1. Declare an action method (return type `IBAction`) in the header file of the class whose instance is the target.
2. Select the object representing this class in the Interface Builder document window (it's often File's Owner).
3. Right-click (or Control-click) to display the connections window; locate the action method under Received Actions.
4. Drag a connection line from the circle after the action name to the toolbar item in the allowed set.
5. Implement the action method.

Finally, make a connection between the `delegate` property of the toolbar and the object (if any) that responds to the delegation messages sent by the toolbar.

Setting a Toolbar Item's Representation

The `label` attribute of a toolbar item is displayed as the text for the toolbar item when the toolbar is in Icon & Text Mode or Text Only Mode. By default, the label is also used in the overflow menu as the title of the menu item representing the toolbar item. The `paletteLabel` attribute of an item is used instead of label in the customization palette. (If `paletteLabel` is not set, the customization palette does not use `label` by default.)

Clicking on the label of an image item in Text Only Mode or selecting an image item's overflow menu item simply invokes the image item's action.

View items, on the other hand are more complex and can't usually be handled so simply. Primarily to give view items more flexibility, NSToolbarItem provides a `menuFormRepresentation` attribute.

Every toolbar item has a menu form representation, which is a menu item. The toolbar provides an initial default menu form representation that uses the toolbar item's label as the menu item's title. The application can provide a custom menu form representation which can have a submenu and which can have a title different from the toolbar item's label. If a toolbar item's custom menu form representation has a submenu, then the toolbar drops down that submenu under the toolbar item in Text Only Mode and displays the submenu in the overflow menu.

If a view item's menu form representation attribute has not been set, NSToolbar disables the overflow menu item as well as the toolbar item's text in Text Only Mode. If the attribute has been set and the menu form representation has a submenu, NSToolbar enables the overflow menu item as well as the toolbar item's text in Text Only Mode. If the attribute has been set and the menu form representation does not have a submenu, the toolbar validates the menu item to determine whether to enable the toolbar item's text in Text Only Mode and the toolbar item's overflow menu item. For more detail on how the validation works, see "Validating Toolbar Items" (page 29).

Image items as well as view items can use a simple `menuFormRepresentation` without a submenu merely for the purpose of replacing `label` as the title of the toolbar item's overflow menu item. The only situation in which you should use this feature is when the label is empty.

Validating Toolbar Items

The toolbar validates every visible image item and every invisible item's overflow menu item.

By default toolbar items have click-through behavior, which is to say that toolbar items remain enabled and clickable when the user switches to another window or even another application. To change this behavior for a toolbar item, your validation code needs to take the larger environment into account. See *Aqua Human Interface Guidelines* for more information on click-through behavior.

Image item validation

The toolbar automatically takes care of darkening an image item when it is clicked and fading it when it is disabled. All your code has to do is validate the item. If an image item has a valid target/action pair, then the toolbar will call NSToolbarItemValidation's `validateToolbarItem:` on target if the target implements it; otherwise the item is enabled by default.

In the absence of a custom menu form representation, NSToolbar validates an image item's overflow menu in the same way that it validates the toolbar item in the toolbar.

View item validation

Validation for view items is not automatic because a view item can be of unknown complexity. To implement validation for a view item, you must subclass NSToolbarItem and override `validate` (because NSToolbarItem's implementation of `validate` does nothing for view items). In your override method, do the validation specific to the behavior of the view item and then enable or disable whatever you want in the contents of the view accordingly. If the view is an NSControl you can call `setEnabled:`, which will in turn call `setEnabled:` on the control.

In the absence of a custom menu form representation, NSToolbar by default disables a view item's overflow menu item.

Overflow menu item validation

If a toolbar item has a custom menu form representation with no submenu, then the toolbar will validate the overflow menu item and the toolbar item's text in Text Only Mode differently than it would with the default menu form representation: If the menu form representation menu item's target implements `validateMenuItem:` (part of NSMenuValidation), the toolbar calls that method to validate both the overflow menu item and the toolbar item's text in Text Only Mode.

Setting a Toolbar Item's Size

When we speak of toolbar item size or toolbar height in this section, we're referring to the image area or view area when the toolbar is displaying in Icon & Text or Icon Only Mode.

The displayed resolution of an image item is dependent on the sizeMode of the toolbar. You should provide image representations specific to the default, regular and small size modes in a single image that supports multiple image representations such as icns or tiff. The appropriate image representation is automatically displayed for the toolbar's current sizeMode. If an appropriate representation is not available, the toolbar scales the a representation to the appropriate size for the current mode, at a cost in performance and appearance. Images that are not square are scaled to fit. An image item's image is also scaled down and used in the image item's overflow menu item. For more information, see the section on toolbar icons in "Icons" in *Aqua Human Interface Guidelines*.

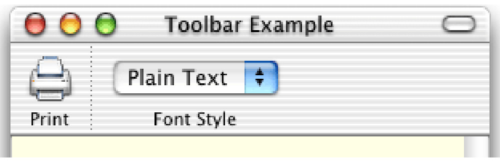
The `minSize` and `maxSize` toolbar item properties are for use only by view items. They must not be left unset (or the view will not display), and unless you are implementing intelligent stretching behavior in a view item, both the `minSize` and `maxSize` properties should equal the size of the item's `view`. The toolbar takes care of providing space between toolbar items, so a view should be just big enough to enclose the frames of its content objects. It's OK for a view item's `minSize` height to be less than the usual 32 pixels (to work optimally with possible future toolbar enhancements).

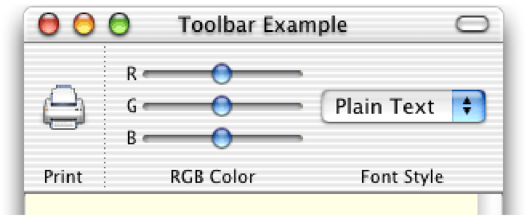
Note: As of Mac OS X v10.5, if you call `setView:` on an NSToolbarItem object without also calling `setMinSize:` or `setMaxSize:`, the toolbar item sets its minimum and maximum size equal to the view's frame.

The height of the toolbar is the height of the greatest `minSize` height of any item visible in the toolbar at the time. If a view item's `maxSize` height is less than the height of the toolbar, then the toolbar centers the item's view within the available vertical space. If a view item's `view` does something intelligent when it is stretched, then you will set its `maxSize` greater than its `minSize` in height, width, or both. Horizontally-stretchable view items, including the Flexible Space standard item, compete equally for available horizontal space. An example of a view item that stretches horizontally is the "Search Mailbox" item in the Apple Mail application.

An example of a view item that stretches vertically is the Separator standard toolbar item. When we insert a tall RGB Color custom item into the toolbar, the Separator item adds more dots to itself, but the image item and the nonstretchable view item don't change (the toolbar merely centers them vertically):

Figure 1 Vertically stretched toolbar item

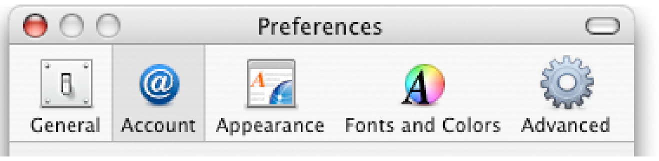



Toolbar Example
R
G
B
Plain Text
Print
RGB Color
Font Style

Selectable Toolbar Items

NSToolbar allows you to specify that certain items in the toolbar can indicate a selected state. This is often used in conjunction with an NSTabView that is configured to have no visible tabs. Figure 1 contains an example implementation similar to that of Safari and the Finder.

Figure 1 Selectable NSToolbar items used as preferences navigation



Toolbars that need to indicate item selection must specify the items that can be selected by implementing the delegate method `toolbarSelectableItemIdentifiers:`. This method returns an array containing the identifiers of the items that can be selected. The example Listing 1in returns all the identifiers for the preferences implementation.

Listing 1 Example implementation of toolbarSelectableItemIdentifiers:

```
- (NSArray *)toolbarSelectableItemIdentifiers: (NSToolbar *)toolbar;
{
    // Optional delegate method: Returns the identifiers of the subset of
    // toolbar items that are selectable. In our case, all of them
    return [NSArray arrayWithObjects:GeneralPreferences,
                                    AccountPreferences,
                                    AppearancePreferences,
                                    FontsAndColorsPreferences,
                                    AdvancedPreferences, nil];
}
```

Your application can specify the currently selected toolbar item using the method `setSelectedItemIdentifier:` passing the identifier for the desired toolbar item. The currently selected toolbar item is returned by the method `selectedItemIdentifier:`. If there is no currently selected, `nil` is returned.

Subclassing NSToolbarItem

To provide enabling behavior or to modify click-through behavior on a view item, you must override `validate` and may want to override `isEnabled` and `setEnabled:`.

To enable multiple copies of the item in the toolbar, you must override `allowsDuplicatesInToolbar` to return `YES`.

Your subclass must conform to the NSCopying protocol.

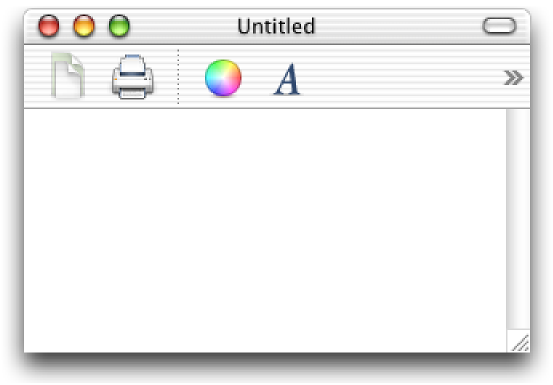
Techniques for Toolbar Management

You can determine the height of a toolbar and whether it has overflow items by following the procedures described below.

Determining if a Toolbar Has Overflow Items

If a toolbar is unable to display all the user's currently configured toolbar items, it pushes the additional items into the overflow menu and displays the overflow menu icon as shown in Figure 1.

Figure 1 A toolbar indicating items in the overflow menu



An application can determine if a toolbar has overflow items by comparing the number of items returned by the method `items` with the number of items returned by the `visibleItems` method as shown in Listing 1. If these values differ, then the toolbar has items in the overflow menu.

Listing 1 Example code to test if a toolbar has overflow items

```
int numberOfItems=[[theToolbar items] count];
int numberOfVisibleItems=[[theToolbar visibleItems] count];

if (numberOfItems != numberOfVisibleItems) {
    // toolbar has overflow items
}
```

Calculating a Toolbar's Height

Although NSToolbar does not currently provide a method for returning a toolbar's height, it is easy to compute that value. You subtract the height of the window's content view from the window's height.

The Objective-C function in Listing 2 calculates the height of the toolbar in a window, returning 0 if the toolbar is hidden.

Listing 2 Objective-C function to calculate toolbar height

```
float ToolbarHeightForWindow(NSWindow *window)
{
    NSToolbar *toolbar;
    float toolbarHeight = 0.0;
    NSRect windowFrame;

    toolbar = [window toolbar];

    if(toolbar && [toolbar isVisible])
    {
        windowFrame = [NSWindow contentRectForFrameRect:[window frame]
                                styleMask:[window styleMask]];
        toolbarHeight = NSHeight(windowFrame)
                        - NSHeight([[window contentView] frame]);
    }

    return toolbarHeight;
}
```

Document Revision History

This table describes the changes to *Toolbar Programming Topics for Cocoa*.

Date	Notes
2009-02-04	Updated for Mac OS X v10.5.
	Folded "Calculating Toolbar Height" and "Determining if a Toolbar Has Overflow Items" into "Techniques for Toolbar Management" (page 37).
2008-10-15	Made several minor corrections.
2007-01-08	Consolidated and updated articles on creating toolbars into "Toolbar Management Checklist."
2006-06-28	Corrected typos.
2006-05-23	Added link to sample code.
2004-08-31	Clarified toolbar item icon scaling in "Setting a Toolbar Item's Size" (page 31).
2004-03-30	Added "Selectable Toolbar Items" (page 33) to *Toolbars*.
2003-02-24	Added "Calculating a Toolbar's Height" to the topic overview.
2003-02-19	Added captions to code listings in "Calculating a Toolbar's Height" and figures in "Setting a Toolbar Item's Size" (page 31)
2003-02-14	Added "Adding and Removing Toolbar Items" (page 17) task as a replacement for "Creating a toolbar delegate".
	Added "Techniques for Toolbar Management" (page 37) task.
2002-11-12	Revision history was added to existing topic.