Chapter 6

MENUS

CHAPTER 6 MENUS

Goal

To explore the mechanics of menus—updating, associating with a window or view and dynamically adding and deleting menu items.

Prerequisites

Understanding of target/action including nil-targeted actions and familiarity with the event loop.

Objectives

At the end of this section, you will be able to:

» Explain how nil-targeted menu items are updated

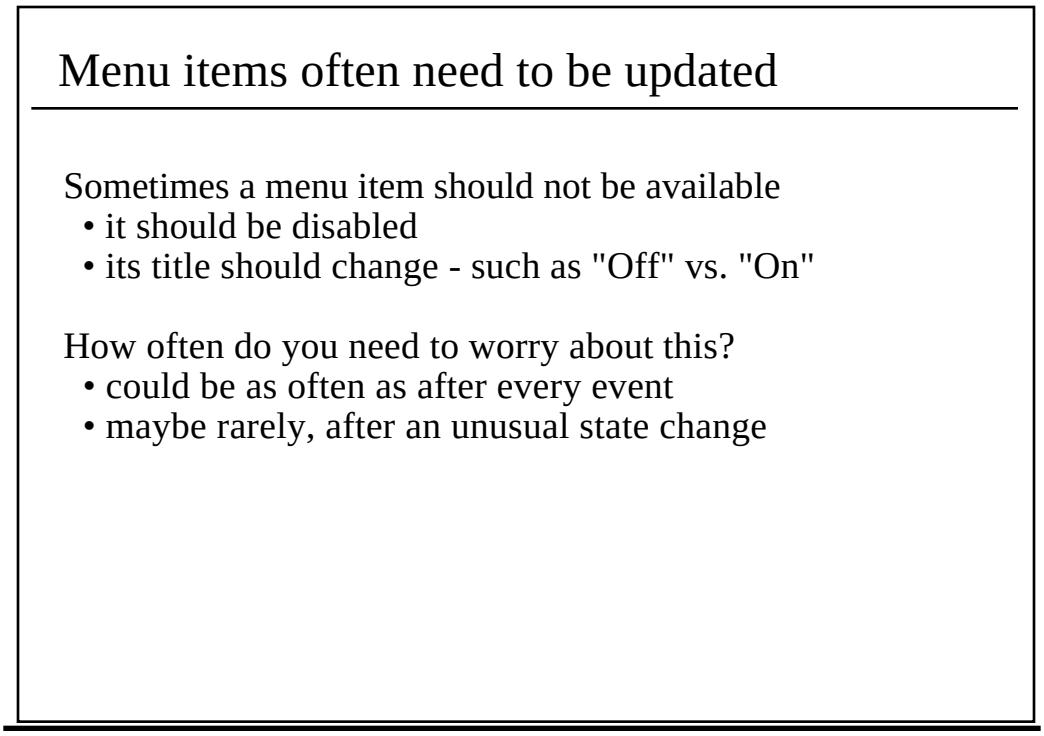
» Specialize menu updating for any menu item regardless of its target

» Add or delete menu items dynamically at run time

Reading

NSMenu and NSMenuItem classes in the Application Kit

NSMenuActionResponder and NSMenuItem protocols in the Application Kit

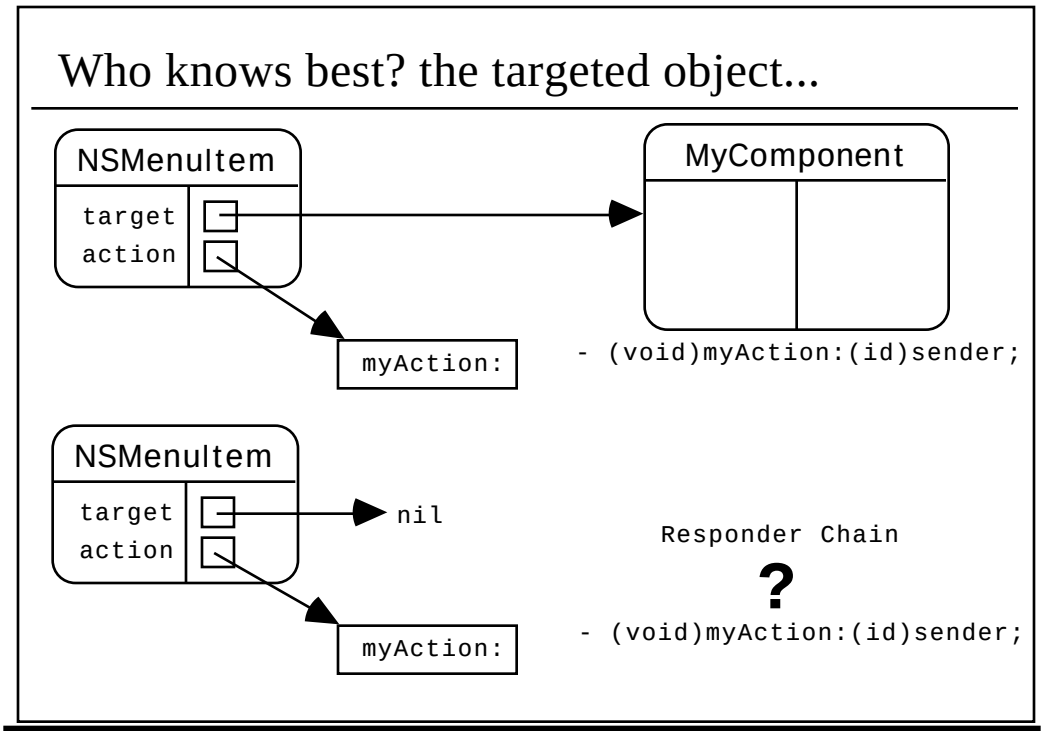


Menu items often need to be updated

Menu items, like other controls, must be sensitive to the state of your application. A particular action may not always be relevant. Some menu items act as two-state toggles, alternating between "on" and "off". Good user interface design implies that a user should not be presented with a meaningless or erroneous option. The option should be clearly disabled or change state to reflect a new context.

How frequent are such state changes? When to they occur? It depends on the function of the menu item and it depends on your design. It could be rare or exceptional, and specific code in your application could change menu items at that specific point in time. It could be as often as after every user event. Menu items that target first responder may suddenly become meaningless when the first responder changes. It can change with the click of a mouse or a <TAB> keystroke.

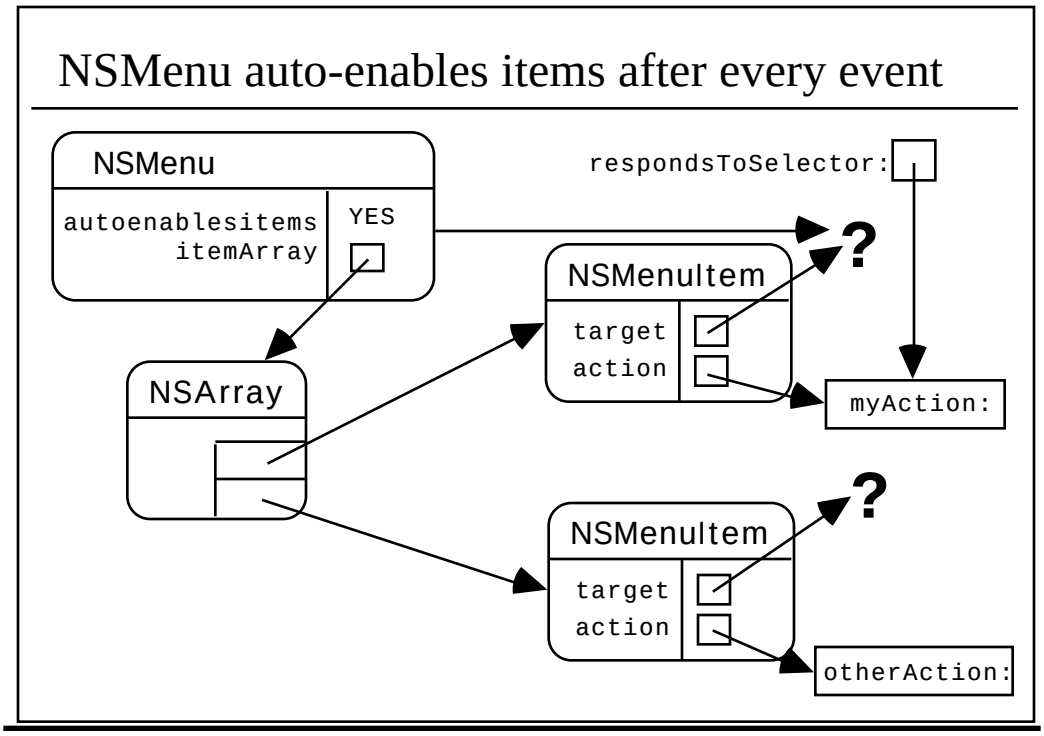
In this case, an automatic updating scheme is required after each user event. After each pass through the event loop, some object should check each menu item to see if its state should change. The best design would let each menu item that cares update itself. How is this done?



Who knows best? the targeted object...

A menu item works by messaging a target object. Menu updating works in conjunction with that target object. Does it actually respond to the selector? Does it want to respond to the selector right now with the application in its particular state?

Your menu item target can be hard-wired or nil, when it refers to an object in the responder chain. In the latter case, the targeted object must be dynamically determined. It may be that no object in the responder chain responds to the action selector. In all cases, if the target specifies an object that responds to the action, the menu item is enabled. Otherwise, it is disabled. Like all controls, when a menu item is disabled, its title changes to light gray and it cannot be pressed by the user.



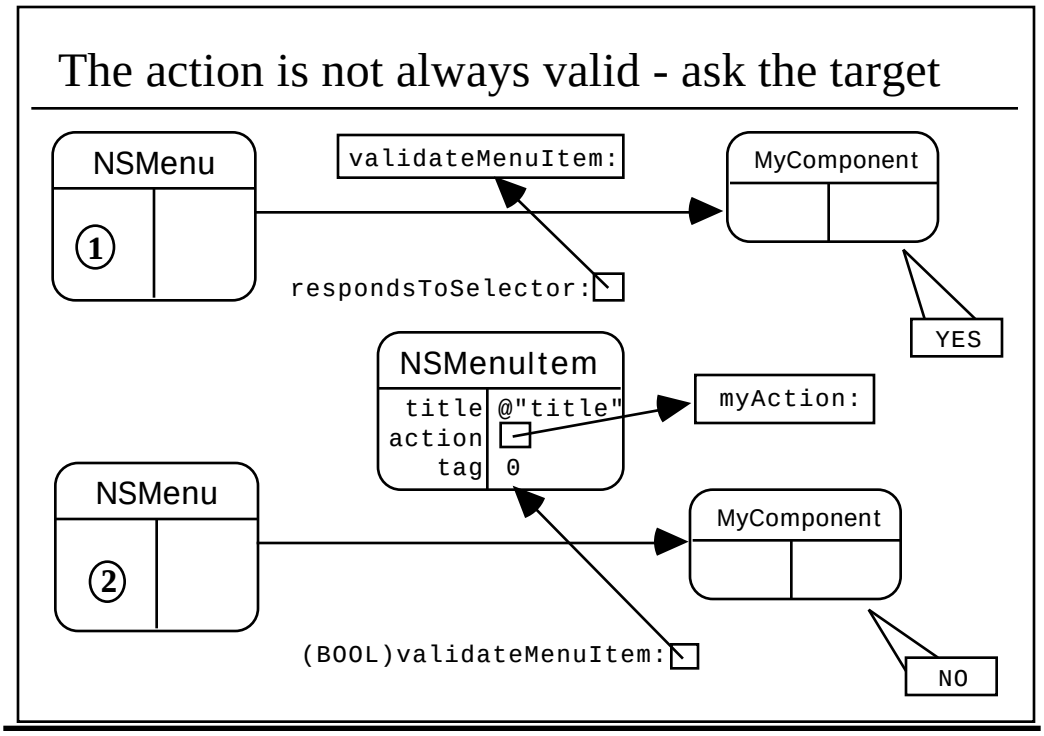
NSMenu auto-enables iterms after every event

Each NSMenu is configured by default to auto-enable its items. Once per event loop iteration, each menu item determines whether its target responds to the menu item's action selector. For hard-wired targets, this status never changes. It is always true. When the target is nil, the target is dynamic, somewhere in the responder chain. The target is the first object that responds to the selector. If no object in the chain responds to the action selector in the menu item, the menu item is disabled. It has no effect at this point in time.

When first responder is a text field, it responds to **cut:**. When it is a button or slider, it does not. A **save:** selector with a nil target might be intended for your window delegate. If you have no open or active window, there is no delegate in the responder chain to respond. In both cases, the menu item will be disabled.

When a menu item's action is nil, there is no message to send at all. This menu item will always be disabled. You would never actually use such a menu item in your application, but it is the case when you first instantiate the menu item, before you set its action. Remember also that a menu item can be permanently disabled in Interface Builder whether or not its target and/or action apply at run time.

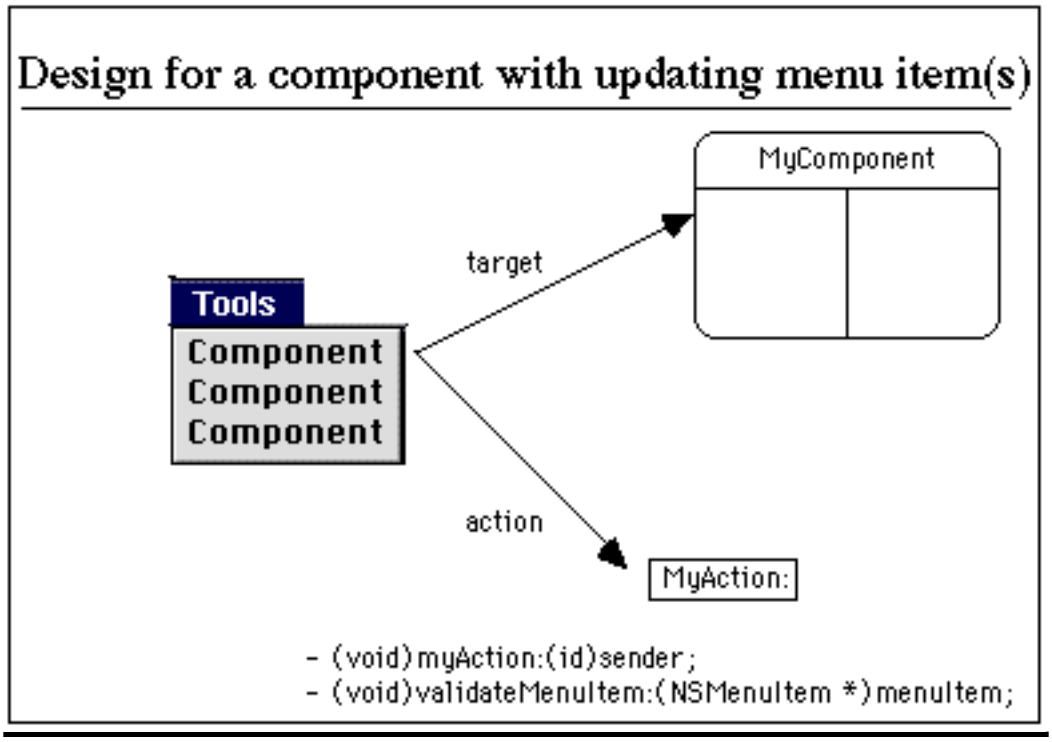
This first phase of updating happens for you automatically. But this is not always enough. A target that responds to the selector may not want to respond at this particular time. Once again, it makes sense to ask the object itself.



The action is not always valid—ask the target

Once the responsive target object is located, auto-enabling checks to see if it responds to **validateMenuItem:**. If so, the message is sent and the return value determines whether the menu item is enabled or disabled. The target object is given the opportunity to consult the relevent application state and decide for itself. If the target object does not implement **validateMenuItem:**, the menu item is automatically enabled.

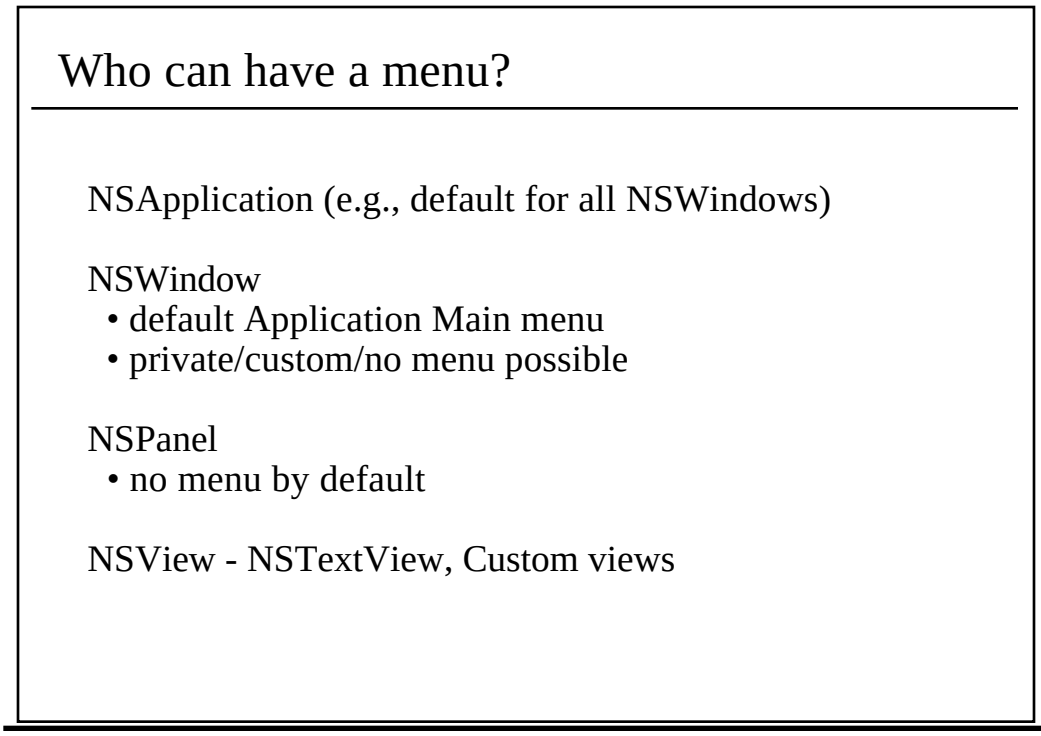
It is typical for a particular controller to be the target for multiple menu items. In this case, the controller may have to decide based on which menu item is being updated. The menu item is passed as a parameter. The controller can identify it by its tag, by its action, by its title, or even by comparing it with an outlet.



Design for a component with updating menu item(s)

Not all menu items need this kind of updating. Often, the first check is sufficient and your target object doesn't need to implement **validateMenuItem:**. If it is, your design should follow this pattern. For multiple actions, your **validateMenuItem:** implementation would need to check for each menuItem, taking all its actions into account.

This is called the NSMenuActionResponder informal protocol. To properly participate, your object should conforms to this protocol.



Who can have a menu?

Your main nib file contains the main menu. It is applied to all windows in your application. It is not common UI design, but a specific window could have its own unique menu, different from the default. It might have no window at all.

NSPanel is a subclass of NSWindow but, by default, has no menu at all.

On Microsoft Windows, any NSView within a window can also have its own menu. It is invisible but pops up dynamically with a mouse click. It is even possible for a view to have multiple menus, each available depending on where the mouse is clicked with the view's geometry.

NSView's provide the following outlets:

» **menu**

» **defaultMenu**

To implement multiple menus, your custom view would need to override **menuForEvent:**. For more information on custom views and overriding event handling, see the section on custom views.

Additional Points to Ponder

NSMenu and NSMenuItem architecture

Here is a detailed look at the NSMenu and NSMenuItem objects and how they are connected to build your application menus.

Each menu has a list of its menu items. Each menu item can be a leaf or a sub menu. In the case of a submenu, its target outlet points to the submenu, a new NSMenu object and its action is the message that makes the submenu appear—**subMenuAction:**. There is a boolean field that flags the menu item as a submenu as well.

Each NSMenu can be separately configured to auto-enable its items. The default it YES.

Each NSMenuItem can be separately enabled or disabled. When useful, they can be uniquely identified by its integer tag. Some may feature a unique keystroke equivalent, also know as a short cut or an accelerator.

Dynamically adding a menu item

```
NSMenu *mm = [NSApp mainMenu];
NSMenu *i;

// instantiates and returns a new NSMenuItem
i = [mm insertItemWithTitle: @"Option"
            action: @selector(performOption:)
            keyEquivalent: @"O" atIndex: 1];
// configure MenuItem
[i setEnabled: YES];
// Default Target is nil, e.g., responder chain
[i setTarget: optionObject];
// Resize NSMenu to accomodate new MenuItem
[mm sizeToFit];
```

Dynamically adding a menu item

Sometimes a useful approach, here is an example of dynamically adding a menu item at runtime. Certain application items might come and go, though enabling and disabling a permanent menu item is more traditional. You might consider this technique for reusable components that can be added to an application and know how to add themselves to the menu automatically.

You can insert an item at a specific index using the insert method shown here, or you can simply append it to the bottom of the menu by using the add variant shown on the following page.

When an item is added or deleted in this way, the menu must be told to resize itself accordingly.

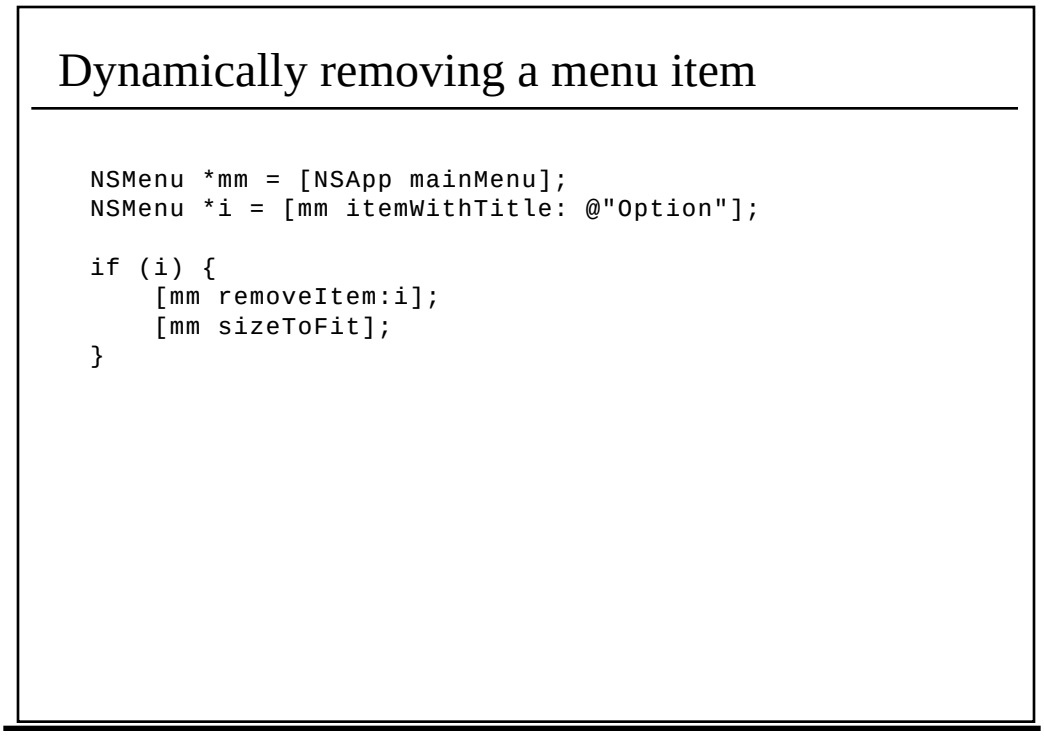
Dynamically adding a submenu

```
NSMenu *new, *mm = [NSApp mainMenu];
NSMenuItem *i;

// Add new NSMenuItem that launches the submenu
i = [mm addItemWithTitle: @"Sub"
        action: (SEL)0
        keyEquivalent: @""];
[i setEnabled: YES];

// Create NSMenu (if not created in IB)
new =  [[NSMenu alloc] initWithTitle: @"Sub"];

// Add submenu items (not shown here)

// Attach it
[mm setSubmenu: new forItem:i];
[mm sizeToFit];
```

Dynamically adding a submenu

Adding an entire submenu requires that you first add a menu item, then attach a new NSMenu instance to it. Now the new menu item will display the submenu when activated.

The NSMenu could be prebuilt or dynamically constructed as demonstrated on the previous page.



Dynamically removing a menu item

Removing an item requires simply that you locate the item, remove it, then ask the containing menu to resize itself. The menu item, and any submenus it contains, will be released.

Important ideas from this section

» By default, menus auto-enable their menu cells during each iteration of the event loop. This can be disabled.

» Auto-enabling involves two steps:

    The first step checks the menu item's target outlet to verify that an object responds to the action selector. If not, the menu item is disabled.

    If it does respond, the second step checks to see if the object responds to **validateMenuItem:** If it does, the message is sent and the return value determines if the menu item is enabled. The target object decides for itself.

» NSViews can have their own popup menus on Microsoft Windows.

» You can dynamically modify menus at runtime, adding or deleting menu items and entire sub menus.

Classes featured in this section

» NSMenu

» NSMenuItem

REVIEW MENUS

1. There are three steps involved in the automatic menu updating scheme. Name them.

2. How often are menu items auto-enabled? Is this always required? Describe in general, another approach a component might use to update its own relevant menu items.

EXERCISE 6.1 MENU UPDATING WITHOUT LIFTING A FINGER

For a final touch of polish to your application's menus, this exercise demonstrates how to provide automatic updating of each menu item. This can be done explicitly, but it can be difficult to remember every time you should re-enable or disable one or more menu items. This facility centralizes the logic so that this aspect of your state management can be held within one method in the corresponding target object.

Objective

After completing this exercise, you'll be able to use the NSMenuValidation protocol to automate menu updating

Exercise

1. In the current version of your application, the Revert to Saved menu item is enabled all the time. When the document is first opened, and hasn't been modified, this is not useful—so the menu validation scheme can be used to do turn this command off.

    - Add a **validateMenuItem:** method to DocController. You need to check either the name of the menu item you are passed, its selector, or its tag to make sure it refers to the **revert:** method.
    - You can move the logic that checked if reverting is necessary from the **revert:** method to **validateMenuItem:**.

2. Recompile the application and check that the menu item is automatically disabled until you modify the document, such as by adding a row.

Enhancements

- Provide automatic updating for the Save All command by adding a **validateMenuItem:** method to AppController.The menu item should be enabled only when there are modified documents.
- What other menu items are automatically updated? What happens when there are no active windows?