Chapter 11

MULTIPLE DOCUMENT APPLICATIONS

CHAPTER 11 MULTIPLE DOCUMENT APPLICATIONS

Goal

To understand the nib files, the objects and their connections used to implement a template for a multi-window application.

Prerequisites

- The ability to use multiple nib files within a single application
- A practical understanding of delegation and notification

Objectives

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

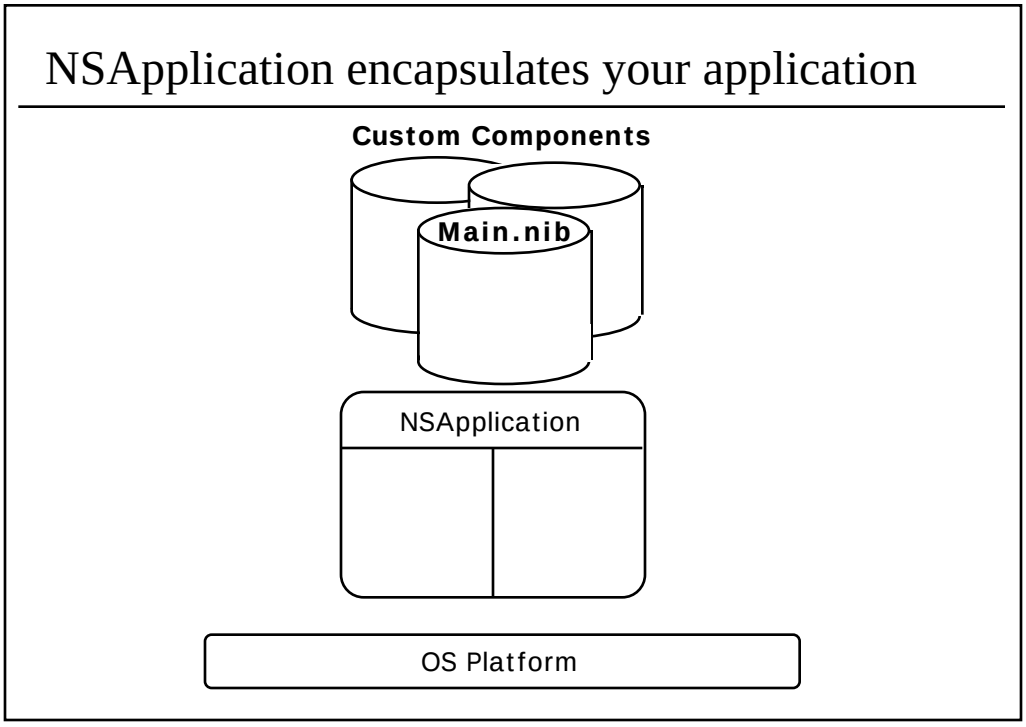
- Control the primary attributes of NSApplication and NSWindow
- Utilize delegation and notification to track state changes in the application
- Implement a functional template for a multi-document application
- Use an Alert panel as a control for the user to direct the flow of your application's behavior

Reading

NSApplication class reference in the Application Kit

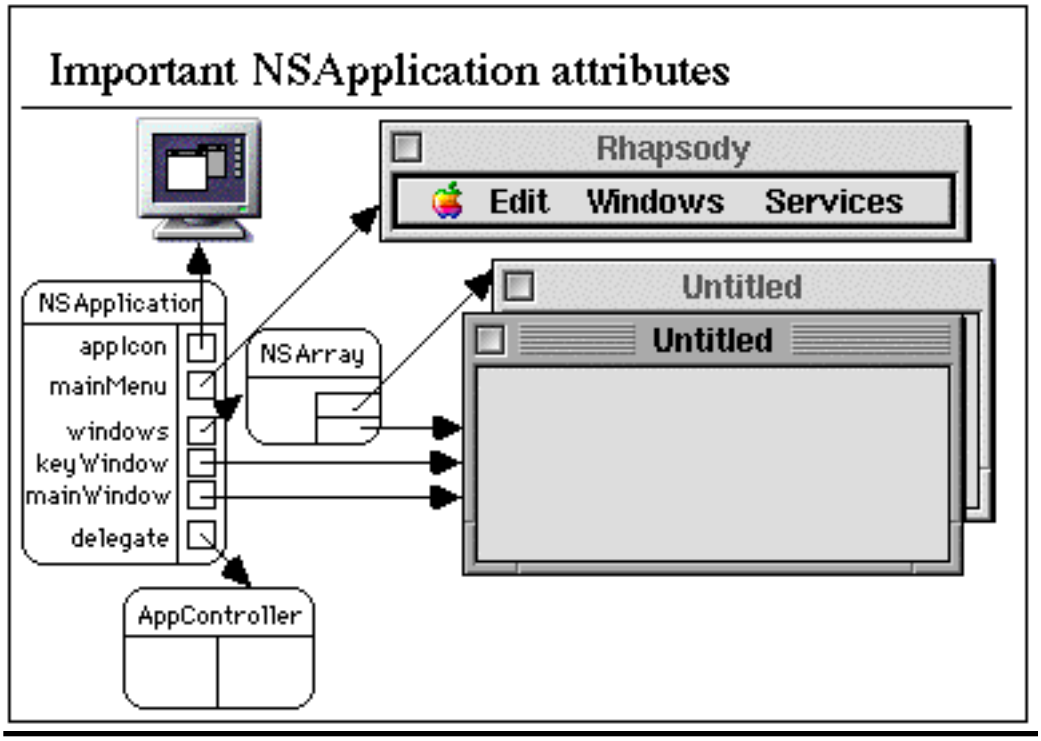
NSWindow class reference in the Application Kit

NSNotification class reference in the Foundation



NSApplication encapsulates your application

Every application contains exactly one instance of NSApplication. It encapsulates your entire application and serves as a kind of bridge to the outside world.To equip your application with all the proper behavior, you should study some of NSApplication's important attributes and understand how it works with a delegate.



Important NSApplication attributes

- **appIcon**—the NSImage for the on-screen application icon
- **mainMenu**—the application main menu, e.g. from the main nib
- **windows**—the list of on-screen windows
- **keyWindow**—the window that receives keyboard input
- **mainWindow**—different than keyWindow if an auxiliary panel is currently keyWindow
- **delegate**—like many Application Kit objects, NSApplication can use a helper object called a delegate

Useful NSApplication delegate methods/notification

Delegate methods

```
- (BOOL)applicationShouldTerminate:(id)sender;
- (BOOL)application:(NSApplication *)app
        openFile:(NSString *)file;
```

Notifications

```
- (void)applicationDidFinishLaunching:
        (NSNotification *)notification;
- (void)applicationWillTerminate:
        (NSNotification *)notification;
```

Useful NSApplication delegate methods and notifications

applicationShouldTerminate: is sent when NSApplication receives the terminate: message. This is typically from a Quit or Exit menu item. By returning NO, your delegate can suppress this allowing the application to continue running as if nothing happened.

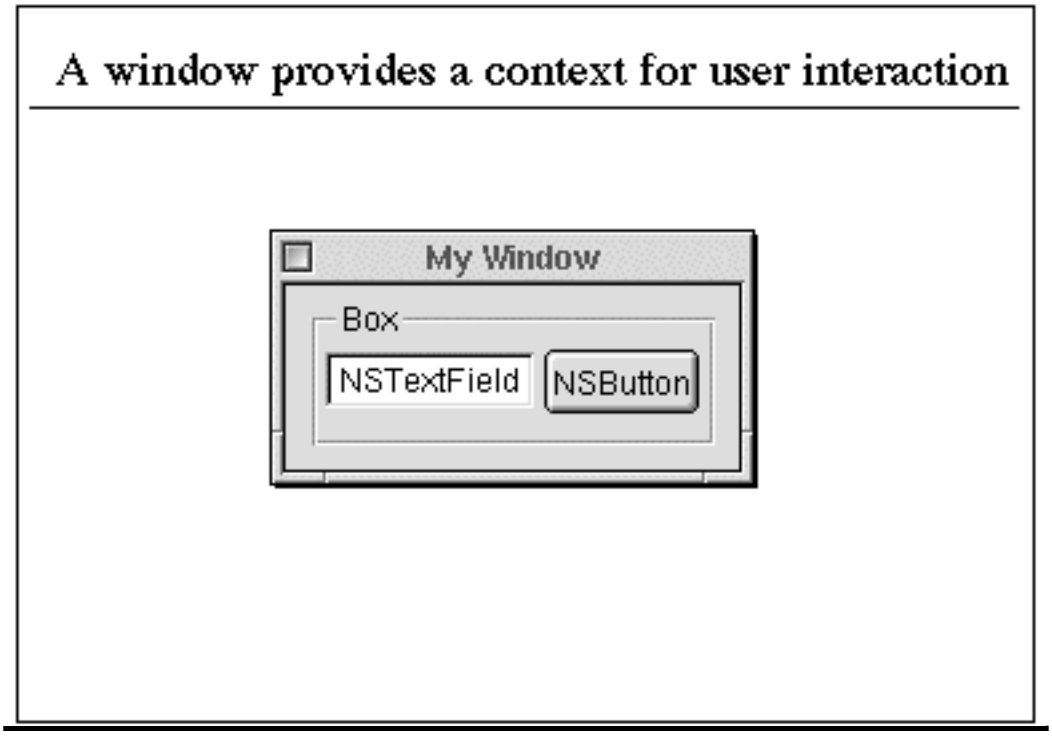
application:openFile: is sent when NSApplication receives a request for your application to open a file. This is the result of a user opening a file in the file viewer which, because of its type, is bound to your application for processing.

applicationDidFinishLaunching: is sent after your application has loaded the main nib, presented the user interface on the screen, but before NSApplication starts the event loop.

applicationWillTerminate: is sent just before the application finally terminates. As a notification, you cannot stop this. If you need to intervene, use applicationShouldTerminate: instead.

The complete set of NSApplication notifications can be summarized by these general classifications:

- Launching, terminating
- Becoming Active, Resigning Active
- Hiding, Unhiding
- Updating



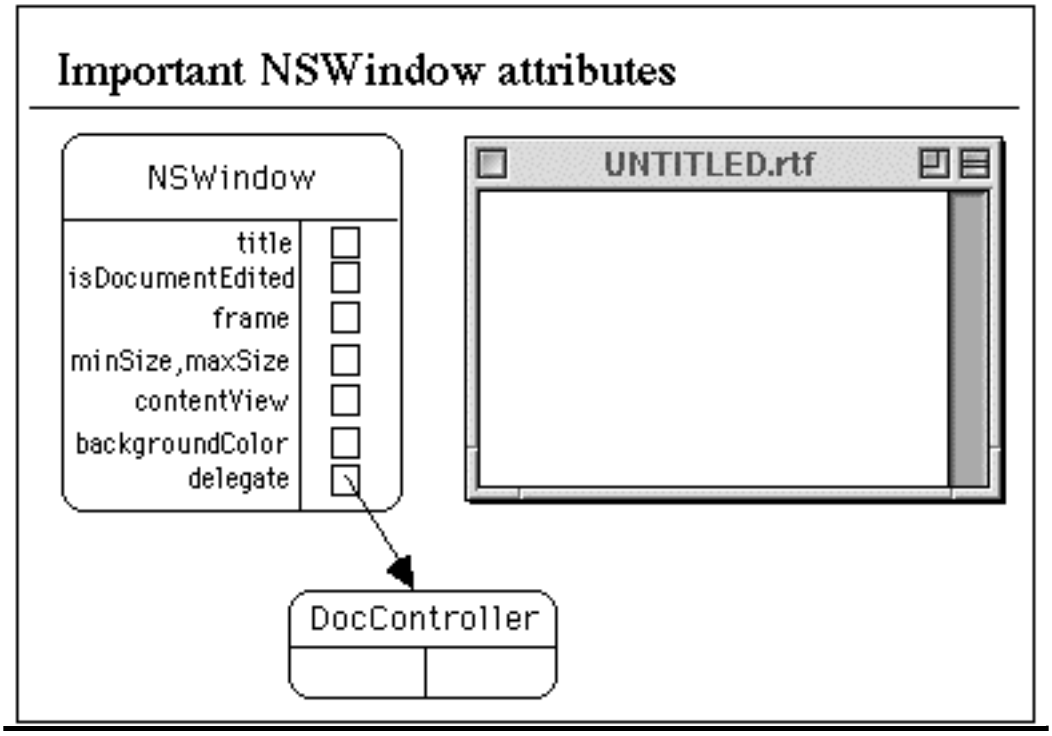
A window provides a context for user interaction

A window is the place where a user interacts with your application.

Some applications use a single main window. Many allow a user to open multiple windows, typically for editing several documents concurrently.

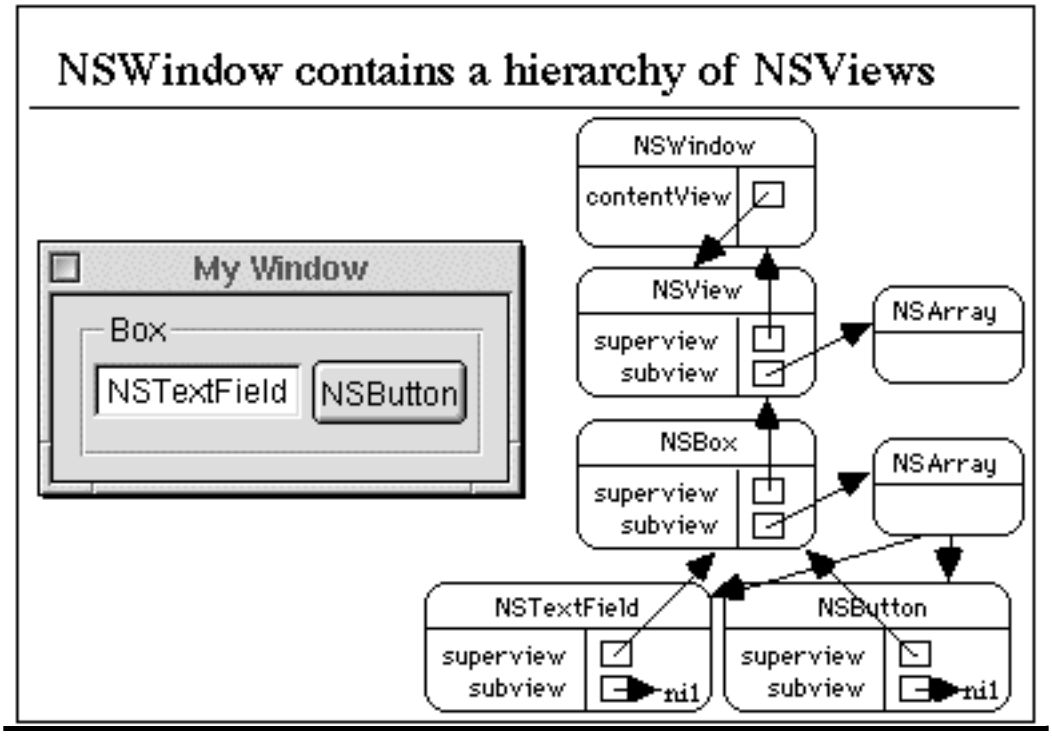
A window features its own built-in controls around the window frame such as buttons for resizing, moving, closing, minimizing or maximizing the window. Within the frame is a blank slate which you can fill with controls and other objects to implement a custom user interface suitable to your application needs.

Several basic attributes allow you to control the appearance and behavior of the window, and to keep your custom objects abreast of important state changes.



Important NSWindow attributes

- **title**—the text in the title bar. Often, this specifically represents a file name with an icon and full pathname. See **representedFileName**.
- **menu**—an outlet to the window's menu. Typically, this is connected to the main menu, but windows can have their own unique menu objects.
- **documentEdited**—a window can keep track of whether its contents have been edited or not and reflects this in the user interface. You must tell the window when document associated with the window becomes edited (e.g. through user interface changes) and when not (e.g. when the document is first loaded or after it is saved).
- **frame**—the location and size of the window on-screen. This is easily set in Interface Builder.
- **minSize**, **maxSize**—limits that a window will enforce. These can easily be set in Interface Builder.
- **contentView**—the topmost view in the view hierarchy of this window.
- **backgroundColor**—what color shows through the contentView.
- **delegate**—NSWindow can use a helper object, a delegate. This would typically be an instance of your custom controller.
- **miniWindow**—you can access the title and icon of the mini window, the on-screen representation of a miniaturized window.

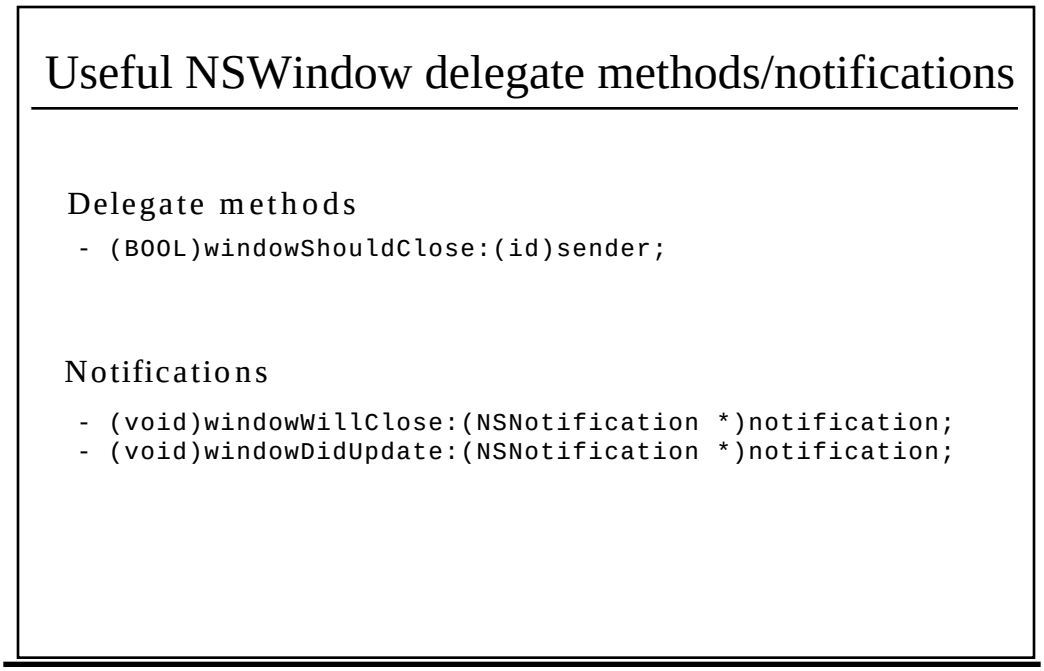


NSWindow contains a hierarchy of NSViews

Any NSControl is a subclass of NSView—among other things, a control is a visual component. To enjoy life on a window, an object must be some sort of NSView. A view lives within a window in a hierarchical arrangement that ties it to other related views—its super view and its subviews. A simple and vivid illustration is a Box which in turn contains a TextField and a Button.

The hierarchy allows a window to recursively display everything within it by telling the topmost view to display and relying on it to pass the message down the line. Each view contains an array of all its subViews and a pointer back up to its superView. The hierarchy is also used for another chain of command—to pass events from a particular view up the hierarchy until it reaches someone who cares. More about this later.

The topmost view is called the contentView. Its main purpose is to provide a background and to completely contain all the subviews that make up the window content. It is possible to dynamically swap the window's content view with another, and then back to the original when asked. This provides a tool for implementing components such as preference panels and tab views creating a sort of multi-window environment all within the boundries of one physical window.



Useful NSWindow delegate methods and notifications

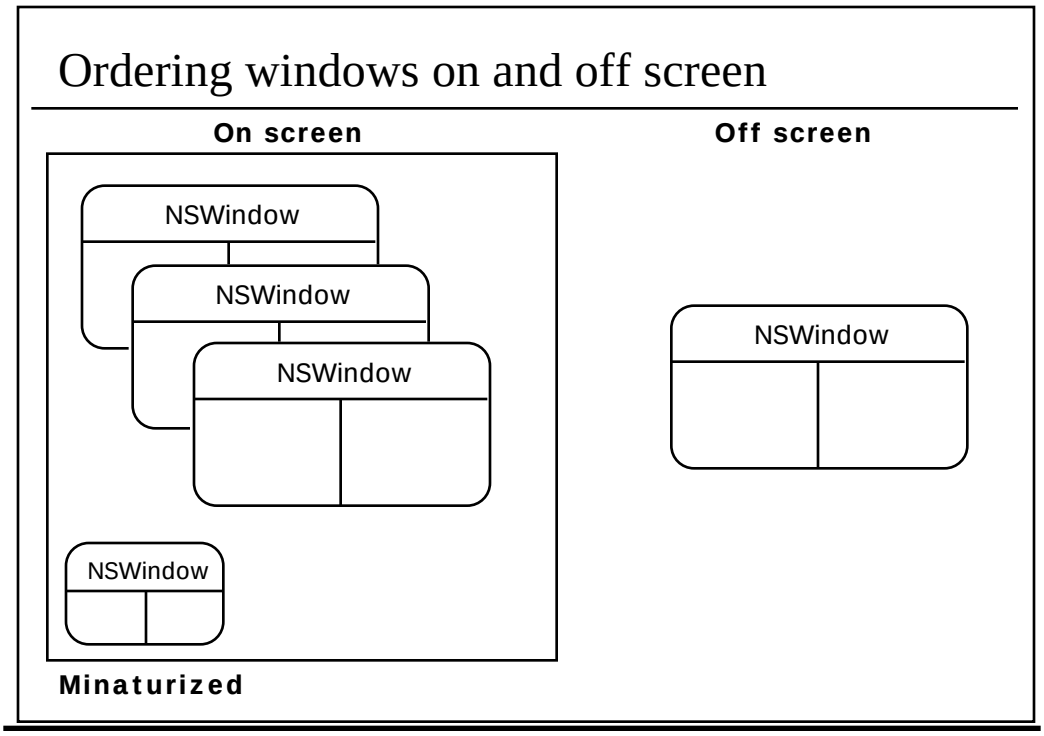
windowShouldClose: is sent when a window wants to close, usually as a result of the user pressing the close button on the window frame. Your delegate can return NO to suppress this. The window will remain open as if nothing happened.

windowWillClose: is sent to notify others that it is about to close. As a notification you can no longer stop this from happening. This is a good place to clean up resources associated with the document, perhaps releasing the document controller itself. If you are going to release the window delegate, be sure to set the window's delegate outlet to nil first.

windowDidUpdate: is sent whenever the window receives an update message. This will be covered in greater detail later. It is useful for implementing inspectors that potentially need to update themselves after every event.

The complete set of NSWindow notifications can be summarized by the following general classifications:

- Key window status (Become, Resign)
- Main window status (Become, Resign)
- Move, Resize
- Miniaturize, Deminiaturize
- Close



Ordering windows on and off screen

Multiple window applications need to control the ordering and visibility of their windows. An NSWindow instance can either be on-screen—visible—or off-screen—invisible. In addition, on-screen windows can be miniaturized when not in use.

Of the on-screen, non-miniaturized windows, only one can be active—the key window. This is the window currently receiving the user's attention, mouse clicks and keystrokes. In the event that an auxiliary panel takes key window status, the window becomes the main window.

There are several window messages that control all these aspects of a window's status. Most do not need to be called manually—they are automatically invoked by the user's actions. A few that may come in handy:

» **makeKeyAndOrderFront:**—bring the window to the front of the window list and make it the key window. This is typically used after loading a nib or when a component is asked to show its window

» **orderOut:**—this makes the window disappear by moving it off-screen, out of the window list. If a window is not configured to be released when closed, it is sent **orderOut:** where it remains until a subsequent **makeKeyAndOrderFront:** message. You can configure whether or not a window or panel should be released when closed in Interface Builder.

Tracking modifications within a window

Pressing a button
- target/action method

Typing in a text field
- target/action method
- delegate/notification

```
- (void)textDidChange:(NSNotification *)notification;
```

Setting the window to reflect modifications

```
[window setDocumentEdited:YES];
```

Asking the window if there were modifications

```
[window isDocumentEdited];
```

Tracking modifications within a window

An NSWindow will remember the state of the document it contains and reflect that state in the user interface. A user needs to know if a document has been edited since the last save. NSWindow does not automatically know when the document is saved nor does it have any idea of what actions result in a modification. You have to tell it. And you have to keep it up to date.

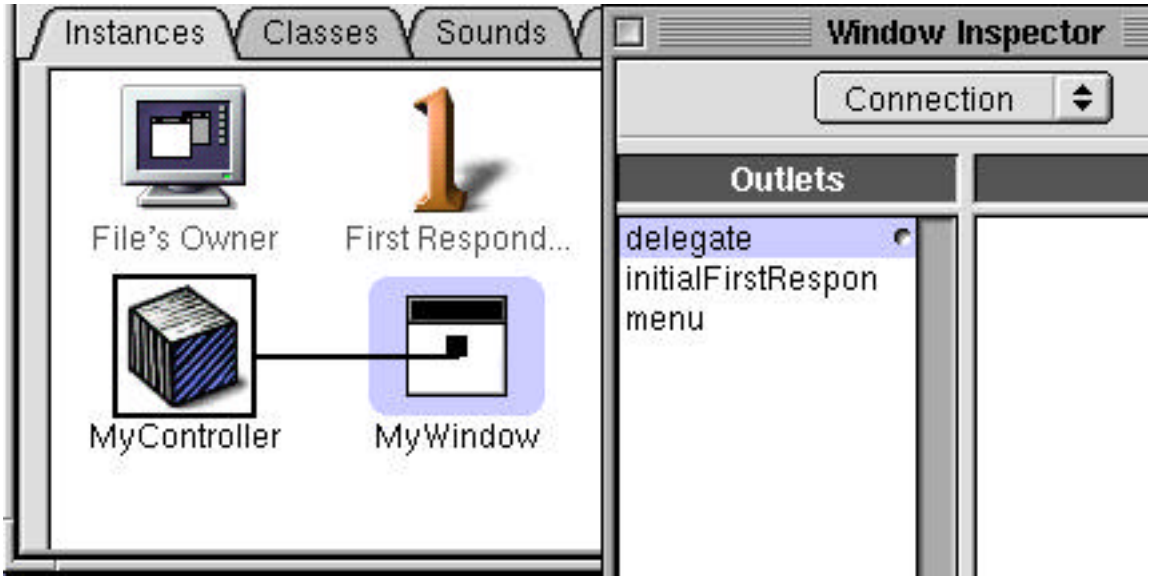
A user edits a document by changing the state of its controls—pressing a button, typing in a text field or text view, modifying a field in a table. Each control or UI component will have its own way of detecting such a change:

» Button—the target/action method knows it was pressed. For 2-state buttons, check to see if the state changed

» Text field—the target/action method knows when <RETURN> is pressed. Text fields post notifications for which a delegate is automatically registered

» Other objects—most other objects such as TextViews and TableViews use notification and delegates like a TextField. Consult the documentation for details.

Once your controller or delegate is informed of a change, it can message the window to report that the document has been edited. When your controller saves the document, all edits are cleared and the window should be messaged to that effect.



Setting a delegate

To configure your object as the delegate of NSApplication, NSWindow, NSTextField or any other object that works with a delegate, you can use Interface Builder to simply connect its outlet. Connect from the object that delegates to the object that is the delegate.

You can also make and break this connection programmatically. To connect a delegate:

```
[object setDelegate: myDelegate];
```

To remove a delegate:

```
[object setDelegate: nil]
```

Note, delegates are not retained by the objects that use them. If you are about to release your delegate object, you must first inform the client object that it no longer has a delegate. Otherwise, future delegate messages and notifications will be sent to an object that has been freed. In a similar fashion, notification centers do not retain their observers.

Registering for notifications

```
- (void)awakeFromNib
{
   [[NSNotificationCenter defaultCenter]
      addObserver:self
      selector:@selector(textDidChange:)
      name:NSControlTextDidChangeNitification
      object:tableView];
}


- (void)textDidChange:(NSNotification *)notification
{
   // process notification here
}
```

Registering for notifications

Notifications are posted to a NotificationCenter which messages all the registered observers that meet the proper criteria. You can add yourself as an observer, supplying the following details:

- selector: the message the notification should send to your object.
- name: the name of the notification you want. If nil, you will get all notifications from the specified object.
- object: the object that will post the notification. This may be a specific object instance or nil in which case you will receive notifications from any object that posts the specified notification names.

Many objects will automatically add their delegates for a set of typical notifications. This may not include all possible notifications that the client object posts. Consult the documentation on a per class basis.

Posting a notification: removing an observer

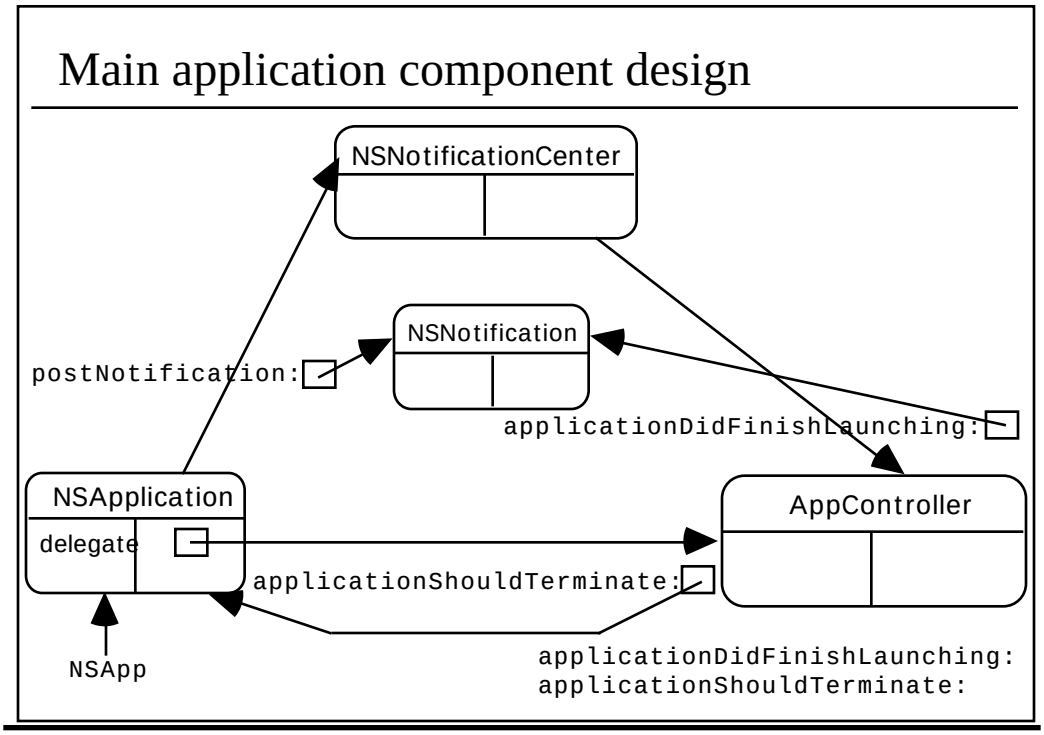
```
- (void)myMethod
{
   NSNotificationCenter *center =
           [NSNotificationCenter defaultCenter];

   [center postNotificationName:@"name"
           object:self];
}

- (void)dealloc
{
   NSNotificationCenter *center =
           [NSNotificationCenter defaultCenter];
   [center removeObserver:self];
   // other class specific dealloc code
   [super dealloc];
}
```

Posting a notification; removing an observer

An object posts a notification using the notification center. The **postNotificatonName:object:** method will generate an NSNotification instance with the specified attributes values and post it to all relevent registered objects.

Removing an Observer

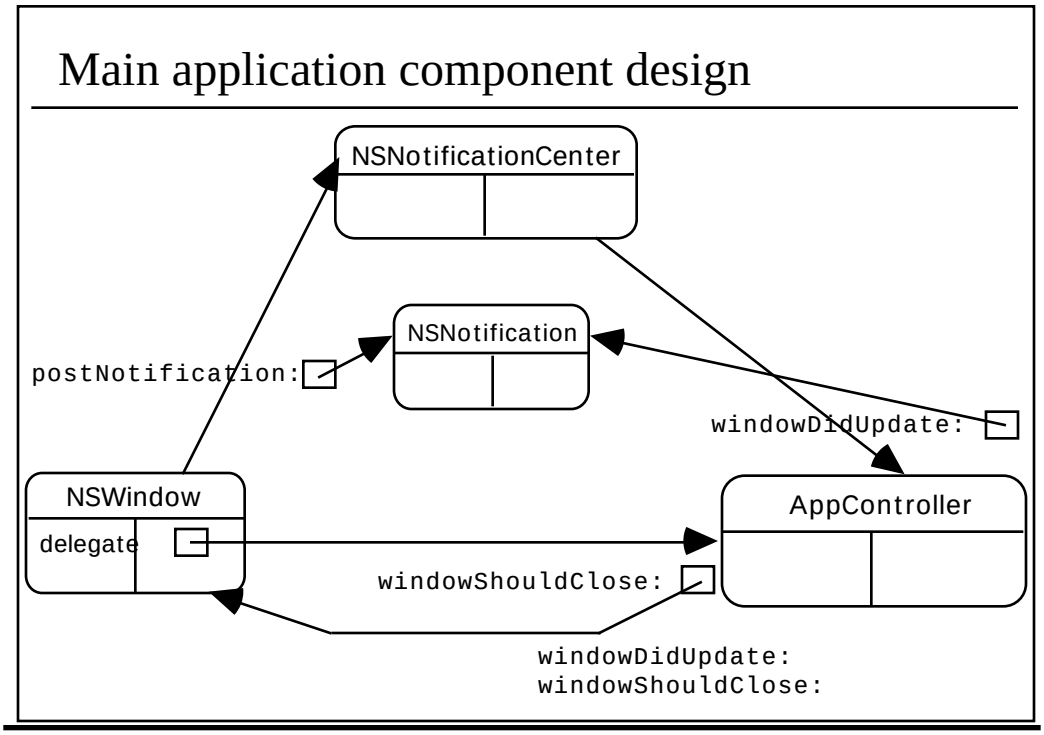
When no longer interested, you can remove your observer object using **removeObserver:** or **removeObserver:name:object:** Notification centers do not retain their observers. If you are going to release an object that is registered for notification, you must first remove it from the center. Otherwise, a future notification will be sent to an object that has already been freed. A good place to do this is in the object's own dealloc message.



Main application component design

The main application component involves NSApplication and its delegate—your AppController. Your AppController does not need an outlet back to NSApplication—it can use the global variable NSApp. This is useful for getting at NSApplication attributes such as the **keyWindow** or the **appIcon**.

As the delegate, your AppController will receive delegate messages directly from NSApplication, such as **applicationShouldTerminate:**. It will indirectly receive notifications such as **applicationDidFinishLaunching:** from the default notification center.



Replicated document component design

A typical document component involves NSWindow and its delegate—your document controller. Your controller will need an outlet back to the window so that it can manipulate window attributes such as the **title** and the **documentEdited** status.

As the delegate, your document controller will receive delegate messages directly from NSWindow, such as **windowShouldClose:**. It will indirectly receive notifications such as **windowDidUpdate:** via the default notification center.

NSRunAlertPanel before closing or quitting

```
ret = NSRunAlertPanel(@"Close",
                      @"Document has been edited. Save?",
                      @"Save",
                      @"Don't Save",
                      @"Cancel");
switch (ret) {
  case NSAlertDefaultReturn:
    [self save: nil];
    break;
  case NSAlertAlternativeReturn:
    break;
  case NSAlertOtherReturn:
    return NO;
    break;
}
return YES;
```

NSRunAlertPanel before closing or quitting

What happens if a user tries to close a window whose document has been edited without being saved? What about quitting the entire application under these circumstances? As a delegate of either NSWindow or NSApplication, you will be informed. You can warn and solicit further input from the user.

NSRunAlertPanel does just this. It presents a modal panel, a dialog, which forces the user to make a choice before interacting with the application in any other way. Using arguments, you can customize the panel for different scenarios:

- Title—defaults to “Alert”.
- Message—takes a **printf** style format string with optional arguments passed after the three button labels.
- Buttons and their labels—up to three, referred to as Default, Alternate, and Other. These appear from right to left on the panel and are passed as arguments from left to right. If you need only 1 or 2 buttons, pass nil for those not needed.

The return value from **NSRunAlertPanel** indicates which button was pressed. You check it against the constants defined by the Application Kit: NSAlertDefaultButton, NSAlertAlternateButton, etc.

Important ideas from this section

- Every application has one instance of NSApplication which serves as a bridge to the OS platform
- Your custom AppController should be a delegate of NSApplication to track and service important state changes in the application
- Your custom document controller should be a delegate of NSWindow to track and service important state changes in the window
- Windows are easily ordered on and off screen
- Delegates are automatically registered for related notifications if they respond to them
- NSAlertPanel is a common and customizable UI component for messaging the user or providing the user with control over the course of action

Classes featured in this section

- NSApplication
- NSWindow
- NSNotificationCenter, NSNotification
- NSRunAlertPanel (function)

REVIEW MULTIPLE DOCUMENT APPLICATIONS

1. What is the fundamental difference between an NSApplication notification message and a delegate message?

2. You intend for your AppController to get the **applicationWillTerminate:** message but it does not. Name some of the possible reasons why.

3. Your document controller is the delegate of its associated window. It is time to release the controller. Are there any concerns given its relationship to the window object?

4. Why might the concept of a document be abstracted from the reality of a window object? Why might your AppController use its own collection of document objects rather than simply the NSApplication's window list?

EXERCISE 11.1 A MULTIPLE DOCUMENT APPLICATION TEMPLATE

A common application model involves the concept of a document. The application can open several documents at one time, allow modifications to them in any arbitrary sequence, and save the new versions to permanent storage. Usually, each document will have a separate window.

In this exercise you evolve the previous application to supporting multiple documents. The AppController is extended to manage the list of documents, and a new class, DocController, is introduced to manage each individual document. The AppController will be the delegate of NSApplication. The DocController will be the delegate of each document window. It responds to certain window delegate messages in order to catch certain actions, such as the window closing, in order to intervene appropriately.

Objectives

After completing this exercise, you will be able to:

- Build a multiple-window application
- Use delegation to control the behavior of document windows

Exercise—Stage 1

1. Make a copy of the previous project to keep a reference copy. Open the new project and change the project name to **MultiDoc**, in order to distinguish it from earlier versions. Save it.

2. Create a new Component subproject called **Documents**. The document code and interface go into this subproject to keep it distinct from your main application files. **DocController.h** and **DocController.m** are provided for you in the **Exercise Materials** area. Drag these files into the Classes suitcase of the Documents subproject.

3. Using Interface Builder, design the document interface—for now it will simply be an empty window:

    - Create a new—empty—nib file. Select New Empty under New Module from the File menu.
    - Drag and drop a window from Interface Builder's Windows palette into the instances window.
    - Using the inspector, make sure the "Release when closed flag" is set. When a document window is closed, the NSWindow instance should be released.

4. Incorporate the DocController class into the nib file:

    - Drag your copy of **DocController.h** into the classes window
    - Make the File's Owner class the DocController class
    - Connect the File's Owner **window** outlet to the document window
    - Connect the window's **delegate** outlet to File's Owner—DocController is the delegate of its window
    - Save the nib file as **Document.nib** in the Document subproject folder

5. Open **DocController.m** using using Project Builder. You will see it is a template and you need to complete the methods in the file. Don't worry about finishing all the methods at once—they will be finished in later parts of the exercise, or as enhancements.

    - Complete the **init** and **dealloc** methods. In **init** you need to load the nib file so that for every new document, a new window is created.
    - Implement the **show:** method. It displays the window by making it key and ordering it to the front of the screen.

6. Use Project Builder's File Attributes Inspector and make **DocController.h** and **AppController.h** Project Header files. This makes them visible to the rest of the project. You can now add an import statement for the DocController class to your **AppController.m** file, so that the methods for the DocController class are declared. Notice **AppController.h** is already included in **DocController.m**.

7. Next you need to modify the AppController so that it can create new documents upon request:

    - Add the **newDocument:** target/action method to the AppController class **.h** and **.m** files. It should create a new DocController instance, and then send it **show:** to make the new document window appear on the screen.

8. Now you can set up the connections in **Main.nib** to the AppController for the desired behavior.

    - Open the **Main.nib** in Interface Builder if it's not open already.
    - Read the new **AppController.h** into Interface Builder by dragging it into the instances window.
    - If the menu does not already have one, add the File submenu to your application's main menu from Interface Builder's menu palette.
    - Connect the New menu item in the File menu to the AppController's **newDocument:** action.
    - Connect your AppController as NSApplication's delegate. The NSApplication instance is the File's Owner of the main nib. This will enable it to receive NSApplication delegate and notification messages.

9. Add an **applicationDidFinishLaunching:** method to AppController which opens a new document when the application starts up.

10. Delete the main window from the Main.nib since it isn't needed any more. You can also delete any methods and instance variables from the Strings application.

11. Make sure everything is saved and then build the project.

At this point, the application should start and present a new empty document. Your New menu item should create new documents. Run the program and see if you can create several documents.

Stage 2

1. Now, you can extend the application to understand modification of documents. This involves the document interface and the DocController.

    - Implement the **isDocumentEdited:** and **setDocumentEdited:** methods which indicate if the document is modified or not. To keep track of the document's state, use the NSWindow methods (**isDocumentEdited:**, **setDocumentEdited:**). The window provides visual feedback that it has been modified. The controller can query the window when it needs to determine the current state.
    - Implement the **modify:** target/action method. This should change the state of the document to edited using the **setDocumentEdited:** method. You may want to include a log message—using NSLog()—to indicate it has been invoked.
    - Position a button on the document window. Change its label to **Modify**, as in the picture. This provides a simple user interface to test the behavior of the DocController in terms of tracking document modifications.

2. Update the list of actions for File's Owner in Interface Builder by re-reading the DocController header file.

3. Connect the Modify button File's Owner's **modify:** action.

4. Re-build the application and create a few new documents. Change one and check that it is modified. You will find you can close a modified without warning. Normally, an application should warn and prompt the user to save changes. The next stage of the exercise adds this capability.

Stage 3

1. Return to the DocController class and implement a **windowShouldClose:** window delegate method. The method should check if the document is modified and, if so, display an alert panel requesting the user to confirm that changes should be saved, or not, or to cancel the operation. Ensure you have set up the delegate connection correctly. Use **NSRunAlertPanel**.

2. Rebuild the application and check that a new window comes up, that you can modify it and that when you try to close it, you get an alert panel warning.

Enhancements

» Currently each new window appears on top of the previous one. **Exercise Materials** contains a code snippet called **staggerWindow.m**. Add this code to your DocController and use it to position the window after nib loading.

» You may notice in the pictures that each window has a different title. Modify DocController's **init** method so that each new document window has a unique title.

» The AppController is managing a group of documents. Currently the only way for it to access each individual document is via the application's window list, by reference to each window's delegate. This works in the simple case of one window per document, but not for others. To track each document directly, modify the AppController so it holds an array of documents. Add two methods r**egisterDocument:** and **unregisterDocument:** to add to and remove documents from the AppController's array. New documents should be registered and documents should unregister themselves before they disappear.

Note: Up until this point, each new document has probably been leaking away when closed. Think carefully how you now deal with keeping the retain count correct and about how you might avoid leaks even without the above registration. This might, for example, involve getting the DocController to release itself in the **windowShouldClose:** method. Be careful in this case to inform the window that you are no longer its delegate.

» Although your DocController will catch attempts to close a modified window, what happens when you try to quit the entire application in this state? Implement an **applicationWillTerminate:** method in your AppController to catch the event and fix this problem. You can use DocController's **close:** method to request documents to close. If you want to be very sophisticated, you can go through the document list first to see if any documents are modified, and then only display an alert panel if there are modified documents.