Chapter 5

NOTIFICATION AND DELEGATION

CHAPTER 5 NOTIFICATION AND DELEGATION

A framework can never be complete. It can cover much of the terrain, but it can't anticipate all the details of every application or what additional structure they'll need. Therefore, frameworks generally provide ways for you to hook your own objects up to framework objects. —Object-Oriented Programming and the Objective-C Language

Goal

To use notification and delegation to extend the behavior of a framework.

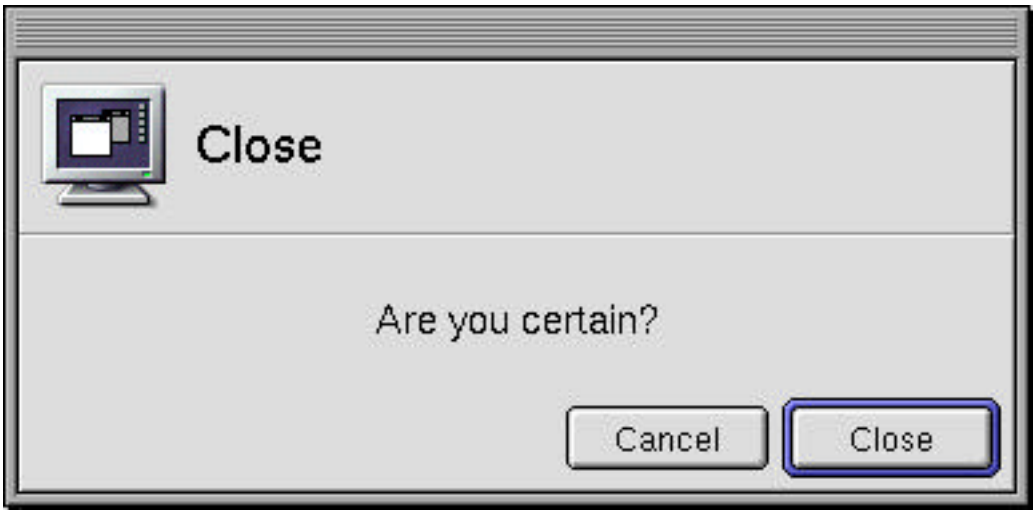
Prerequisites

A basic understanding of objects and messaging.

Objectives

After completing this chapter, you'll be able to:

- Create objects that register for and receive notifications
- Create objects that act as delegates

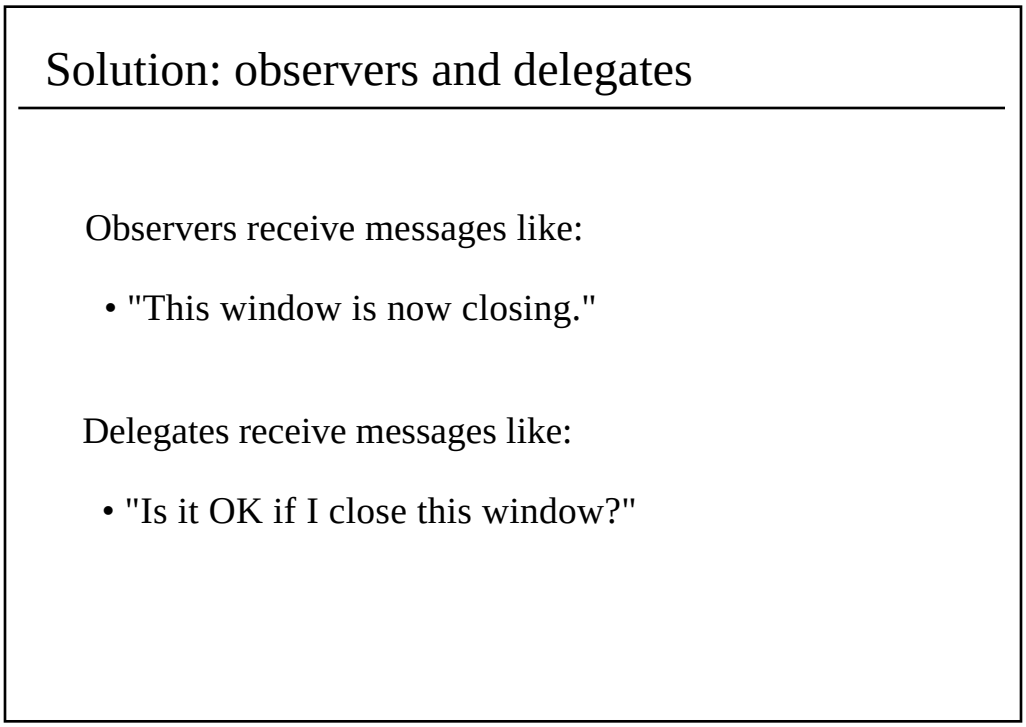


The problem

You've developed an excellent application for editing 1040EZ tax forms. When the user tries to close a window containing an edited document, you want to raise an alert panel asking if they want to save the document. How can you get informed that the window is about to close?

There are many situations like this—you want to be notified about an event that happens to some other object. For example, you want to know when someone types something in a text field. Or when the value of an object changes. Or when a window resizes. Sometimes you also want to be able to control what happens. That window is not allowed to close right now, because it's displaying vital information. Or the user should be allowed a chance to change their mind before the application quits.

The problem is that your objects need this information even when they didn't initiate the action. For example, closing a window typically occurs because the user pushed the close box on the window. How can you find out about this kind of event?



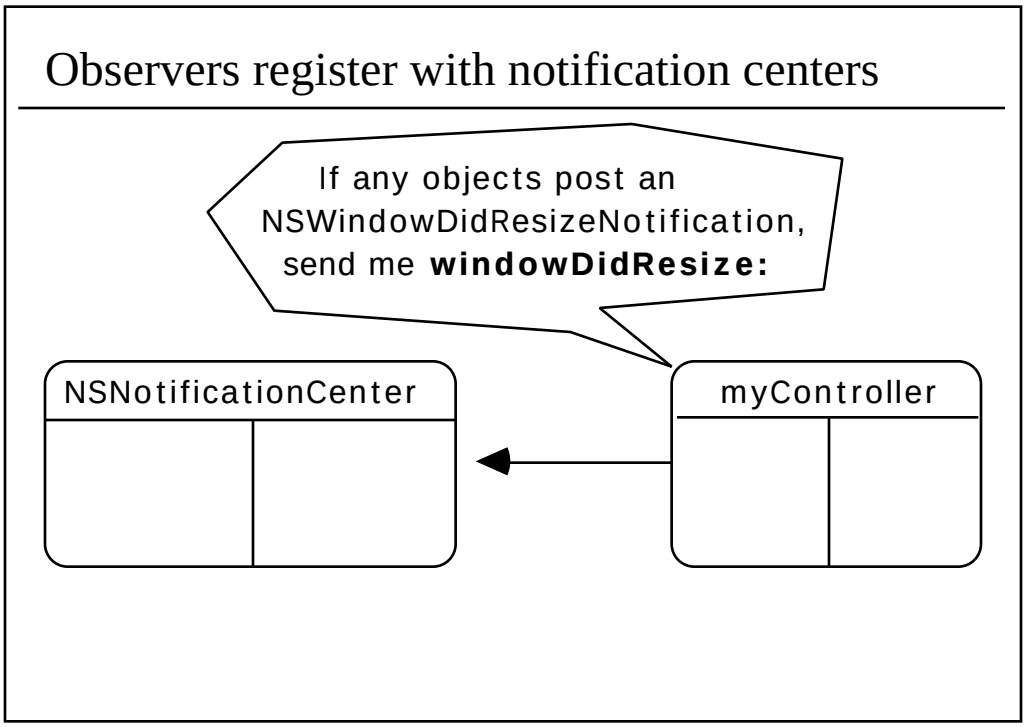
Solution: observers and delegates

The solution to the problem is to provide two mechanisms by which an object can find out about events that happen to other objects.

Observers are simply informed of an event. They have an opportunity to execute code, but are not permitted to interfere with the event. For example, an observer object might see that the window was about to close and save the document, but it can't prevent the window from closing.

Delegates can actually affect the outcome of the event. For example, a delegate might see that the window was about to close and raise an alert: "Do you really want to close this window?" The delegate could then prevent the window from closing.

An object can have many observers, but only one delegate. Many objects might want to know that something is happening, but only one is allowed to affect the outcome.



Observers register with notification centers

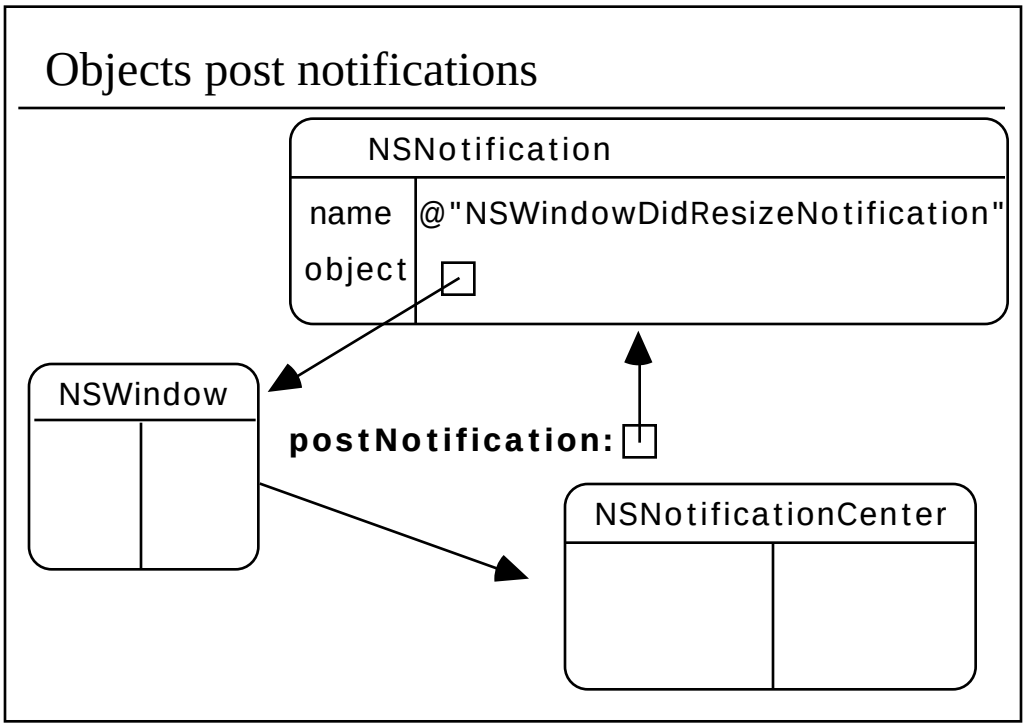
The notification center is like a bulletin board. Objects register themselves as being interested in a certain type of notification from a certain object.

Usually there is only one notification center in the whole application. It is an instance of the class NSNotificationCenter, and can be accessed using NSNotificationCenter's **defaultCenter** method.

The method used for registering an observer with an NSNotificationCenter is:

```
- (void)addObserver:(id)anObserver
      selector:(SEL)aSelector
      name:(NSString *)aName
      object:(id)anObject
```

This registers **anObserver** such that it receives the message **aSelector** when a notification with the name **aName** is posted by **anObject**. If **aName** is nil, the observer receives all notifications from **anObject**. If **anObject** is nil, the observer receives all notifications called **aName**.



Objects post notifications

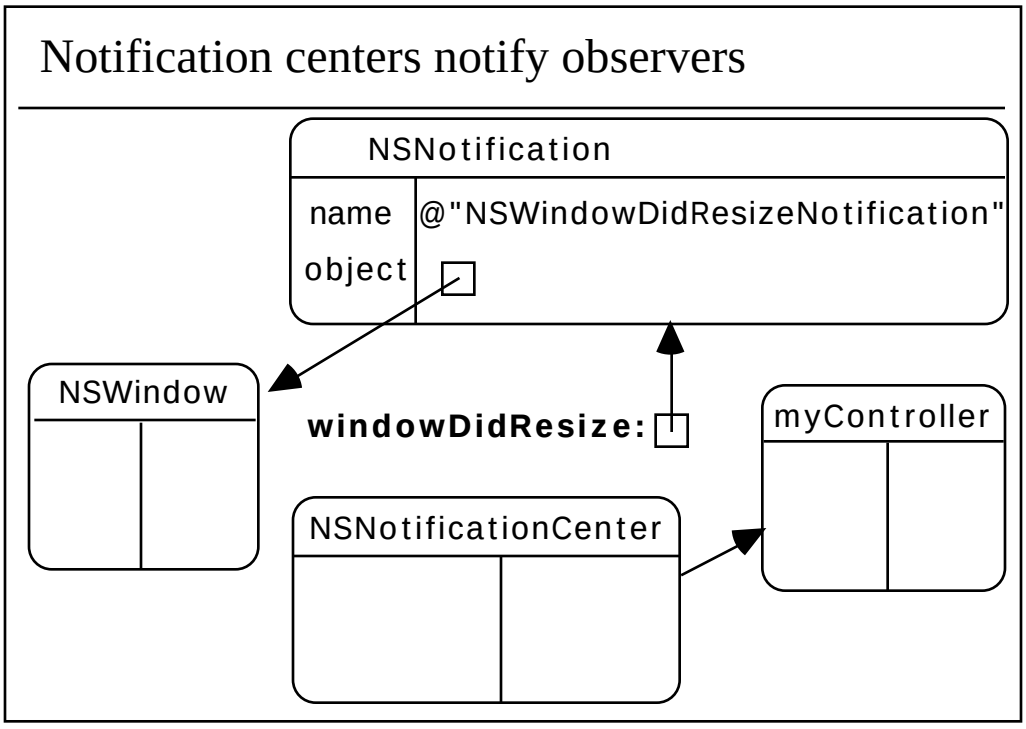
While the application is running, objects can post notifications to the notification center. The notification center looks at its list of observers and sends messages to those that registered as being interested in the specific type of notification from the object that posts it.

A notification is an instance of NSNotification. It has a name and an object—generally the object that posted the notification. Notifications are sent to observers. Because the notification has a pointer back to the object that posted the notification, observers can use the notification to find out what object posted it and communicate with that object directly, if they need to.

Three objects are involved in posting a notification:

- The object posting the notification—NSWindow in this case
- The notification—an instance of NSNotification
- The notification center—an instance of NSNotificationCenter, probably the default center

Note that observers are essentially passive. They rely on the objects they are observing to post the appropriate notifications. If an object doesn't post notifications, it does no good to observe it.



Notification centers notify observers

When the notification center receives a notification that some observer is interested in, it forwards the notification to the observer. The notification sends the message requested by the observer when it registered, including the notification as the argument.

Notification is really a broadcast service -- a notification center forwards notifications to any object that's interested, and will notify them in whatever way they want.

Registering an observer

There are two steps to registering an object as an observer.

First, you get the default instance of NSNotificationCenter using NSNotficationCenter's **defaultCenter** class method. All the objects provided with OPENSTEP post their notifications to the default notification center. To receive these notifications, you need to register with the default center.

Then, register your object as an observer with the default notification center. NSNotificationCenter declares the following method for registering observers:

```
(void)addObserver:(id)anObserver
   selector:(SEL)aSelector
   name:(NSString *)notificationName
   object:(id)anObject
```

Note that you can specify what message you want sent with the notification. In this way registering as an observer is very much like setting yourself up as a target in the target/action design pattern. The key difference is an object can have many observers, but only one target.

When you register as an observer, the arguments **anObserver** and **aSelector** must be non-nil. The name or the object to observe, however, can be nil. When the observed object is nil, the observer gets the specified notifications regardless of what object posted them. Similarly, the name of the notification can be nil. When the name is nil, the observer gets notifications from the observed object regardless of what type they are.

Objects typically register themselves as observers in their initialization method. Here is an **init** method that registers the object being initialized as an observer for NSWindowDidResizeNotification's posted by any object:

```
- (id)init
{
   NSNotificationCenter *defaultCenter;

   self = [super init];
   defaultCenter = [NSNotificationCenter defaultCenter];
   [defaultCenter addObserver:self
      selector:@selector(windowDidResize:)
      name:@"NSWindowDidResizeNotification"
      object:nil];
   return self;
}
```

Initialization methods are discussed in more detail in Chapter 12: Inheritance.

Removing an observer

```
- (void)dealloc
{
    NSNotificationCenter *defaultCenter;

    defaultCenter = [NSNotificationCenter
                        defaultCenter];
    [defaultCenter removeObserver:self];
    // other deallocation code
    [super dealloc];
}
```

Removing an observer

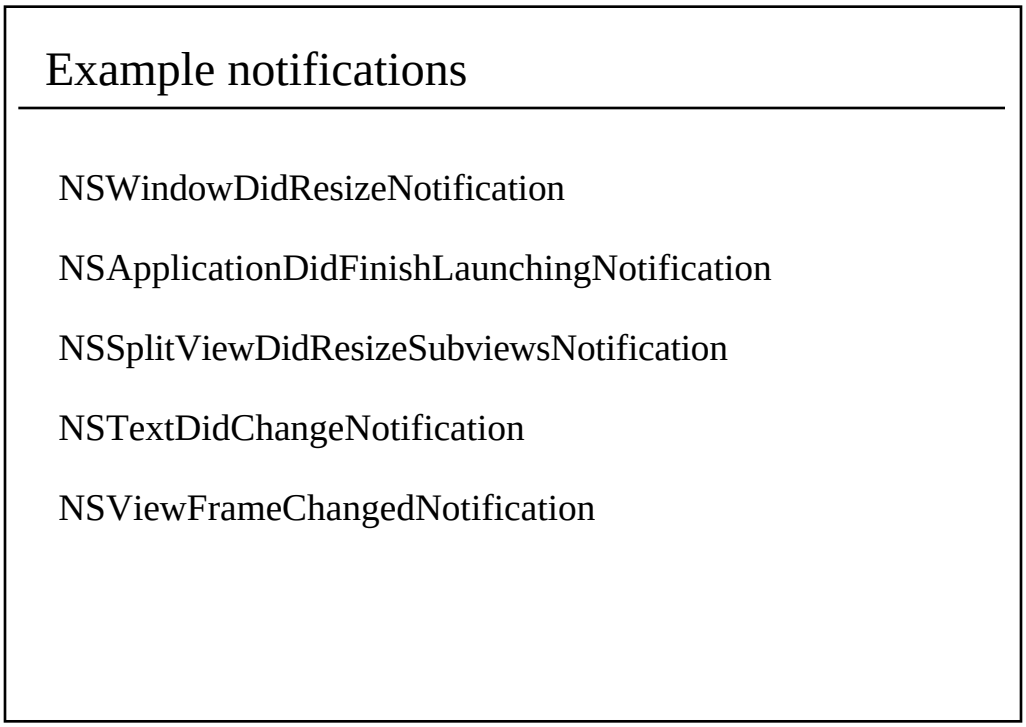
Notification centers do not retain observer objects, so you should be careful to remove any observers before they are deallocated. The reason notification centers don't retain their observers is the notification center has no way of knowing when the observer should go away. So the notification center would never release the observer, and your program would leak memory.

The rule to follow is: If objectA registers objectB with a notification center, objectA must remove objectB from the notification center before objectA releases objectB.

Typically, objects register themselves as observers in their initialization method. In this case, the object should remove itself from the notification center in its **dealloc** method. For example:

```
- (void)dealloc
{
   NSNotificationCenter *defaultCenter;

   defaultCenter = [NSNotificationCenter defaultCenter];
   [defaultCenter removeObserver:self];
   // other deallocation code
   [super dealloc];
}
```

The **dealloc** method is discussed in more detail in Chapter 12: Inheritance.

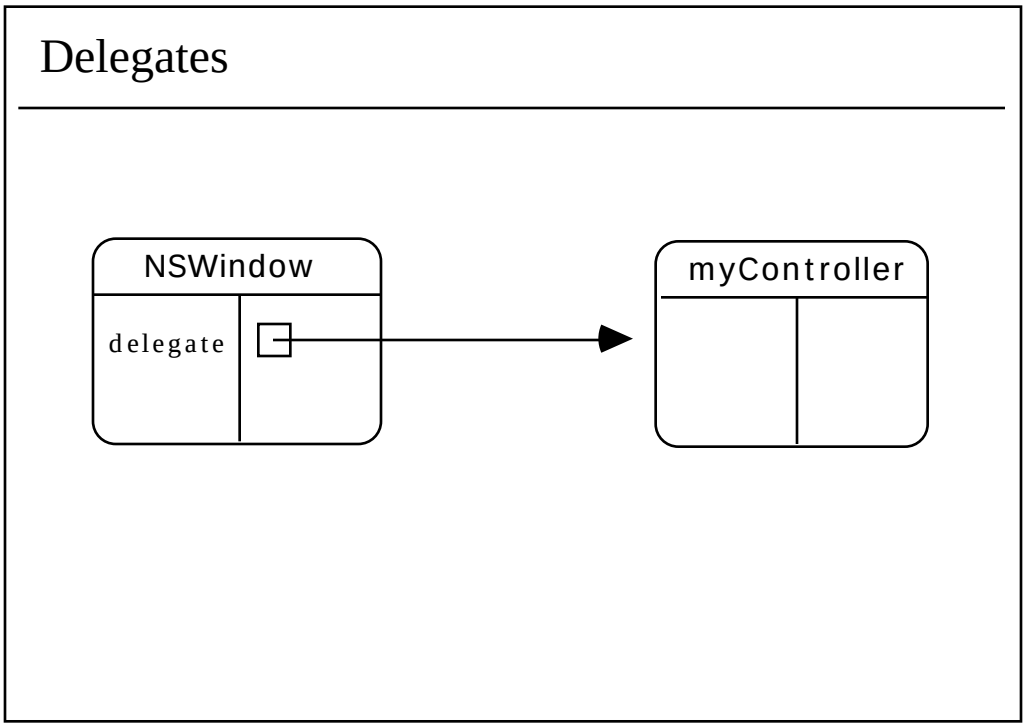


Example notifications

Several objects in the Application Kit post notifications to the notification center. These objects include:

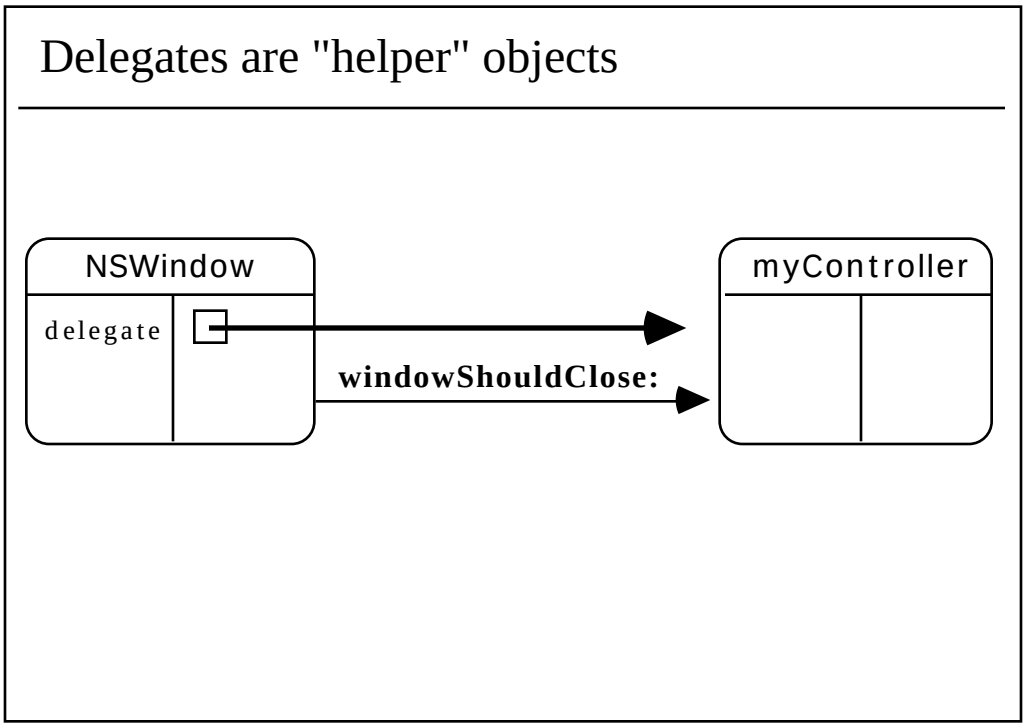
- NSWindow
- NSApplication
- NSSplitView
- NSText
- NSView
- NSTableView

When you use a user interface object from the Application Kit, you should familiarize yourself with the notifications it posts. Using notifications gives you a much more sophisticated level of control over how your application works.



Delegates

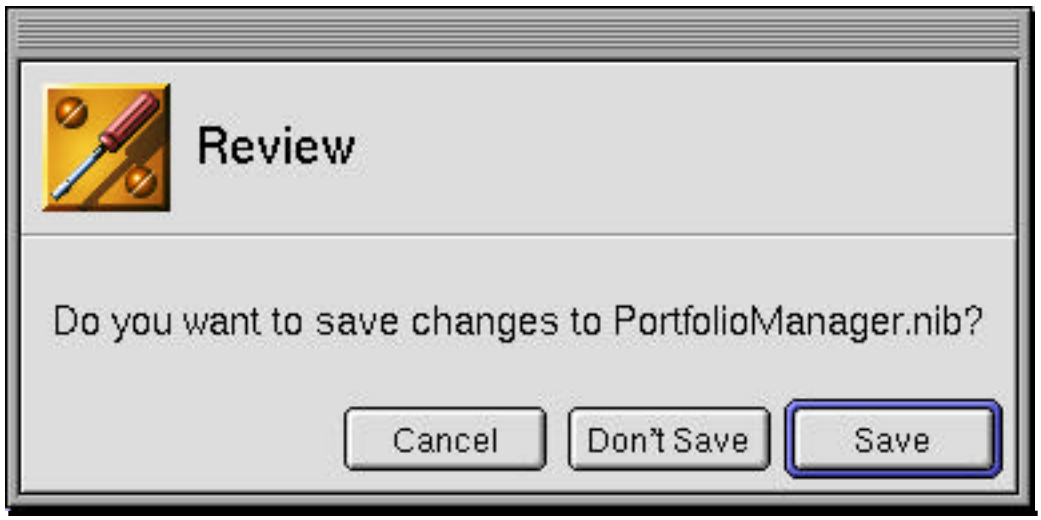
Several objects in the Application Kit have an outlet called **delegate**. This outlet can be set to refer to any sort of “helper” object.



Delegates are “helper” objects

The object sends requests to its delegate, allowing the delegate to influence its behavior and aid in decision making. For example, NSWindow has a delegate outlet. It sends messages to its delegate like **windowShouldClose:**, asking the delegate if the window should in fact close when someone asks the window to close. This message is sent when the user pushes the close box. The delegate then gets to decide if the window should close or not.

Delegation is very different from notification. Notification is a broadcast service, notifying observers when an event takes place. Delegation is a way of delegating decision making power to some other object. Essentially, delegation allows you to modify the behavior of objects provided in the Application Kit without having to subclass them yourself.



Example of delegation at work

Here is an example of delegation at work. In applications where you can edit documents, there's usually a mechanism for preventing you from losing work by carelessly closing a window. If you try to close a window containing an unsaved document, the delegate of the window brings up an alert panel before allowing the window to close.

Implementing a delegate

A window delegate can give input about some window events

- window resizing
- window hiding
- window moving
- 14 other events in the life of a window

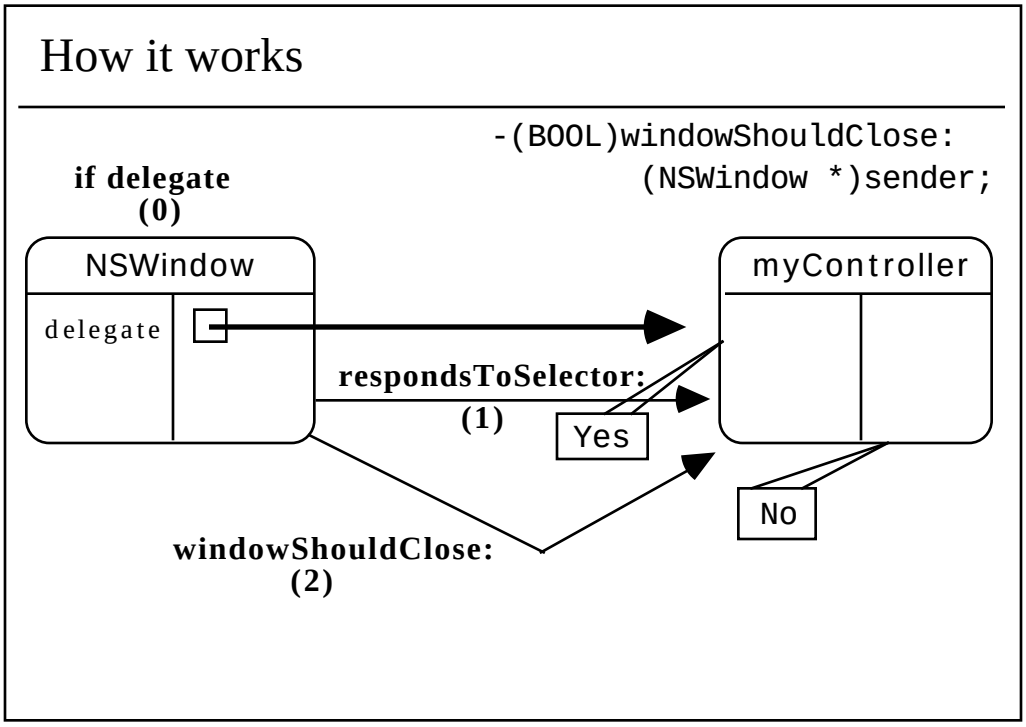
Not every delegate cares about all window events... the delegate implements methods for events that it is interested in.

Implementing a delegate

Windows can do lots of things. Some of the more interesting things are resizing, closing, moving, and hiding. In general, a delegate might want to take action for any of these window events. For example, before a window closes, the delegate might want to give the user the opportunity to save changes, or prevent the window from closing altogether.

For each separate type of event in the life of a window that is suitable for input by a delegate, NSWindow declares a delegate method. For example, the **windowShouldClose:** method is the delegate method for window closing. Before the window closes, it sends a **windowShouldClose:** message to its delegate. The delegate can put code that allows the user to save changes in its **windowShouldClose:** method.

Not every delegate wants to provide input for all seventeen different occurrences in the life of a window. You can implement only the delegate methods you want called, and to leave the other delegate methods unimplemented.



How it works

Before an object sends any message to its delegate, it first makes sure the delegate implements the appropriate method. It does this by using the **respondsToSelector:** method. This is a method all objects inherit from NSObject. The method simply checks to see if the object knows how to respond to a specified message.

For example, before an instance of NSWindow sends a **windowShouldClose:** message to an instance of MyController, it first does something like this:

```
[delegate
respondsToSelector:@selector(windowShouldClose:)];
```

This is marked as step (1) in the diagram. The instance of MyController checks to see if it implements **windowShouldClose:**. Assuming it does, it returns YES.

The instance of NSWindow then sends the **windowShouldClose:** message. This is marked as step (2) in the diagram. The instance of MyController is now free to allow the user to save changes, cancel the close, or perform whatever other action it wants to take. In the example, the user cancels the close so the instance of MyController returns NO.

If the delegate doesn't implement **windowShouldClose:**, no message is sent.

Examples of delegate methods

-(BOOL)windowShouldClose:(id)sender;

-(BOOL)applicationShouldTerminate:(id)sender;

-(BOOL)textShouldEndEditing:(id)sender;

Examples of delegate methods

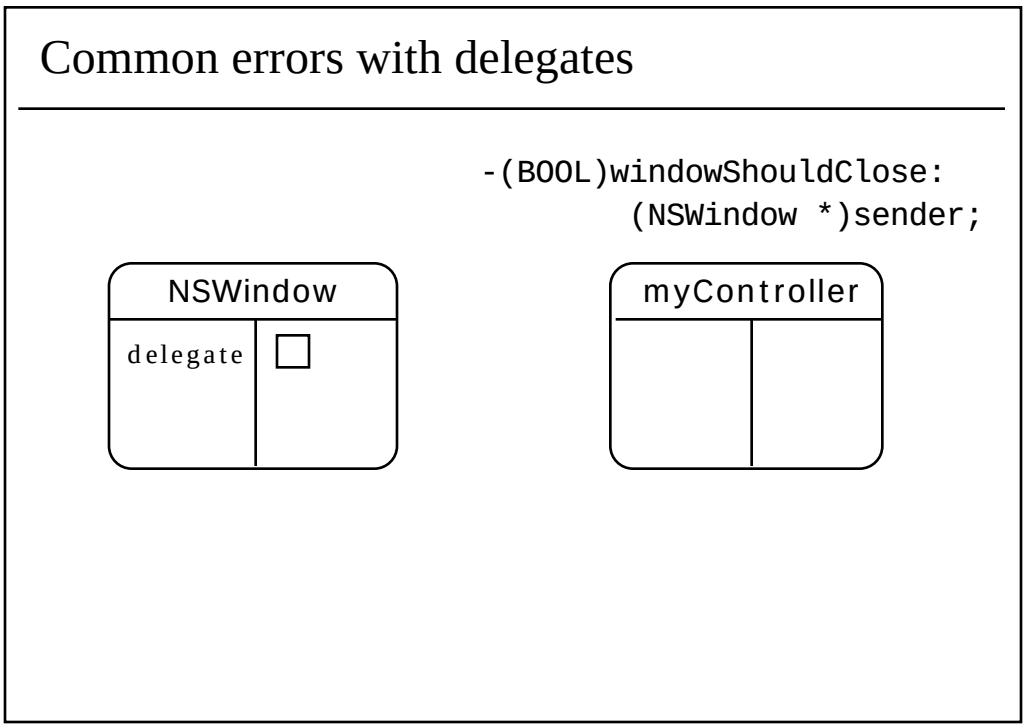
A few classes in the Application Kit have delegates:

- NSWindow
- NSApplication
- NSText

Implementing a delegate method

Here is an example implementation of the delegate message **windowShouldClose:**.

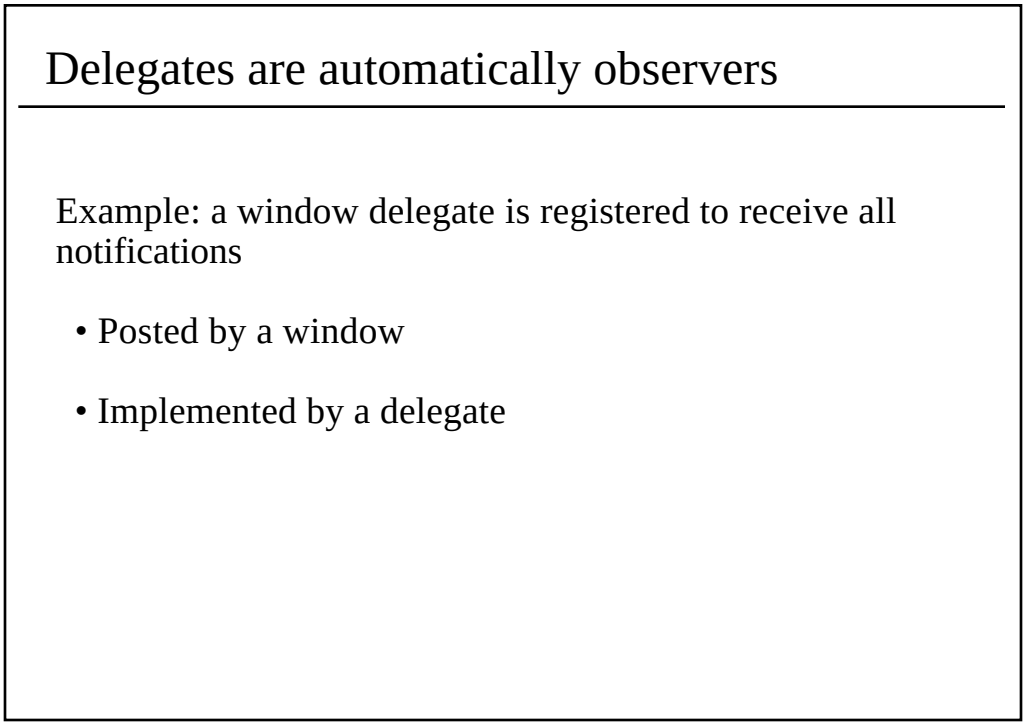
```
- (BOOL)windowShouldClose:(NSWindow *)sender
{
   int answer;

   answer = NSRunAlertPanel(@"Close", @"Are you certain?",
                            @"Close", @"Cancel", nil);
   switch (answer) {
      case NSAlertDefaultReturn:
         return YES;
      default:
         return NO;
   }
}
```



Common errors with delegates

A very common complaint is, “My delegate method isn’t getting called.” This is typically because the name of the method is misspelled, or the **delegate** outlet was never set.

You can set the **delegate** outlet in two ways—in Interface Builder, or programmatically by using the object’s **setDelegate:** accessor method.

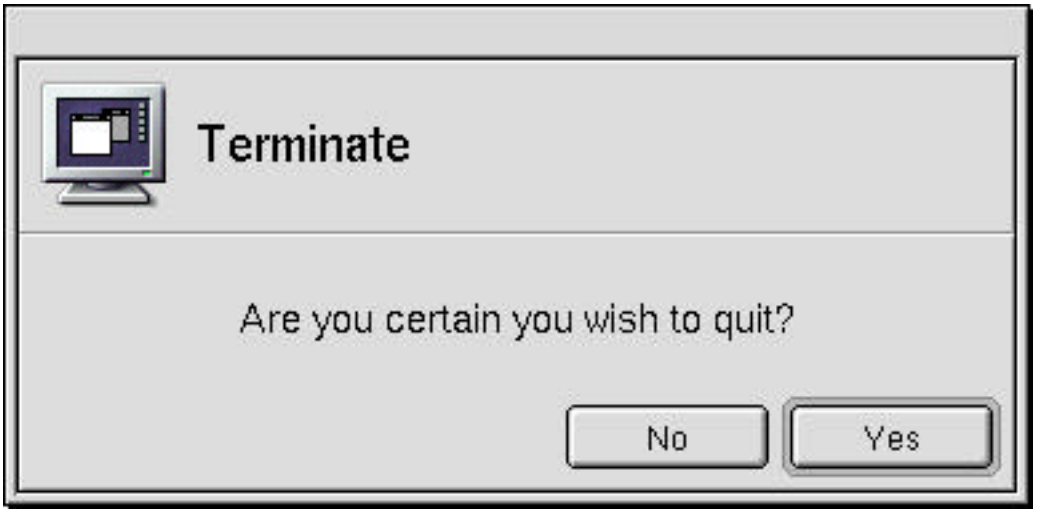


Delegates are automatically observers

Often, a delegate is interested not only in delegate messages, but also notifications. To make this easier, delegates are automatically registered as observers for notifications if they implement a correspondeing notification method.

For example, assume the object **myController** has been set as the delegate of an NSWindow. If **myController** implements the method **windowDidMove:**, it is automatically registered as an observer for NSWindowWillMoveNotifications posted by the window.

DEMONSTRATION 5.1 DELEGATION: A DELEGATE OF NSAPPLICATION



Notification and delegation are powerful design patterns that give you a great deal of control over how an application runs. This demonstration shows how to simply and cleanly create an application that asks the users if they really want to quit after they choose the Quit command.

In this demonstration you create a controller object called ApplicationController. ApplicationController is the controller for the application as a whole. This is where functionality that applies to the entire application goes. In this demonstration you write code to make ApplicationController notice when the application is about to terminate, and give the user a chance to interrupt the process.

Objectives

After completing this demonstration, you'll be able to:

- Create a controller object for the PortfolioManager application
- Make a connection from the File's Owner in Interface Builder
- Make your controller the delegate of NSApplication
- Implement a method that allows users to prevent the application from quitting

Demonstration

1. Create a new application project called **PortfolioManager**. Open **PortfolioManager .nib**.

2. Create a new subclass of NSObject called ApplicationController. Create the .m and .h files for ApplicationController using the Create Files command in Interface Builder's Classes menu.

3. Create an instance of ApplicationController by choosing Instantiate in the Classes menu.

4. Connect the **delegate** outlet of File's Owner to ApplicationController. The File's Owner of the main nib file for an application is the NSApplication object. At run-time, the ApplicationController instance will be NSApplication's delegate and will therefore receive delegate messages.

5. Add the following method declaration to **ApplicationController.h**:

```
- (BOOL)applicationShouldTerminate:(NSApplication *)sender;
```

6. Add the following method definition to **ApplicationController.m**:

```
- (BOOL)applicationShouldTerminate:(NSApplication *)sender
{
   int answer;

   answer = NSRunAlertPanel(@"Terminate",
            @"Are you certain you wish to quit?",
            @"Yes",
            @"No", nil);
   switch (answer) {
      case NSAlertDefaultReturn:
         return YES;
      default:
         return NO;
   }
}
```

   This method runs an attention panel when ApplicationController receives an **applicationShouldTerminate:** message. If the user presses the default button, Yes, the method returns YES and the application quits. Otherwise, the method returns no and the application continues running.

7. Make sure all the files are saved. Build and test the project. Ensure that attempting to quit the application brings up the alert panel, and that the two buttons function as intended.

Important ideas from this section

» Before closing, a window does two things:

  Sends its delegate a **windowShouldClose:** message, if the delegate implements the corresponding method

  Posts an **NSWindowWillClose** notification on the default notification center

» Observers receive notifications of events they can't change

» Delegates can alter events they are informed of

» Notification is done in three parts:

  Observers register with the notification center

  Posters post notifications on the notification center

  The notification center informs the "interested" observers

» Remove observers from the notification center before they are deallocated

» A delegate acts as a helper object for several Application Kit classes

» An object can have many observers, but only one delegate

» Delegates only implement delegate methods they are interested in

» Delegates are automatically registered to receive notifications

REVIEW DELEGATION AND NOTIFICATION

1. What is an observer? How do you make an object an observer?

2. Describe the notification process.

3. What is a delegate? How do you make an object a delegate?

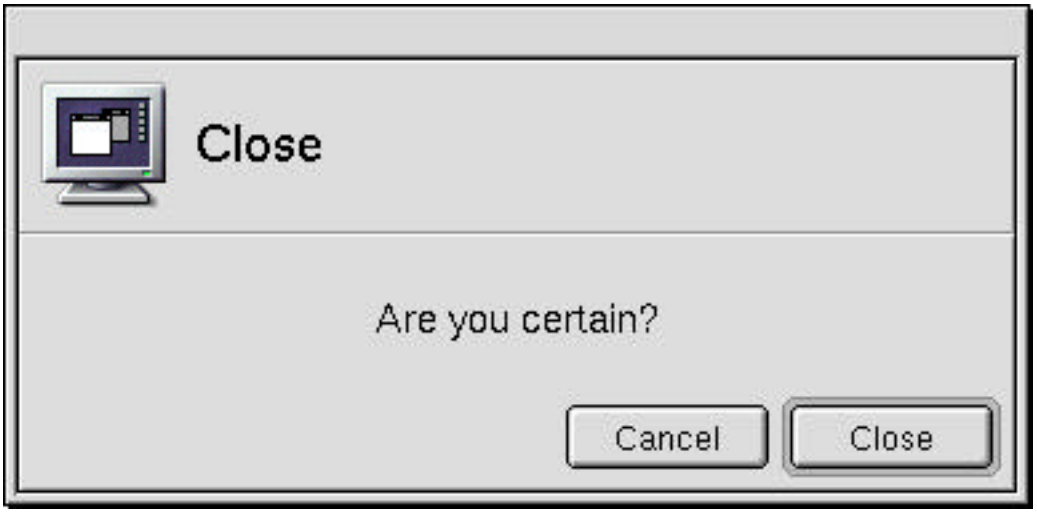
4. What are the following classes for?

    a. NSNotificationCenter

    b. NSNotification

5. Name three classes in the Application Kit that can have both delegates and observers.

EXERCISE 5.1 DELEGATION: A DELEGATE OF NSWINDOW



You've seen how to set up an application delegate in Interface Builder and implement a delegate method. In this exercise, you set the main window's delegate outlet to the instance of ApplicationController. This allows the application to bring up an alert panel if the user attempts to close its main window.

Objectives

After completing this exercise, you'll be able to:

- Set a delegate of NSWindow in Interface Builder
- Add a delegate method to a class you've written
- Use a window delegate to control window closing

Exercise

1. Open the PortfolioManager project from ExerciseMaterials/Provided.

2. Add the method windowShouldClose: to the ApplicationController class. Make it bring up an alert panel that asks the user if it's OK to close the window.

3. Set up the necessary connections in Interface Builder so your instance of ApplicationController receives delegate messages from the application's main window.

4. Build and test the application. Make sure the alert panel comes up when you try to close the main window and that the buttons do what they're supposed to.