Delegates and Data Sources

A delegate is an object that acts on behalf of, or in coordination with, another object when that object encounters an event in a program. The delegating object is often a responder object—that is, an object inheriting from `NSResponder` in AppKit or `UIResponder` in UIKit—that is responding to a user event. The delegate is an object that is delegated control of the user interface for that event, or is at least asked to interpret the event in an application-specific manner.

To better appreciate the value of delegation, it helps to consider an off-the-shelf Cocoa object such as a text field (an instance of `NSTextField` or `UITextField`) or a table view (an instance of `NSTableView` or `UITableView`). These objects are designed to fulfill a specific role in a generic fashion; a window object in the AppKit framework, for example, responds to mouse manipulations of its controls and handles such things as closing, resizing, and moving the physical window. This restricted and generic behavior necessarily limits what the object can know about how an event affects (or will affect) something elsewhere in the application, especially when the affected behavior is specific to your application. Delegation provides a way for your custom object to communicate application-specific behavior to the off-the-shelf object.

The programming mechanism of delegation gives objects a chance to coordinate their appearance and state with changes occurring elsewhere in a program, changes usually brought about by user actions. More importantly, delegation makes it possible for one object to alter the behavior of another object without the need to inherit from it. The delegate is almost always one of your custom objects, and by definition it incorporates application-specific logic that the generic and delegating object cannot possibly know itself.

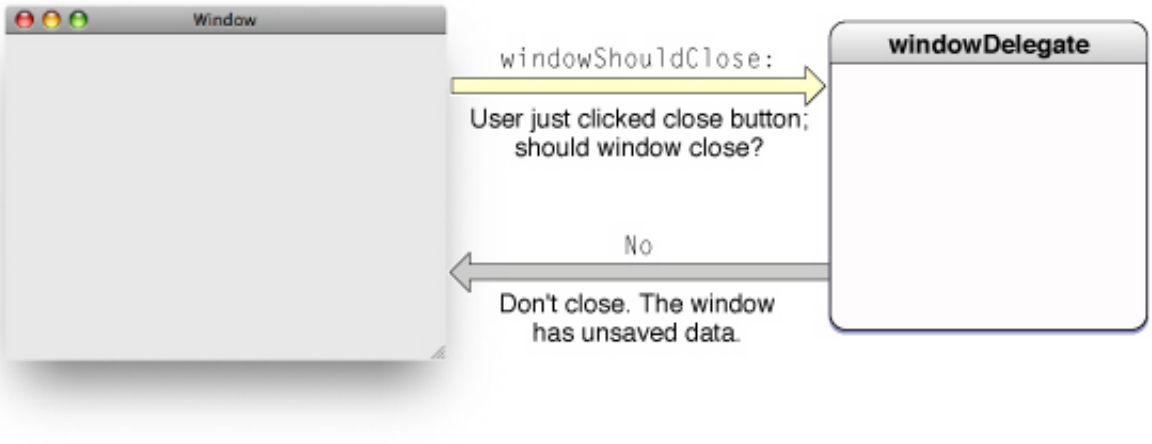
How Delegation Works

The design of the delegation mechanism is simple—see Figure 3-1. The delegating class has an outlet or property, usually one that is named `delegate`; if it is an outlet, it includes methods for setting and accessing the value of the outlet. It also declares, without implementing, one or more methods that constitute a formal protocol or an informal protocol. A formal protocol that uses optional methods—a feature of Objective-C 2.0—is the preferred approach, but both kinds of protocols are used by the Cocoa frameworks for delegation.

In the informal protocol approach, the delegating class declares methods on a category of `NSObject`, and the delegate implements only those methods in which it has an interest in coordinating itself with the delegating object or affecting that object's default behavior. If the delegating class declares a formal protocol, the delegate may choose to implement those methods marked optional, but it must implement the required ones.

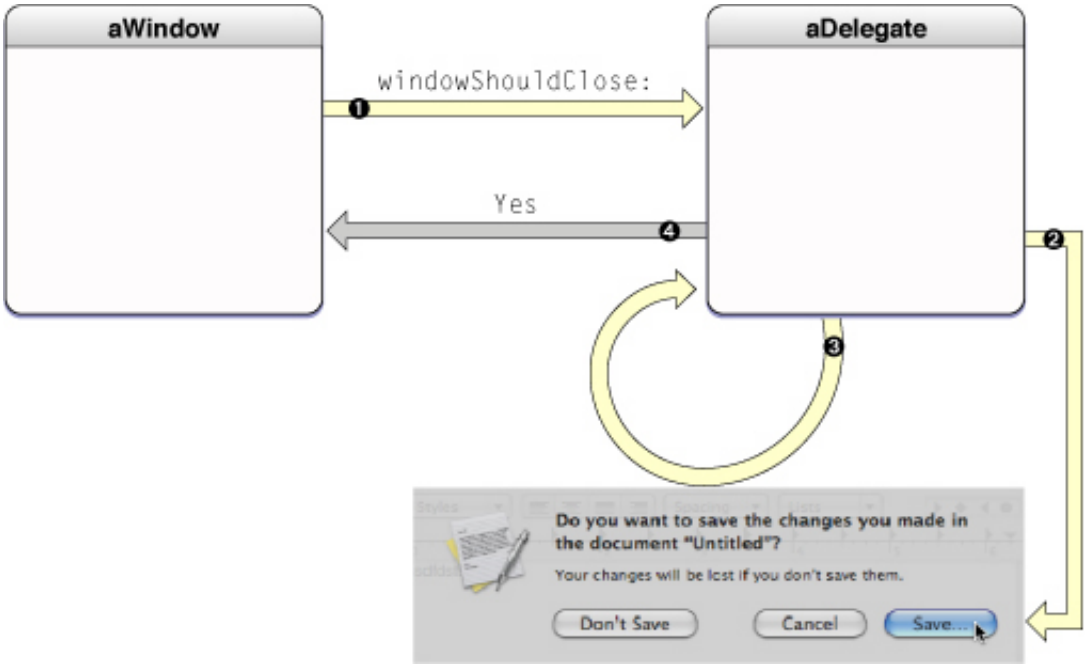
Delegation follows a common design, illustrated by Figure 3-1.

Figure 3-1 The mechanism of delegation



The methods of the protocol mark significant events handled or anticipated by the delegating object. This object wants either to communicate these events to the delegate or, for impending events, to request input or approval from the delegate. For example, when a user clicks the close button of a window in OS X, the window object sends the `windowShouldClose:` message to its delegate; this gives the delegate the opportunity to veto or defer the closing of the window if, for example, the window has associated data that must be saved (see Figure 3-2).

Figure 3-2 A more realistic sequence involving a delegate



The delegating object sends a message only if the delegate implements the method. It makes this discovery by invoking the `NSObject`method `respondsToSelector:` in the delegate first.

The Form of Delegation Messages

Delegation methods have a conventional form. They begin with the name of the AppKit or UIKit object doing the delegating—application, window, control, and so on; this name is in lower-case and without the “NS” or “UI” prefix. Usually (but not always) this object name is followed by an auxiliary verb indicative of the temporal status of the reported event. This verb, in other words, indicates whether the event is about to occur (“Should” or “Will”) or whether it has just occurred (“Did” or “Has”). This temporal distinction helps to categorize those messages that expect a return value and those that don’t. Listing 3-1 includes a few AppKit delegation methods that expect a return value.

Listing 3-1 Sample delegation methods with return values

```
- (BOOL)application:(NSApplication *)sender
    openFile:(NSString *)filename;                          // NSApplication
- (BOOL)application:(UIApplication *)application
    handleOpenURL:(NSURL *)url;                             // UIApplicationDelegate
- (UITableRowIndexSet *)tableView:(NSTableView *)tableView
    willSelectRows:(UITableRowIndexSet *)selection;         // UITableViewDelegate

- (NSRect)windowWillUseStandardFrame:(NSWindow *)window
    defaultFrame:(NSRect)newFrame;                          // NSWindow
```

The delegate that implements these methods can block the impending event (by returning `NO` in the first two methods) or alter a suggested value (the index set and the frame rectangle in the last two methods). It can even defer an impending event; for example, the delegate implementing the `applicationShouldTerminate:`method can delay application termination by returning `NSTerminateLater`.

Other delegation methods are invoked by messages that don’t expect a return value and so are typed to return `void`. These messages are purely informational, and the method names often contain “Did”, “Will”, or some other indication of a transpired or impending event. Listing 3-2 shows a few examples of these kinds of delegation method.

Listing 3-2 Sample delegation methods returning `void`

```
- (void) tableView:(NSTableView*)tableView
    mouseDownInHeaderOfTableColumn:(NSTableColumn *)tableColumn;      // NSTableView
- (void)windowDidMove:(NSNotification *)notification;                // NSWindow
- (void)application:(UIApplication *)application
    willChangeStatusBarFrame:(CGRect)newStatusBarFrame;              // UIApplication
- (void)applicationWillBecomeActive:(NSNotification *)notification;  // NSApplication
```

There are a couple of things to note about this last group of methods. The first is that an auxiliary verb of “Will” (as in the third method) does not necessarily mean that a return value is expected. In this case, the event is imminent and cannot be blocked, but the message gives the delegate an opportunity to prepare the program for the event.

The other point of interest concerns the second and last method declarations in Listing 3-2 . The sole parameter of each of these methods is an `NSNotification` object, which means that these methods are invoked as the result of the posting of a particular notification. For example, the `windowDidMove:` method is associated with the `NSWindow` notification `NSWindowDidMoveNotification`. It's important to understand the relationship of notifications to delegation messages in AppKit. The delegating object automatically makes its delegate an observer of all notifications it posts. All the delegate needs to do is implement the associated method to get the notification.

To make an instance of your custom class the delegate of an AppKit object, simply connect the instance to the `delegate` outlet or property in Interface Builder. Or you can set it programmatically through the delegating object's `setDelegate:` method or `delegate` property, preferably early on, such as in the `awakeFromNib` or `applicationDidFinishLaunching:` method.

Delegation and the Application Frameworks

The delegating object in a Cocoa or Cocoa Touch application is often a responder object such as a `UIApplication`, `NSWindow`, or `NSTableView` object. The delegate object itself is typically, but not necessarily, an object, often a custom object, that controls some part of the application (that is, a coordinating controller object). The following AppKit classes define a delegate:

- `NSApplication`
- `NSBrowser`
- `NSControl`
- `NSDrawer`
- `NSFontManager`
- `NSFontPanel`
- `NSMatrix`
- `NSOutlineView`
- `NSSplitView`
- `NSTableView`
- `NSTabView`
- `NSText`
- `NSTextField`
- `NSTextView`
- `NSWindow`

The UIKit framework also uses delegation extensively and always implements it using formal protocols. The application delegate is extremely important in an application running in iOS because it must respond to application-launch, application-quit, low-memory, and other messages from the application object. The application delegate must adopt the `UIApplicationDelegate` protocol.

Delegating objects do not (and should not) retain their delegates. However, clients of delegating objects (applications, usually) are responsible for ensuring that their delegates are around to receive delegation messages. To do this, they may have to retain the delegate in memory-managed code.

This precaution applies equally to data sources, notification observers, and targets of action messages. Note that in a garbage-collection environment, the reference to the delegate is strong because the retain-cycle problem does not apply.

Some AppKit classes have a more restricted type of delegate called a *modal delegate*. Objects of these classes (`NSOpenPanel`, for example) run modal dialogs that invoke a handler method in the designated delegate when the user clicks the dialog's OK button. Modal delegates are limited in scope to the operation of the modal dialog.

Becoming the Delegate of a Framework Class

A framework class or any other class that implements delegation declares a `delegate` property and a protocol (usually a formal protocol). The protocol lists the required and optional methods that the delegate implements. For an instance of your class to function as the delegate of a framework object, it must do the following:

- Set your object as the delegate (by assigning it to the `delegate` property). You can do this programmatically or through Interface Builder.
- If the protocol is formal, declare that your class adopts the protocol in the class definition. For example:

```
@interface MyControllerClass : UIViewController <UIAlertViewDelegate> {
```

- Implement all required methods of the protocol and any optional methods that you want to participate in.

Locating Objects Through the delegate Property

The existence of delegates has other programmatic uses. For example, with delegates it is easy for two coordinating controllers in the same program to find and communicate with each other. For example, the object controlling the application overall can find the controller of the application's inspector window (assuming it's the current key window) using code similar to the following:

```
id winController = [[NSApp keyWindow] delegate];
```

And your code can find the application-controller object—by definition, the delegate of the global application instance—by doing something similar to the following:

```
id appController = [NSApp delegate];
```

Data Sources

A data source is like a delegate except that, instead of being delegated control of the user interface, it is delegated control of data. A data source is an outlet held by `NSView` and `UIView` objects such as table views and outline views that require a source from which to populate their rows of visible data.

The data source for a view is usually the same object that acts as its delegate, but it can be any object. As with the delegate, the data source must implement one or more methods of an informal protocol to supply the view with the data it needs and, in more advanced implementations, to handle data that users directly edit in such views.

As with delegates, data sources are objects that must be present to receive messages from the objects requesting data. The application that uses them must ensure their persistence, retaining them if necessary in memory-managed code.

Data sources are responsible for the persistence of the objects they hand out to user-interface objects. In other words, they are responsible for the memory management of those objects. However, whenever a view object such as an outline view or table view accesses the data from a data source, it retains the objects as long as it uses the data. But it does not use the data for very long. Typically it holds on to the data only long enough to display it.

Implementing a Delegate for a Custom Class

To implement a delegate for your custom class, complete the following steps:

- Declare the delegate accessor methods in your class header file.

```
- (id)delegate;
- (void)setDelegate:(id)newDelegate;
```

- Implement the accessor methods. In a memory-managed program, to avoid retain cycles, the setter method should not retain or copy your delegate.

```
- (id)delegate {
    return delegate;
}


- (void)setDelegate:(id)newDelegate {
    delegate = newDelegate;
}
```

  In a garbage-collected environment, where retain cycles are not a problem, you should not make the delegate a weak reference (by using the `__weak` type modifier). For more on retain cycles, see *Advanced Memory Management Programming Guide*. For more on weak references in garbage collection, see Garbage Collection for Cocoa Essentials in *Garbage Collection Programming Guide*.

- Declare a formal or informal protocol containing the programmatic interface for the delegate. Informal protocols are categories on the `NSObject` class. If you declare a formal protocol for your delegate, make sure you mark groups of optional methods with the `@optional` directive.

  The Form of Delegation Messages gives advice for naming your own delegation methods.

- Before invoking a delegation method, make sure the delegate implements it by sending it a

`respondsToSelector:` message.

```
- (void)someMethod {
    if ( [delegate respondsToSelector:@selector(operationShouldProceed)] ) {
        if ( [delegate operationShouldProceed] ) {
            // do something appropriate
        }
    }
}
```

The precaution is necessary only for optional methods in a formal protocol or methods of an informal protocol.