Chapter 13

EVENTS AND RESPONDERS

CHAPTER 13 EVENTS AND RESPONDERS

Goal

To understand how user events reach application objects through the responder chain and utilize the features of the first responder.

Prerequisites

A solid understanding of delegation and a good sense of how a user interacts with graphical user interfaces.

Objectives

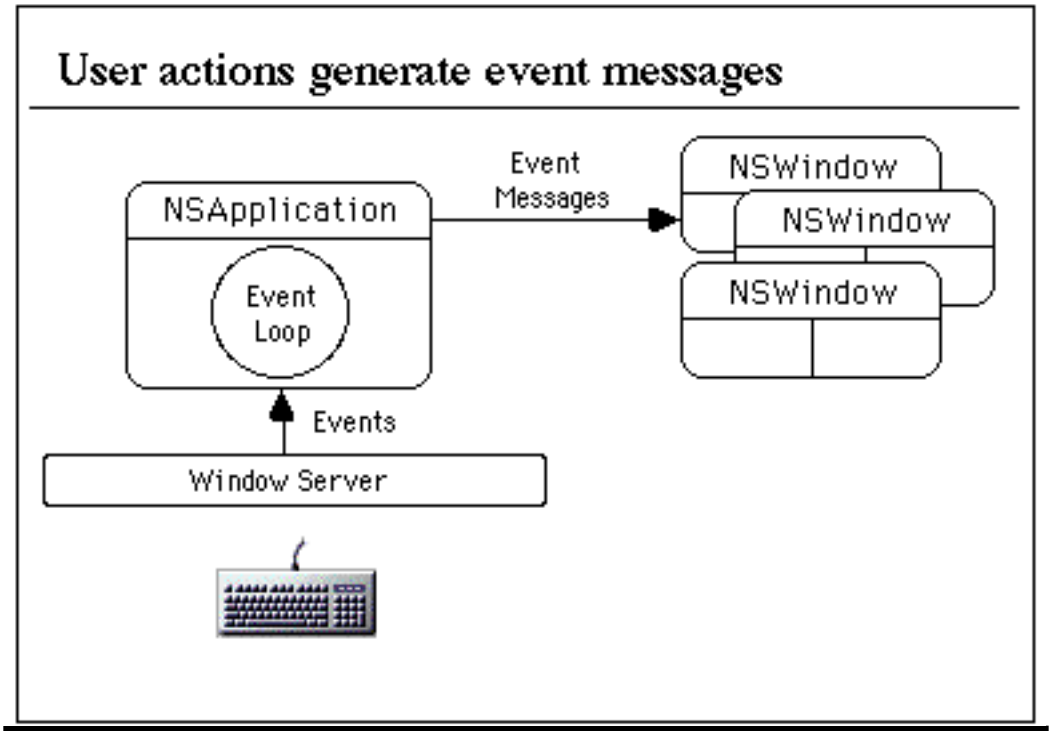
At the end of this section, you will be able to:

- Describe how user events result in messages to responders and enumerate the objects in the responder chain
- Explain the utility of the key window and first responder for messaging the currently active user interface element
- Implement multi-document menu actions that target a document controller using first responder

Reading

Developer Tutorial

NSResponder class in the Application Kit.



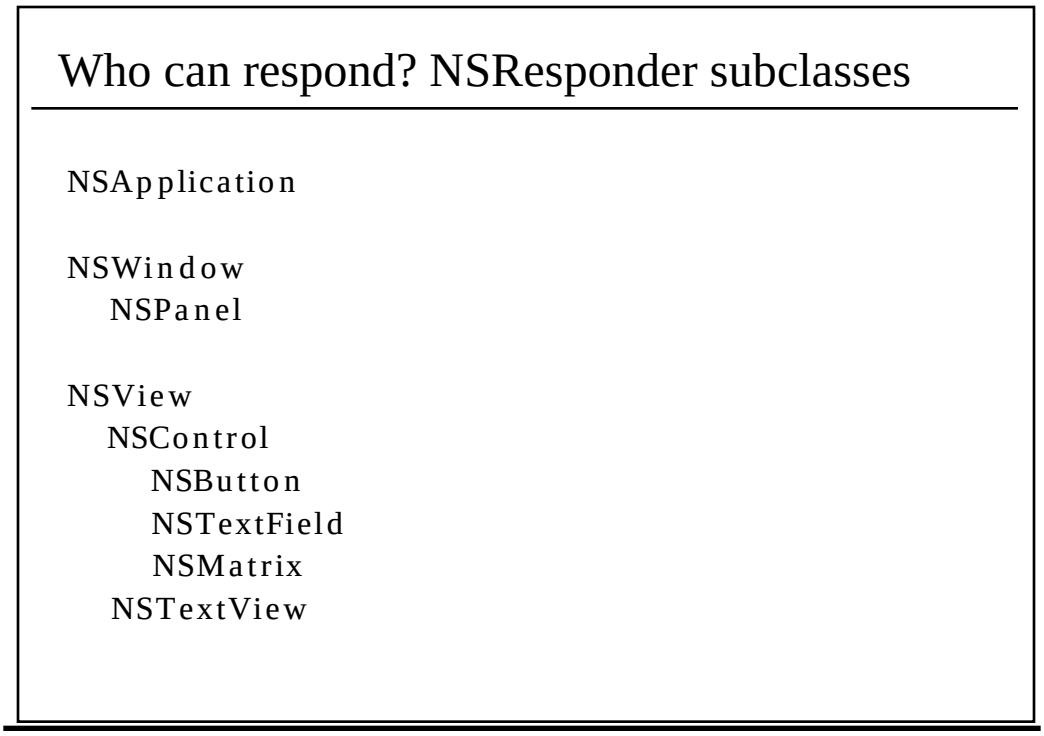
User actions generate event messages

Graphical interfaces are driven by user events—mouse clicks and key strokes. A user event is a request for action from the application. In an object-oriented world, actions generally mean sending a message to an object.

NSApplication is, in a sense, the master controller for your application. At the core of its responsibilities is the **event loop**. One by one, it picks an event from those queued by the underlying platform and decides which object is responsible for handling the event. Then it sends a message, passing an NSEvent object describing the particulars. The event message passes from NSApplication to the right window to a view, commonly a control, within the window and eventually to your target object.

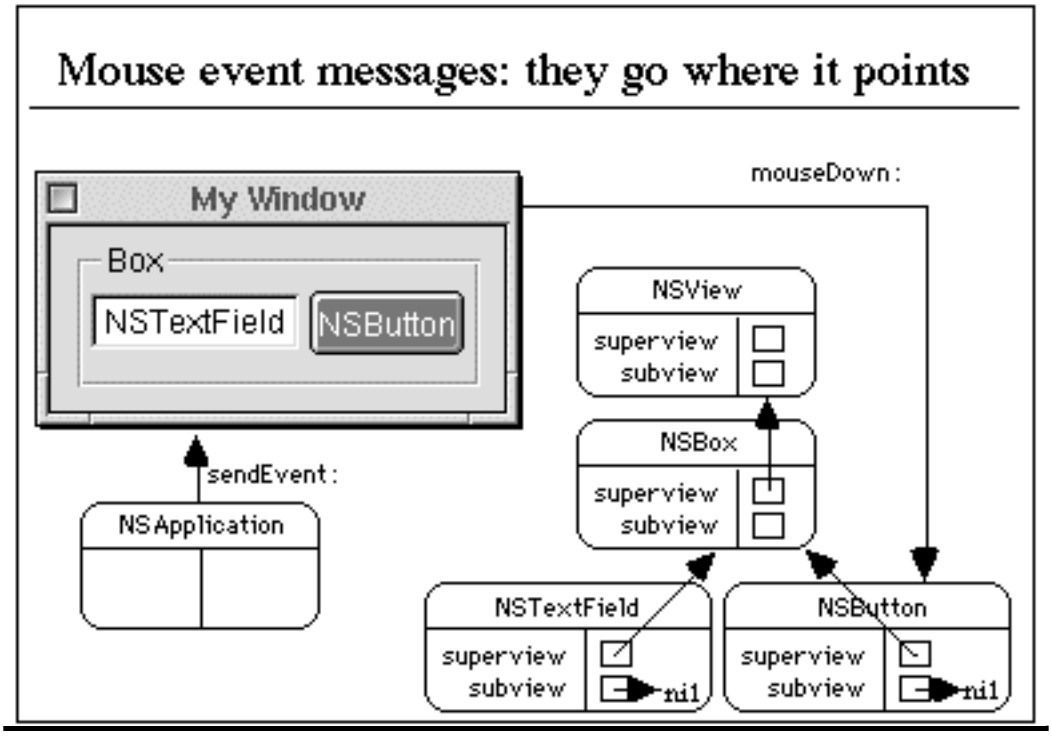
This is how a button knows that it has been pressed. This is how your custom objects gain control—through target/action, delegation or, in the case of custom views or controls, directly.

When your application objects are finished responding to the message, control unwinds and returns to NSApplication where it loops again, ready to process the next queued event. Many interesting things happen during the course of an event cycle. You will learn more details later on and will see how this basic clockwork drives the entire application.



Who can respond? NSResponder subclasses

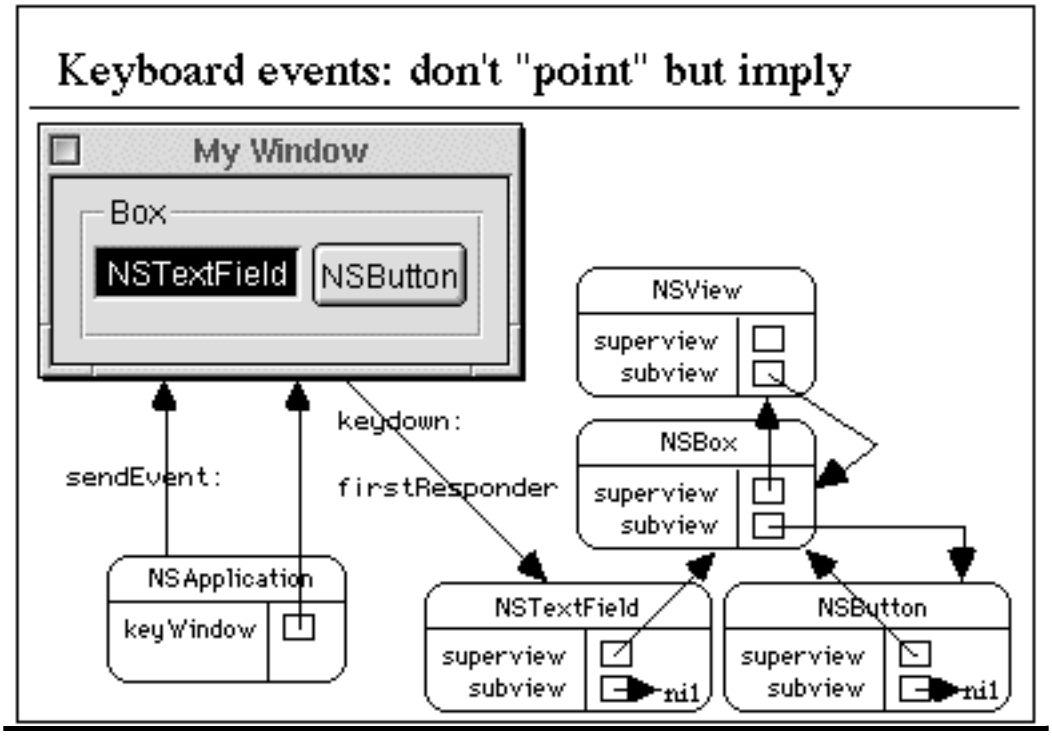
What kinds of objects are sent these event-based messages? Any subclass of NSResponder—NSApplication, NSWindow, NSView including NSControl. Each of these classes serve an important role in responding to specific events. Each class inherits the generic attributes and behaviors of NSResponder, extending and overriding them to achieve further specialized behavior. For your custom objects, NSView and its subclasses are the most important since NSApplication and NSWindow will rarely be subclassed by you.



Mouse event messages: they go where it points

Where does a mouse event message go? A mouse click points to a spot on the screen which can be precisely mapped to a window and a view within that window. Click on a button and that is the object that gets the message.

Through target/action, your controller object gets a message and your code is activated.



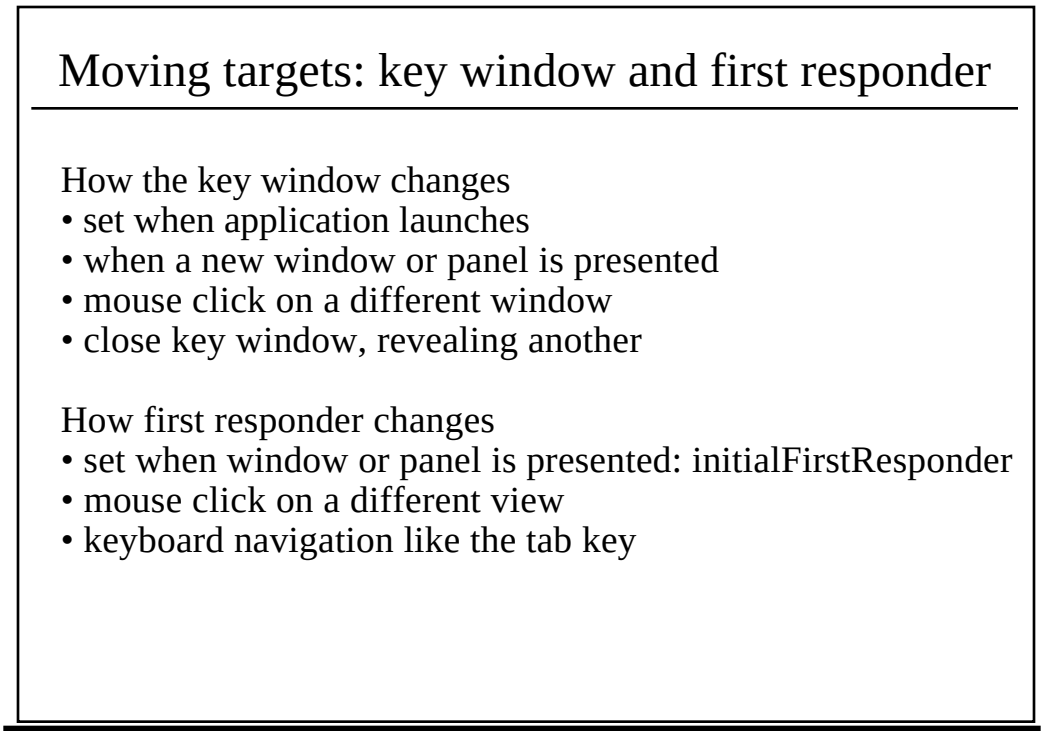
Keyboard events: don't "point" but imply

Key strokes from the user's keyboard do not point anywhere on the screen. It is necessary to designate a default object that gets the first crack at those event messages and it is called the **first responder**. Every NSWindow instance has an outlet to its first responder. At any point in time, only one of these windows is the active. The application-wide first responder is precisely the first responder on the **key window**.

The key window is the active window. You can think of the first responder as the user interface object that is currently "in focus". It is where the user's attention is. This object is often visually apparent—it shows some highlighting or outlining.

A text editor provides a clear illustration. You click on a text view and start typing. The text view is in focus, it has your primary attention. It is natural that any keystrokes should be passed to that text view object. The text view is the first responder.

The interesting part is that this is all very dynamic. As time passes, the key window changes. Within the window, as the user iteracts, the first responder changes as well. It is worth considering how and when this happens.



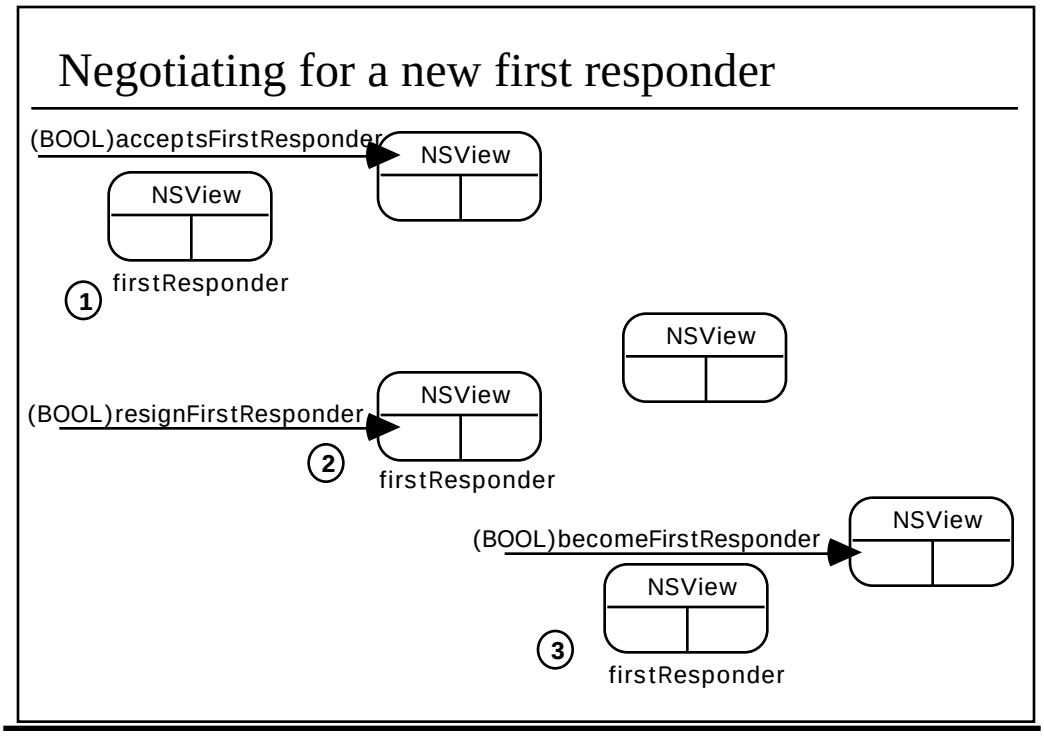
Moving target: key window and first responder

In multi-windowed desktop environments, there are many open windows on your screen. You use the mouse to precisely select a window. It's active and you're ready to type. The window has just become the key window and that window has a first responder.

Using your mouse or perhaps a navigation keystroke like Tab, you precisely select another control on the same window. You have now changed the first responder.

If no object has been selected, or if the window has no controls, the window is its own first responder. You can configure the **initialFirstResponder** so that, when a window appears, the first logical control capable of using keystrokes is brought into focus as the first responder. The default object selected by the window, when **initialFirstResponder** is nil, is typically quite reasonable.

A first responder is usually one that handles keystrokes. Not all user interface objects do so—a slider or a color well for example. Not all objects become the first responder when you point and click on them. Alternatively, once the first responder, an object may not be wiling to give it up. Consider a text field that performs validation. You type something invalid. It won't let you Tab away or press OK. The first responder may not be willing to give up its privileged spot to another responder object.



Negotiating for a new first responder

Whenever a user action attempts to change the first responder, a negotiation ensues to verify that this is acceptable. It is important to ask the objects that will be affected.

The new view (remember, NSControls are NSViews) is asked to see if it will accept first responder status. By default, buttons say YES. A disabled button will say NO. A button can be configured to always say NO. Any NSControl that is configured to be non-selectable will also answer NO. Changes to the first responder are affected by state changes even within a single object.

If at any time in the negotiation, an object returns NO, the dialog terminates and the old responder remains in control.

Assuming the new view will accept first responder status, ask the old first responder if it is willing to relinquish its position. Various kinds of text editing cells may likely enforce validation by refusing to give up first responder status unless the user's entry is acceptable. There might be other reasons.

Once the old responder is willing resign, the new responder is assigned first responder status and informed of state change with a message. Many controls need to do a little setup at this point. A text cell gets ready for editing, another control may want to change its appearance. Most controls indicate this status with some kind of highlighting or outlining.

Messaging first responder: target/action

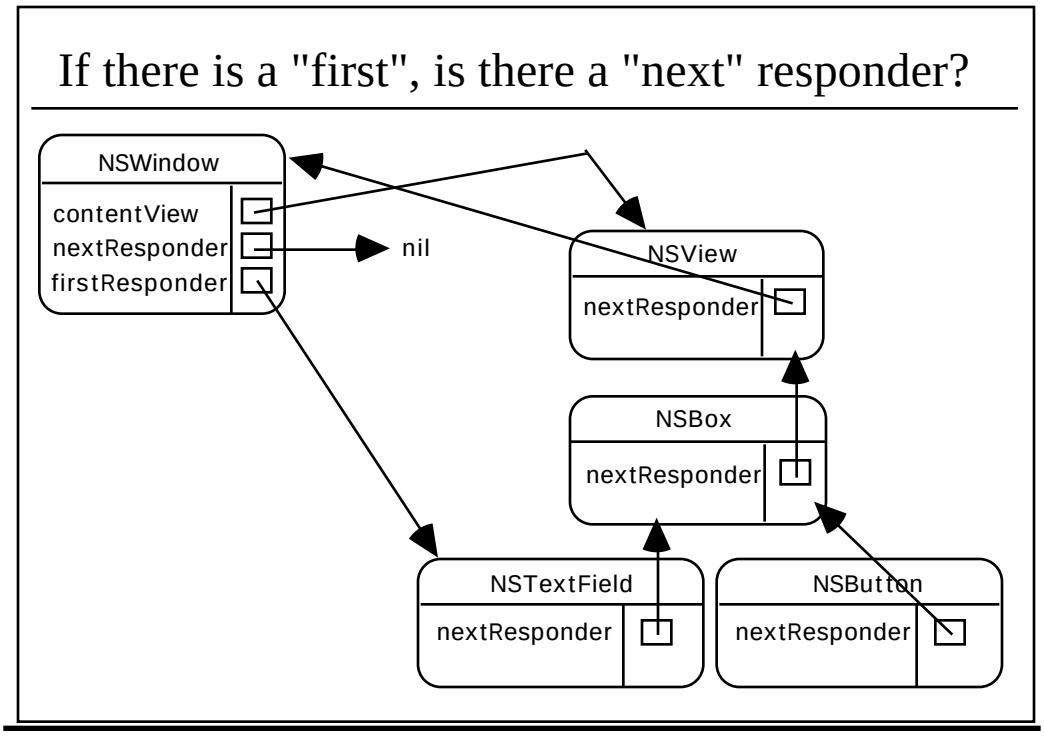
How about connecting a button to message first responder? While many target/action connections are hard-wired to specific object instances, it is very useful to be able to message this dynamically changing target. Consider the traditional editing menu selections cut, copy and paste. Which object actually gets the **cut:** message when you press the button? The message is sent to the first responder. Since many objects support the notion of "cutting", the menu item can store the **cut:** selector as its **action** and send it to the first responder, if it responds. Whoever knows how to **cut:** is able to respond to a user's mouse click on the menu. This is a practical example of the power of polymorphism.

But what is stored as the **target**? You can connect any NSControl to the first responder icon in Interface Builder and get the desired behavior. The value of the **target** outlet in the NSControl is quite simply nil. You can programmatically set the **target** outlet using

```
[control setTarget: nil]
```

and get the same result. Notice that this is different than simply sending a message to nil. Using logic built-in to NSControl and its subclasses, a control interprets a nil target to mean the current first responder. For this reason, this special situation is called a nil-targeted action.

Nil-targeted actions start by messaging your application's first responder.

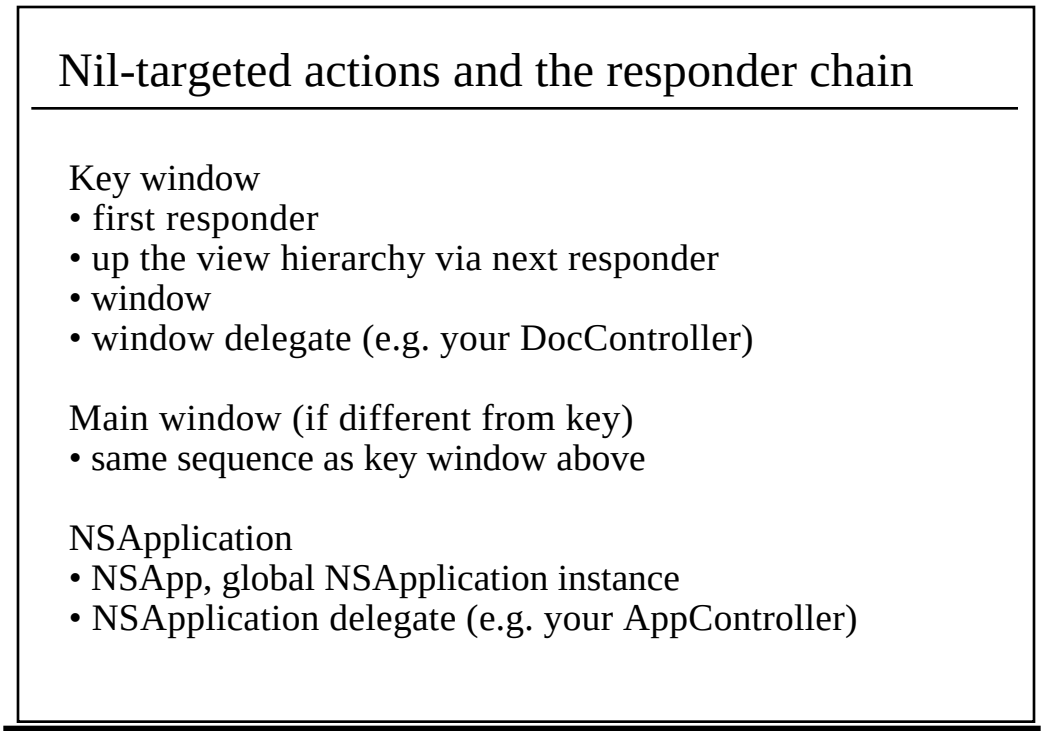


If there is a “first”, is there a “next” responder?

First responder, as its name implies, is actually just a starting point. All the views on a window are linked together into a hierarchy that leads from any one view upward until it stops at the window. This hierarchy should look familiar—it’s the same as the one that connects super and sub views. Recursive display messages flow down the hierarchy but the responder links flow upward. If one view can’t or doesn’t want to respond, the opportunity moves on to the **nextResponder** until someone is willing to take control. The buck stops there. The message initiates the desired action and no other objects in the chain of responders notices the message.

Key event messages will proceed up this chain until they reach the window. If unclaimed, the event is effectively discarded and processing resumes with the next event.

A nil-targeted action sent by an NSControl object also uses this chain. It searches up from first responder until it reaches the first object in the chain that responds to the selector. And if unclaimed, it does not stop at the window but continues on a more extensive journey.

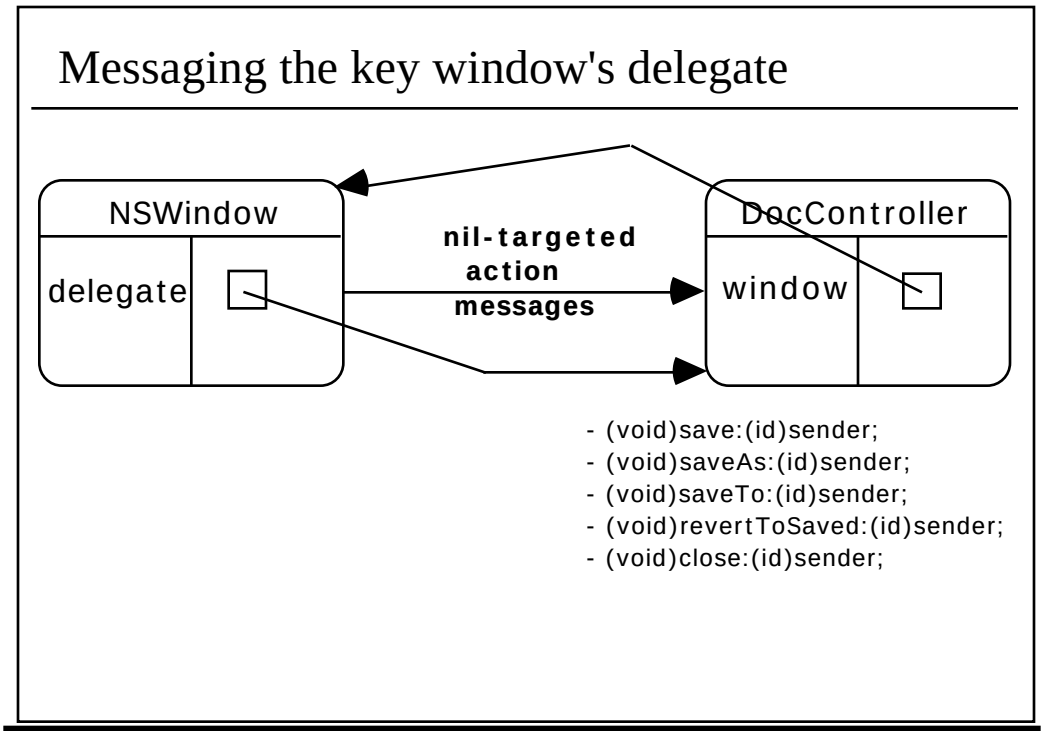


Nil-targeted actions and the responder chain

For nil-targeted actions, the responder chain extends beyond just the key window. Once it reaches the window it conveniently checks with its delegate. This is typically your controller and it might well respond to the message at hand. If not, the search moves on. If the main window is different than the key window, it gets the next shot, following a similar chain beginning with its first responder. Once again, your controller may enter the picture as a window delegate. Finally, your application's NSApplication instance is queried, followed by its delegate. Your application controller object is the last stop in the responder chain for nil-targeted action messages. If the message was not intended for your object, it goes unclaimed.

It is interesting to note that all the objects in the responder chain are subclasses of NSResponder except for your custom delegates, application and document ontrollers. They are typically subclasses of NSObject. In these two cases, your controllers can participate in responding, regardless of their class type.

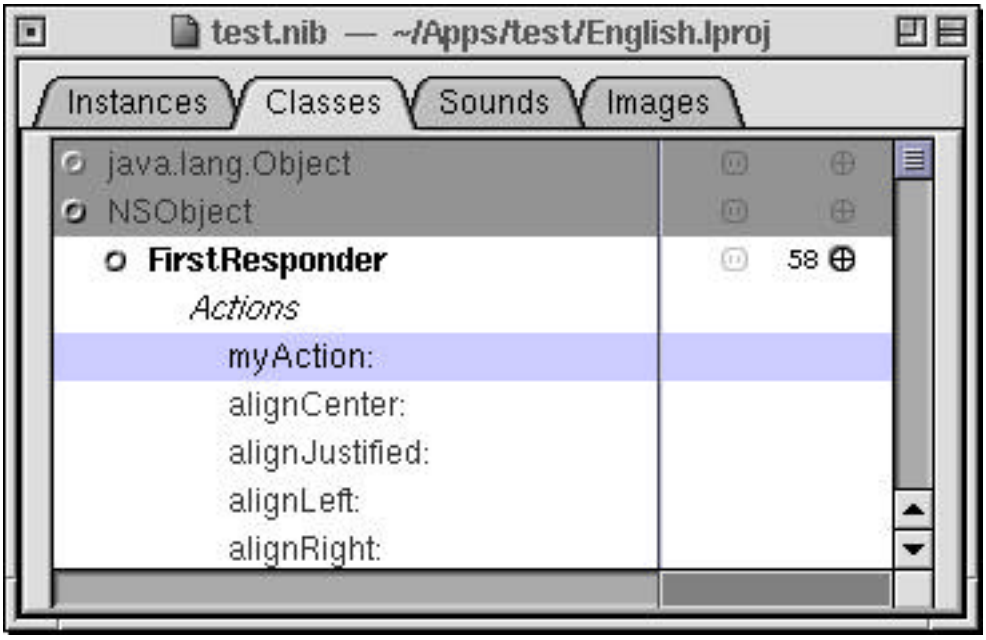
It is possible to have a menu but no open windows at all. In this case, the responder chain is quite short—first NSApplication, then its delegate object, typically your AppController.



Messaging the key window's delegate

Consider a "Save" menu item. Its action selector is **save:** and its target is nil. When pressed, a search for a responder begins with the key window's first responder. moving up the responder chain until it reaches the window. If no object responds so far, your delegate now gets involved. If your controller responds to **save:**, it gets the message and performs the job of saving the document.

A multi-window application has a dynamically changing key window and each window has a delegate—your document controller. You can design a main menu with items that apply to the relevent document controller whenever they are pressed.



Adding actions to first responder in Interface Builder

If you use this approach to message your delegate objects, or you add new methods to custom controls or views in the manner of cut:, you will want to graphically make first responder connections in Interface Builder. Your methods are likely to have unique names. Although first responder is not a real object or class, it is located in the Interface Builder classes browser for this reason. It has a list of default actions commonly used by Application Kit objects. You can add your own actions here and they will be available choices in the Interface Builder inspector when you connect NSControl subclasses to the first responder icon.

Nil-targeted action message caveats

The first object in the chain that respondsToSelector:wins

For custom action messages (e.g. DocController), use unique selectors

No object responds to misspelled selectors!

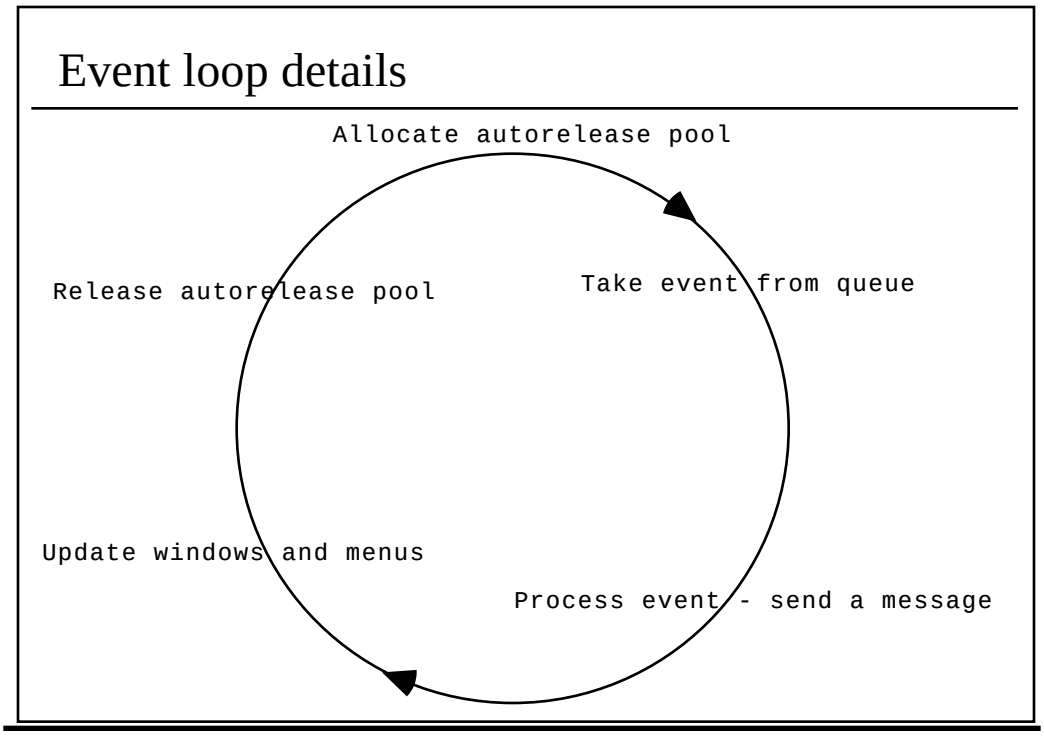
The responder chain is specific to NSControl and target/Action

vs.

simply messaging nil such as `[nil someSelector:sender];`

Nil-targeted action messages caveats

Nil-targeted actions are powerful and useful. These are a some points to bear in mind when using this technique. If your custom object it not getting the message you expect, you might check your application against these points.



Event loop details

The event loop is the heartbeat that drives graphical applications. Here is the sequence of steps carried out with each iteration of the event loop:

- Allocate as autorelease pool
- Take an event from queue
- Process the event—send a message
- Update the windows—including menus
- Release the autorelease pool

Important ideas from this section

» User events are transmitted to your application objects as messages. There are two types of user events:

    mouse

    keyboard

» Keyboard event messages are sent to the first responder on the key window.

» The key window and first responder change frequently. First responders negotiate before switching—some views can't become first responder and some are not always in a state to give it up. Many views need preparation to become first responder.

» An NSControl with a nil target outlet is said to have a nil-targeted action. The actual target will be dynamically located using the responder chain when the NSControl is activated

» Nil-targeted actions start with first responder and move through the responder chain until an object that responds is found. The responder chain includes views, the key and main windows, their delegates, NSApp and its delegate. The delegates are your custom objects that can participate in servicing nil-targeted actions.

» The event loop is the heartbeat of your application. It is useful to understand each step in one iteration of the event loop, especially in terms of autorelease pools and event updating.

Classes featured in this section

» NSResponder

» NSView

» NSWindow

REVIEW EVENTS AND RESPONDERS

1. What's the use of the key window?

2. Explain the role of the first responder.

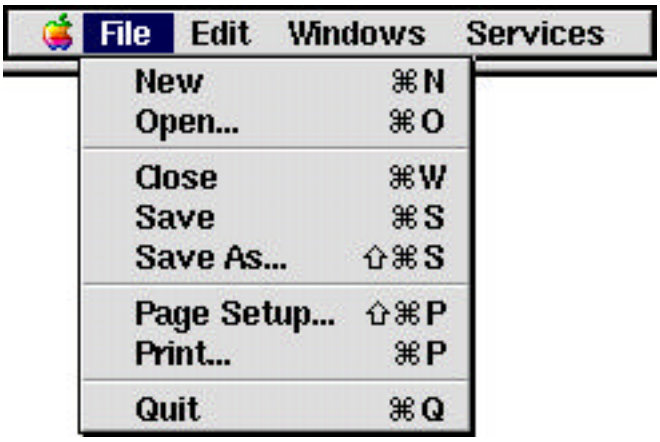
3. Why would you use a nil-targeted action?

4. Name the places in the responder chain where your custom objects might participate.

5. Is this an example of a nil-targted action: `[nil doSomething: self]` ?

EXERCISE 13.1 EXPENSE REPORT—ADDING THOSE MISSING MENU ITEMS

Returning to the expense report application you built earlier, you may recall that certain menu actions were left disabled. In this exercise, you will extend the Expense Report application by enabling these important menu items and implementing the corresponding methods. You use first responder and the responder chain to get the message to the right object. As a window delegate, your DocController is in the responder chain. Messages sent to first responder—nil-targeted action messages—can target the DocController for the key or main window.



Objectives

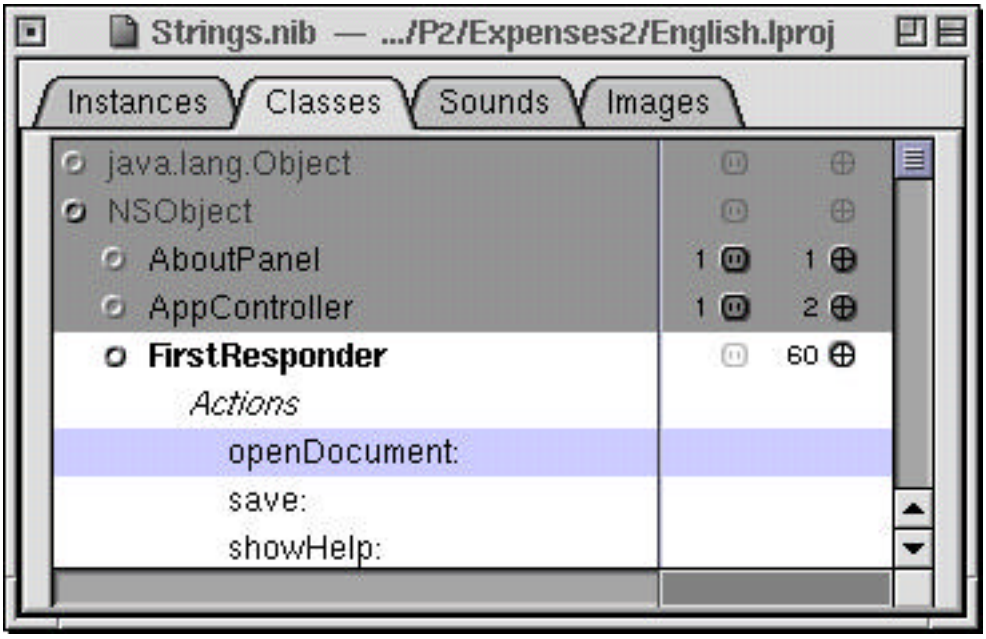
After completing this exercise, you'll be able to:

- Use the First Responder to process menu actions
- Exploit the Responder Chain to simplify interface design

Exercise—Stage 1

1. As before, make a backup copy of the previous version of your application.

2. Open the Main nib file, and select the First Responder. Switch to the class browser view and add **save:** and **openDocument:** to the First Responder list of actions.



» Connect the relevant menu items in the Document or File submenu to the First Responder icon.

» Add the **openDocument:** method to the AppController class. Have it simply log a message that it was called using NSLog(). Do the same for the **save:** method in the DocController class.

Note: When the user clicks on the Open command in the menu, the top document in the application will be asked if it understands **openDocument:**. Since DocController has no **openDocument:** method, the answer is no. The menu cell will then try the next candidate in the responder chain, which in this case is NSApplication, and then the application delegate. AppController does have an **openDocument:** method, so it will be called.

3. Build the project and try it out. Are your new methods being called? Which of the methods you connected isn't being called? Do you understand why not?

Stage 2

1. The DocController will implement the reading and writing of files. It will need to know about the table view data source, in order to set and archive the information it holds—its array of expense items.

» Add a **dataSource** outlet to **DocController.h**

» Open the document nib and read **DocController.h** back into Interface Builder to update the outlet list

» Connect the **dataSource** outlet to the ExpenseDataSource in the nib

2. Now you can flesh out DocController's **save:** method:

» Save the document to a file. Most applications save whether or not the document has actually been modified. Obtain the array from the data source, with the **array** accessor method, and archive it using document's filename. If the document is new and has never been saved, you will need to use an NSSavePanel to obtain the filename. NSArray and the Expense object can archive and unarchive themselves. To archive the array object, use the following method:

```
[NSArchiver archiveRootObject:array toFile:filename];
```

» Once a document has been saved, the window's **representedFilename** attribute is set. If the document being saved is not new, you can save it using the **representedFilename** attribute as provided by the window (see below). Remember, after saving, to update the document's modification status.

Hint: To maintain the correspondence between a document and a filename, you will want to use NSWindow's **representedFilename** and **setTitleWithRepresentedFilename**: methods. The **representedFilename** and the title need not be the same. By default, a window's **representedFilename** is @"". New unsaved document windows can have a title but will have an empty string for their filenames. You can use this as a flag to determine if the document is new:

```
if ([[window representedFilename] isEqual: @""])
```

3. Add an **initFromFile:** method to DocController so that you can create a new array directly from an archive file and pass it to the data source. A document initialized from a file is not new. It has a valid filename. You may want to modify the **init** method to invoke **initFromFile:,** with a nil filename, so that **initFromFile:** becomes the designated initializer. The code in the data source's **init** method that generates a default set of data may now be deleted so that a new document is indeed empty. To unarchive an array, use the following method:

```
array = [NSUnarchiver unarchiveObjectWithFile:file];
```

4. Now you can extend the AppController class with its two methods:

» **openDocument:** This should use NSOpenPanel to obtain a filename and then create a new document using DocController's **initFromFile:** method. In **ExerciseMaterials**, you will find a test expense report file named **sample.expense**.

» **saveAll:** You should enumerate over the array of documents, or the window list to obtain the delegate, and send each one a save: message.

Enhancements

» If you haven't already done so, add the name of the relevant document to the alert panel displayed when a modified document is being closed, so that the user can tell which document it is referring to. This is primarily useful when the user quits the application with lots of windows open. Typically, the corresponding window should also become key and ordered to the front of the screen.

» Currently the user can open the same document twice. See if you can enforce some sort of document uniquing policy.

» Unarchiving from a file can fail so that **unarchiveObjectWithFile:** returns nil. This might be due to the simple fact that the file does not exist or is unreadable by the current user. Test for this common error and report the problem to the use—using an alert panel. See NSFileManager in the Foundation framework for a way to check for the existence and readability of files. What problems might you check for in the archiving case?