CommunicatingWithOtherObjects:OutletsandActions;¬Communicating With Other Objects: Outlets and Actions

Outlets

An outlet is an instance variable that points to another object. Objects use outlets to communicate with other objects; they simply send messages to the object identified by the outlet.

Using Interface Builder, you can declare and set outlets for the custom objects in your application. You can also set ready-made outlets in many Application Kit objects, such as browsers. Once initialized, the connection information for the outlet is stored in the nib file. At run time, the nib file is unarchived and the outlet is reinitialized with the connection information.

The Application Kit defines two types of outlets that you can use to establish specialized connections with other objects: delegates and targets.

Delegates

A delegate is an object that acts on behalf of another object. Many Application Kit classes define delegate outlets as an alternative to subclassing. All your object must do is register itself as a delegate of the Application Kit object. At important junctures in its life cycle, the Application Kit object sends messages to its delegate, giving it an opportunity to participate in processing and sometimes the chance to veto behavior.

For example, browsers ask their delegates to supply the cells for browser columns, and the application informs its delegate when it is initialized, hidden, and activated.

Targets

Targets are a special kind of outlet. They identify objects that can respond to action messages. When a user activates an NSControl object (for instance, clicking a button or moving a slider), that object sends an action message to the target. The action message gives application-specific meaning to the original mouse or key event.

For example, you could connect a custom object in your application as the target of a button so that when the button is clicked, your object performs a method that fills all of the text fields in the window with appropriate information.

Like a delegate, a target must implement methods to respond to the messages it's sent. But unlike a delegate, which receives messages chosen from a limited set defined by another object, a target responds to any action message you choose to define.

You can also make one object a target of a second object programmatically by sending **setTarget:** to the second object.

Outlets
OutletConnection.eps ¬

Actions

When a user manipulates an NSControl object, the object receives an event message, which it translates into a message that is meaningful within the application. It then send this message to another object. These application-specific messages initiated by an NSControl object are called *action messages*, and the methods they invoke are called *action methods*.

NSControl, an abstract class, defines for its many subclasses (such as NSButton, NSScroller, NSTextField, and NSForm) a paradigm for inter-object communicationÐaction messages. But NSControl objects don't act alone: they always contain one or more NSActionCells (or one of its subclasses). The NSActionCell superclass defines instance variables for the two elements essential to an action message:

- **target** the object that's responsible for responding to the user's action
- **action** the method that specifies what the target is to do

Action methods take a single argument, the **id** of the NSControl object that sends the message. This argument enables the receiver to ask the control for more information, if it's needed.

An NSControl can send a different action message to a different target for each NSActionCell it contains. Different

NSControls dispatch action messages differently; for instance, an NSButton generally sends action messages on a mouse-up event, but an NSSlider usually sends action messages continuously, as long as the mouse button is pressed.

Actions
ActionConnection.eps ¬

101306_TableRule.eps ¬

TheModesoftheInstancesDisplay;¬The Modes of the Instances Display

When you open a nib file in Interface Builder, the Instances display of the nib file window first shows objects as icons. This icon mode doesn't show all objects, just the *top-level objects*Ðthose objects that are not contained by another object. Windows and panels and most controller objects (that is, objects that manage an application or a window) are top-level objects; although they may contain other objects (for instance, a window contains one or more views), no other object contains them.

The graphical representation of objects in icon mode makes it an ideal interface for many operations. Its simple, intuitive, and uncluttered nature makes it easy to do the basic things, such as making connections between top-level and interface objects.

For more complex operations, the Instances display has another modeÐoutline modeÐthat shows more detail about objects in the nib file, including their connections with each other.

Icon Mode

_InstancesDisplayMode1.eps ¬

The most important advantage of the outline mode is that it shows *all* objects in the nib file, not merely the top-level objects. It also shows all connections, both connections into an object and connections from an object to other objects.

The outline mode starts by listing the top-level objects in the nib file. By clicking the open button next to an instance,

you can see what other objects it contains. Click a connection button (triangle button) to see what connections go into or out of an object.

You can connect objects in outline mode; there's no need to drag a connection line to the interface. Outline mode also has facilities that make it easy to identify objects in the interface and to disconnect objects.

Objects in outline mode are identified first by class name and then, in parentheses, by title. If the title is obscured, you can resize the nib file window until it is visible.

Outline Mode
_InstancesDisplayMode2.eps ¬

Expanding Objects in Outline Mode

In outline mode, objects that contain other objects have a small circle button to their left that is filled with gray. The subordinate objects are usually subviews of a window, panel, or another view object, but can be objects that are part of another object not visible on the screen. You display these contained objects by expanding the container object.

Click a circle button to expand an object into a list of its component objects; click it again to collapse the list. Expansions can be nested many levels. To expand everything within an object, Command-click the circle button. Collapse the list back to the original level by Command-clicking the circle button again.

See ªThe View Hierarchyº in this chapter for a description of the relationship between superview and subview.
;ConnectionsConcepts.rtfd;TheViewHierarchy;¬

_ExpandObjectsOutMode.eps ¬

101306_TableRule.eps ¬

StandardObjectsintheInstancesDisplay:File'sOwner,FirstResponder,andFontManager;¬Standard Objects in the Instances Display:
File's Owner, First Responder, and Font Manager

FilesOwnerObject.tiff ¬ **File's Owner**

Every nib file has one owner, represented by the File's Owner icon. The owner is an object, external to the nib file, that channels messages between the objects unarchived from the nib file and the other objects in your application.

Not only must the owning object be external to its nib file, it must exist before the nib file is unarchived. This is because the same method that loads a nib file (**loadNibNamed:owner:** and its variants) also specifies the file's owner.

The typical owner of an auxiliary nib file (such as one containing an Info panel) is an instance of the class you assign to File's Owner in Interface Builder. This class is almost always a custom class, and is frequently the class of the object that manages your application. Once you make the assignment, File's Owner serves as a proxy instance of your class, which you can then connect to the interface. (By the way, the typical owner of an application's main nib file is NSApp, the global NSApplication object.)

See Chapter€11, ªDynamic Loading,º for more on the role of File's Owner in the loading of auxiliary nib files and for details on assigning classes to File's Owner. ;../../05_SpecialTasks/11_DynamicLoading/DynamicLoading.rtfd;;¬

File's Owner
FilesOwner.eps ¬

FirstResponder8bit.tiff ¬

First Responder

The First Responder is the object within a window that first receives keyboard events, mouse-moved events, and action messages from NSControl objects that don't have an explicit target (for example, cut and paste). The First Responder object is the active window's focus for future events. Although technically an object, First Responder is really a status conferred on an object.

Usually, when you click an object that accepts key events (such as a text field), that object becomes the window's First Responder. First Responder status also changes when you make another window key in your application. (Because of this, First Responder can be useful when you build multiple-document applications.) Over time, many different objects can become the First Responder, but at any one time only one object has this status. The First Responder icon stands for the object that currently has this status, no matter which actual object it is within your application.

The First Responder figures into the event-handling behavior defined by the NSResponder class. In a window, objects inheriting from NSResponder (including NSView, NSApplication, and NSWindow) are part of a linked list of event-handling objects called a *responder chain*. The responder chain contains (in this general order) a view, the view's superview, the view's window, the main window, and then the application. (The application and window delegates are in this chain as well, although they aren't NSResponders.) If the First Responder can't respond to an event message, its next responder is given a chance to respond. If an NSResponder can't handle the message, the message continues to be passed up the chain from object to object in search of an NSResponder that can. Messages are passed in one direction only: up the view hierarchy toward the window and application.

In Interface Builder you can connect an NSControl object in the interface to the First Responder icon. Thereafter, when the user manipulates this NSControl (say, by clicking a menu item entitled Copy) an action message (**copy:**) is sent to the object that is currently First Responder. If you examine in Interface Builder the default connections from the Edit menu, you'll discover that its menu cells are all connected to First Responder.

First Responder
_FirstResponder.eps ¬

fontmanager8bit.tiff ¬

Font Manager

The Font Manager icon represents an instance of the NSFontManager class that is shared among the objects of an application. Interface Builder automatically creates and adds this object to your project when you drag the Font menu into your application's menu. The Font Manager is the center of activity for font conversion. It accepts messages from font conversion user-interface objects (such as the Font menu or the Font panel) and appropriately converts the current font in the selection by sending a **changeFont:** message up the responder chain. See the documentation on the NSFontManager class for more information.
;/NextLibrary/Frameworks/AppKit.framework/Resources/English.lproj/Documentation/Reference/Classes/NSFontManager.rtfd;;¬

101306_TableRule.eps ¬

WhenYouDon'tWanttoDisconnect;¬When You Don't Want to Disconnect

After all of the objects in your interface are connected the way you want them, you may want to make sure that they stay that way. When you delete an object from the interface, all of the connections to that object are broken. If all you're doing is fine-tuning the interface's appearance, you want to make sure this doesn't happen.

To prevent someone from accidentally changing connections, set the Lock all connections preference on the General preferences panel display. (Choose Preferences from the Info menu to bring up the Preferences panel.) When this preference is set, you can't connect objects, disconnect objects, or delete objects that have connections.

When you're localizing an application, it's a good time to use this connection locking feature. When you localize a nib file, you want the interface objects to behave the same way, but you want their titles to change. Sometimes, it's necessary to move and resize the interface objects to make room for titles in other languages that tend to have longer words. By locking connections, you make sure that you don't make a change to the interface that will change the way the application behaves.

101306_TableRule.eps ¬

TheViewHierarchy;¬The View Hierarchy

When you expand an NSWindow object in outline mode and then expand the NSView objects indented beneath, you are looking at a *view hierarchy*. All the NSView objects within a window are linked together in this hierarchy, an abstract tree structure similar to the class inheritance hierarchy.

Within every window's content rectangleÐthe area enclosed by the border, title bar, and resize barÐ is its *content view*. The content view is at the top of the view hierarchy. All other views of the window descend from it. Each view has one other view as its *superview* and can be the superview for any number of *subviews*.

ViewHierarchy.eps ¬

What physically determines a view's place in the hierarchy is *enclosure*. A superview encloses its subviews. NSView stores pointers to three objects that reflect a view's physical relationships to other views in the window and locate the view in the hierarchy:

- **window** identifies the view's window (the window points to the content view)

- **superview** identifies the view's superview
- **subviews** a list of the view's subviews

ContentView.eps ¬

The defining relationship of enclosure makes it easier to draw a view:

- It allows you to construct a view object (the superview) from its subviews.
- Views are positioned within the coordinates of their superviews, so when a view is moved or its coordinate system is transformed, all its subviews are moved and transformed with it.
- Each view has its own coordinate system for drawing. Since a view draws within its own coordinate system, its drawing instructions can remain constant no matter where it or its superview moves on the screen.

Two other attributes, the frame and bounds rectangles, set the location, dimensions, and coordinate systems of a view. **frame** holds the position and size of a view within its superview's coordinate system. The **frame** rectangle defines the area in which drawing can occur. The origin point of a frame locates the lower-left corner of the rectangle in the superview's coordinates. The **bounds** rectangle occupies the same area as the frame rectangle, but it is stated in a different coordinate system; the frame's origin becomes the origin (0.0, 0.0) of the view's drawing coordinates (**bounds.origin**). The bounds rectangle is thus expressed in the view's own drawing coordinates.

Another attribute, inherited from the NSResponder class, determines how events are handled within the view hierarchy. The **nextResponder** by default identifies a view's superview. If a view receives an event message (for example, **mouseDown:**) and cannot handle it, that message is passed on to the view identified by **nextResponder**. See the specifications of the Application Kit's NSView ;/NextLibrary/Frameworks/AppKit.framework/Resources/English.lproj/Documentation/Reference/Classes/NSView.rtfd;;¬ and NSResponder ;/NextLibrary/Frameworks/AppKit.framework/Resources/English.lproj/Documentation/Reference/Classes/NSResponder.rtfd;;¬ classes in the *Application Kit Reference* for more information on the view hierarchy and event handling.