Chapter 14

USING PANELS

CHAPTER 14 USING PANELS

Goal

To cover several key distinctions between windows and panels, and to explore the specific cases of updating for inspectors and designing custom modal panels.

Prerequisites

Experience using NSWindow with multiple nib files and an understanding of first responder and the event loop.

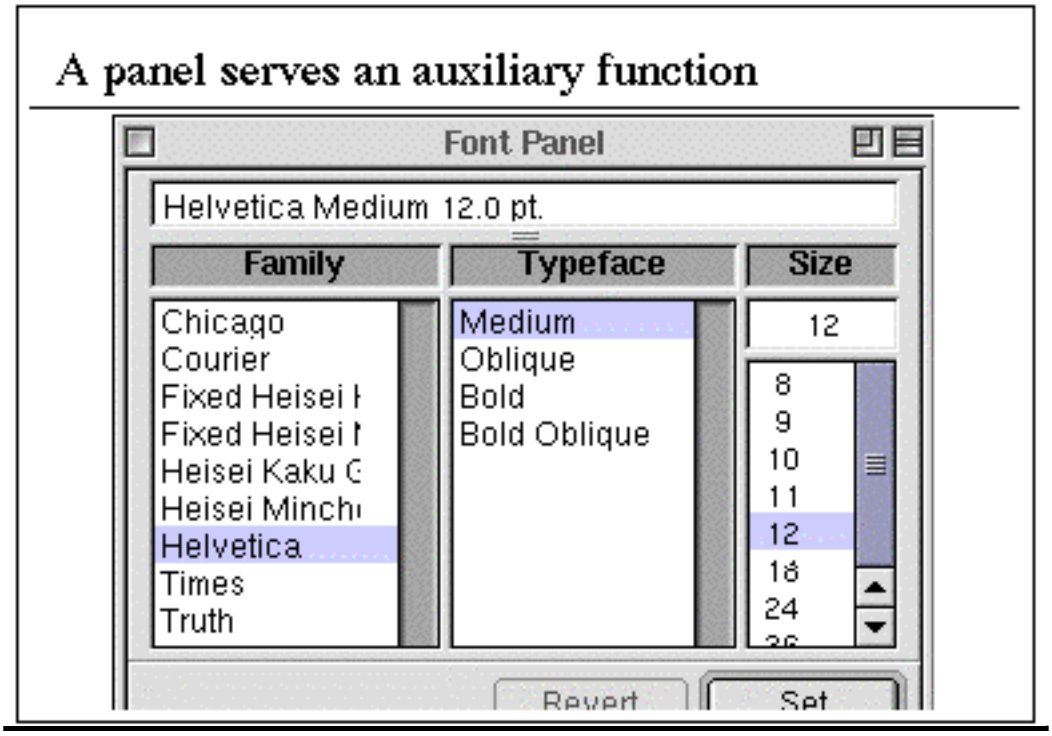
Objectives

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

- List key differences between NSWindow and NSPanel including configurable attributes usually used with one or the other
- Implement a dynamically updating inspector panel
- Design a custom modal panel

Reading

- **NSPanel class in the Application Kit**
- **NSWindow class in the Application Kit**

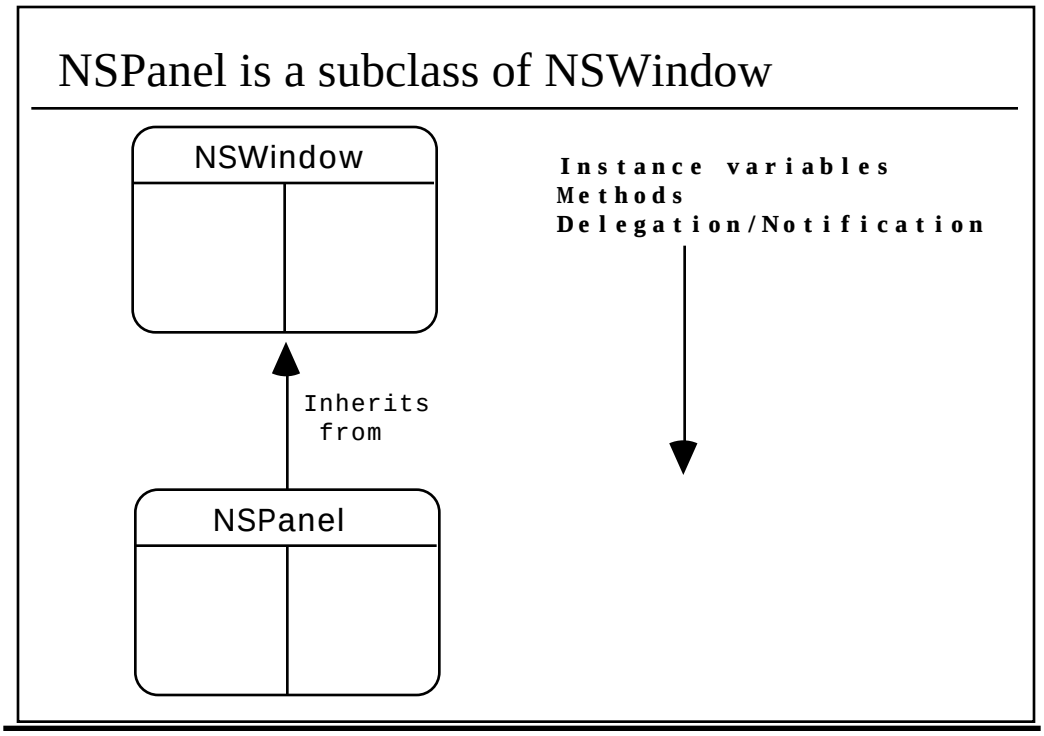


A panel serves an auxiliary function

Your graphical application focuses primarily on a window where all the important action is. It might be a document or a monitor screen or some set of controls for driving a particular business process. But it is likely that your application provides additional details, configuration options, customizable preferences, helper tools, even just pop up warnings, questions or status indicators.

These are usually implemented with NSPanel, a subclass of NSWindow. Often called secondary windows or dialogs, these serve an auxiliary function to the primary window or the application as a whole. They operate on a primary window such as changing the font or color of a selected object. They may provide additional detail, again depending on the primary window or something selected within. A panel might change the behavior of all windows or some mode in the application itself.

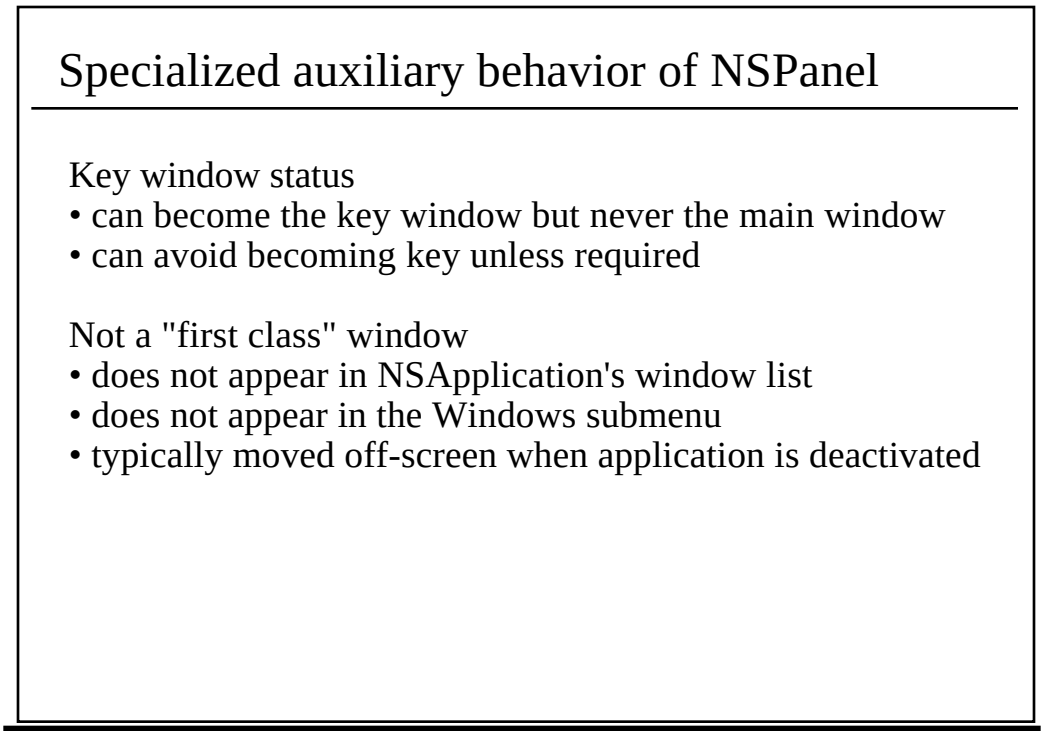
Because of this auxiliary role, panels have unique properties. A panel is a specialized window with configurable attributes for a variety of purposes.



NSPanel is a subclass of NSWindow

NSPanel inherits much of its identity from NSWindow. It has a frame with window decorations and controls. A panel has a content view which is the top of a view and responder hierarchy. A panel can have a delegate. It is important to remember that a large number of methods, attributes and behaviors that apply to NSPanel will be found in the documentation and understanding of NSWindow. You already know a great deal about panels.

What makes a panel different from a window?

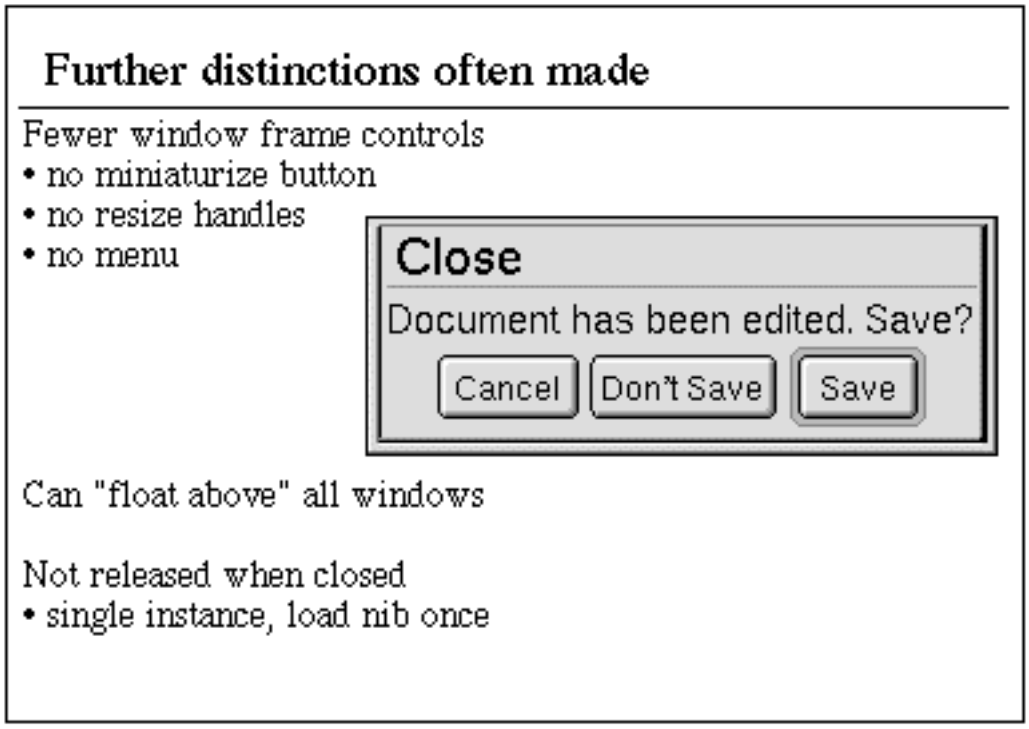


Specialized auxiliary behavior of NSPanel

While a panel can become the key window, the place where keystrokes go, it cannot become the main window. A panel is auxiliary and never the main focus. A panel, even when it is the key window, often needs to communicate with the main window to get its job done. Once a panel becomes the key window, the user must click back on the main window to continue typing there. You can configure your panel so that its controls can be used without taking key status from the main window as a convenience. It is useful that you can display your panel with **orderFront:** instead of **makeKeyAndOrderFront:**.

Panels are not added to NSApp's **windows** list. You will not get confused about which are your document windows and which are your inspectors, font, and print panels. Panels are not listed in the Windows submenu for the same reason.

Because panels usually provide extra detail or are common between applications, such as the save panel, you can configure a panel to disappear off-screen when the application is deactivated. When the application is re-activated, the panel appears again, maintaining its previous state—its position and its appearence.



Further distinctions often made

Panels often have a less flexible user interface. They may be fixed in size, unable to minimize or maximize. They may not have a close control other than responding to the dialog itself—pressing “OK” or “Cancel”. Most often, a panel does not have a menu and will not default to using your application’s main menu like windows do. Because a panel applies to the main window which is usually one of many, your panel can float above all of them to avoid being obscured.

A very important distinction is whether the panel should be released after the user closes it. A document window should be. If the user opens that document again, your application will allocate a new controller which loads its nib file which instantiates a new window for the occasion. Closing your inspector or preferences panel is not the same. You load the nib once since it is a single instance component. When the user closes and re-opens it, your controller merely brings the panel back on-screen with **orderFront:**. If you do set the Panel to be released on close, you would have to set your outlet to nil and reload the nib each time. For complex and seldom used panels, this may make your application more efficient.

This attribute is configurable using the Interface Builder inspector or programmatically with **setReleasedWhenClosed:**.

Standard application kit panels

Alert

Save

Open

Print

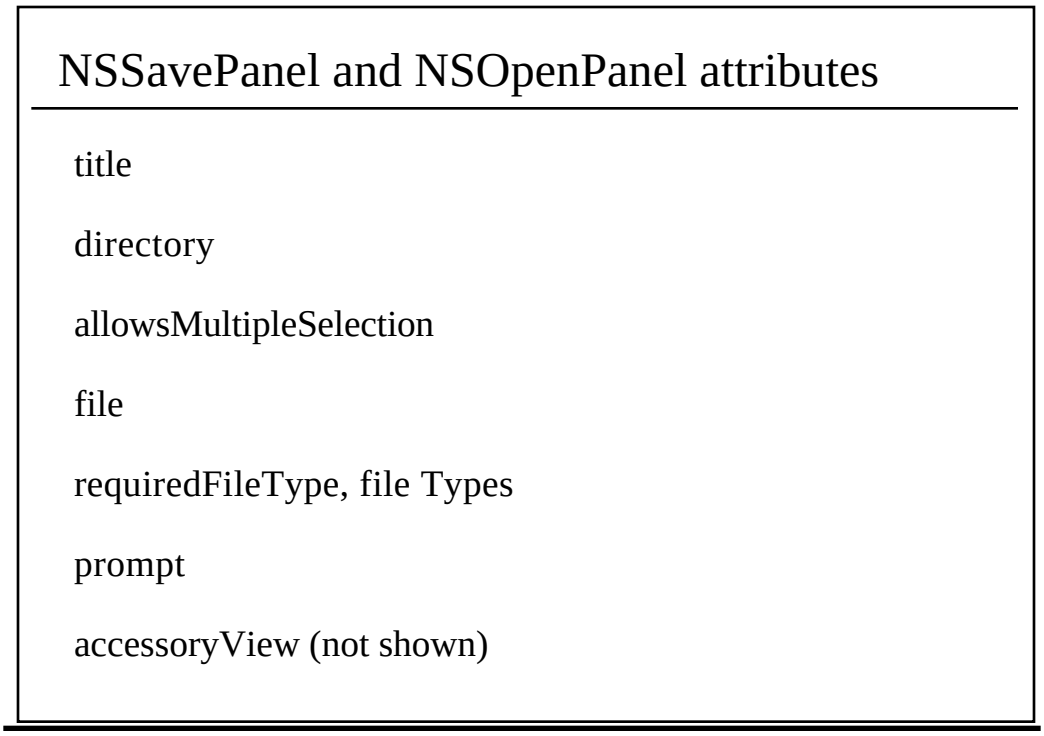
Page Layout

Font

Color

Standard Application Kit panels

You should be aware of the standard, built-in panels provided. You certainly don’t need to implement them yourself and they reveal some of the features you can build into your application in an effortless and consistent way. Many of these are pre-connected to corresponding menu items available on the Interface Builder palette.



NSSavePanel and NSOpenPanel attributes

Although these are provided for you as Application Kit Framework objects, the Save and Open panels are quite configurable. You can change the title, restrict the available types, customize the prompt and even attach a custom view for additional controls specific to your application. A practical example would be a set of radio buttons selecting one of many possible file formats for saving.

NSSavePanel and NSOpenPanel methods

Getting the panel

```
- (id)savePanel;
- (id)openPanel;
```

"Running" the panel

```
- (int)runModal;
- (int)runModalForDirectory:(NSString *)d file:(NSString *)f;
- (id)runModalForTypes:(NSArray *)types;
```

Return Values -- NSOKButton or NSCancelButton

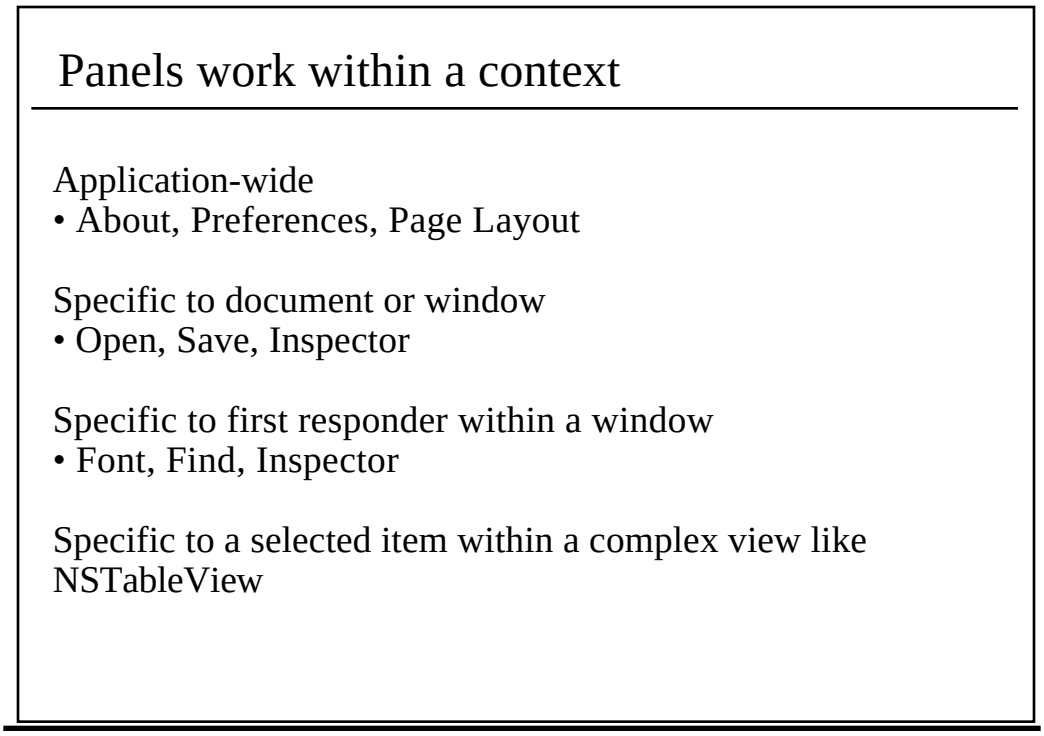
Getting the User's Selection

```
- (NSString *)filename;
- (NSArray *)filenames;
- (NSString *)directory;
```

NSSavePanel and NSOpenPanel methods

There is one instance of the save panel shared throughout your application. It is instantiated once and returned every time you ask for it. You do not have to release it.

To display the panel, you run it as a modal panel. This means that, unlike a font panel or your other windows, the save panel takes control of the application prohibiting the user from interacting with any other window until it is dismissed. There are a number of different “run” methods depending on where in the file system the panel should first come up and whether your application has specific file types.

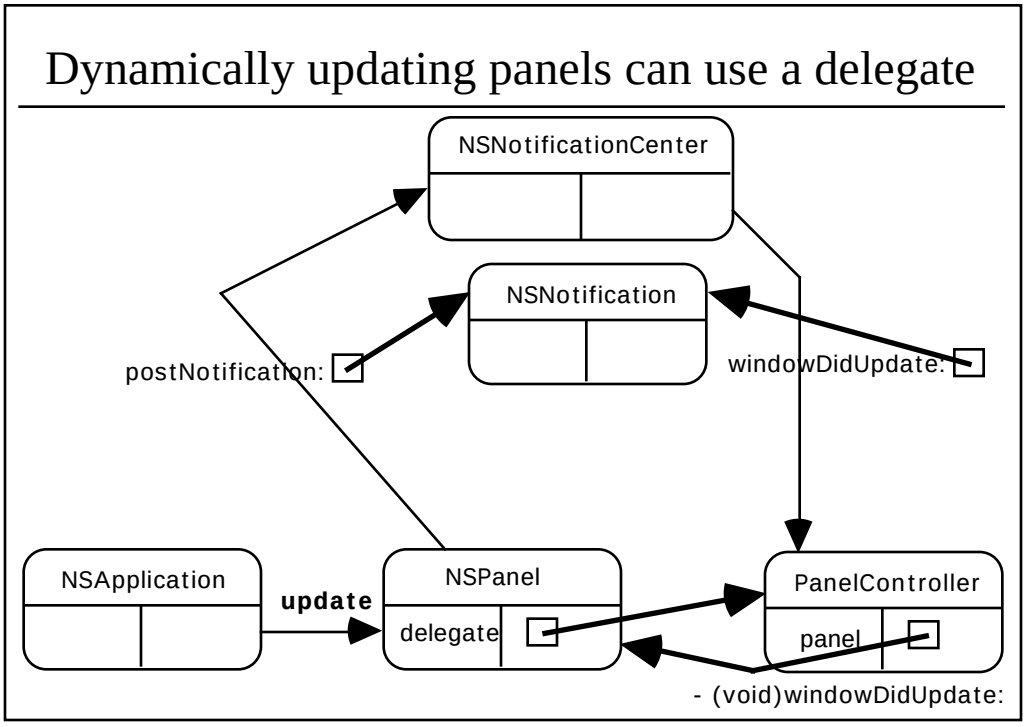


Panels work within a context

As you think about designing your panel, it is useful to consider the different contexts within which a panel might work. It highlights a few interesting design concerns:

- Does the panel apply to your application as a whole or to a specific document—the main window for example?
- Does it apply to the window as a whole or the first responder or even one of many possible selections in the first responder such as a row in a table view?
- Will you need to update your panel to follow a moving target—the main window, the first responder, or the selection in a table view?

Like menu items, many panels need to be updated as frequently as after each event.



Dynamically updating panels can use a delegate

There are essentially three options for implementing updating panels:

- Message the panel controller directly when a state change requires it
- Post a notification and let any number of concerned objects all register to find out when a state change requires it
- Let the panel update itself after every event, regardless of whether there was a relevant state change or not

The third option is similar to auto-enabling menu items and requires the following implementation details:

- Your panel controller is the panel's delegate
- Your panel controller implements **windowDidUpdate:**

With this setup, your panel controller will receive an message after every event. Its job is to track its item of interest and update the panel to reflect any relevant changes.

Which choice you make depends on several factors: how often the panel's state must change and how much processing it takes to do so. You should avoid expensive processing that will always occur on every iteration of the event loop. If the panel does not need to change, the controller should avoid redundant processing.

Tracking other objects of interest

Accessing the Main window and/or Delegate

```
[[NSApp mainWindow]delegate];
```

Accessing firstResponder

```
[mainWindow firstResponder];
```

Searching the responder chain

```
[NSApp targetForAction:selector];
```

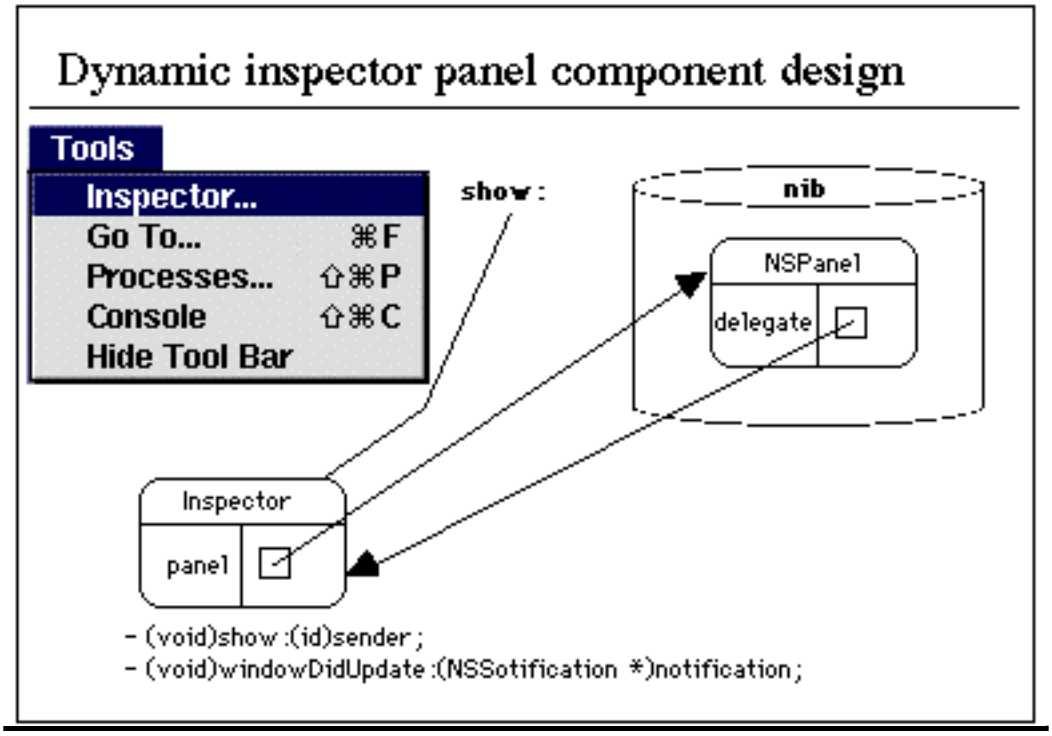
Basic Queries

```
[anObject isKindOfClass:class];
[anObject respondsToSelector:selector];
[anObject conformsToProtocol:selector];
```

Tracking other objects of interest

Your panel will likely need to message other objects in its context, dynamically tracking them when they are moving targets. Here are some common approaches. You can easily get to the main window and its delegate which is typically the one of many instances of your own document controller. Once on the main window, you can get to the first responder or if you are interested in following the full responder chain for nil-targeted actions rather than just keyboard events, you can message NSApp find the target. The methods for locating selected characters or cells with controls, like text fields and table views, are specific to each object; consult the documentation.

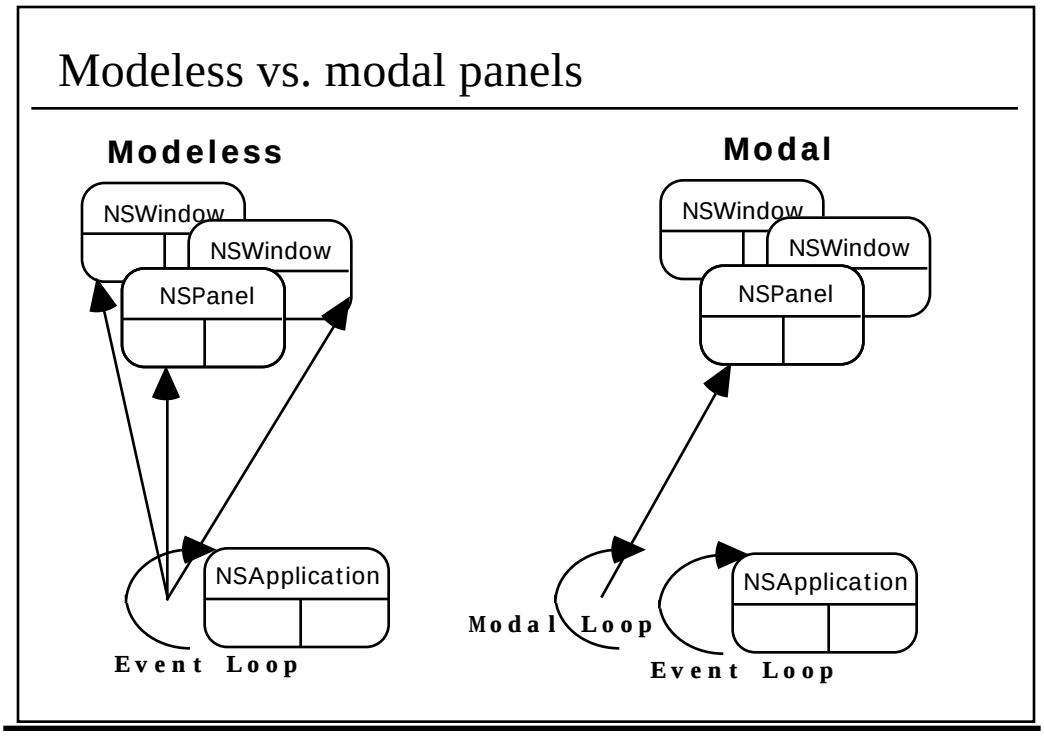
Once you have the object you are interested in, you might need to query it to determine its class type or message interface in terms of a selector or a protocol. A find panel might use these techniques to determine which first responders it can actually search and ignore those that are incapable.



Dynamic inspector panel component design

The basic picture is familiar—a **show:** message to a controller which is File's Owner for a nib containing its user interface. The panel controller is connected as the panel's delegate and responds to **windowDidUpdate:**. The controller can update the state of the panel after every event. Since this reflects the intention of tracking a dynamic target, it fits a category of panels generally called "inspectors".

Additional Points To Ponder



Modeless vs. modal panels

Panels fall into two categories—modeless and modal. Like windows, on-screen modeless panels receive events intended for them and sit idle and unobtrusive while the user interacts with other windows and panels. Each one is independently available and unconcerned with events targeted at any of the others.

Modal panels put the application into a specific mode. They become the primary and exclusive focus of the application, seizing control and prohibiting events from reaching any other on-screen window or panel. They must be dealt with before the user is allowed to move on and freely interact with other on-screen objects. Modal behavior is required when the application has to carry out a dialog with the user— to obtain confirmation, to post a warning before proceeding, or to retrieve a piece of information such as a file name.

Examples of modal panels: NSAlertPanel, NSOpenPanel, Print panel, a login or authentication panel. Examples of modeless panels: Font panel, inspectors, preferences panels, version information.

NSApplication modal panel support

Methods

```
- (int)runModalForWindow:(NSWindow *)theWindow;
- (void)stopModalWithCode:(int)returnCode;
- (void)stopModal;
- (void)abortModal;
```

Return Value Constants
NSRunStoppedResponse
NSRunAbortedResponse
NSOKButton
NSCancelButton

NSApplication modal panel support

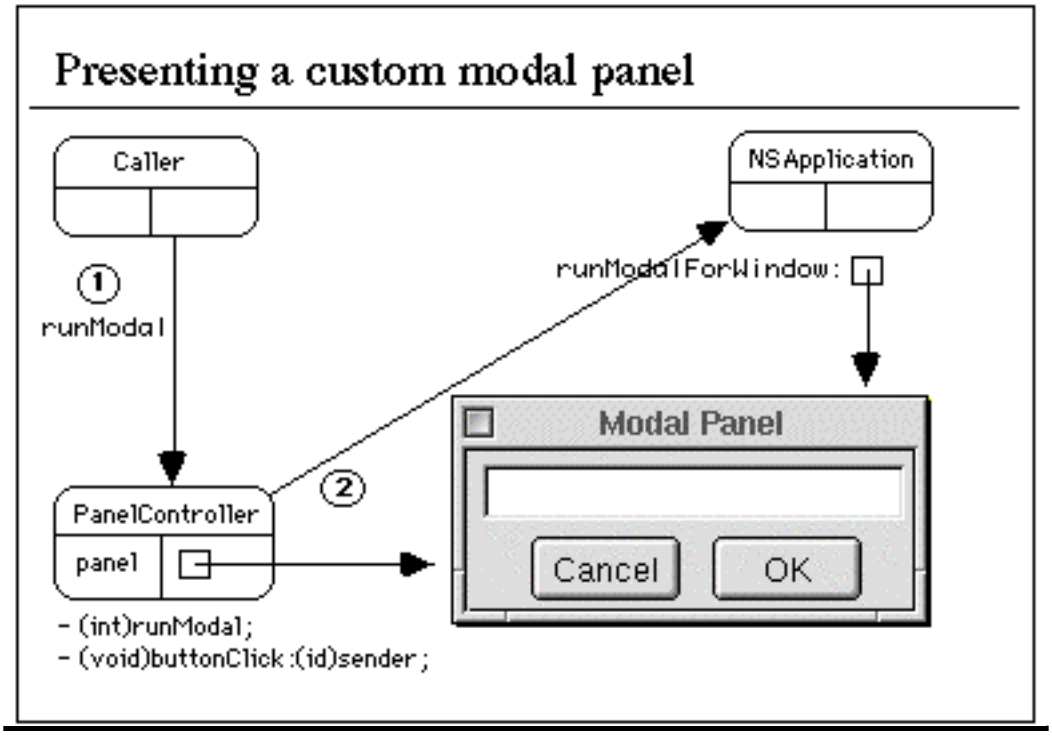
Modal panel support requires special handling of the event loop. NSApplication must restrict its fetching and processing of events to those belonging to the current modal panel, ignoring all others until the modal session is terminated. NSApplication provides an API for starting and stopping a modal session on a designated window. In addition, the Application Kit defines constants to be used to report the modal session termination code
or condition.

Methods:

- **runModalForWindow:**—runs the window or panel in a modal session
- **stopModalWithCode:**—stops the modal session returning code
- **stopModal**—stops the modal session returning NSRunStoppedResponse
- **abortModal**—aborts the modal session returning NSRunAbortedResponse

Standard button codes:

- NSOKButton
- NSCancelButton

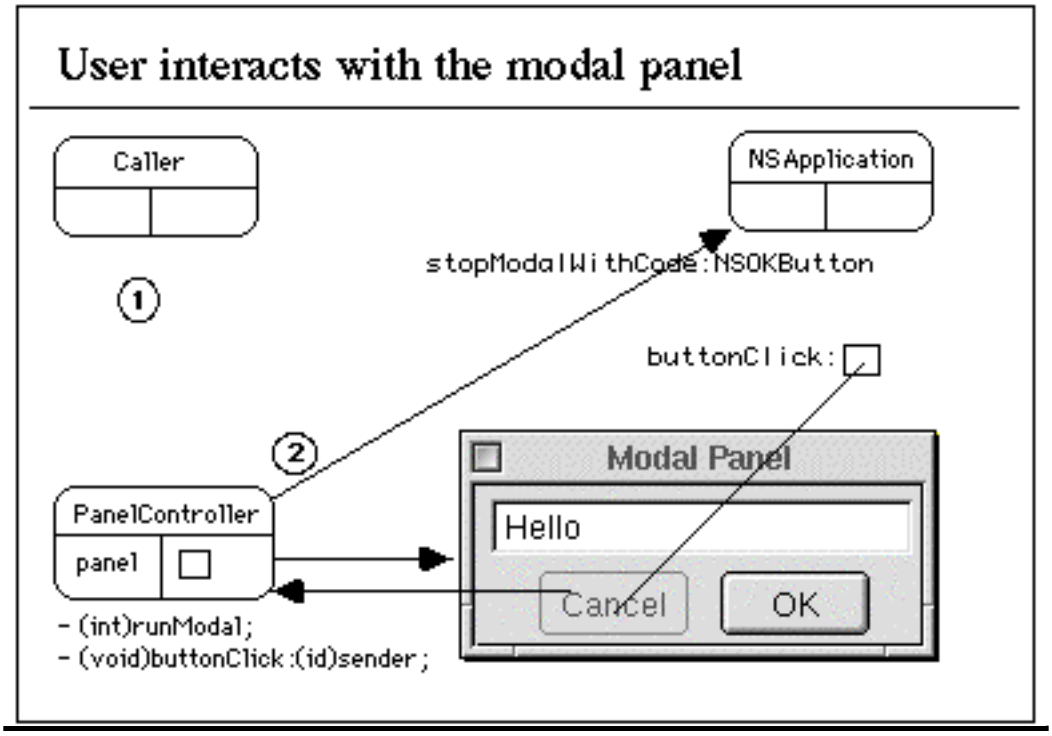


Presenting a custom modal panel

The sequence of events is more complex for a modal session because it does not follow a simple event flow. Rather, the complete modal session spans multiple event loop iterations when the design is pleasantly encapsulated:

1. A caller object messages the modal panel controller to run a modal loop.

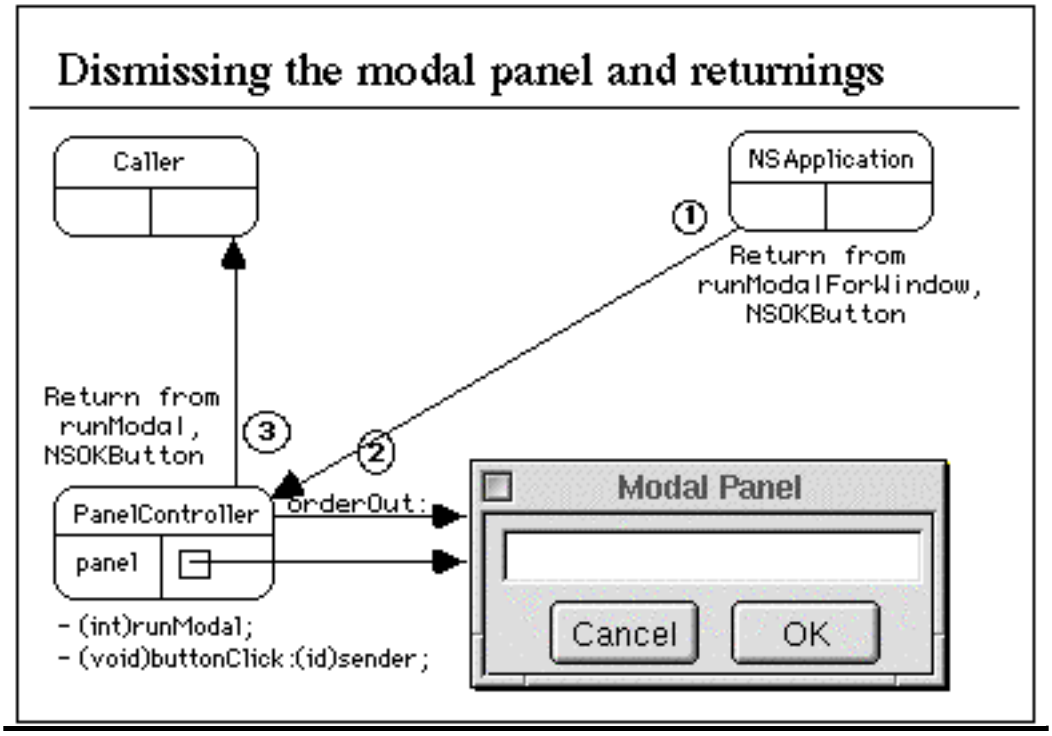
2. The modal panel controller, after loading its nib, messages NSApplication to start the modal session. The initial caller is now blocked inside **runModal** which in turn is blocked inside **runModalForWindow:** until user actions terminnate the modal loop.



User interacts with the modal panel

1. The user now interacts with the modal panel, typing some input and eventually pressing one of the panel buttons, in this case, OK

2. Target/action from the button messages the modal panel controller which messages NSApplication to terminate the modal loop and return the standard button token NSOKButton.



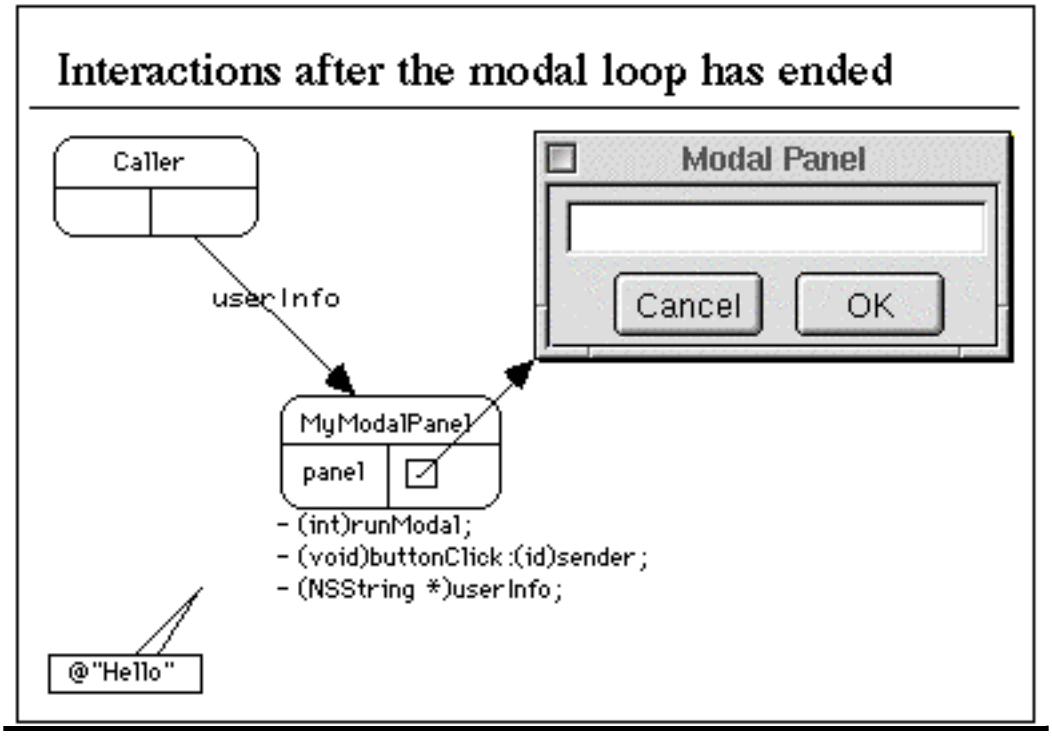
Dismissing the modal panel and returning

1. **runModalForWindow:** returns back from its original call and returns the button token to the caller, the modal panel controller.

2. Now that the modal session has technically finished, the panel itself should be removed from the screen with the **orderOut:** message.

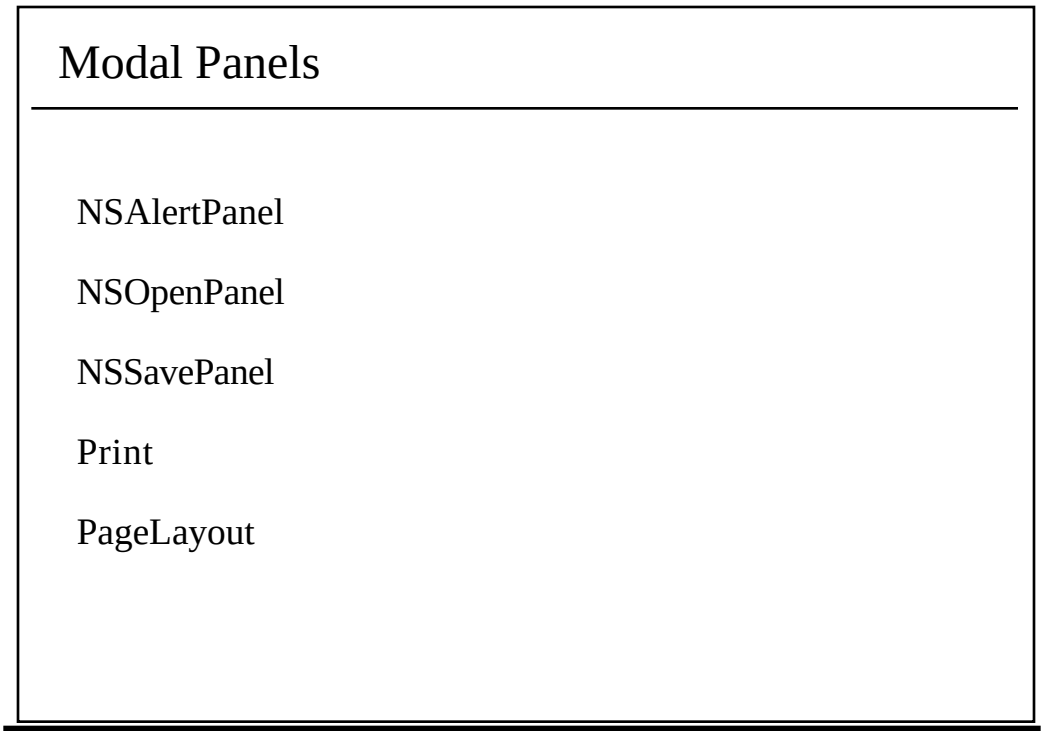
3. Control continues in the original **runModal** method which finally returns the NSOKButton token back to the caller object. The modal session is complete.

All is done. The caller discovers that the user pressed OK. What about the user's data input in the text field?



Interactions after the modal loop has ended

A typical design has the caller message the panel again, asking for pertinent data from the modal session. A hypothetical **userInfo** message fetches the value from the panel's control and returns it. This assumes that the panel was merely ordered off screen, not released. In the latter case, the panel controller might cache the desired values before releasing the panel. A more generalized form of the **userInfo** method might return an array or dictionary in the event that the panel presented a multi-control dialog.



Modal panels

There are a number of modal panels for standard Application Kit functions. NSAlertPanel in particular is very general and can be customized for a variety of basic purposes—warnings and errors, confirmations, and other simple choices with up to three options.

Important ideas from this section

» A panel serves an auxiliary function in the application or for a window.

» It is a subclass of NSWindow that has specialized behavior supporting this auxiliary role.

» The Application Kit provides a number of standard panels, some highly configurable to a variety of customized uses.

» Panels often track a moving or changing target and need to be updated:

    Directly

    Via notification(s)

    Via window notification/delegation after every event

» Modal panels take control of the application until they are dismissed. NSApplication provides the support for running modal panels.

Classes featured in this section

» NSPanel

» NSWindow

» NSOpenPanel, NSSavePanel

» NSApplication

REVIEW USING PANELS

1. Name at least three ways in which NSPanels typically differ from NSWindows.

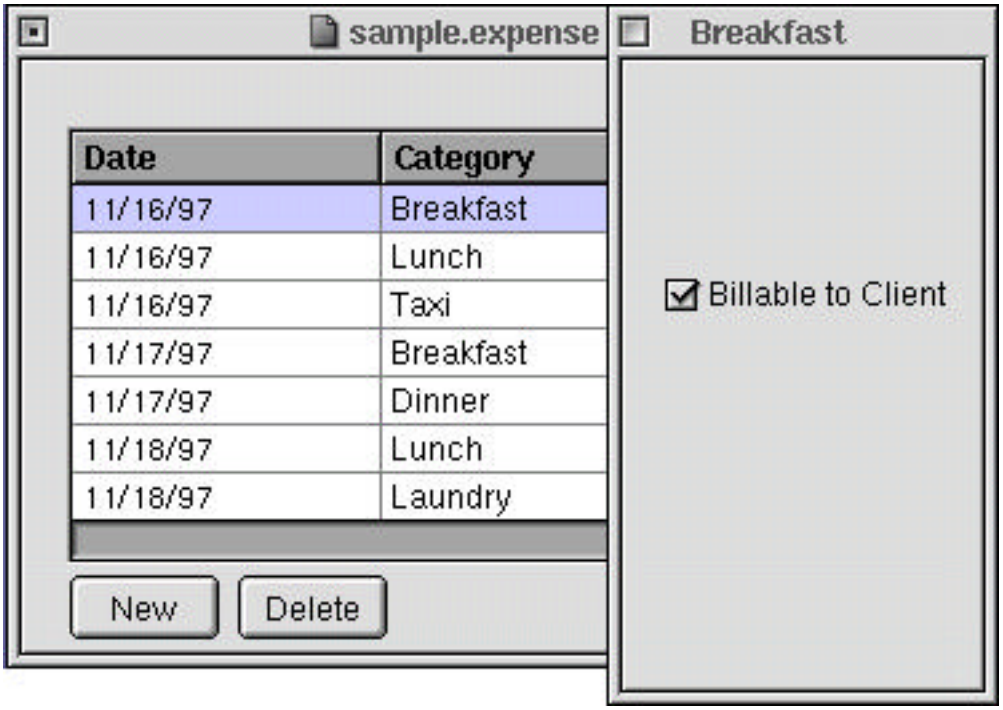
2. What is the difference between a modal and modeless panel?

3. What is the difference between the main window and the key window? How does this distinction relate to panels?

4. The essence of an inspector-type panel is that it stays in sync with the object it is inspecting. There are many different ways to achieve this in terms of design and features. Describe two.

EXERCISE 14.1 UPDATING INSPECTOR PANEL

In an application involving the presentation of a wide range of data, it often makes sense to present only the key information to the user, to avoid cluttering up the user interface. The application can provide a mechanism for viewing the detailed information for the particular item the user has selected. A common paradigm is an Inspector panel that tracks whatever the user has selected. It can be dismissed if not needed. Inspectors are most suitable when not all users need to see the information all the time or where the data is difficult to present in the context of the rest of the information. In this exercise, you add an updating inspector panel to the Expense Report application. It displays an additional boolean attribute of the Expense object—**isBillable**.



Objectives

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

- Understand the distinction between Panels and Windows and when to use one or the other
- Create Inspector-style panels

Exercise—Stage 1

1. Return to your expenses application. As usual, make a backup if you wish to keep a copy of the previous version.

2. The inspector component will be part of the Documents subproject, as it is specific to this kind of document. A template for the Inspector class—the panel controller—can be found under **ExerciseMaterials**.

    - Drag a copy of **Inspector.m** from **ExerciseMaterials** into the classes suitcase of the **Documents** subproject
    - Using Project Builder's file attributes inspector, make **Inspector.h** a Project header file

3. Use Interface Builder to create a new, empty nib module.

    - Drag and drop a panel from the palette into the interface
    - Ensure the Inspector panel has a close box
    - Add a switch to the panel and label it "Billable to Client"
    - Save the nib as **Inspector.nib** in the Interfaces suitcase of the Documents subproject

4. The Inspector component needs a controller object. It will be the File's Owner for this nib.

    - Read **Inspector.h** into Interface Builder
    - Make the File's Owner class Inspector
    - Connect the File's Owner's **panel** outlet to the NSPanel instance
    - Connect File's Owner as the panel's delegate
    - Connect File's Owner to the switch on the panel using the **billableSwitch** outlet
    - Connect the switch toFile's Owner with **billableChanged:** as the action

5. Like all File's Owner objects, in order to display the panel, some methods must be implemented—in this case, in the Inspector class. The automatic updating is handled in Stage 2 of the exercise—here you simply ensure that the inspector panel can be activated from the menu. Here is a rough sketch of what the methods should do:

    - **init**—load the Inspector's nib file
    - **show:**—display the panel
    - **billableChanged:**—the action when the user changes the switch. For now, just log a message using NSLog()
    - **windowDidUpdate:**—the window notification sent to its delegate. For now, log a message to provide you with some feedback. This will be finished in Stage 2 of the exercise

6. Now, you update AppController to handle the inspector component:

   - Import **Inspector.h** into AppController. Add an **inspector** instance.
   - Add a **showInspector:** method. It shows the inspector panel. Since the inspector is not always used, it makes sense to create the inspector object lazily the first time it is needed, and then message it to show its panel.

7. Return to Interface Builder and in the Main nib, add a Tools submenu to the application menu.

   - Label the first item in the submenu **Inspector**
   - Connect it to AppController directly, with **showInspector:** as the action

8. Build the project and check that the application behaves as you would expect—it brings up the inspector panel upon request.

Stage 2

1. The inspector should reflect the state of the currently selected expense item in the table view of main window. In order to implement this, you need to locate the main window, then the window delegate, and then the delegate's data source. From this you can obtain the selected expense object—if one is actually selected. You need to add some additional accessor methods to the pre-existing classes:

   - Add a **dataSource** accessor method to DocController that returns its data source. With the data source, you can obtain the array of expense objects using the **array** accessor method.
   - Add and implement a **selectedObject** method in the data source that returns the Expense object—or nil—thereby encapsulating the table view handling and its mapping to the array of objects.

2. To complete the inspector, you must implement the following methods in the Inspector class:

   - **billableChanged:**—the user has requested a change to the currently selected object via the switch. Use the Expense class's **setIsBillable:** method to do this. Remember to notify the document or the window that it has been edited.
   - **windowDidUpdate:** should locate the currently selected object, if there is one, determine its billable status, and set the switch accordingly. You can set the panel title to be the category of the selected object. If there is no selected object you might set the title to "Nothing selected".
   - Remember to import **Expense.h** as you need to invoke some methods it declares.

   Hint: The **windowDidUpdate:** method is called after each event—anything could have changed each time it is called. To determine which row is selected you need to use the methods in DocController and data source mentioned above.

3. Re-build the project and check that your inspector tracks the selected item in the table view.

Enhancements

- What if nothing is selected or there is not a main window? Ideally, the inspector should disable the switch—re-enabling when appropriate—and reflect that there is nothing to inspect in the panel title.
- You might wish to use the inspector panel title to reflect the state of the inspector—whether something is selected in the table view and if so, which expense is it. Create a title using the expense attributes, the date and category for example, and updating it along with the other status of the inspector.
- Since the inspector is not always needed by a user, it makes sense to make it entirely dynamic. See if you can convert the inspector and its panel into a bundle-loaded subproject. Refer to the Enhancements provided in the About Panel exercise.