Application File Management

Cocoa > File Management

2006-11-07

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures and Listings

Introduction to Application File Management

This topic describes the Application Kit's facilities for representing file system objects (files and directories) and allowing users to interact with the file system.

Organization of This Document

This document has the following articles:

- Concepts
  - "File Wrapper" (page 11) describes the object that encapsulates a file, directory, or link.
  - "The Save and Open Panels" (page 9) describes the panels that you run to let users specify files (and sometimes directories) to save and open.
- Tasks
  - "Working With File Wrappers" (page 13) describes how to make and manage file wrappers.
  - "Working With Directory Wrappers" (page 15) describes how to make and manage directory wrappers.
  - "Using a Save Panel" (page 17) explains how to create and display a save panel.
  - "Using an Open Panel" (page 19) explains how to create and display an open panel.
  - "Getting the Current Selection" (page 21) describes how to find out what the currently selected file or directory is.
  - "Filtering Out Browser Items" (page 23) explains how to make specific items in the browser unselectable.
  - "Configuring a Choose Dialog" (page 25) describes how to configure an Open panel as a Choose dialog.
  - "Managing Accessory Views" (page 27) explains how to make, add, and manage accessory views.
- See Also

  Document-Based Applications Overview
  Low-Level File Management Programming Topics

The Save and Open Panels

`NSSavePanel` creates and manages a Save panel, and allows you to run the panel in a modal loop. The Save panel provides a simple way for a user to specify a file to use when saving a document or other data. It can restrict the user to files of a certain type, as specified by an extension. It also allows you to do several other things with the Save panel:

- Add an accessory view to the panel.
- Customize the user interface of the panel, including the Hide Extension check box and the New Folder button.
- Modify the behavior of the panel through messages exchanged with a delegate.

`NSOpenPanel` provides the Open panel for the Cocoa user interface. Applications use the Open panel as a convenient way to query the user for the name of a file to open. The Open panel can only be run modally.

Most of this class's behavior is defined by its superclass, `NSSavePanel`. `NSOpenPanel` adds to this behavior by:

- Letting you specify the types (by file-name extension or encoded HFS file type) of the items that will appear in the panel
- Letting the user select files, directories, or both
- Letting the user select multiple items at a time

File Wrappers

An NSFileWrapper holds a file's contents in dynamic memory. In this role it enables a document object to embed a file, treating it as a unit of data that can be displayed as an image (and possibly edited in place), saved to disk, or transmitted to another application. It can also store an icon for representing the file in a document or in a dragging operation.

Instances of this class are referred to as file wrapper objects, and when no confusion will result, merely as file wrappers. A file wrapper can be one of three specific types: a regular file wrapper, which holds the contents of a single actual file; a directory wrapper, which holds a directory and all of the files or directories within it; or a link wrapper, which simply represents a symbolic link in the file system (sometimes called a shortcut or alias).

Because the purpose of a file wrapper is to represent files in memory, it's very loosely coupled to any disk representation. A file wrapper doesn't record the path to the disk representation of its contents. This allows you to save the same file wrapper with different paths, but it also requires you to record those paths if you want to update the file wrapper from disk later.

Working With File Wrappers

You can create a file wrapper from data in memory using `initWithSerializedRepresentation:` or from data on disk using `initWithPath:`. Both create the appropriate type of file wrapper based on the nature of the serialized representation or of the file on disk. Three convenience methods each create a file wrapper of a specific type: `initRegularFileWithContents:`, `initDirectoryWithFileWrappers:`, and `initSymbolicLinkWithDestination:`. Because each initialization method creates file wrappers of different types or states, they're all designated initializers for this class—subclasses must meaningfully override them all as necessary.

Some NSFileWrapper methods apply only to a specific type, and an exception is raised if sent to a file wrapper of the wrong type. To determine the type of a file wrapper, use the `isRegularFile`, `isDirectory`, and `isSymbolicLink` methods.

A file wrapper stores file system information (such as modification time and access permissions), which it updates when reading from disk and uses when writing files to disk. The `fileAttributes` method returns this information in the format described in the NSFileManager class specification. You can also set the file attributes using the `setFileAttributes:` method.

NSFileWrapper allows you to set a preferred filename for save operations and records the last filename it was actually saved to; the `preferredFilename` and `filename` methods return these names. This feature is more important for directory wrappers, though, and so is discussed under "Working With Directory Wrappers" (page 15).

When saving a file wrapper to disk, you typically determine the directory you want to save it in, then append the preferred filename to that directory path and use `writeToFile:atomically:updateFilenames:`, which saves the file wrapper's contents and updates the file attributes. You can save a file wrapper under a different name if you wish, but this may result in the recorded filename differing from the preferred filename, depending on how you invoke the `writeToFile:atomically:updateFilenames:` method.

Besides saving its contents to disk, a file wrapper can re-read them from disk when necessary. The `needsToBeUpdatedFromPath:` method determines whether a disk representation may have changed, based on the file attributes stored the last time the file was read or written. If the file wrapper's modification time or access permissions are different from those of the file on disk, this method returns `YES`. You can then use `updateFromPath:` to re-read the file from disk.

Finally, to transmit a file wrapper to another process or system (for example, over a distributed objects connection or through the pasteboard), you use the `serializedRepresentation` method to get an NSData object containing the file wrapper's contents in the `NSFileContentsPboardType` format. You can safely transmit this representation over whatever channel you desire. The recipient of the representation can then reconstitute the file wrapper using the `initWithSerializedRepresentation:` method.

Working With Directory Wrappers

A directory wrapper contains other file wrappers (of any type), and allows you to access them by keys derived from their preferred filenames. You can add any type of file wrapper to a directory wrapper with `addFileWrapper:` or `addFileWithPath:`, and remove it with `removeFileWrapper:`. The convenience methods `addRegularFileWithContents:preferredFilename:` and `addSymbolicLinkWithDestination:preferredFilename:` allow you to add regular file and link wrappers while also setting their preferred names.

A directory wrapper stores its contents in an NSDictionary, which you can retrieve using the `fileWrappers` method. The keys of this dictionary are based on the preferred filenames of each file wrapper contained in the directory wrapper. There exist, then, three identifiers for a file wrapper within a directory wrapper:

- Preferred filename. This doesn't uniquely identify the file wrapper, but the following identifiers are always based on it.
- Dictionary key. This is always equal to the preferred name when there are no other file wrappers of the same preferred name in the same directory wrapper. Otherwise, it's a string made by adding a unique prefix to the preferred filename (note that the same file wrapper can have a different dictionary key for each directory wrapper that contains it). You use the dictionary key to retrieve the file wrapper object in memory, in order to get its contents or its filename (to update it from disk). You can get a file wrapper's dictionary key by sending a `keyForFileWrapper:` message to the directory wrapper that contains it.
- Filename. This is usually based on the preferred filename, but isn't necessarily the same as it or the dictionary key. You use the filename to update a single file wrapper relative to the path of the directory wrapper that contains it. Note that the filename may change whenever you save a directory wrapper containing the file wrapper (particularly if the file wrapper has been added to several different directory wrappers); thus, you should always retrieve the filename from the file wrapper itself each time you need it rather than caching it.

When working with the contents of a directory wrapper, you can use a dictionary enumerator to retrieve each file wrapper and perform whatever operation you need. Note that with the exceptions of saving and updating, a directory file wrapper defines no recursive operations for its contents. To set the file attributes for all contained file wrappers, or to perform any other such operation, you must define a recursive method that examines the type of each file wrapper and invokes itself anew for any directory wrapper it encounters.

Using a Save Panel

Typically, you access an `NSSavePanel` by invoking the `savePanel` class method. A typical programmatic use of NSSavePanel requires you to:

- Invoke `savePanel`
- Configure the panel (for instance, set its title or add a custom view)
- Run the panel in a modal loop
- Test the result; if successful, save the file under the chosen name and in the chosen directory

The following Objective-C code fragment demonstrates this sequence. (Two objects in this example, *`newView`* and *`textData`*, are assumed to be defined and created elsewhere.)

```
NSSavePanel *sp;
int runResult;

/* create or get the shared instance of NSSavePanel */
sp = [NSSavePanel savePanel];

/* set up new attributes */
[sp setAccessoryView:newView];
[sp setRequiredFileType:@"txt"];

/* display the NSSavePanel */
runResult = [sp runModalForDirectory:NSHomeDirectory() file:@""];

/* if successful, save file under designated name */
if (runResult == NSOKButton) {
    if (![textData writeToFile:[sp filename] atomically:YES])
         NSBeep();
}
```

When the class receives a `savePanel` message, it tries to reuse an existing panel rather than create a new one. When a panel is reused its attributes are reset to the default values so the effect is the same as receiving a new panel. Because a Save panel may be reused, you shouldn't modify the instance returned by `savePanel` except through the methods listed below. For example, you can set the panel's title and required file type, but not the arrangement of the buttons within the panel. If you must modify the Save panel substantially, create and manage your own instance using the `alloc...` and `init...` methods rather than the `savePanel` method.

Using an Open Panel

Typically, you access an NSOpenPanel by invoking the `openPanel` method. When the class receives an `openPanel` message, it tries to reuse an existing panel rather than create a new one. If a panel is reused, its attributes are reset to the default values so that the effect is the same as receiving a new panel. Because Open panels may be reused, you shouldn't modify the instance returned by `openPanel` except through the methods listed below (and those inherited from NSSavePanel). For example, you can set the panel's title and whether it allows multiple selection, but not the arrangement of the buttons within the panel. If you must modify the Open panel substantially, create and manage your own instance using the constructors or the `alloc...` and `init...` methods rather than the `openPanel` method.

The following Objective-C code example shows the NSOpenPanel displaying only files with extensions of ".td" and allowing multiple selection. If the user makes a selection and clicks the OK button (that is, `runModalInDirectoryrunModalForDirectory:file:types:` returns `NSOKButton`), this method opens each selected file:

```
- (void)openDoc:(id)sender
{
    int result;
    NSArray *fileTypes = [NSArray arrayWithObject:@"td"];
    NSOpenPanel *oPanel = [NSOpenPanel openPanel];

    [oPanel setAllowsMultipleSelection:YES];
    result = [oPanel runModalForDirectory:NSHomeDirectory()
                    file:nil types:fileTypes];
    if (result == NSOKButton) {
        NSArray *filesToOpen = [oPanel filenames];
        int i, count = [filesToOpen count];
        for (i=0; i<count; i++) {
            NSString *aFile = [filesToOpen objectAtIndex:i];
            id currentDoc = [[ToDoDoc alloc] initWithFile:aFile];
        }
    }
}
```

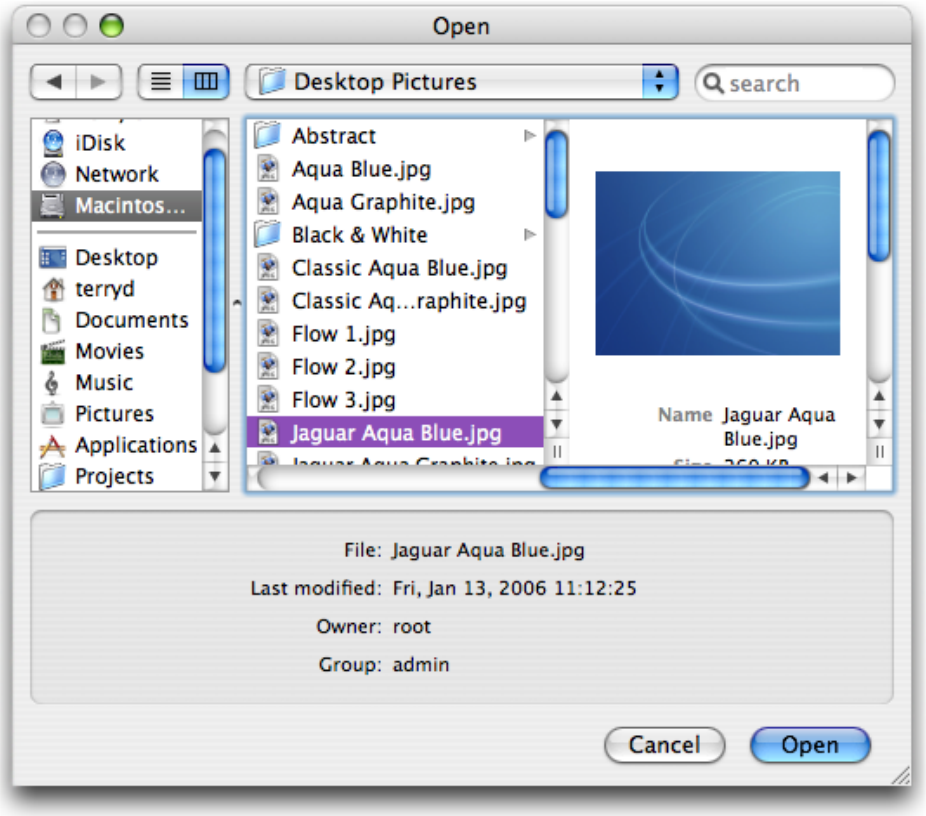
NSOpenPanel can accept file types specified as either filename extensions or encoded HFS file types. To encode an HFS file type into an acceptable NSString use the function `NSFileTypeForHFSTypeCode`. (See "HFS File Types" for details.) When specifying file types for NSOpenPanel, you should include any allowed HFS file types as well as the filename extensions. For example, if you want to open text files, specify a file types array like this:

```
NSArray *fileTypes = [NSArray arrayWithObjects: @"txt", @"text",
                        NSFileTypeForHFSTypeCode( 'TEXT' ), nil];
```

Getting the Current Selection

You can get the currently selected item in the browser of an `NSSavePanel` or `NSOpenPanel` object by having the delegate of the panel object implement the `panelSelectionDidChange:` delegation method. The delegate then can perform some operation based on the user's selection, such as displaying metadata about a chosen file in an accessory view, as illustrated in Figure 1.

Figure 1 Displaying information about the current selection



In its implementation of `panelSelectionDidChange:`, the delegate sends a message back to the `NSOpenPanel` or `NSSavePanel` object (*sender*) to get the current filename. For `NSSavePanel`, this message is `filename`; for `NSOpenPanel`, send `filenames` and then get the first item in the array. Listing 1 shows how you might implement the method to display in the information in the accessory view in Figure 1.

Listing 1 Getting the selection in the panel browser

```
- (void)panelSelectionDidChange:(id)sender {
    NSArray *curFiles = [sender filenames];
    if ([curFiles count] == 1) { // ignore multiple selections
        NSString *curPath = [curFiles objectAtIndex:0];
```

```
        if (curPath != nil) {
            NSDictionary *fAttrs = [[NSFileManager defaultManager]
fileAttributesAtPath:curPath traverseLink:YES];
            if (fAttrs != nil) {
                [infoFile setStringValue: [curPath lastPathComponent]];
                [infoMod setStringValue: [[fAttrs
objectForKey:NSFileModificationDate] descriptionWithCalendarFormat:@"%a, %b %d,
 %Y %H:%M:%S" timeZone:nil locale:nil]];
                [infoOwner setStringValue: [fAttrs objectForKey:
NSFileOwnerAccountName]];
                [infoGroup setStringValue: [fAttrs objectForKey:
NSFileGroupOwnerAccountName]];
            }
        }
    }
}
```

In this code example, the delegate uses the `NSFileManager` method `fileAttributesAtPath:traverseLink:` to fetch information about the currently selected file. It then sets the string value of various text fields in the accessory view.

Filtering Out Browser Items

Suppose you want to prevent certain files in an `NSOpenPanel` object from being selected by the user. You don't want to filter them out by their extension—other files with the same extension are valid selections—but by some other characteristic. The files could be temporary files or files that contain data you don't want users to have access to. You can filter files from being selectable in a browser by implementing the delegation method `panel:shouldShowFilename:`.

Let's assume that the convention of an initial underscore for a filename marks it as a private file, and you don't want users to open these files in your application. There are two of these files in a certain directory:

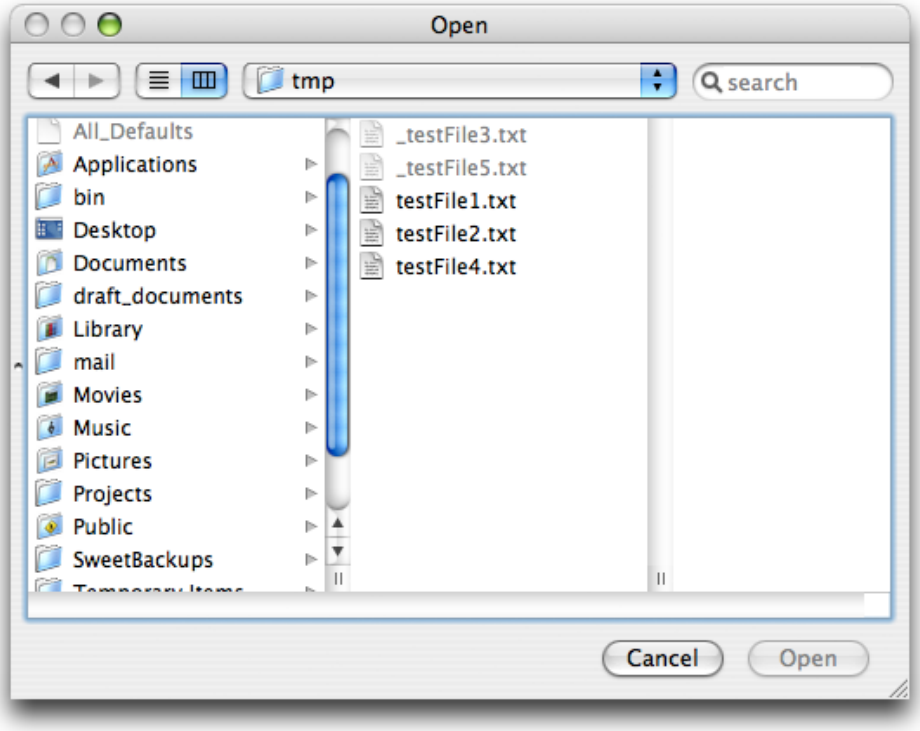
```
testFile1.txt
testFile2.txt
_testFile3.txt
testFile4.txt
_testFile5.txt
```

When you configure an `NSOpenPanel` object, set a delegate for it; then implement the `panel:shouldShowFilename:` method as illustrated in Listing 1. This method is called for each filename to be displayed in the panel's browser; return `NO` if the file should not be selectable.

Listing 1 Implementing the `panel:shouldShowFilename:` method

```
- (BOOL)panel:(id)sender shouldShowFilename:(NSString *)filename {
    NSString *lpc = [filename lastPathComponent];
    if ([lpc characterAtIndex:0] == '_')
        return NO;
    return YES;
}
```

When you next run the open panel and select the directory containing these files, you'll find that he files with the underscore prefixes are now grayed out and are not selectable.



Configuring a Choose Dialog

The Choose dialog, according to *Apple Human Interface Guidelines*, "lets a user select an item as the target of a task." The item is a file-system item, such as a file or a directory. But unlike an Open panel, a Choose dialog allows users to select files or directories without necessarily opening them. If you need to display a Choose dialog in your application, you can do so with a specially configured Open panel. A Choose dialog is an Open panel that:

- Has a title of "Choose *ObjectOrAction*", where *ObjectOrAction* identifies the type of items or signifies the task to be performed on the items

  Ideally, the title of a Choose dialog and the title of the menu item or control initiating the command should match (for example, "Choose Picture"). Also, it is helpful to include some instructional text in the panel, such as "Choose a picture to display in the background."
- Restricts the selection of items to a particular type or types
- Starts browsing at the user's home directory
- Allows multiple selection of items (if appropriate)
- Can have an accessory view (see "Managing Accessory Views" (page 27)) with a Show pop-up menu to filter the types of files that are selectable

Some requirements of a Choose dialog are supported by `NSOpenPanel` by default, such as document preview and the ability to resize the dialog.

Listing 1 shows you might configure, run, and manage a Choose dialog.

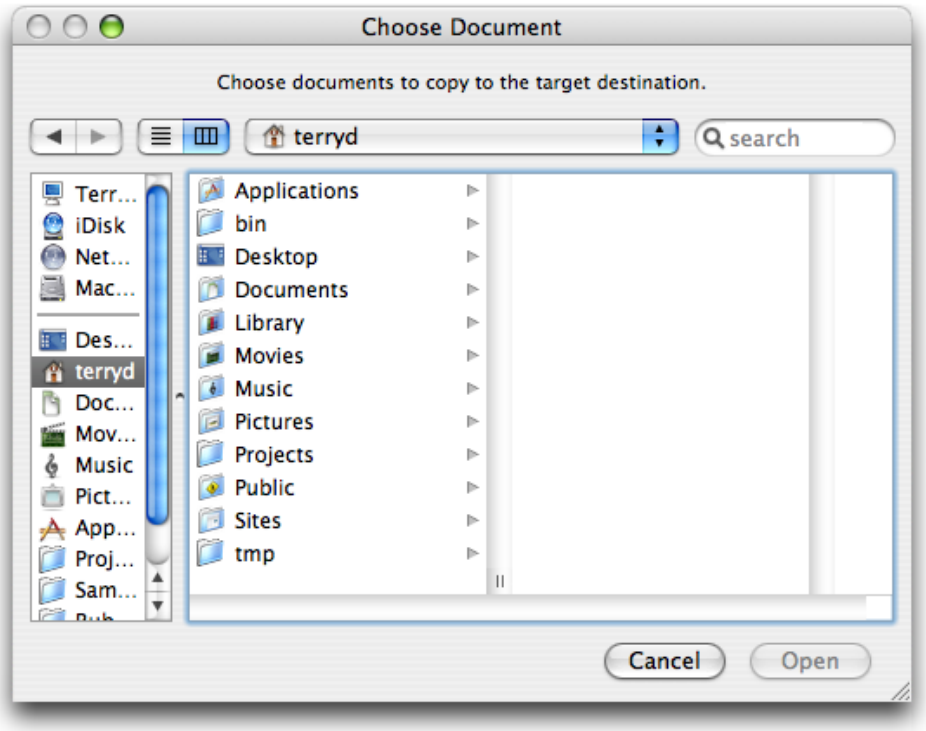
Listing 1 Configuring and running a Choose dialog

```
- (IBAction)copyFiles:(id)sender
{
    int result;
    NSArray *fileTypes = [NSArray arrayWithObjects:@"txt", @"rtf", @"doc", nil];
    NSOpenPanel *oPanel = [NSOpenPanel openPanel];

    [oPanel setAllowsMultipleSelection:YES];
    [oPanel setTitle:@"Choose Document"];
    [oPanel setMessage:@"Choose documents to copy to the target destination."];
    [oPanel setDelegate:self];
    result = [oPanel runModalForDirectory:NSHomeDirectory() file:nil
types:fileTypes];
    if (result == NSOKButton) {
        NSArray *filesToCopy = [oPanel filenames];
        int i, count = [filesToCopy count];
        NSFileManager *fm = [NSFileManager defaultManager];
        for (i=0; i<count; i++ ) {
            NSString *filePath = [filesToCopy objectAtIndex:i];
            NSString *destPath = [[self destinationPath]
stringByAppendingPathComponent:[filePath lastPathComponent]];
            [fm copyPath:filePath toPath:destPath handler:self];
```

```
            }
        }
}
```

This code displays a Choose dialog similar to the one depicted in the following screen shot:



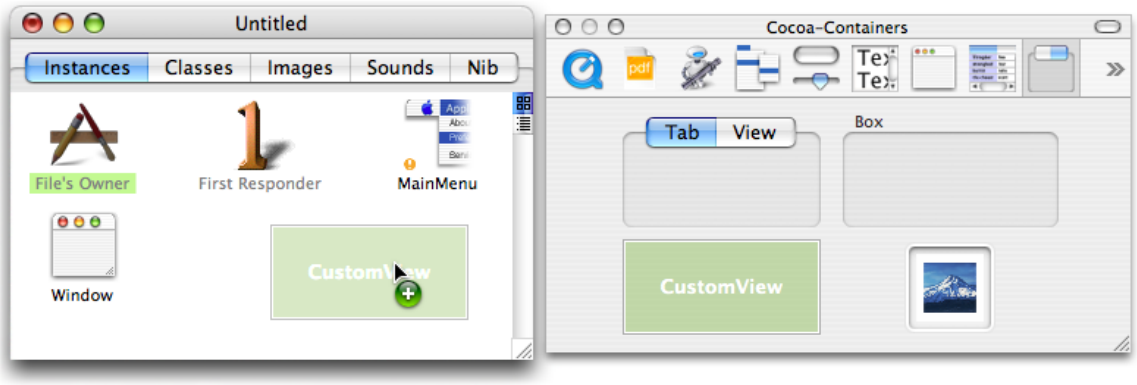
Managing Accessory Views

An accessory view is a view containing controls and other views that you can add to an existing Cocoa panel. The controls affect in some way the item (or items) chosen in the panel, and the views may display an image or other data related to the selected item. Many Cocoa classes allow you to add an accessory view to their objects via the `setAccessoryView:` method. These include `NSSavePanel`, and its subclass `NSOpenPanel`), `NSFontPanel`, `NSColorPanel`, `NSPrintPanel`, `NSPageLayout`, `NSSpellChecker` (in its spelling-correction panel), `NSAlert`, and `NSRulerView`. The location of the accessory view varies from object to object. However, the procedure for creating, adding, and accessing an accessory (summarized in the sections below) is essentially similar for all of these classes.

Creating an Accessory View

To create an accessory view in Interface Builder, start by dragging a CustomView object from the Containers palette to a nib file window.

Figure 1 Adding a view to the top level of a nib file

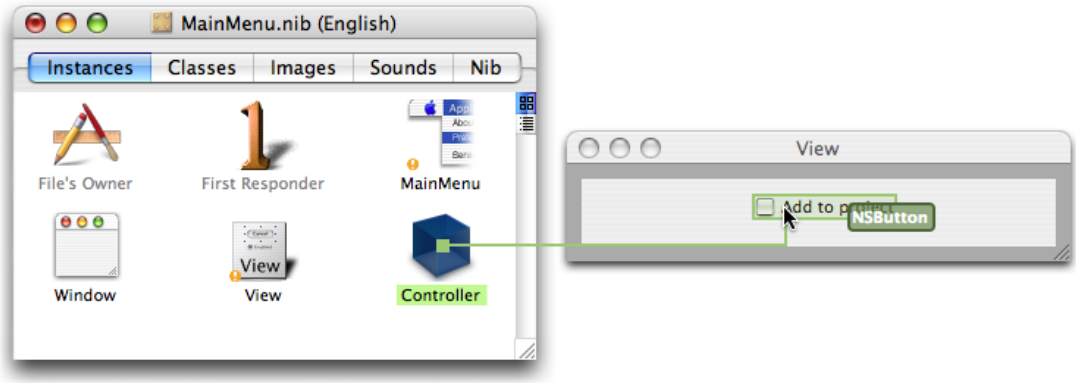


Change the size of the view to generally fit the width of the panel it's going to be added to. Add all required controls, text fields, image views, and other palette objects to the accessory view.

Note: The panel classes of the Application Kit embed an accessory view in an `NSBox` object . It also sets their auto-resizing characteristics: the width of the accessory view is constant, but the length varies as the host panel is resized, with the objects in the accessory view centered.

You next need to specify outlets from some controller object to both the accessory view and its individual controls and connect those outlets. Instead of outlets, you could also define attributes in a controller or model object and then establish bindings between the controls of the accessory view and those attributes. Figure 2 shows the former approach.

Figure 2 Connecting an outlet



Save the nib file. The remainder of the procedure takes place in the Xcode application.

Adding an Accessory View to a Panel

In your application's method that responds to the action message requesting the opening of a file, get the shared instance of `NSOpenPanel` and configure it appropriately, as described in "Using an Open Panel" (page 19). As part of panel configuration, send the `setAccessoryView:` message to the panel object, passing in the outlet to the accessory view. Then run the Open panel and, when the user clicks the OK button, check the state of the controls on the accessory view (via outlets or bindings). Process the selected files accordingly.

Listing 1 illustrates how you might do this.

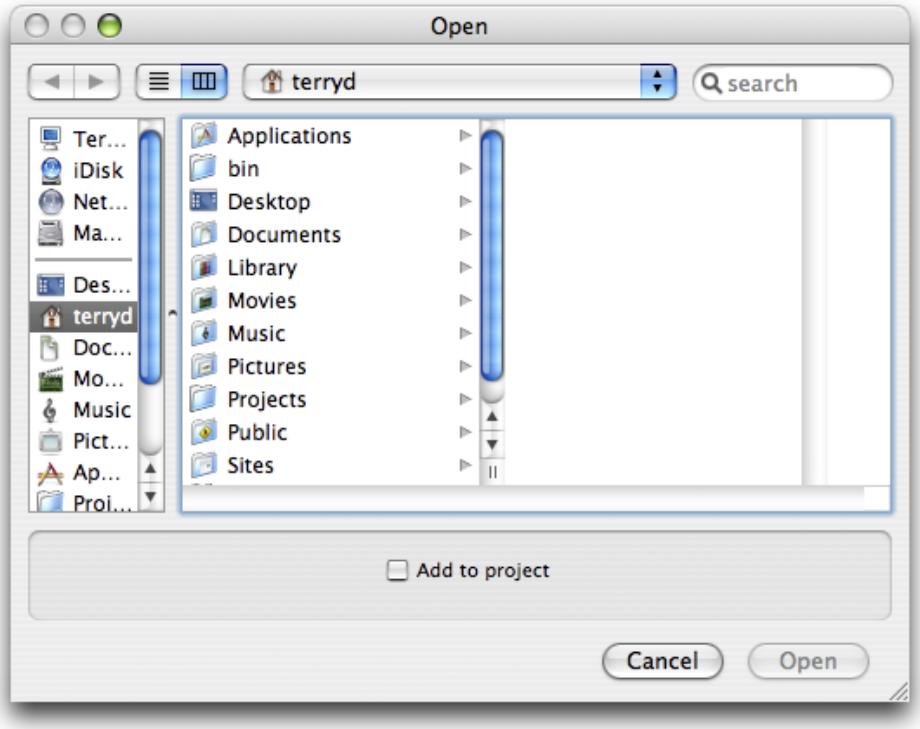
Listing 1 Adding an accessory view and accessing its control

```
- (IBAction)openFile:(id)sender
{
    int result;
    NSArray *fileTypes = [NSArray arrayWithObject:@"xml"];
    NSOpenPanel *oPanel = [NSOpenPanel openPanel];

    [oPanel setAllowsMultipleSelection:YES];
    [oPanel setAccessoryView:accessView]; // add the accessory view to the open
 panel
    result = [oPanel runModalForDirectory:NSHomeDirectory() file:nil
types:fileTypes];
    if (result == NSOKButton) {
        NSArray *filesToOpen = [oPanel filenames];
        int i, count = [filesToOpen count];
        for (i=0; i<count; i++ ) {
            NSString *aFile = [filesToOpen objectAtIndex:i];
            if ([addToProj state] > 0) {    // is check box in accessory view
checked?
                [self addToProject:aFile];
            }
            [[NSWorkspace sharedWorkspace] openFile:aFile
withApplication:@"Sweet.app"];
        }
```

```
        }
}
```

This code causes an Open panel similar to the following to be displayed



Document Revision History

This table describes the changes to *Application File Management*.

Date	Notes
2006-11-07	Added articles on managing accessory views, filtering browser items, configuring Choose dialogs, and managing the current selection.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.