11 Dynamic Loading

...to divide is not to take away.
Percy Bysshe Shelly

Choosing each stone, and poising every weight,
Trying the measures of the breadth and height;
Here pulling down, and there erecting new,
Founding a firm state by proportions true.
Matthew Arnold

How many things I can do without!
Socrates

Multiple Nib Files: Good Things in Small Pieces

Why have multiple nib files in an application? Why not put everything in the main nib file? The answer is simple: Because multiple nib files enhance the performance of the application.

You can strategically store the resources of an application (including pieces of the interface) in several nib files. When the application needs a resource, it loads the nib file containing it. Because you don't have to load the entire application into memory at once, the program is more efficient. The application also will launch faster.

When many sophisticated applications start up, they load only a minimum of resources in the main nib file—the main menu and perhaps a window. They display other windows (and load other nib files) only when users request it or when conditions warrant it.

Types of Auxiliary Nib Files

Nib files other than an application's main nib file are sometimes called *auxiliary nib files*. There are two general types of auxiliary nib files: special-use and document.

Special-use nib files contain objects (and other resources) that *might* be used in the normal operation of the application (like a Preferences panel). Document nib files contain objects that represent some repeatable entity, such as a word-processor document. A document nib file functions as a template for documents: it contains the UI objects and other resources needed to make a document. (Creating document nib files is described at length in the book *Discovering OPENSTEP*.)

File's Owner

The key step in creating applications with multiple nib files is assigning the auxiliary nib file's File's Owner. The file's owner object is always external to the nib file it owns. It channels messages between the objects unarchived from the nib file to the other objects in your application.

The global NSApplication object owns the main nib file. Special-use nib files are often owned by the application's controller object, which you typically define in the main nib file. A document nib file is typically owned by a separate controller object, a document controller.

The main job of the File's Owner object is to load the auxiliary nib file. To do so, it sends the message **loadNibNamed:owner:** to the NSBundle class object. In the main nib file you define an action method in the controller class and hook that action up to a control in the interface. That action method's implementation sends the **loadNibNamed:owner:** message. In this way, the nib file is loaded only if the user requests it.

Creating Auxiliary Nib Files

To create an auxiliary nib file, you use one of the commands on the New Modules menu (which is under the Document menu) in Interface Builder. New Modules gives you several choices of the type of nib file to create:

- New Info Panel Creates an info panel.
- New Attention Panel Creates an attention panel.
- New Empty Creates an empty nib file.
- New Palette Creates a static palette.
- New Inspector Creates an inspector panel.

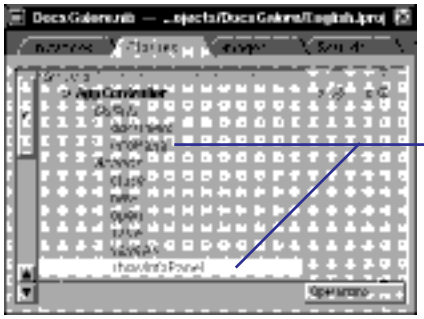
The last two commands (New Palette and New Inspector) are used when creating static palettes. If you're not working on a palette project, you use the New Empty command to create a nib file, unless you're specifically creating an info panel or an attention panel.

You might have noticed that the New Application command also creates a nib file. This command creates a main nib file—one that contains a main menu and is owned by the NSApplication object. However, you usually let Project Builder create the main nib file for you when you create an application project.

Loading nib files dynamically: an info panel

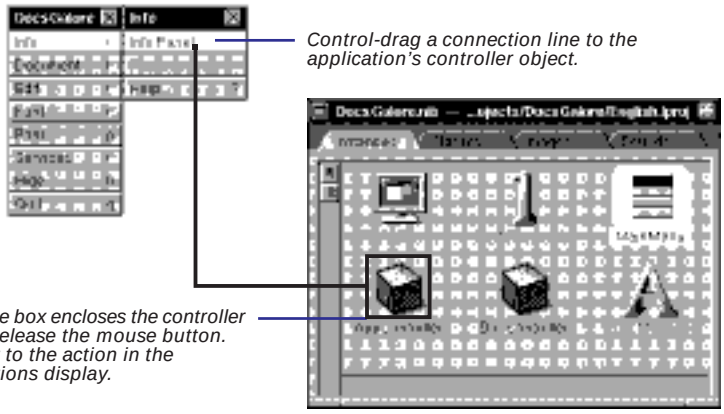
1. **Create an outlet for the Info Panel and an action method that displays the Info panel in the application's controller class.**
2. **Connect the Info Panel command to the controller object.**
3. **Choose Document ► New Modules ► New Info Panel to create a nib file for the Info Panel.**
4. **Add the controller class to the new nib file.**
5. **Assign the controller class to File's Owner.**
6. **Connect the Info Panel to the File's Owner outlet for the panel.**
7. **Implement the action method that loads the Info Panel's nib file.**

The steps you follow to create, load, and manage an Info Panel are common to creating any special-use nib file. First, in the main nib file, you create an object that knows about the Info Panel. Usually, this object is the application's controller object. Define the class of this object in the Classes display of the main nib file. When you do, specify the necessary outlet and action for the Info Panel.



Specify an outlet to identify the panel and an action message that the Info Panel command sends when users click it.

Instantiate the controller class, and connect the action to the menu command.



Control-drag a connection line to the application's controller object.

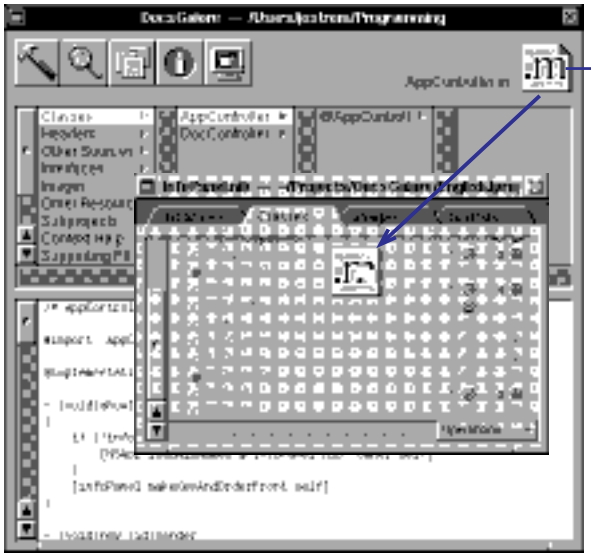
When the box encloses the controller object, release the mouse button. Connect to the action in the Connections display.

Chapter 6, "Subclassing," describes how to add outlets and actions to your custom class and shows how to connect them to instances of your class.

Now choose the New Info Panel command. When you do, Interface Builder displays a template panel and creates an untitled nib file to contain it. Be sure to save the file (as, for instance, **InfoPanel.nib**).

Modify the template Info Panel to contain the application icon and name, your name, and version and copyright information.

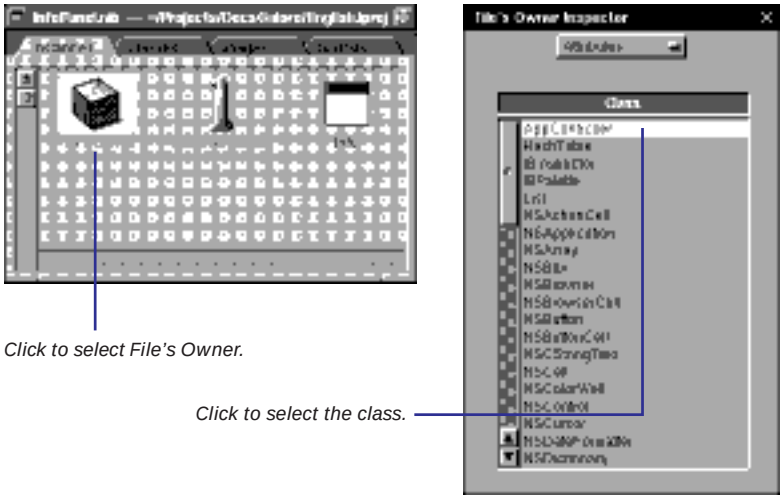
You cannot connect the Info Panel to the controller object in the main nib file because the panel is in the new auxiliary nib file. You must assign the controller class to File's Owner in the auxiliary nib file and then make the outlet connection between File's Owner and the panel. The first step in this direction is to insert the class definition of the controller class into the auxiliary nib file.



From Project Builder, drag the header file or the implementation file for the controller class and drop it over the nib file window.

You can also copy a class definition between nib files using the Edit menu's Copy and Paste commands. Copy the class in one Classes display, select the superclass in the other Classes display, and then paste the class into the nib file.

Next, assign the controller class to File's Owner.



Now make a connection in Interface Builder between File's Owner and the title bar of the panel. Select the info panel outlet in the Connections display and click Connect.

The final step is to write the code (in the **.m** file of the controller class) that implements the action method invoked by the Info Panel command.

```
- (void)showInfoPanel:(id)sender
{
    if (!infoPanel)
      [NSBundle loadNibNamed:@"InfoPanel" owner:self];
    [infoPanel makeKeyAndOrderFront:self];
}
```

Notes on the code: Once an Info Panel is loaded, it is kept in memory until the user quits the application. The code tests the infoPanel outlet to determine if the auxiliary nib file containing the panel has already been loaded. If it hasn't, it loads it with **loadNibNamed:owner:**. It is important to specify **self** as owner (**self** being the object that implements the method). Display the panel by sending it the **makeKeyAndOrderFront:** message.

Displaying an attention panel

- **Call NSRunAlertPanel().**
  Or
- **Create an attention panel in Interface Builder and load it dynamically.**

When you can accomplish an end programmatically or in Interface Builder, the recommended course is almost always Interface Builder. A notable exception is displaying attention panels. You display attention panels to tell the user something about the current context (such as an error that occurred), to clarify or complete an action the user is taking, or to give the user a chance to take corrective steps.

Displaying Attention Panels Programmatically

For most situations requiring attention panels, the easiest and most appropriate thing to do is call a function: **NSRunAlertPanel()**. In the following example, the application informs users that, because of hardware incompatibility, it cannot proceed:

```
if(![LiveVideoView doesWindowSupportVideo:bufWindow
                  standard:&type size:&vidSize])
  {
    NSRunAlertPanel(@"No Video Present", @"This machine is not
     capable of running video applications. Since this program
     is exclusively for Video,it will now exit.", @"OK", nil, nil);
    [self terminate:self];
  }
```

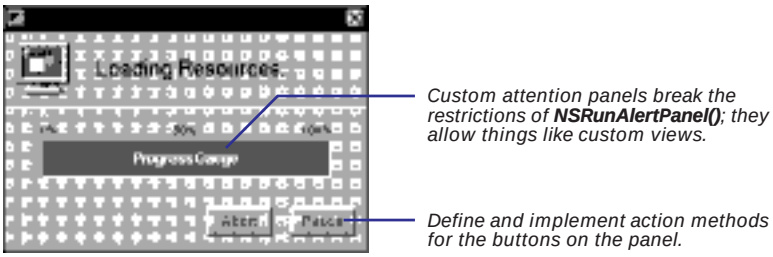
Notes on the code: The arguments of **NSRunAlertPanel()** determine what appears on the panel. The first argument is the heading (above the dividing line), and the second is the text (below the line). The next three arguments are the titles of the buttons that appear across the bottom of the panel. The first of these titles goes to the default button, which has a carriage return associated with it. You can remove a button by giving **nil** as its title, but you must specify something for all three arguments. The declaration of **NSRunAlertPanel()** permits a variable number of arguments, so you can have **printf()**-style format specifiers in the panel heading and text and variables following the third button argument.

The Application Kit defines other functions related to **NSRunAlertPanel()**. For more information on these functions, see the "Functions" section of the *Application Kit Reference*.

The call to **NSRunAlertPanel()** in the example above creates the following panel:

Loading Attention Panels Created in Interface Builder

The panel created by **NSRunAlertPanel()** might not be adequate for certain situations. For example, you might want to display an attention panel that has a special view object, say one that shows the progress of some lengthy process (such as a progress bar for loading or copying files). And you want to give the user the options of aborting or pausing that process. You'd want something like this:



To implement a custom attention panel, you perform almost identical steps as you do to create an Info panel:

1. Pick a custom class, typically the application's controller, to manage the panel.
2. Specify an action and outlet in the controller class.
3. Connect the action in the main nib file.
4. Create a nib file for the attention panel by choosing Document ► New Modules ► New Attention Panel.
5. Compose the text, graphics, and other UI elements of the panel.
6. Drag the controller's header file to the attention panel's nib file window.
7. Assign the controller class to File's Owner.
8. Assign the attention panel to the File's Owner attention panel outlet.
9. In the action method, load the panel's nib file with **loadNibNamed:owner:**.

There are some important differences between attention panels and Info panels. With attention panels, you typically load the nib file not as the result of a user action (for instance, clicking an panel Panel command), but because of internal conditions in your code. Also, you dismiss an Info Panel by clicking its close box; you usually dismiss an attention panel by clicking a button on the panel. This means that, for custom attention panels, you will have to define and implement action methods for the buttons on the panels. (This is something **NSRunAlertPanel()** simulates by returning a code indicating the button clicked.)

Creating a window with multiple displays

1 **In a nib file, create a window with an empty box in the place where the display should change.**

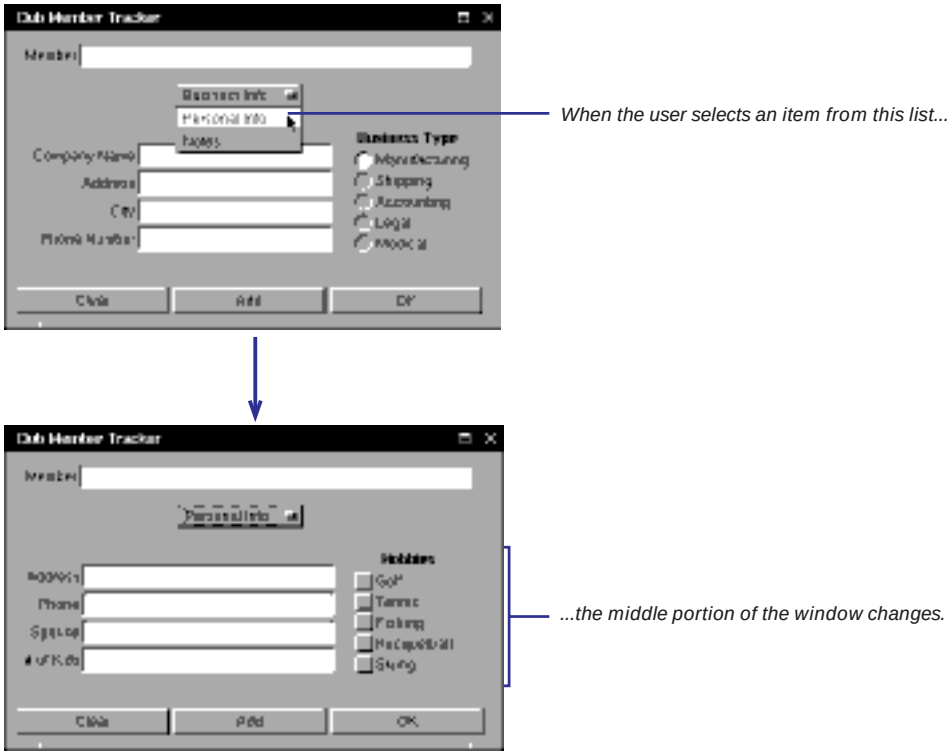
2 **Add control objects that allow the user to change the display, and hook them to action methods in a controller class.**

3 **For each display, use Document ► New Modules ► New Empty to create a nib file containing a window with just that display.**

4 **Assign the controller to be the nib file's owner and connect one of its outlets to the display.**

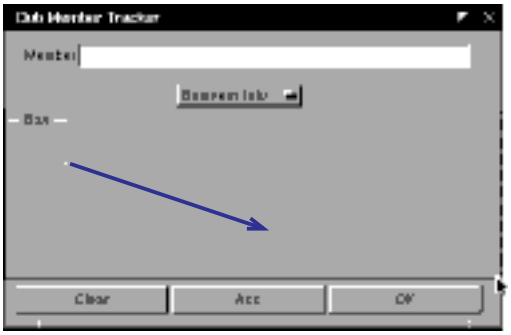
5 **In the action method's implementation, load the appropriate nib file.**

One common interface style is to have a window whose display changes upon a user action, such as clicking a button. For example, Interface Builder's Inspector panel changes its display when you choose a different item in the pop-up list. Another example is Project Builder's Preferences panel. Both of these panels have infrequently-used displays, thus it makes sense to store these displays in nib files that are loaded only if needed.



The key to creating a window with multiple displays is the content view attribute. NSBox, NSScrollView, and NSWindow all have a content view attribute. The content view is the superview of all of the view objects, such as button and text fields, inside of the box, scroll view, or window. You can send a **setContentView:** message to a box to swap out the entire contents of the box and replace them with new contents. The rest of this task uses the window shown above to show you how to create a window with multiple displays.

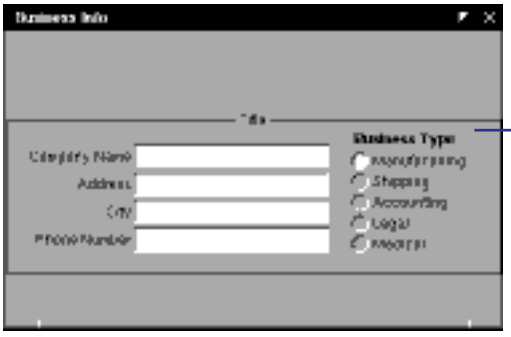
You can define each of the window's displays in separate nib files. In the main nib file, place a box in the area that you want to be changeable.



Drag a box from the Views palette and resize it to be the same width as the window. In the Attributes inspector, set the box to have no title and no border.

Also in the main nib file, define the controller class. Give the controller class an outlet for each view (in this example, Business Info, Personal Info, and Notes) plus outlets for the main window and the box on the main window. Also define an action for the controller class, named something like **setContents:**, and connect the pop-up list to that action.

Next, use the New Empty command to create a nib file for each of the displays that the window can show. In each of these nib files, create a window by dragging one from the Windows palette. Your application never displays these windows; they exist only to hold the view objects that the main window will display.

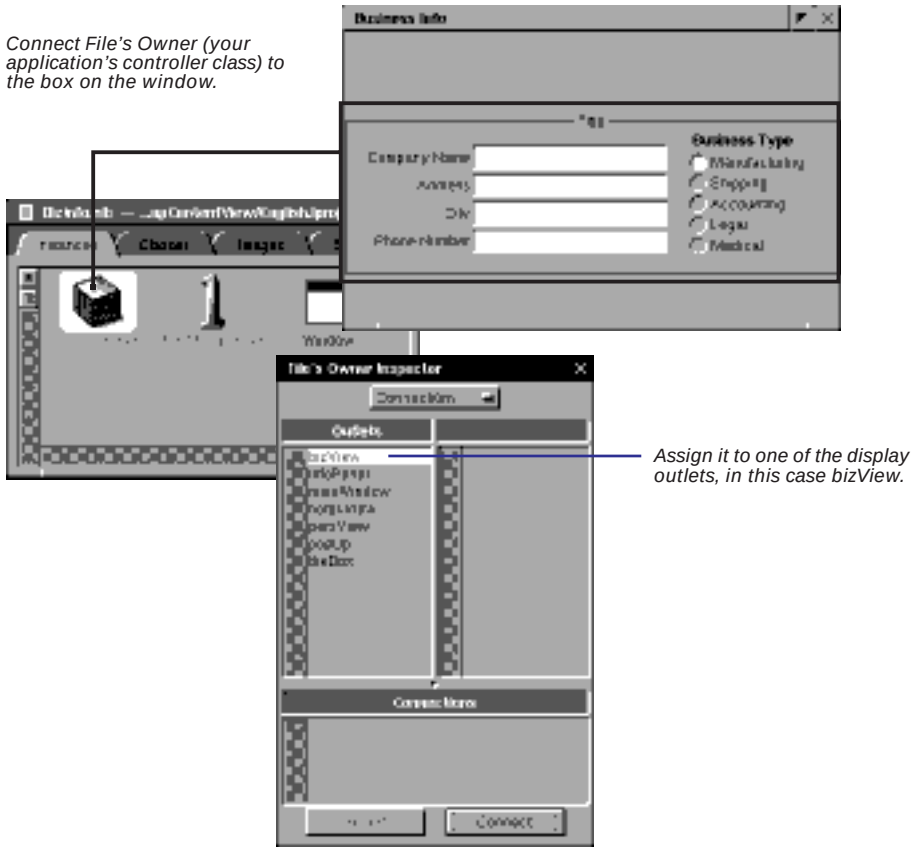


Place interface objects in a box in the auxiliary nib file's main window. When you're done, make the box invisible by choosing no border and no title in the Attributes display.

"Grouping objects" in Chapter 2 describes how to group objects inside of a box. "Setting box (group) attributes" in Chapter 3 describes the box's Attributes display in the Inspector panel.

Tip: Use the Size display of the Inspector panel to make sure that this box is just smaller than the box on the main window. It also helps to make the auxiliary nib file's window the same size as the main nib file's window.

Now you need to connect this display to the controller class. Add the controller class to the nib file and assign it to the File's Owner object. Then, connect File's Owner to the box you just created.



In this example, we would create two more auxiliary nib files, one for the "Personal Info" display and one for the "Notes" display, in the same manner as the **BizInfo.nib** file shown above.

Note: If one of the displays is going to be used frequently, you might want to create it in an off-screen panel in the main nib file rather than creating a separate nib file and incurring the overhead of reading it in.

After all of the nib files have been created, implement the action method that you connected to the pop-up list. In this example, the action method is named **setContents:**. Its implementation is shown here.

```
- (void)setContents:(id)sender
{
   switch((InfoType)[[sender selectedItem] tag]) {
     case BUSINESS:
      if (!bizView) {
          [NSBundle loadNibNamed:@"BizInfo" owner:self];
          [bizView retain];
      }
      [theBox setContentView:bizView];
      break;
     case PERSONAL:
      if (!persView) {
          [NSBundle loadNibNamed:@"PersInfo" owner:self];
          [persView retain];
      }
      [theBox setContentView:persView];
      break;
     case NOTES:
      if (!notesView) {
          [NSBundle loadNibNamed:@"Notes" owner:self];
          [notesView retain];
      }
      [theBox setContentView:notesView];
      break;
   }
}
```

Notes on the code: Based on the pop-up list's selection, this method loads the appropriate nib file, if necessary, then sets the contents of the box to the view defined in that nib file. As a view becomes the box's content view, it is removed from the window in its nib file. The **setContentView:** method releases the box's previous content view and retains the new one. To ensure that the previous content view isn't deallocated before the next user event, you must retain each view. By preventing the views from being deallocated, you allow your users to switch back and forth between them. (Be sure to release the views in your class's **dealloc** method.)

This method uses an enumerated type (**InfoType**) to give meaning to the pop-up list items' tags. For more information on using tags, see "Using tags" in Chapter 3, "Setting an Object's Attributes."

Finally, you need to have the Business Info display appear when the application starts up. To set this up, in the main nib file assign your Controller class to be the NSApplication delegate (NSApplication is the main nib file's owner). Implement the delegate method **applicationDidFinishLaunching:** as shown:

```
- (void)applicationDidFinishLaunching:(NSNotification *)notify
{
    [self setContents:popUp];
    [mainWindow makeKeyAndOrderFront:self];
}
```

Notes on the code: This method is invoked immediately after the NSApplication object has finished initializing itself. It invokes **setContents:** to load the nib file for the default display (Business Info) and set the contents of the box on the main window to be the view defined in that nib file. Then it displays the main window.

Tip: Make sure that the main window's "Visible at launch time" attribute is deselected. Otherwise, there will be a slight lag between the time the window appears on the screen and the time that the box's contents appear on screen.

Inside the NSBundle Class

If you look at the NSBundle class specification in the *Foundation Framework Reference*, you'll notice that NSBundle can tell you a lot of useful things: where your program's resources are, where its frameworks are, which framework defines a particular class. It can even tell you how your application's interface ought to be localized. Why is it so smart?

Every bundle contains a property list that defines the bundle's attributes. This property list is the real brains behind the NSBundle class; NSBundle is simply reading the property list and returning the information it contains. Project Builder uses the information you specify in Project Builder's Inspector panel to create and update this property list.

The Principal Class

At the very least, the property list contains the name of the bundle's executable. Most property lists (in fact, all of them besides those used for frameworks) must contain one other important piece of information: the principal class's name.

The principal class is the class that performs the main work of the bundle. For applications, the principal class is either NSApplication or a subclass of NSApplication. NSApplication runs the application event loop, during which the custom code you have written for your application is executed.

For Loadable Bundle projects, the principal class is often a controller-style class. It knows about all of the other objects inside of the bundle and can send them messages to have them perform work. If the bundle contains a nib file, the bundle's principal class is often the appropriate choice for the owner of that nib file (just as NSApplication owns the main nib file of an application bundle).

The principal class is important because the NSBundle class uses it to load a bundle into memory. Loading a bundle is typically a two step process. First you create an NSBundle object using the location of the bundle in the file system as input. Then, you send that bundle the message **principalClass**. This method returns the principal class in the bundle. In order to do this, it must read the property list, which in turn means it must load the bundle into memory if that bundle has not already been loaded. Thus, asking an NSBundle for its principal class is the main way you load a bundle into memory. From there, you can create an instance of the principal class and send it a message to have it perform work.

If NSBundle can't find out the name of the principal class from the property list, it assumes that the first class loaded is the principal class. This is determined by the order in which the object files are linked.

Application Property Lists

A simple application project contains two more pieces of information in addition to the executable name and principal class name: the name of the main nib file, and a list of file formats the application can read and write. Most applications also have a line that identifies the application's icon.

Adding Information to the Property List

Your project is not limited to the information that Project Builder stores in the property list. You can use this list to store other information specific to your application. However, because Project Builder maintains this list, you should never update it directly. Instead, create a file named **CustomInfo.plist** and add it to the project under Other Resources. Project Builder looks for such a file and merges it with the other information to create property list. Two reasons that you would create a **CustomInfo.plist** file are to advertise a service that your application performs (on the Services menu of other applications) or to add on-line help to your application.

Creating dynamically loadable bundles

1 **Create a project or subproject of type Loadable Bundle.**

2 **In the Project Attributes Inspector, enter the name of the bundle's controller class in the Principal Class field.**

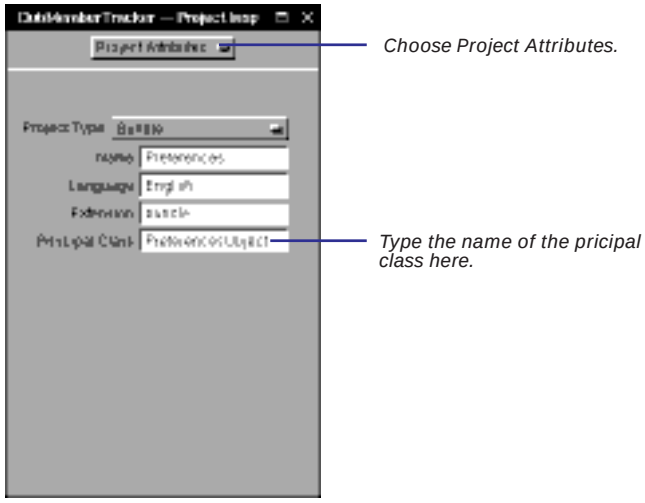
3 **Add classes, interfaces, and resources as you would for any other project.**

4 **Create a class outside of the bundle that loads the bundle.**

The other tasks in this chapter show how to separate the interface into multiple nib files so that infrequently used parts of the interface are loaded only if needed. You can do the same thing with the application's executable code—divide it into wholly contained pieces that are loaded only if needed.

To separate out a portion of executable code, you create a loadable bundle. *Loadable bundles* are file packages that can contain executable code, resources, and nib files. The main difference between a loadable bundle and an application is that an application has a **main()** function and an NSApplication instance. Loadable bundles typically don't have **main()** functions.

The key attribute of a bundle project is its principal class. The principal class is essentially the controller class for the bundle. You must specify the principal class in the bundle project's attributes inspector.



Although Loadable Bundle projects can be stand-alone projects, they are often created as subprojects of an application or framework. For more on subprojects, see "Grouping projects" in Chapter 1.

Loading the Bundle Programmatically

Because a loadable bundle doesn't have a **main()** function, you must write code that loads the bundle and starts executing. The following method does just that:

```
- (void)showPreferences:(id)sender
{
    Class bundleClass;
    id newInstance;
    NSBundle *bundleToLoad =
      [NSBundle bundleWithPath:[[NSBundle mainBundle]
      pathForResource:@"Preferences" ofType:@"bundle"]];

    if (bundleClass = [bundleToLoad principalClass]) {
        newInstance = [[bundleClass alloc] init];
        [newInstance loadPanel];
     }
}
```

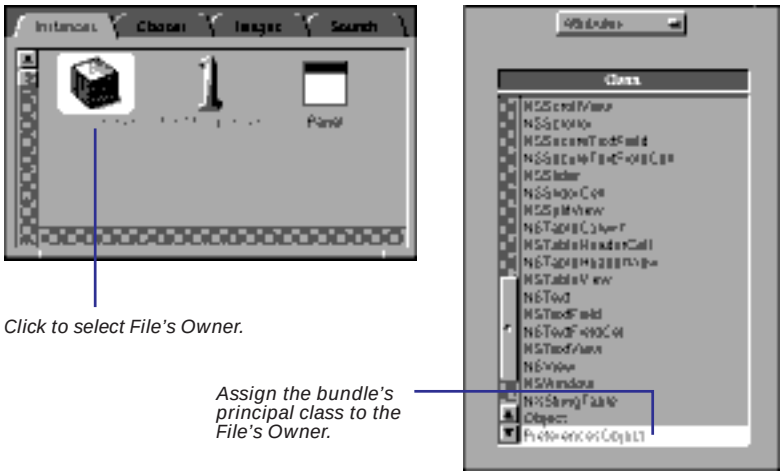
Notes on the code: This method loads the bundle into memory. It starts by telling the NSBundle class where to find the bundle. In this case, the bundle is in a subproject, which means it resides in the **Resources** directory inside the main bundle, so sending **pathForResource:ofType:** to the main bundle returns the correct location. The **principalClass** method finds out the bundle's principal class, loading the bundle if necessary. Once the principal class is known, this method creates an instance of that class and sends it a message. (In this example, the message is to load a panel defined in the bundle.)

Adding a Nib File to a Bundle Project

Because loadable bundles can contain nib files, it's often convenient to create a bundle containing an infrequently used part of the interface and the code that controls it. For example, you could put the Preferences panel and an object that controls it in a separate bundle project.

"Loading nib files dynamically: an info panel" in this chapter walks through the major steps of creating a nib file from the New Module menu and assigning the File's Owner.

To create a nib file in a bundle project, use the Interface Builder command Document ► New Module ► New Empty. Add the bundle's principal class to the nib file and set the File's Owner to be that class.



You'll need to connect this part of the interface to the main nib file. To do so, have the application's controller object load the bundle in response to an action message. In the example shown here, the bundle is loaded when the user chooses the Preferences command. (**showPreferences:** method is shown above.)