

OPENSTEP DEVELOPMENT: TOOLS & TECHNIQUES

Apple Computer, Inc.

Tools & Techniques

© 1997 Apple Computer, Inc.

All rights reserved.

May 1997

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc., except to make a backup copy of any documentation provided on CD-ROM. Printed in the United States of America.

The Apple logo is a trademark of Apple Computer, Inc. Use of the “keyboard” Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple-labeled or Apple-licensed computers.

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for printing or clerical errors.

Apple Computer, Inc.

1 Infinite Loop

Cupertino, CA 95014

408-996-1010

Apple, and the Apple logo are trademarks of Apple Computer, Inc., registered in the United States and other countries. NeXT, the NeXT logo, NEXTSTEP, the NEXTSTEP logo, Digital Librarian, NeXTmail, and Workspace Manager are trademarks of NeXT Computer, Inc. All other trademarks mentioned belong to their respective owners.

1

Creating and Managing a Project

Creating a project

Setting indexing preferences

Grouping projects

Changing a project's name

Setting the application icon

Setting document icons

Setting system-defined
document icons

I have a bit of FIAT in my soul,
And can myself create my little world.

Thomas Lovell Beddoes

But from these create he can
Forms more real than living man,
Nurslings of immortality!

Percy Bysshe Shelley

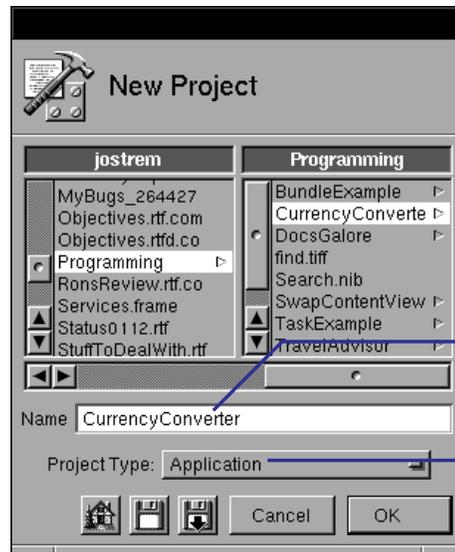
I must Create a System, or be enslav'd by another Man's;
I will not Reason and Compare: my business is to Create.

William Blake, *Jerusalem*

Creating a project

- 1 In Project Builder, choose **New** from the **Project** menu.
- 2 In the **New Project** panel, choose the project's type from the **Project Type** pop-up list.
- 3 Name the project.
- 4 Choose **OK**.

A *project* is a set of files that produces a given end product, such as an application, a tool, a library, or a loadable bundle. When you create a project in Project Builder, you create a directory that will hold all of the project's code files and resource files. Project Builder adds several supporting files, such as project makefiles and templates that you can use to create source files, to that directory.



Project Builder creates a directory named **CurrencyConverter** and places the project's supporting files in it.

Choose the project type from this list. Remember to choose the project type; you can't change the type of an Application project later.

Project Types

Project Builder can create these types of projects:

Application A standalone OPENSTEP application.

Tool A server or command-line tool.

Loadable Bundle A directory of resources, such as images, sounds, character strings, nib files, and dynamically loadable executable code, to be used by one or more applications.

Library A static or dynamic shared library.

Framework A bundle that contains a dynamic shared library plus resources. See "Frameworks: Easy to Use, Easy to Create" in this chapter.

Palette A static Interface Builder palette—a palette with code that you must compile before it can be used.

Legacy A project for which Project Builder doesn't maintain the makefile. Use this when you have created your own makefile. See "Legacy Projects" in this chapter.

Aggregate A collection of loosely related projects. See "Grouping projects" in this chapter.

For the most part, the only difference between project types is the kind of executable they produce. However, there are some special issues involved in frameworks and libraries. See Chapter 12 for more information.

Use the **Open** command on the **Project** menu to open a project that already exists. In the **Open** panel, select the project's folder.

Managing Project Files With Project Builder

Project Builder makes it easy for you to manage a project's files. It organizes your project for you by grouping files into the following types (not all types are available for all projects):

Classes The “.m” files that implement an Objective-C class.

Headers The “.h” files that define a class's interface or declare C functions, data types, and variables.

Documentation The documentation files for framework projects, which must be RTF files.

Context Help The help files for this project, which must be RTF files. This only exists for application projects and subprojects.

Other Sources Objective-C files that don't implement a class or files containing code in other languages, such as C, C++, Objective-C++, or PostScript.

Interfaces Interface Builder nib files that define the user interface. Nib files are described further in Chapter 2.

Images Image files, such as TIFF or EPS files, other than icons.

Other Resources Files containing other resources (such as sound files or eomodels) that the project uses.

Subprojects Subprojects. See “Grouping projects” in this chapter.

Supporting Files Makefiles and other files the project does not use directly.

Libraries Libraries that the project links to, such as those in `/usr/lib` or `/lib`.

Frameworks Dynamic shared libraries that the project links to, such as the Application Kit. OPENSTEP-supplied frameworks are in `/NextLibrary/Frameworks`. (See “Frameworks: Easy to Use, Easy to Create” in this chapter.)

Non Project Files Files that you have opened that aren't part of the project.

In addition, Project Builder creates and maintains some files for you. For all projects, it creates a makefile and templates for a class's interface (“.h”) and implementation (“.m”) files. For application projects, it also creates a `Project_main.m` file, which contains the `main` function, and a nib file, which defines your application's interface. You can customize both of these files for your application.

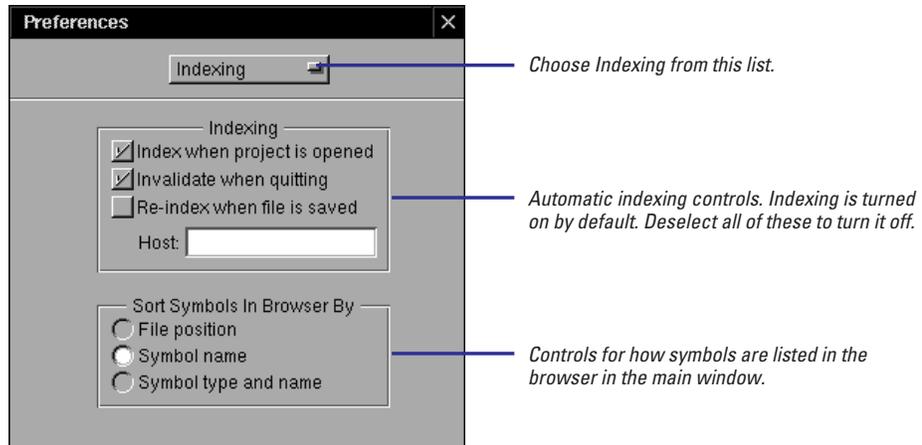
Of course, you can add your own files to the project. You can also remove files, rename files, create new files, and even open files that aren't part of the project using commands on the File menu.



Setting indexing preferences

- 1 Choose **Info** ► **Preferences**.
- 2 In the **Preferences** panel, choose **Indexing** from the pop-up list.
- 3 **Select or deselect** the preferences you want.

For you to take full advantage of Project Builder, the source code must be indexed. When a project is indexed, Project Builder keeps track of each symbol in the project, what that symbol defines (such as a class or a function), where it is declared, and where it is used. The Indexing Preferences panel allows you to control when and how a project is indexed.



By default, “Index when project is opened” and “Invalidate when quitting” are selected. These preferences cause Project Builder to create a new index in memory each time it is opened and to delete the index when you quit the application. (This ensures that the index is updated at least when you quit Project Builder.) If you deselect “Invalidate when quitting,” the project index will persist until you reboot.

Indexing requires overhead. To improve performance, you can turn indexing off; however if you do this, many useful features, such as project-wide name completion and language-based searching, aren’t available. Another option is to have the indexing process run on another computer on your network. To do this, enter that computer’s host name in Host field of the Indexing Preferences panel.

Chapter 9, *Building*, describes the Build Attributes inspector and how to build the project on another host.

Tip: If the project indexes incorrectly, make sure that you have set the information in the Project Attributes and Build Attributes inspectors. Use the same host to index the project as you use to build the project.

When you index, the browser can show you information about a file's contents.

The classes, protocols, and other symbols the selected file defines.



The methods defined for that class or protocol.

The editor jumps to the class or method you select.

The Project Server

Indexing is actually performed by a background process called the project server. The project server is the brains behind Project Builder. When you request information stored in the index, Project Builder asks the project server for that information, then relays it to you.

The project server starts up as soon as you open a project. In just a short time, project server can build a cache of symbol information about your project.

The project server is a continually running process. Even after you quit Project Builder, the project server may continue running. This saves time at startup; Project Builder only has to

start up a project server when you reboot.

When you set the Host attribute on the Preferences panel, the project server runs on that host. If other people on the network use the same computer for their project servers, one project server is created per user.

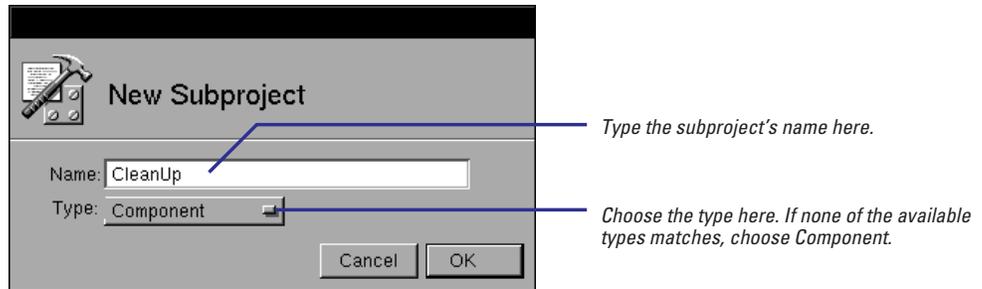
If you need to control the project server, use the commands on the Indexing menu under the Project menu. The command Purge Indices kills the current project server. After you use this command, use Index Source Code to start a new project server and reindex your project.

Grouping projects

- ▶ To incorporate the build result of one project into the build result of another project, create a subproject.
- ▶ To group together related projects of any type, create an Aggregate project.

If you have many projects that relate to each other, you may want to organize them as a single project in Project Builder. You can group projects into a single project in two ways: by creating a subproject and by creating an Aggregate project.

If one of the projects is clearly a part of the other, create it as a subproject of that other project. To create a subproject within a project, choose **New Subproject** from the **Project** menu to bring up the **New Subproject** panel, choose the type of subproject you want to create, type a name for the subproject in the **Name** field, and click **OK**.

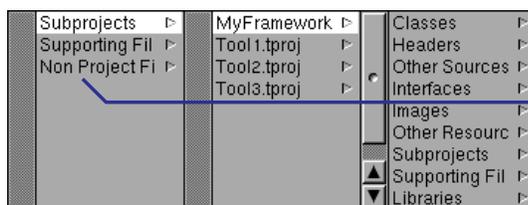


When you create a subproject, you are creating a project whose build result (executable or bundle) is incorporated into the build result of the main project. (A *bundle* is simply a file package directory that contains all of the resources the user needs to execute the program.) If the main project produces a bundle as its build result, the subproject's executable or bundle is placed inside the main project's bundle. Component subprojects produce an object file that is linked with the main project's executable. A component's resources are merged into the resource directory of the main project.

For example, if you are creating an application named **MyApp** and one of the application's commands invokes a tool named **aTool**, you might want to have the **aTool** project be a subproject of the application project. When you build the application, **aTool** is built as well. After all of the code in both projects has been compiled and linked, Project Builder creates a directory (bundle) named **MyApp.app**. That directory contains **aTool** as well as the application's executable and its resources.

By creating a subproject, you are creating a project that is subordinate to the main project. In some cases, a subordinate relationship doesn't make sense. For example, you can't have one application be a subproject of another application.

If the projects' executables don't need to be tied to each other but you want to group them together as a convenience, create an Aggregate project. Then make those projects be subprojects of the aggregate. (You create an Aggregate project the same way you create any other type of project.)



An aggregate project contains only subprojects and a makefile that builds all of the subprojects at once.

The only purpose of an aggregate project is to group other projects together. The aggregate itself produces no executable or bundle. Because of this, you can have any type of project, even applications and frameworks, be a subproject of an aggregate.

For example, suppose you've created several tools to test a framework that you're working on. If you want to manage all of these projects as a single unit, you can create an aggregate project and include the tools and the framework as subprojects of the aggregate.

Tip: To have all subprojects be listed in the top level of the browser (rather than under Subprojects), set the preference on the Miscellaneous Preferences panel.

Frameworks: Easy to Use, Easy to Create

Frameworks are new in OPENSTEP 4.0. A framework is just a simpler, more convenient way to package a dynamic shared library and the resources associated with it.

In previous releases, you had to install essential library components in three different locations: the library file in **/usr/local/lib**, header files in **/LocalDeveloper/Headers**, and documentation in **/LocalLibrary/Documentation**. Now, you can store all three of these components in one directory, the framework directory, which you install in **/LocalLibrary/Frameworks**. Plus, you can package other resources (such as nib files and images) in your framework.

All OPENSTEP kits, including the Application Kit, are now distributed as frameworks. You can find these in **/NextLibrary/Frameworks**.

Another big advantage to frameworks is that Project Builder can see inside the framework package. For example, if you select **AppKit.framework** in Project Builder's browser, you can access its header files and documentation. This means you can look up how to use a method without ever leaving Project Builder!

To find out how to create your own framework, see Chapter 12 in this book.

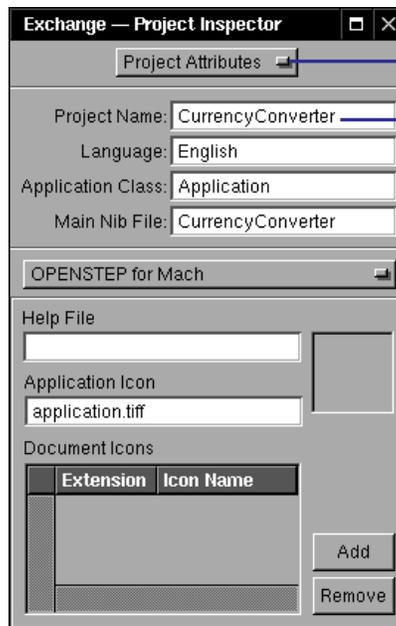
Changing a project's name

- 1 Click the Inspector button.
- 2 In the Project Inspector panel, choose Project Attributes.
- 3 Type the new name in the Project Name field.
- 4 Press Return.

By default, the project's name is the name of the directory you chose when you created the project. The same name is used for the executable or bundle that the project creates. If this isn't the name you want to use, change it in the Project Attributes inspector.



Click this button, or choose Inspector from the Tools menu.



Select Project Attributes.

Type the new name here.

The Project Attributes inspector contains information Project Builder needs to know to build the project and to maintain the project makefiles. Different types of projects have different Project Attributes inspectors, but all of them have the Project Name attribute.

For application projects, this panel contains the language used to develop the application, the class for the application object, the name of the main nib file, and the icons used by the application in addition to the project name. You can change these attributes as well.

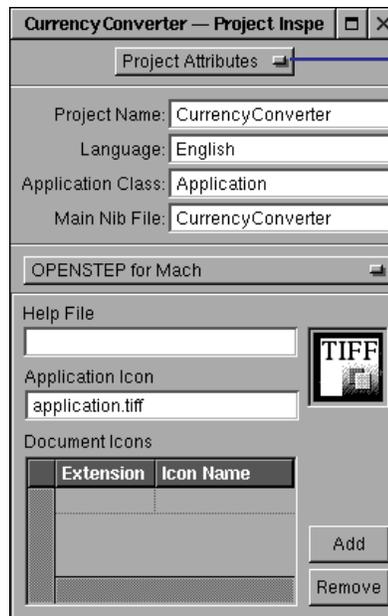
Setting the application icon

- 1 Add the icons to the project under Images.
- 2 Bring up the Inspector and choose Project Attributes.
- 3 Place the cursor in the Application Icon field.
- 4 Drag the application's icon from the main window into the icon well.

For most applications, you'll want to use a unique icon. You can create your application icon using any graphics tool, but it must be a 48 X 48 pixel TIFF image. Once you've created the icon for your application, add it to the project and to the Project Attributes inspector.



Click here, or choose Inspector from the Tools menu.



Select Project Attributes.



Legacy Projects

If Project Builder can't understand a project's Makefile, it decides that the project is a *legacy project*, a project created using a previous release. Project Builder won't create or maintain the Makefile for legacy projects; you must do that yourself.

You don't have to use the legacy project type every time you want to control the Makefile. Instead, you can make your changes in the

Makefile.preamble or **Makefile.postamble** files. The project Makefile includes **Makefile.preamble** at the beginning of the build and **Makefile.postamble** at the end of the build. Project Builder won't overwrite these files, so making changes to them is safe.

For more information about Makefiles, see Chapter 9 in this book.

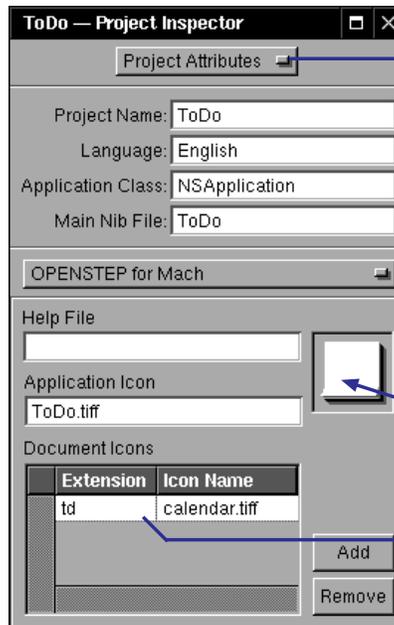
Setting document icons

- 1 **Add the icons to the project under Images.**
- 2 **Bring up the Project Inspector and choose Project Attributes.**
- 3 **Click Add Row.**
- 4 **Drag the icon from the main window into the icon well.**
- 5 **Type the file extension to be represented by this icon in the Name field of the Document Icons table.**

If an application creates documents of a unique type, you should create an icon for that document type in addition to the icon for the application itself. Like the application icon, the document icon must be a 48 X 48 pixel TIFF image. Again like the application icon, you add the document icon to the project, and then to the Project Attributes inspector.



Click here, or choose Inspector from the Tools menu.



Select Project Attributes.



Type the file extension here.

Are you creating a multi-document application? Be sure to read the section on multi-document applications in *Discovering OPENSTEP*.

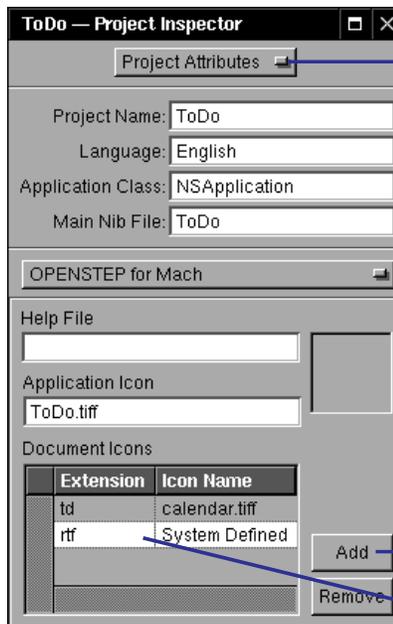
Setting system-defined document icons

- 1 **Bring up the Project Inspector and choose Project Attributes.**
- 2 **Click Add Row.**
- 3 **Type the file extension in the Name field of the Document Icons table.**

In addition to listing the file types that the application can create, the Document Icons table should list the file types that the application can read but cannot create. For example, if you are creating a word processing application that can open RTF files and translate them into its own unique file type, you should list the RTF extension in this table.



Click this button, or choose *Inspector* from the *Tools* menu.



Select *Project Attributes*.

Click here.

Type the file extension here.

2

Composing the Interface

Opening a nib file

Creating a nib file

Using palettes

Placing interface objects

Initializing text

Removing objects

Duplicating objects

Sizing interface objects

**Shrinking objects to their
minimum size**

**Making interface objects the
same size**

**Positioning and sizing
precisely**

Sizing windows and panels

**Moving objects to other
windows**

Arranging objects

Grouping objects

Creating matrices of objects

Creating menus

Layering objects

The last thing one knows in constructing a work is what to
put first.

Blaise Pascal, *Pensées*

Measurement began our might.

W. B. Yeats, “Under Ben Bulben”

Opening a nib file

- ▶ Double-click a nib file in Project Builder.

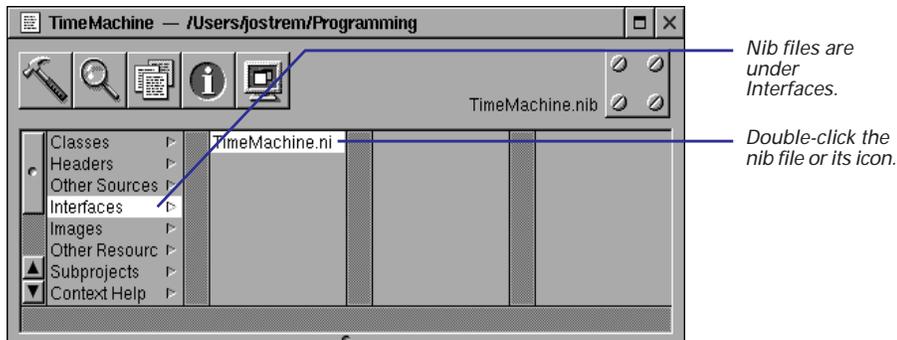
Or

- ▶ Double-click a nib file in the Workspace's File Viewer.

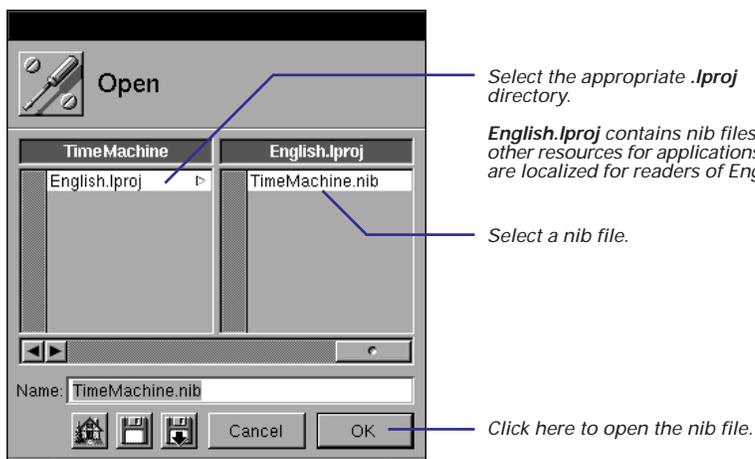
Or

- ▶ In Interface Builder, choose Document ▶ Open and select a nib file in the Open panel.

You'll usually open a nib file from Project Builder, since that is the central tool for application development. When you create an application project, Project Builder creates a nib file that has the same name as the project and, like all nib files, ends with the extension `.nib`. Opening a nib file switches control to the Interface Builder application, which you use to create the interface.



You can also open nib files using the standard methods of opening files, such as using the Open panel.



Nib files (files that have a `.nib` extension) are file packages that archive the class definitions, objects, and the connections between objects when you create an interface in Interface Builder. See “What’s in a Nib File” in this chapter for some conceptual background.

When Interface Builder Starts Up

When you open a nib file, Interface Builder displays several windows and panels on your screen.

The palette window Holds all currently loaded palettes of objects. You select a palette by clicking its icon. Then you drag objects from the palette to the appropriate surface.

The interface window This window or panel displays the actual interface that you're working on. If this is the first time you've opened a main nib file in Project Builder, an empty window is displayed.

The nib file window This window contains multiple views that display the contents of the nib file. These views show archived objects, the connections among objects, the current class hierarchy (including any custom classes that you may have

created), and the images and sounds stored in the nib file. Click a tab to switch the view.

The Inspector panel This multiform panel displays the attributes, connections, and size of a selected object. It also presents the object's resizing characteristics, its associated help, and which class it inherits from.

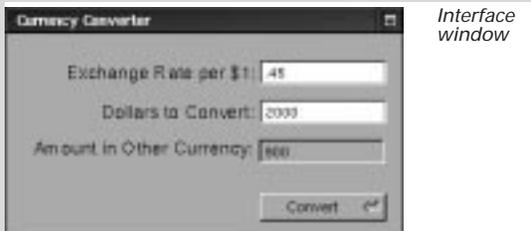
You can control whether the palette window and the Inspector panel appear when Interface Builder starts up by checking the appropriate boxes in the Preferences panel.



Palette window



Nib file window



Interface window



Inspector panel

Creating a nib file

- Choose one of the commands in the **New Modules** submenu of the **Document** menu.

Or

- Choose **New Empty** from the **New Modules** submenu, then drag a window or panel from the **Windows** palette.

Or

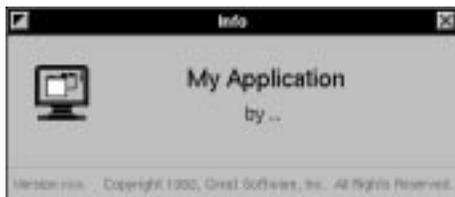
- Choose **New Application** from the **Document** menu.

You can have auxiliary nib files, such as an Info panel, that you load into your program only when you need to. The programming technique of loading nib files on demand (lazy instantiation) is described in Chapter 11, “Dynamic Loading.”

One reason to use **New Application** is to create a version of your interface for Microsoft Windows. For more information see Chapter 18.

Sometimes you need to create nib files directly in Interface Builder, typically when you want to add additional windows and panels to your application.

Most commands of the **New Module** submenu create nib files that contain a special kind of ready-made panel; your application can later load these nib files when it needs them. For example, if you choose **New Info Panel**, you’ll get the following template panel:

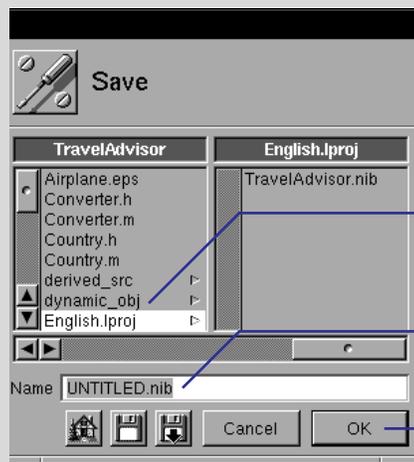


The **New Empty** command just creates an empty nib file; you must create the windows and panels for it by dragging these objects from the **Windows** palette. The **Document** ► **New Application** command can create your application’s main nib file (a nib file with the owner of `NSApplication`) if that hasn’t already been done for you in **Project Builder**.

Saving the Nib File

An **UNTITLED** nib file window appears for each newly created nib file. After you make changes to an interface, remember to save the nib file. Choose **Save** from the **Document**

menu and specify a path and file name in the **Save** panel. Interface Builder may ask if you want to insert the file into your project; confirm by clicking **Yes**.



Select a localized resource subdirectory (`.lproj` extension) of the project directory.

Type the name of the nib file. The `.nib` extension is automatically added if you leave it off.

Click here to save the file.

What's in a Nib File

Every application has at least one nib file (actually a file package). The main nib file contains the main menu and often a window and other objects. An application can have other nib files as well. Each nib file contains:

Archived Objects Encoded information on OPENSTEP objects, including their size, location, and position in the object hierarchy (for view objects, determined by superview/subview relationship). At the top of the hierarchy of archived objects is the File's Owner object, a proxy object that points to the actual object that owns the nib file. (For a description of File's Owner, see the concept summary on File's Owner, First Responder, and Font Manager in Chapter 4.)

Sounds and Images Any sound or image files (TIFF or EPS) that you drag and drop over the nib file window.

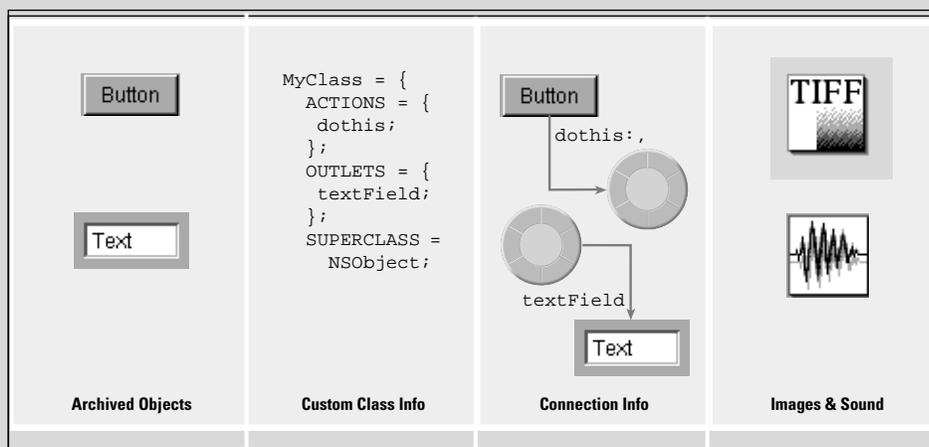
Class References Interface Builder can store the details of OPENSTEP objects and objects that you palettize (static palettes), but it does not know how to archive instances of your custom classes since it doesn't have access to the code. For these classes, Interface Builder stores a proxy object to which it attaches class information.

Connection Information Information about how objects within the object hierarchy are interconnected. Connector objects special to Interface Builder store this information. When you save the document, connector objects are archived in the nib file along with the objects they interconnect.

When You Load a Nib File

In your code, you can load a nib file by sending the `NSBundle` class the message `loadNibNamed:owner:orLoadNibFile:externalNameTable:withZone:.` When you do this, the following things happen:

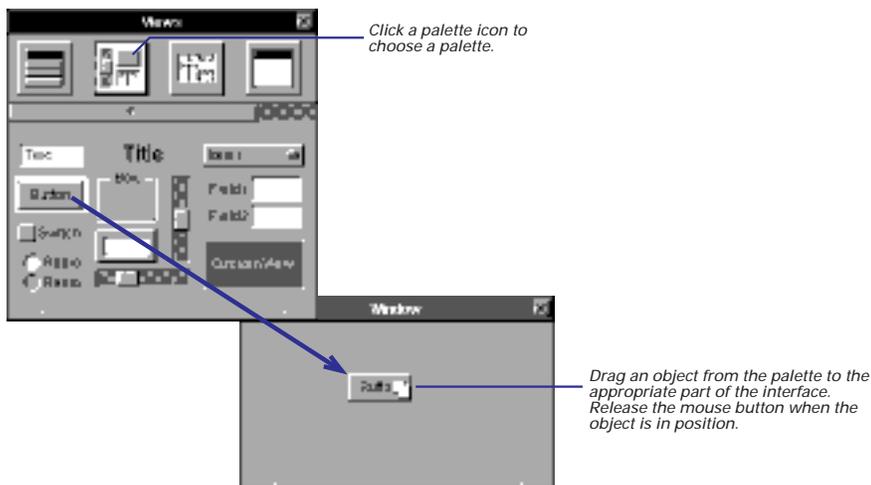
- The run-time system unarchives the objects from the object hierarchy, allocating memory for each object and sending it an `initWithCoder:` message.
- It unarchives each proxy object and queries it to determine the identity of the class that the proxy represents. Then it creates an instance of this custom class (`alloc` and `init`) and frees the proxy.
- The system unarchives the connector objects and allows them to establish connections, including connections to File's Owner.
- As the final step, the run-time system sends `awakeFromNib` to all objects that were derived from information in the nib file, signalling that the loading process is complete.



Using palettes

- 1 Choose the palette you want.
- 2 Drag an object from the palette to the appropriate “surface.”
- 3 Release the mouse button.

The palette window of Interface Builder displays all currently loaded palettes. Each palette is represented in the window by an icon.



“Where Palette Objects Go” in this chapter illustrates the proper “surfaces” for interface objects.

See the *Enterprise Objects Framework Developer's Guide* for more information on Enterprise Objects Framework palettes.

See “Managing palettes” in Chapter 5 for instructions on loading and installing palettes.

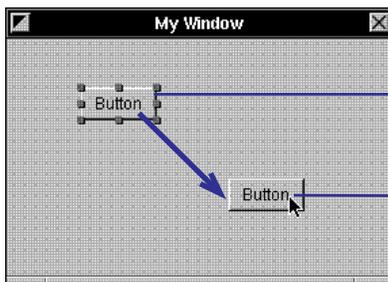
The palettes for the Application Kit are loaded by default. These palettes provide windows, panels, browsers, scroll views, buttons, text fields, and a number of other interface objects. You can also load palettes for other frameworks, such as Enterprise Objects , and you can load your own custom palettes.

Note: Where you “drop” a window or panel is important, because it sets its initial position on the screen—the location where the window appears when the application starts up or when this particular nib file is loaded.

Placing interface objects

- 1 **Select the object you want to move.**
- 2 **Drag the object to the new location in the window or panel.**

To move an object around the “surface” of a window or panel, select the object and drag it with the mouse. The currently selected object has resize handles—small, gray rectangles—around its perimeter.



Click the object so the resizing handles appear.

When you drag the object, you can move it anywhere inside the window or panel.

You can adjust the size and location of objects precisely by specifying their origins, width, and height in the Size display of the object’s Inspector.

See “Positioning and sizing precisely” in this chapter for details.

When you move an object, make sure that the mouse pointer is inside the object and not on a resize handle.

For greater precision, select an object and press the arrow keys; this moves the object an incremental distance in the required direction. If the alignment grid is off, this distance is one pixel; if it is on, the distance is equal to the size of the grid.

Selecting Multiple Objects

You can select multiple objects and then move, copy, or do other things with them as a group. There are two ways to select more than one object:

- Hold down the Shift key while you click objects in succession.
- Click in an empty area, then draw a “rubberbanding” rectangle around all objects you want selected.

After making the selection, press (**don’t** momentarily click) the mouse pointer on one of the objects and drag the group to the new location. (Or do another suitable operation, such as copy and paste.)

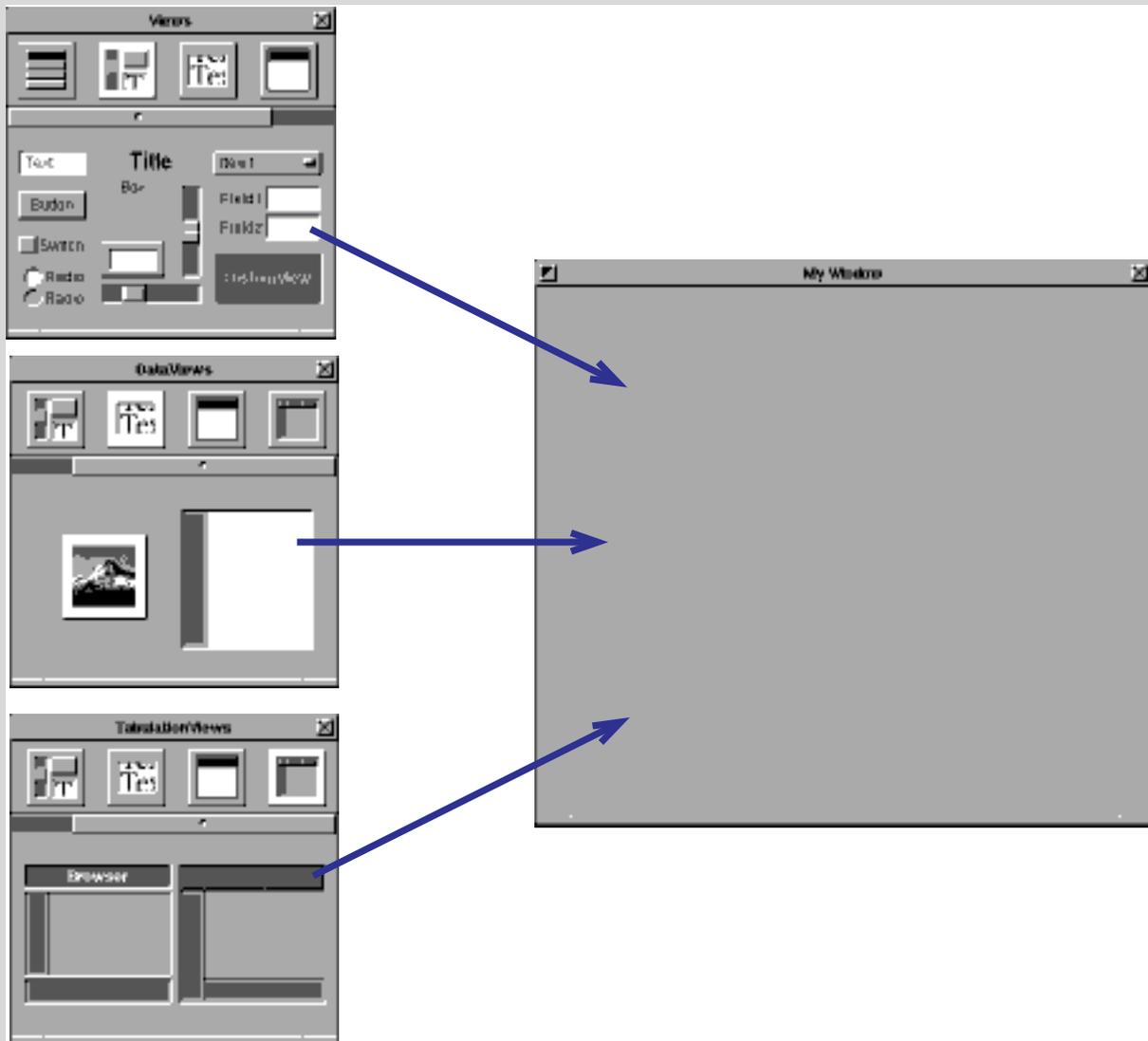
To deselect an object in a grouped selection, hold down the Shift key and click that object.

You cannot do sizing operations on multiple selected objects.

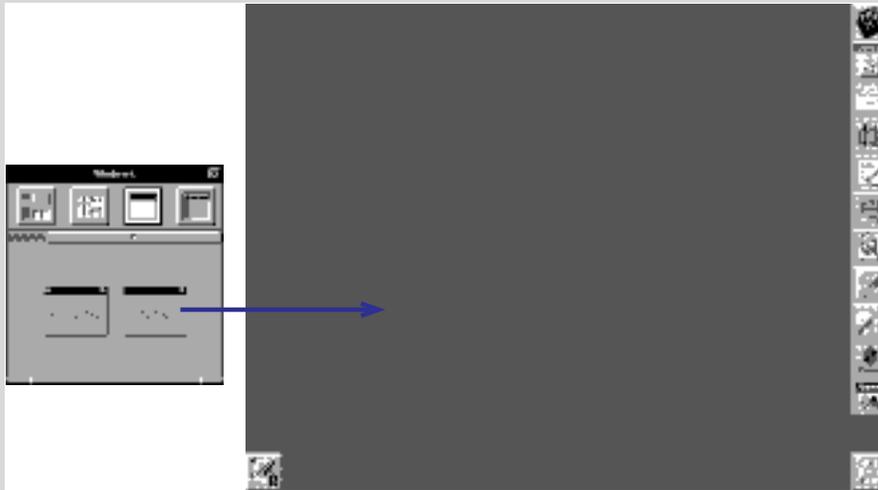
To select all objects in a window or panel, first select the window or panel, and then choose Select All from the Edit menu. You can also use this command to select all items in the Instances or Classes display of the nib file window. The key equivalent for Select All is Command-a.

Where Palette Objects Go

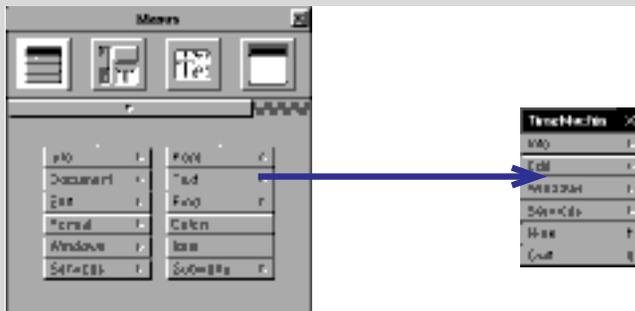
You put items from the Views, TabulationViews, and DataViews palettes anywhere within the bounds of a window or panel. These items include buttons, labels, fields, boxes, text fields, scroll views, browsers, and custom views.



You can put windows and panels anywhere in the work space.
Nothing contains them except the screen.

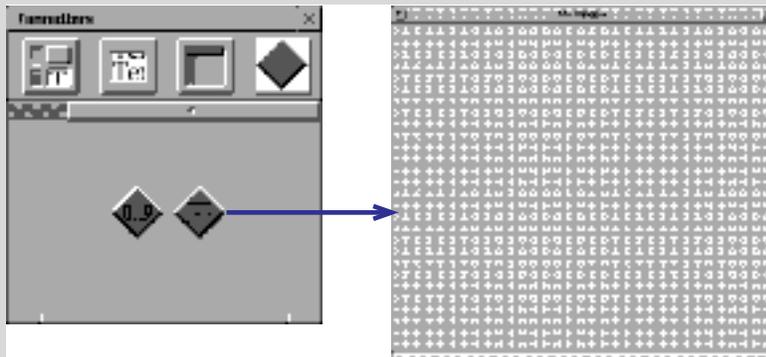


You drag a menu item from the Menus palette and drop it in the application's menu. When you release the mouse button, Interface Builder inserts the item between the two menu commands underneath it.

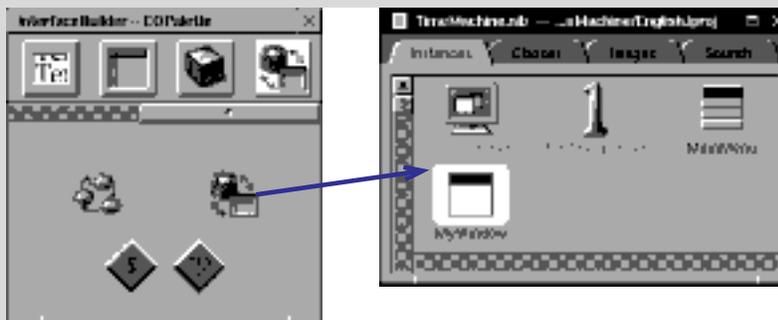


Where Palette Objects Go (continued)

You drag a formatter from the Formatters palette and drop it on a text field or a cell in a tableview. That formatter will control the formatting for all of the cells in that column.



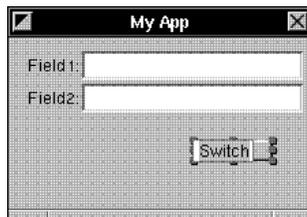
Some palettes, like the one for the Enterprise Objects Framework, carry objects that do not appear on an interface. These usually are controller objects that perform management or computational functions. Drop these objects anywhere on the Instances display of the nib file window.



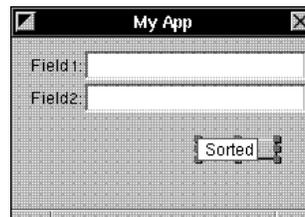
Initializing text

- 1 **Select the object.**
- 2 **Double-click the text inside the interface object.**
- 3 **Edit the text.**
- 4 **Deselect the text by clicking outside of the object.**

Many of the palette objects include text as a component. Buttons of all sorts usually have titles, boxes usually name the elements they group, and so on. Interface Builder initially sets the text in most of these objects to the name of the object itself (such as “Button” or “Text”). After you drag the palette object onto a window or panel, you will probably want to delete these text strings or rename them to something meaningful. This text is what is initially displayed when your application loads the nib file; the text can change later if one of the objects in your application requests it.



Double-click the line to select it.



Type the new text. When finished, click outside the object to set the text.

When text is selected, you can move the cursor among the characters by pressing the left and right arrow keys. You can delete characters by pressing the Delete key.

Matrices—compound objects, such as radio buttons and form fields—need a slightly different procedure for selecting text for initialization: You must double-click the embedded text item twice, the first time to select the embedded object, and the second time to select the text inside the object.

“Creating matrices of objects” in this chapter describes how to create these compound objects. Also see “Compound Objects” in Chapter 3 for a conceptual summary of matrices and other compound objects.



In matrices, double-click the text to select the embedded object.

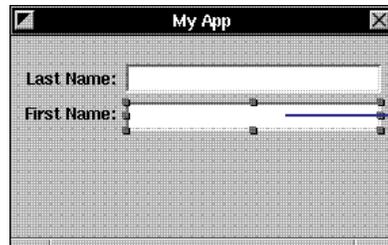


Double-click again to select the text.

Removing objects

- 1 **Select one or more objects.**
- 2 **Choose Cut from the Edit menu.**

To delete objects from an interface, select the objects and choose the Cut command. You can also use the Delete command, but the differences are significant. Cut saves the selected objects to the pasteboard, so you can retrieve the objects with the Paste command (Command-v). The Delete command permanently deletes the selected items.



Select one or more objects.

Choose Cut from the Edit menu or press Command-x.

The Coordinate System in Interface Builder

The Size display of an object's Inspector panel shows that object's precise location and dimensions. The **x** and **y** fields hold the origin point (horizontal and vertical) for the object within the drawing context of its enclosing window, panel, or superview. The **w** and **h** fields hold the width and height. All values are in pixels.

Within a window or panel, the lower left corner is origin 0,0. This is the point of reference for objects within that window or panel.

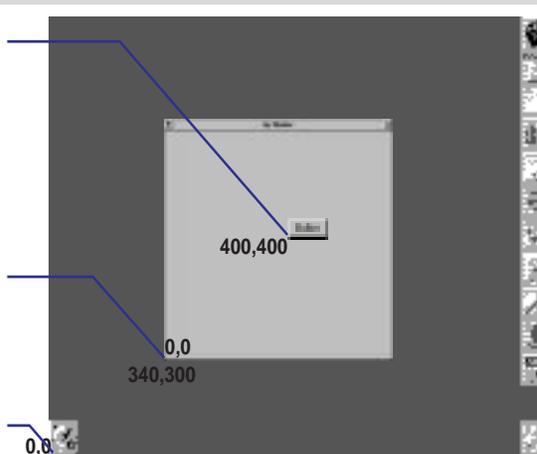
Therefore, when you move or size objects downward or to the left, the values in the Size display are decreased.

The point of reference for a window or panel (or origin 0,0) is the lower-left corner of the screen. This means that the same relationship applies: if you decrease its **x** value in the Size display, it moves to the left; if you decrease its **y** value, it moves toward the bottom of the screen.

The origins and dimensions of view objects are based on the origin point of the window or panel that contains them. (The origin does not include the resize bar.)

In this example the window's placement is 340,300 relative to the screen origin. This same point on the screen is the origin for views in the window.

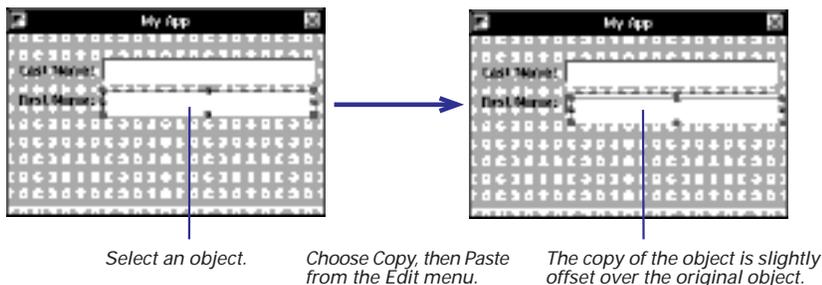
The origins and dimensions of windows and panels are based on the screen's origin point.



Duplicating objects

- 1 **Select an object.**
- 2 **Choose Copy from the Edit menu.**
- 3 **Choose Paste from the Edit menu.**
- 4 **Position the new object.**

You can duplicate an object just as you would with geometric shapes in a drawing application. The copied object has the dimensions and most other attributes of the original object.



In addition to the objects that appear on the interface, you can copy your custom non-UI objects—represented as cubes in the icon mode of the nib file window Instances display—as well as your windows and panels. Just click to select them and then copy and paste.

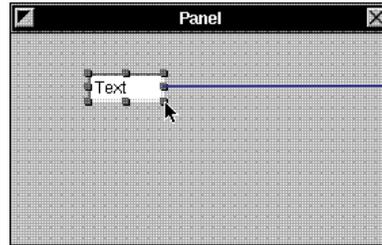
Tip: Instead of choosing Copy and Paste from the Edit menu, you can press Command-c (Copy) and Command-v (Paste).

You can also duplicate groups of selected objects by copying them and then pasting them. See “Selecting Multiple Objects” in this chapter for details on making multiple selections of objects.

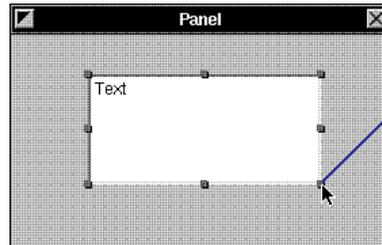
Sizing interface objects

- 1 **Select an object.**
- 2 **Drag a resize handle in the desired direction.**

Interface objects in Interface Builder resize to any practical dimension. You can, for instance, increase the size of a button so it fills a window. Most interface objects, however, do not resize below a certain minimum size of usefulness.



Click an object to select it, then drag a resize handle in the direction of growth.



Release the mouse when the object reaches the desired size.

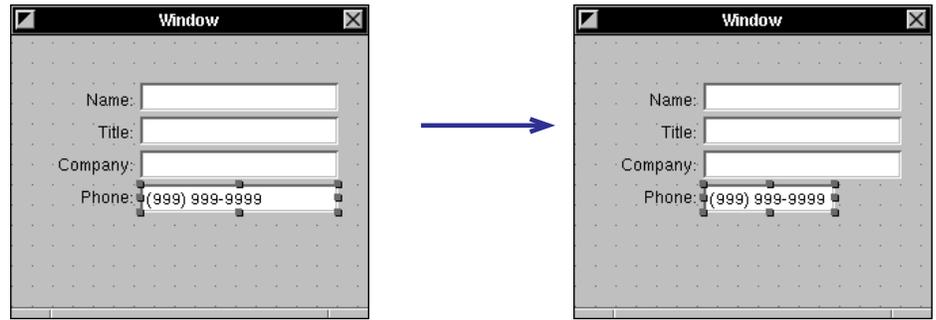
To affect just one dimension of the object, drag a top, bottom or side handle. To adjust both dimensions simultaneously, drag one of the corner handles. To size both dimensions proportionally, hold down the Shift key while you drag a corner resize handle.

You can adjust the size and location of objects precisely by specifying their origins, width, and height in the Size display of the object's Inspector panel. See "Positioning and sizing precisely" for details.

Shrinking objects to their minimum size

- 1 **Select one or more objects.**
- 2 **Choose Format ►
Size ► Size to Fit.**

To conserve screen real estate, or to enhance the appearance of your interface, you might want to have view objects just large enough for any text they contain. You can do this with the Size to Fit command.



Select an object that you want to shrink.

*Choose Format ►
Size ► Size to Fit*

*The copy of the object is slightly
offset over the original object.*

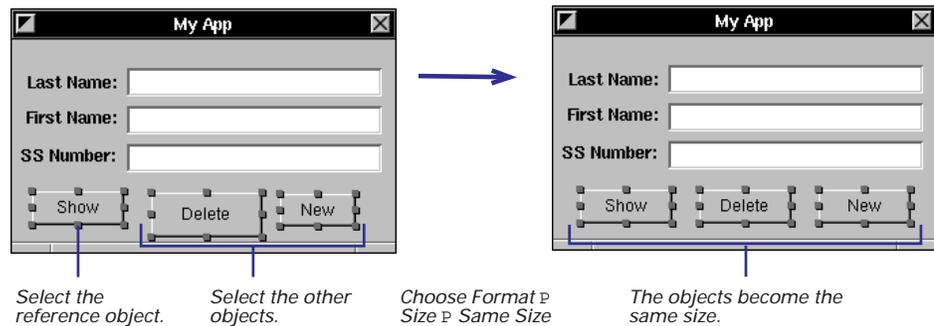
The Size to Fit command has no affect on matrices, custom views, and some other objects. If you delete the text from a button, text field, or other object that holds text, and then apply the Size to Fit command to it, that object shrinks to its minimum (and probably unusable) size.

Making interface objects the same size

- 1 **Select the reference object.**
- 2 **Add to the same selection the objects that you want resized.**
- 3 **Choose Format ► Size ► Same Size.**

To lend a look of consistency to your interface, you often want to make similar objects have the same size. Buttons across the bottom of an attention panel, for instance, should be the same exact size. Interface Builder gives you an easy way to do this, allowing you to resize selected objects to a *reference* object. You designate the reference object differently, depending on your method of selection:

- If you press the Shift key while clicking objects in succession, the first object clicked is the reference object.
- If you draw a selection rectangle around a group of objects, selecting the objects simultaneously, the topmost object in the selection (often the most recently added object) is the reference object.

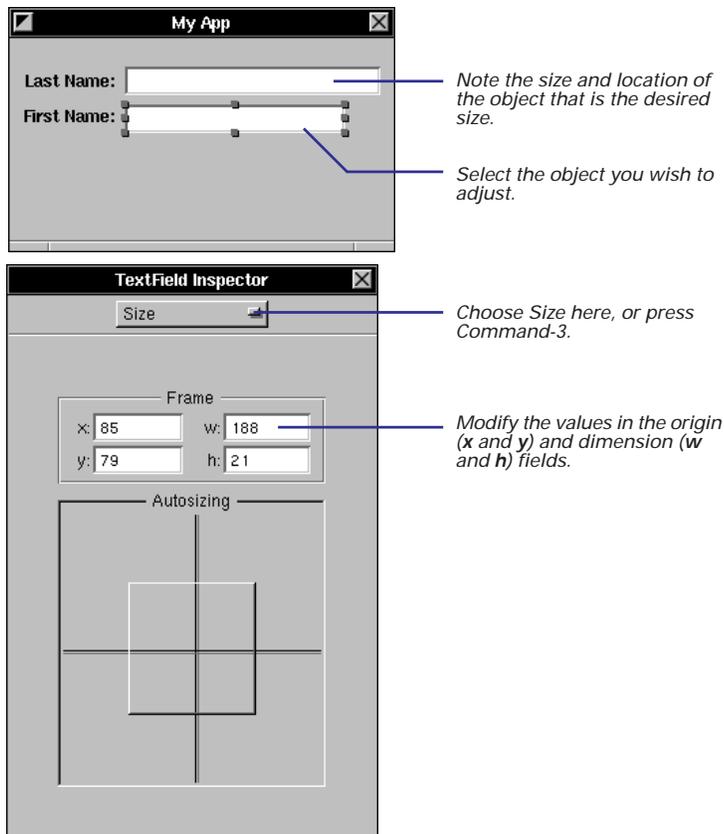


Tip: In most situations, you should select multiple objects by Shift-clicking them because this method gives you more control (you know which object will be the reference object).

Positioning and sizing precisely

- 1 **Select an object.**
- 2 **Choose Tools ► Inspector to bring up the Inspector panel.**
- 3 **Choose Size from the Inspector pop-up list.**
- 4 **Modify the object's origin point or its dimensions.**

You can move and resize objects in your interface with numerical exactness using the Inspectors for those objects. You'll occasionally find need for such exactness, such as when you want to size an image view to the same dimensions as the image that it will display. More frequently, you'll use this method to align objects or make sure they're the same size.



See “Automatically resizing objects” in Chapter 3 for information on the Autosizing area.

When you press Return in an origin or dimension field, the object moves to the new position or expands or contracts to the new size.

Tip: You can also move selected objects incrementally—and precisely—by pressing the arrow key that points in the required direction. Each incremental “nudge” moves the object the distance of the grid or, if the grid is turned off, one pixel.

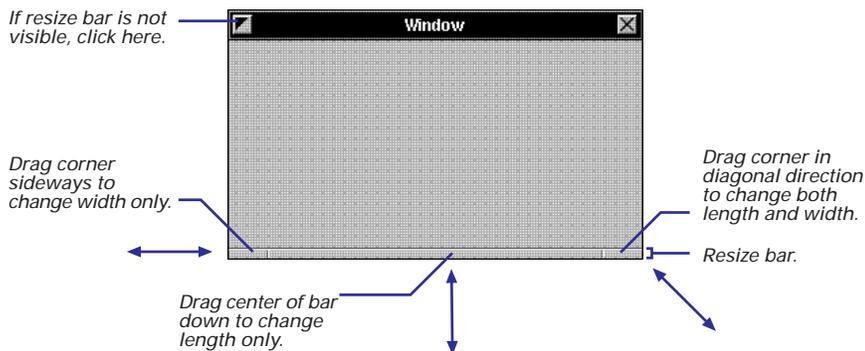
Sizing windows and panels

- ▶ Drag the resize bar in the direction you want the window to grow.

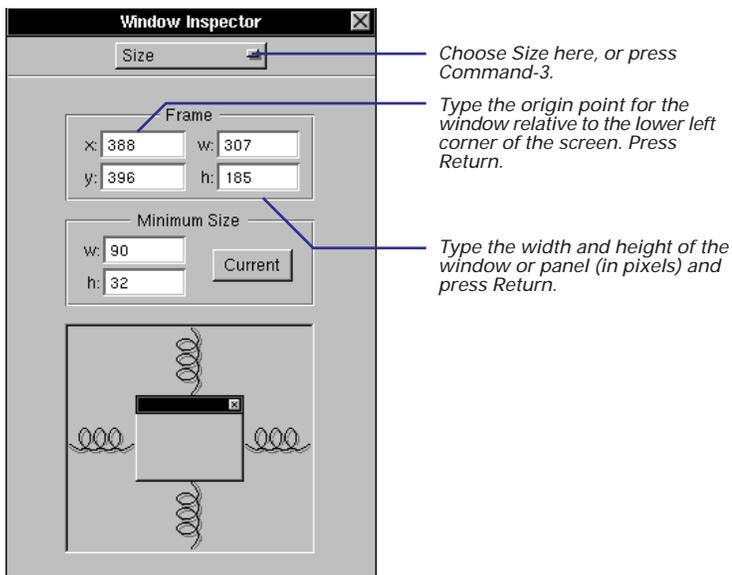
Or

- 1 Select the window by clicking its titlebar.
- 2 Choose Tools ▶ Inspector to bring up the inspector window.
- 3 Enter the dimensions in the Size display of the Inspector panel.

After you drag a window or panel from the Windows palette and drop it on the screen, you'll probably want to resize the object to a suitable dimension. To resize a window, drag the resize bar in the required direction.



To resize windows or panels with greater precision, enter the exact dimensions in the Size display of the Inspector panel.



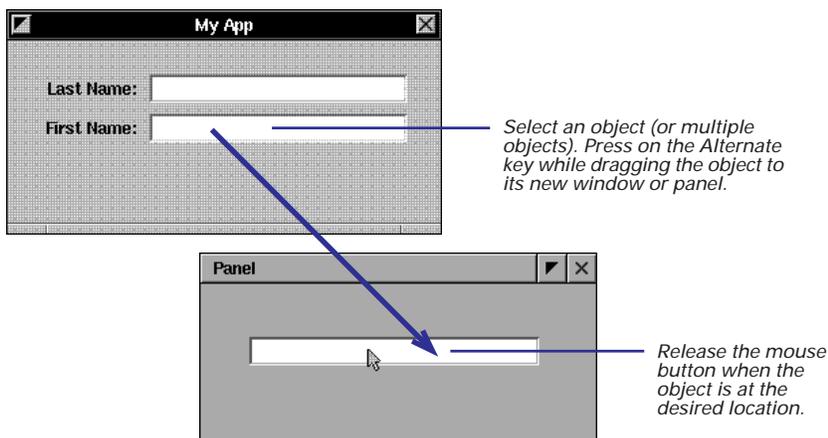
You can also use the Inspector panel to size interface objects with numerical exactness. See "Positioning and sizing precisely" for more information.

Also see "The Coordinate System in Interface Builder" for some conceptual background.

Moving objects to other windows

- 1 **Select one or more objects.**
- 2 **Alternate-drag the objects to the other window or panel.**

To move objects from one window or panel to another within the same nib file, hold down the Alternate key and drag them between windows. This action moves the objects if the windows are in the same nib file and copies them if they are in different nib files.



If the windows are in the same nib file and you want to copy rather than move the selected objects, you have two alternatives:

- Copy the objects using the Copy command; click the other window or panel to activate it, and use the Paste command to copy the objects from the pasteboard.
- Copy and paste the objects in one window, then Alternate-drag the duplicated objects to their new window or panel.

Copying Objects to Other Interfaces

To copy objects to different nib files, simply select a group of objects in one nib file and Alternate-drag those objects to the appropriate “surface” of the other nib file.

You can copy entire windows or panels as well as custom, non-UI objects between interfaces.

The surface you drop objects onto must be compatible:

- Non-UI objects must be dropped over the nib file window.
- View objects are dropped over a window or panel or over the nib file window.
- Windows and panels must be dropped over the nib file window.

The basic technique of Alternate-drag also copies the connections among selected objects. See “Copying interconnected objects” in Chapter 4 for details.

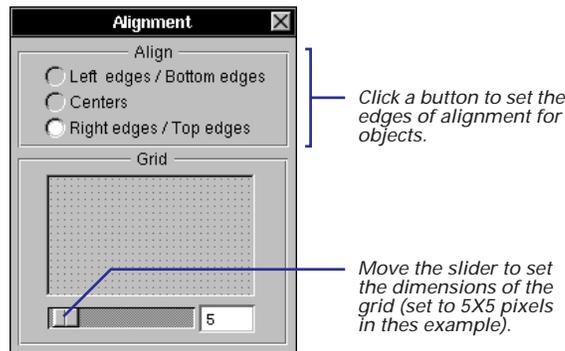
Arranging objects

- 1 **Choose Format ► Align ► Alignment to bring up the Alignment panel.**
- 2 **Set the characteristics of the grid in the Alignment panel.**
- 3 **Choose Format ► Align ► Turn Grid On to turn on the grid.**
- 4 **Align objects with the grid.**

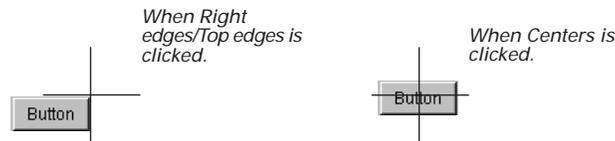
When you compose your interface, you usually want to arrange the objects in that interface in some appealingly regular way. You want buttons, for instance, to be aligned on the same invisible horizontal or vertical line. Or you want the distance between text fields in a form application to be exactly the same. Interface Builder gives you a set of tools for arranging objects.

Every window or panel has a grid associated with it. You may turn this grid off and on. When it is on and you move an object, an edge of that object “snaps,” like a nail to a magnet, to the adjacent intersecting lines of the grid.

You set the dimensions of this grid and the edges of alignment in Interface Builder’s Alignment panel.

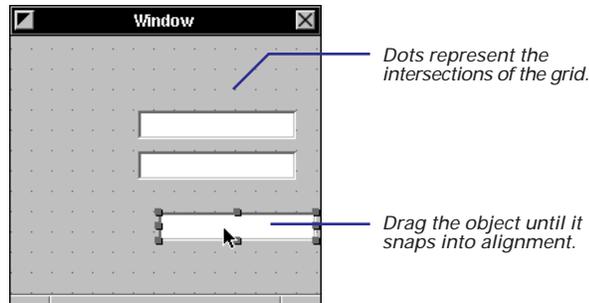


The buttons in the Align section of the Alignment panel determine what point or edge of interface objects snaps to the grid.



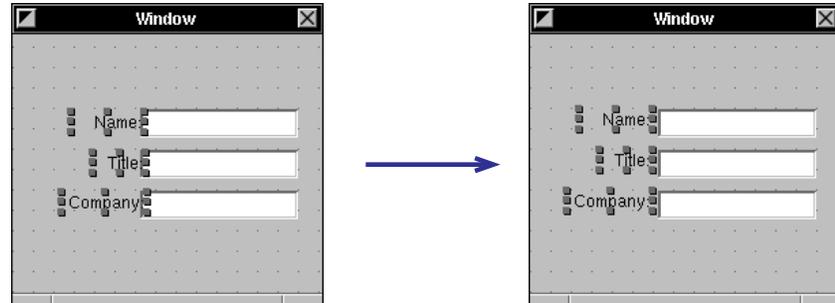
Once you have your grid set up, make sure the grid is turned on: Choose Format ► Align ► Turn Grid On. If you also want the grid visible, choose Show Grid from the same menu.

Now align the objects, either individually or as a group, using the grid.



There are other ways to align objects that don't require using the mouse. With the grid turned off, you can drag view objects from a palette and visually align them as precisely as possible. Then set the grid spacing, turn the grid on, and choose the Align to Grid command.

Once the grid is set and on, align the objects, either individually or as a group.



With grid off, place objects for alignment and then select them.

Choose Format ▾ Align ▾ Align to Grid

Objects become aligned, in this case moving on the grid to their left.

With the Align to Grid command, the direction of alignment is toward the origin point of the window or panel (in other words, toward the lower-left corner). You should be aware of this when placing objects for later alignment.

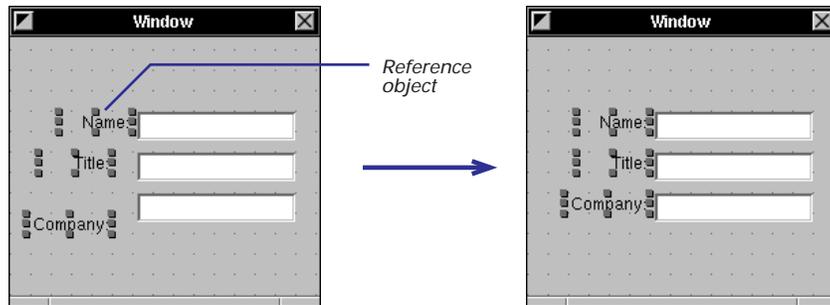
Tip: You can align selected objects to a grid, singly or as a group, by pressing the arrow keys in the direction of alignment. When the grid is turned on, the unit of increment changes from one pixel to whatever the grid spacing is.

Making Columns and Rows of Objects

It is more efficient to align groups of objects than to align single objects successively. With the Make Column and Make Row commands, Interface Builder aligns groups of selected objects to a *reference* object. You designate the reference object by the way you select multiple objects.:

- If you press the Shift key while clicking objects in succession, the first object clicked is the reference object.
- If you draw a selection rectangle around a group of objects, and so select objects simultaneously, the topmost object in the selection (often the most recently added object) is the reference object.

For most purposes, Shift-clicking objects is the preferred method because it permits more control.



Click the reference object first, then Shift-click the remaining objects to select them.

Choose Format ▾ Align
▾ Make Column

The objects become vertically aligned to the reference object.

NeXT's Basic UI Design Philosophy

Composing a user interface involves much more than techniques for placing, sizing, and arranging objects on a window. When you put your application's UI together in Interface Builder, keep in mind the following principles that NeXT has developed, through trial and test, to guide interface designers.

Make It Consistent

When all applications share the same basic interface, each application benefits. Consistency makes each application easier to learn, and so increases the likelihood of acceptance and use. Just as with so many natural "interfaces" in life, conventions count for a great deal. Although different applications are designed to accomplish different tasks, they all share, to some degree, a set of common operations such as selecting, editing, scrolling, and setting options. Reliable conventions are possible only when these operations are carried out the same way for all applications.

Make it Feel Natural

Try to make the screen a visual metaphor for the real world, so that the objects in it reflect the way the represented things actually behave. That's what an "intuitive" interface is – it behaves as we expect based on our experience with objects in the real world.

Modeled objects don't have to mimic every detail of their real counterparts, but they should behave in similar ways. For example, objects in the real world stay where we put them; they don't disappear and reappear again, unless someone causes them to do so. Users should immediately recognize the objects in your interface and should use them for the sorts of operations that people typically use their real counterparts for.

Put the User in Charge

Users should have the widest freedom of action. If an action makes sense, your application should allow it. In particular, avoid setting up arbitrary modes, periods during which only certain actions are permitted. On occasion, however, modes are a reasonable way of solving a problem, particularly in these forms:

- attention panels
- modal tools
- "spring-loaded" mode (while mouse or key down)

But these modes should be freely chosen, provide an easy way out, be visually apparent, and keep the user in control.

At the same time, you should try to anticipate what users will do and ease their way, reducing the actions they must perform. Give them freedom, but still act on their behalf without waiting for their instructions. These helping actions should be simple and convenient, like, in the Open panel, preselecting a directory that is probably in the path of the final selection.

Focus on the Mouse

The mouse is the most appropriate instrument for a graphical interface. The keyboard is principally used for entering text, but the mouse is the instrument by which users manipulate the objects of your interface. Your user interface should support three paradigms of mouse action:

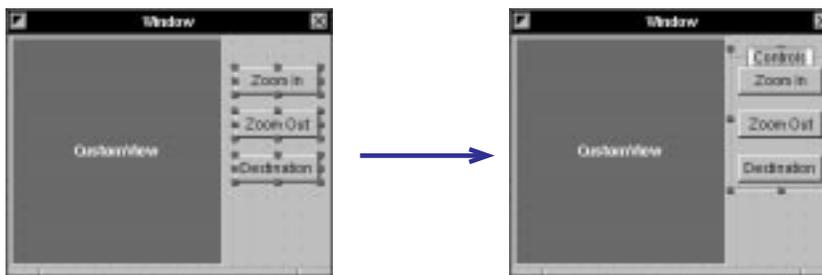
- Direct manipulation
- Targeted action
- Modal tool

Grouping objects

- 1 **Select the objects you want grouped.**
- 2 **Choose Format ► Group ► Group in Box.**

When you group objects, Interface Builder draws a box around them. You select, move, copy, cut, and paste the objects within the box as a group. Interface Builder gives you two ways to group objects.

In the first method, you select the objects of the group and choose a menu command. The box has a title, initially “Title.” To change this, double-click the title to select it (as in the example above, on the right). Then type the new name for the grouped objects.



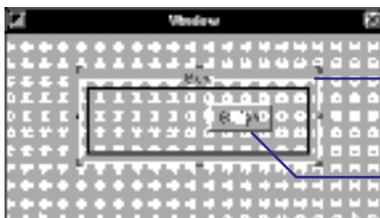
The objects of your group should be adjacent.

Choose Format ► Group ► Group in Box

A box encloses the objects.

To ungroup the objects, making each object individually selectable again, select the group and choose Format ► Group ► Ungroup.

In the second method, you use the box object in the Views palette. First, drag a box onto a window or panel. Then add its contents:



Double-click the box; its resize handles become truncated.

Drag an object from the palette and position it within the box. The black line inside the boundaries of the box indicates that grouping is taking place.

See Chapter 3 “Setting an Object’s Attributes” for other things you can do with box objects.

See the specifications of the NSScrollView and NSSplitView classes in the *Application Kit Reference*.

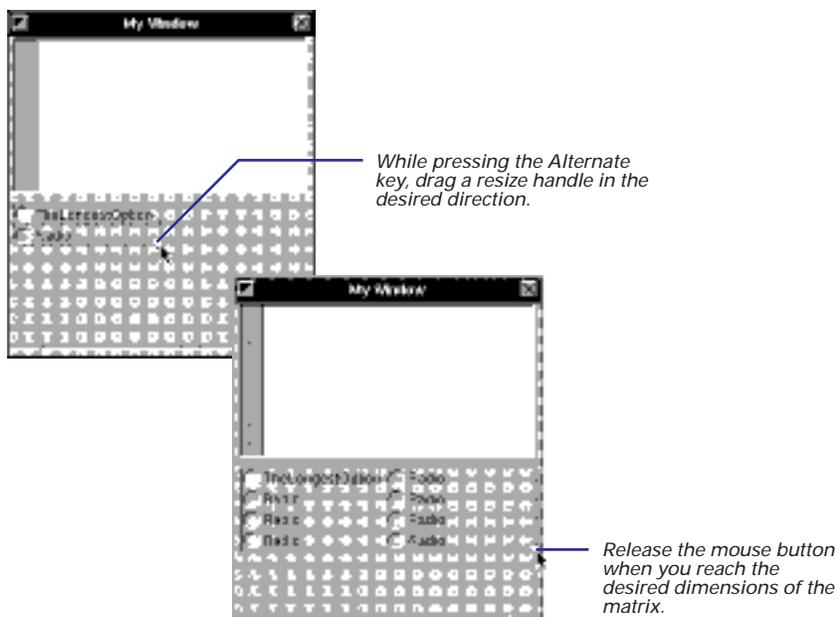
The Group submenu has two other interesting commands. With Group in Scroll View, you can bind an NSText object (or your own custom view object) to horizontal and vertical scrollbars. With Group in Split View, you can group two related views (often a custom view and a browser object) in a split view, which has a sizing bar between the views.

Creating matrices of objects

- 1 Drag a suitable object from the Views or TabulationViews palette.
- 2 Alternate-drag a resize handle of the object.

You can easily transform certain objects in the standard Interface Builder palettes into matrices of those objects. A matrix (defined by class `NSMatrix`) imposes a regular size and intervening distance on a set of identical objects. Matrices afford an easy way to compose forms, arrays of buttons and sliders, and multiple-column browsers. To create a matrix, drag one of these objects to a window or panel and size it to the maximum dimension you anticipate for a cell in the matrix:

- text field
- button
- switch button
- radio button
- form field
- slider (vertical or horizontal)

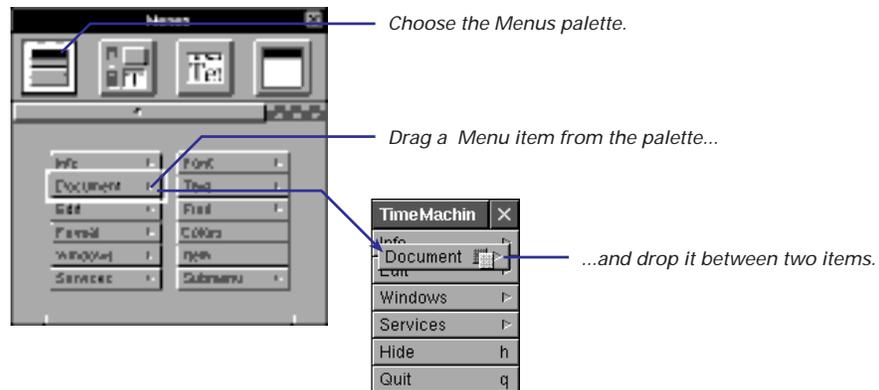


Tip: To make a browser with more than one column, drag a browser object from the TabulationViews palette onto your interface; then Alternate-drag the right resize handle until the desired number of columns appear.

Creating menus

- 1 Drag a menu item from the Menu palette.
- 2 Drop it between two menu items in your application's menu.

Menus are just as important as windows and panels for an interface. Menu commands initiate most of the standard functions of an application, such as printing, opening files, or cutting and pasting text. That's why Interface Builder's Menu palette holds a number of ready-made submenus and menu items.



You can make menu items active or inactive by default. Select the item and set the Disabled button in the Attributes display of the cell's Inspector. See Chapter 3, “Setting an Object’s Attributes” for more information on using the Inspector panel.

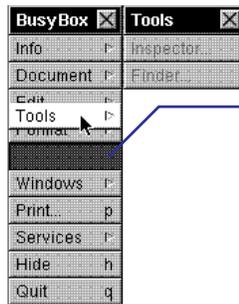
See Chapter 4, “Making and Managing Connections” to learn what you must do to connect menu items with the objects that are to handle menu commands.

Click several menu items in your application’s main menu and note how some cells in the submenus are dimmed. Dimmed cells indicate that, as the default, the command is inactive until some condition occurs in your code that causes your application to activate the command.

You delete a menu item just as you do with any other object in Interface Builder: select it, then choose the Cut command from the Edit menu (Command-x) or press the Delete key. Also, as with other Interface Builder objects that display text, you can easily change the titles of menu items:

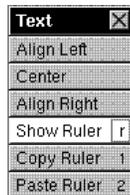
- Double-click the text to select it
- Type the new title or edit the old one
- Click outside the cell to set the new title or press Enter

You can also do two special tasks with menu items: re-sequencing and assigning Command keys. By re-sequencing, you change the order in which items are listed in a menu. By assigning a Command key to an item, you give the user of your application a command key equivalent—a shortcut way to invoke the command (as Command-x is a shortcut for invoking the Edit menu’s Cut command).



Resequencing

Drag the menu item from its old location (shown by a black rectangle) and drop it between two menu items (its new location).



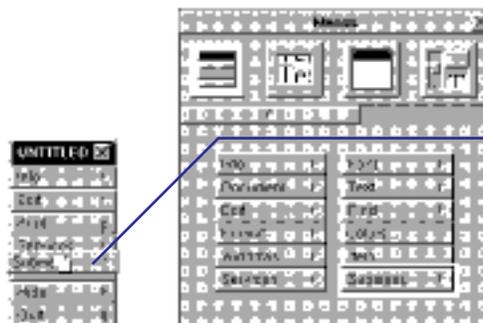
Command-key Assignment

Double-click the menu item near its right edge. In the square, type the letter for the Command key.

Make sure that the Command-key letter is not assigned to some other menu command in your application.

Custom menus

In addition to the standard menu commands and submenus, Interface Builder makes it easy for you to compose your own custom menus. Use the Submenu cell in the Menu palette to create custom submenus and use the Item cell for custom menu commands. The Print command is frequently added as a custom cell.



Drag the Submenu item from the Menu palette and drop it between two menu items in the application's main menu.

Change the title of Submenu and click the cell to expand it. Then add Items from the Menu palette to the new submenu and change their titles.

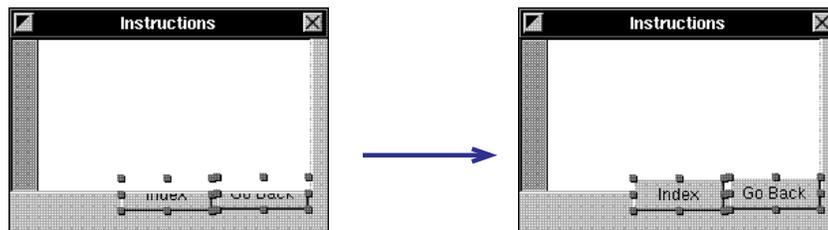
Layering objects

- 1 Select an object.
- 2 Choose **Format ► Bring to Front** or **Format ► Send to Back**.

Every object on a window or panel in Interface Builder is on its own layer. That's why when you move one object over another object, the first object moves in front of the second or moves behind it. The most recently added object is generally on the topmost layer.

Occasionally, you need to alter the layering order to make an object visible or to have it appear behind other objects. To do this, apply the **Bring to Front** command or the **Send to Back** command (on the **Format** menu) to selected objects.

These commands are only useful while you are editing the interface; the final version of your interface must not have overlapping views.



The buttons are behind the ScrollView. To fix this, select the buttons.

*Choose **Format ► Bring to Front***

The buttons appear in front of the ScrollView.

3

Setting an Object's Attributes

Examining an object's attributes

Customizing windows and panels

Setting button attributes

Associating images or sounds with buttons

Managing images and sounds

Customizing titles, text fields, and scroll views

Setting textual attributes

Setting text and background colors

Setting mnemonics

Setting box (group) attributes

Customizing browsers

Setting attributes of menu items and pop-up lists

Setting matrix attributes

Setting up table views

Automatically resizing objects

Using tags

But four young Oysters hurried up,
All eager for the treat;
Their coats were brushed, their faces washed,
Their shoes were clean and neat —
And this was odd, because, you know,
They hadn't any feet.

Lewis Carroll, *Through the Looking Glass*

Is it a world to hide virtues in?

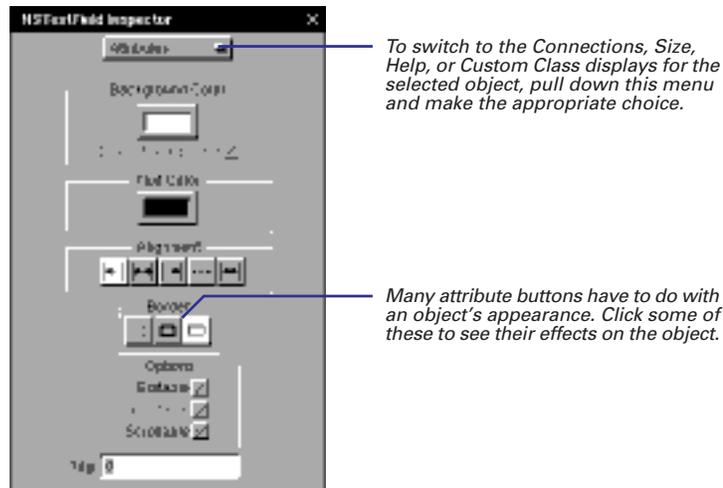
Shakespeare, *Twelfth Night*

Examining an object's attributes

- 1 Choose Inspector from the Tools menu.
- 2 Select an object in the interface.

You can examine the attributes of any object, whether that object is a graphical object such as a button or panel, or a non-UI object in the Instances display.

The Inspector panel displays the attributes of the currently selected object.



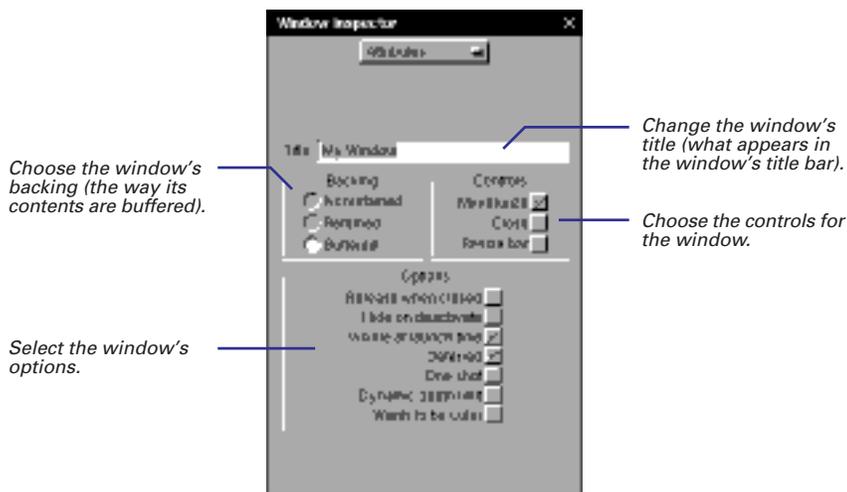
Tip: You can also bring up the Attributes display of the Inspector panel by pressing Command-1.

Once the Inspector panel is visible on the screen, it stays there until you close it. As you select different objects, their attributes are displayed (or dimensions or connections or help links—whatever Inspector display is current).

Customizing windows and panels

- ▶ Set the window title.
- ▶ Determine how the Window Server buffers window contents.
- ▶ Choose the window's controls.
- ▶ Set the window's options.

A single Attributes display serves for both windows and panels.

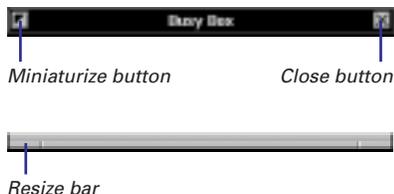


Window Backing

The backing determines how to redraw part of a window when that part is re-exposed after being covered by another window.

- **Nonretained:** The application is responsible for all drawing on the screen because there is no buffer. If the application does nothing, the re-exposed part is replaced by the background color. Nonretained windows are appropriate for transitory images that you don't need to save.
- **Retained:** The Window Server copies the covered part's pixels to a buffer. When the obscured part of the window is later revealed, the Window Server redraws only that part, not the rest of the window. A retained window is the appropriate choice for most situations.
- **Buffered:** The Window Server first draws in the buffer and then copies the buffer to the screen. When an obscured part is revealed, the Window Server refreshes the entire window using the buffer. A buffered window is appropriate when you don't want users watching complicated images being rendered on-screen. It is also the best choice for animation or for redrawing lines of rapidly typed text.

Window Controls



Window Options

Option	Description
Release when closed	The window object is sent a release message when it is closed.
Hide on deactivate	The window should disappear when the application is deactivated.
Visible at launch time	The window should appear when the nib file is loaded.
Deferred	A window device for this object is deferred until it's placed on-screen.
One shot	The window device is released when it is removed from the screen.
Dynamic depth limit	The window's depth limit can change to match the depth of the screen.
Wants to be color	The window is displayed on a color screen (2-monitor systems only).

What's the Difference Between a Window and a Panel?

A panel is a window that serves an auxiliary function within an application. Because it's intended for a supporting role, a panel typically has these features:

- A panel can be the key window, but never the main window.
- When the application is deactivated, the panel moves off-screen (it's removed from the screen list). When the application is reactivated, the panel appears again.
- When a panel is closed, it moves off-screen; it isn't destroyed.
- When instantiated programmatically, panels have a grey background by default, while programmatically created windows have a white background.

Also, a panel usually has fewer controls: only a close button; rarely a resize bar; and sometimes no controls at all.

You can make some panels exhibit special behavior for specialized roles:

- A panel can be precluded from becoming the key window until the user makes a selection in it.
- Some panels (e.g., palettes) can float above windows and other panels.
- You can have a panel receive mouse and keyboard events while an attention panel is on-screen. Actions within the panel can thus affect the attention panel.

Setting button attributes

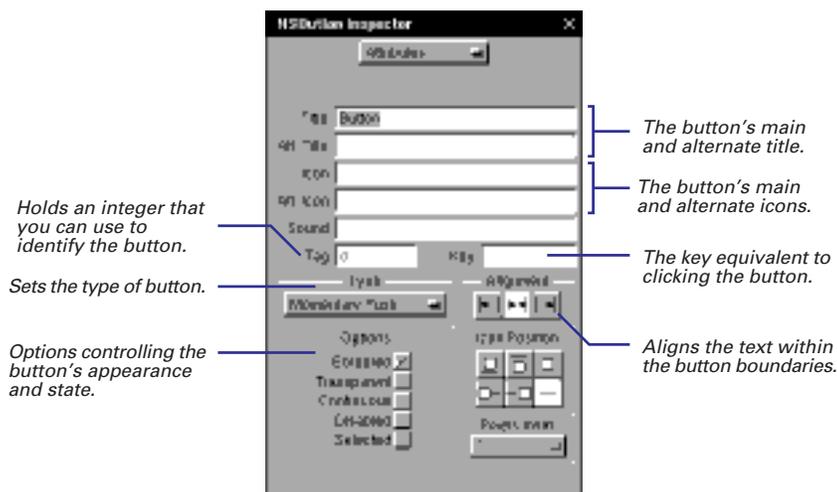
- ▶ Enter the button's main and alternate titles.
- ▶ Select the button type.
- ▶ Specify any key equivalent.
- ▶ Specify button options.

The Attributes display for buttons enables you to set a button's type, title, icon, alternate title and icon, and various other characteristics. The object labeled Button on the Views palette is only one style of button (albeit the most common style). The palette also holds radio buttons and switch buttons. Using the Attributes display for buttons, you can customize any palette button, making it something that is uniquely suitable for a particular circumstance.

The Icon Position and Pixels Inset controls as well as the Sound and Icon fields are described in detail in the next task, “Associating images or sounds with buttons.”

For more information on the Tag field, see “Using tags” in this chapter.

You might think of storing specially configured buttons on a dynamic palette. See Chapter 5, “Using Dynamic Palettes,” for complete information.



The Anatomy of a Button

A button is essentially a two-state NSControl object. When a user clicks a button, an action message is sent to a target object. It is two-state because it is either on or off, and when it is on, it typically sends its action message. For a button, the states are also known as “normal” (off) and “alternate” (on).

Like most objects on the Views palette in Interface Builder, a button is actually a

compound object: an NSButton object and an NSButtonCell object. (See “Compound Objects” in this chapter.) Most of NSButton's methods match identically declared methods in NSButtonCell. Aside from dispatching the action message, NSButton's unique role is to set the font of the key equivalent, and to manage the highlighting or depiction of the NSButton's current state.

Button Titles and Icons

The Title field's value is what appears in most buttons; you can set the title by double-clicking inside the button. The Icon field identifies an image stored in the nib file (Images display) that appears within the button. The alternate title (Alt. Title) and the alternate icon (Alt. Icon) appear when the user clicks a button of type Momentary Change or Toggle.

Button Key

The Key field identifies a keyboard alternative to clicking the button. Possible values are: \e (Escape), \r (Return), and any normal letter or number.

Button Type

Type	Button Behavior When Clicked
Momentary Push	Button is highlighted, appears to be pressed.
Momentary Change	Alternate button title and icon appear (while mouse button is pressed).
Momentary Light	Button is highlighted, but no illusion of being pressed.
Push On/Push Off	First click highlights button with illusion of being pushed in; second click returns it to normal.
On/Off	First click highlights button. Second click returns it to normal.
Toggle	First click displays alternate title and button. Second click returns to normal.

Button Options

Option	Description
Bordered	A line is drawn around the button's border.
Transparent	The button has no border, text, icon, or background color.
Continuous	The button sends its action message continuously when pressed.
Disabled	Prevents activation of the button; title is in gray.
Selected	The button, when initialized, is to be selected (applies to switch and radio buttons).

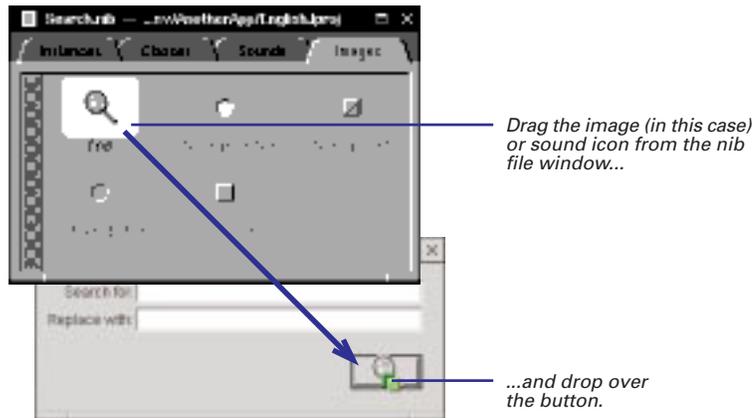
Associating images or sounds with buttons

- ▶ Drag the icon representing an image or sound from the nib file window or the Workspace and drop it over a button.

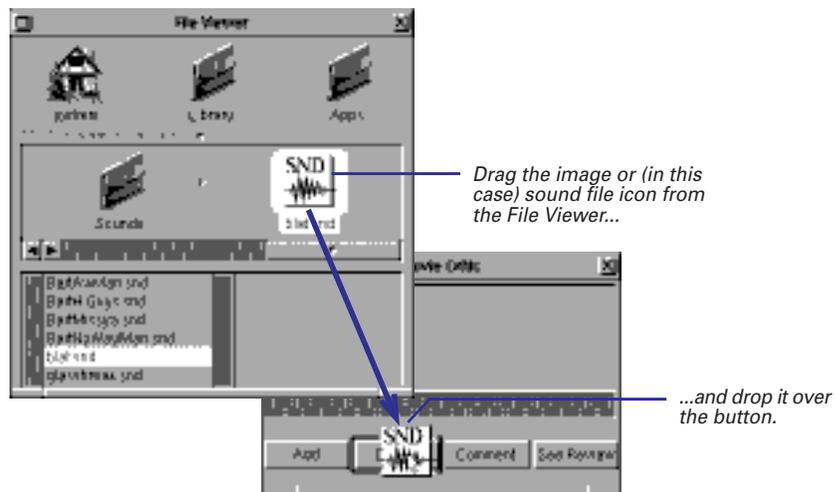
Or

- ▶ Enter the file name of the image or sound in the appropriate field of the button's Attributes inspector.

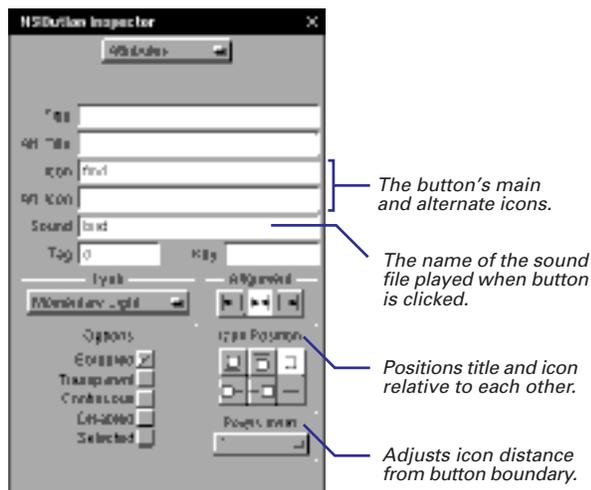
When you click a button that has a sound associated with it, it plays the sound. Images appear in buttons with or without text.



When you drag an image or sound from the File Viewer, it automatically gets added to the Images or Sounds section of the nib file.



Several fields and controls in the Inspector's Attributes display for buttons relate to images and sounds.



Before you type in the file name, you should insert the resource into the nib file or the project. Usually, you want to add the resource to the project. See “Managing images and sounds” for more information.

Note that the name of an image or sound in this display is the file name (**find.tiff** and **Poit.snd**, for example) minus the extension. Instead of dragging and dropping image and sound icons, you can type their file names (minus the extension) in the appropriate field.

Icon Position and Pixels Inset

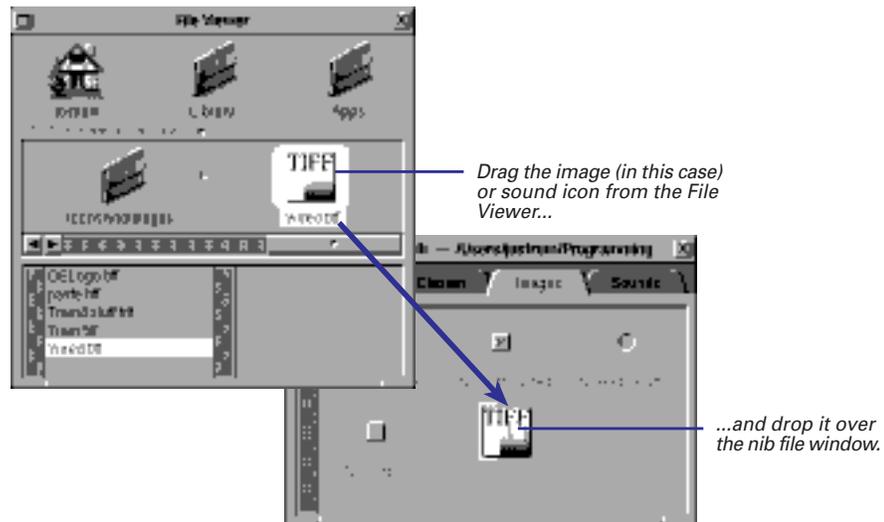
The six buttons in the Icon Position group position the button title and icon relative to each other. Thus, you can have the title above, below, to the left, or to the right of the icon, or show only one or the other. The Pixels Inset pop-up list gives several pixel distances for adjusting the spacing between the icon and the nearest edge of the button.

Tip: If you want to import images into your interface for decorative purposes, use the image view object on the DataViews palette. You simply drop the image on the image view. You’ll probably want to deselect the bordered option and the Editable option in the image view’s Attributes inspector.

Managing images and sounds

- ▶ To add an image or sound, drag its file icon from the File Viewer and drop it over the nib file window.
- ▶ Examine the image or sound in the Inspector's Attributes display.

You can add images and sounds to a nib file. The image or sound is added to the appropriate display no matter what display is currently showing.



As shown in “Associating images or sounds with buttons,” you can add images and sounds to a nib file as a side effect of associating them with a button.

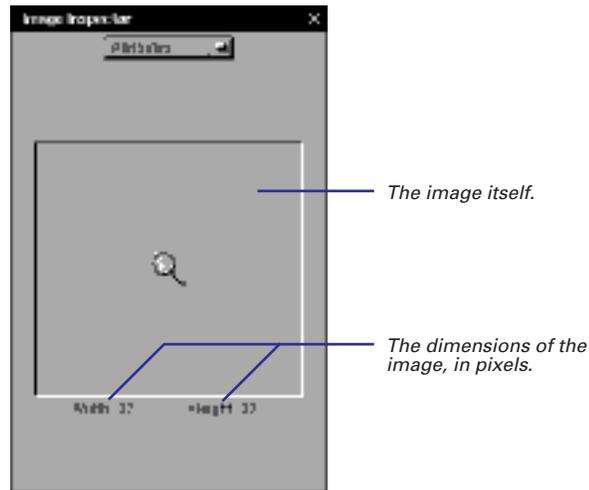
Although the association of images and sounds with buttons is an important reason for putting them into a nib file, there are other reasons. When you composite an image or play a sound in your code, the search path (if your code supplies no path) starts with the application's executable (already loaded resources), the main bundle, and the main bundle's `.proj` directories. Then the standard directories are searched:

- the appropriate subdirectory of the user's `~/Library` directory
- the appropriate directory in `/LocalLibrary`
- the appropriate directory in `/NextLibrary`

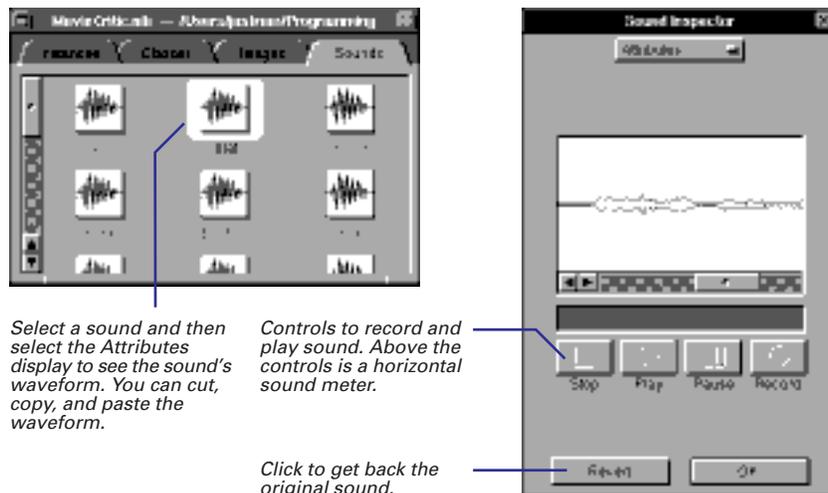
If you do not want to risk an image or sound not being in one of these standard directories, then you should store it in a nib file or in the project.

Tip: For most situations, the recommended course of action is to add images and sounds to your project. If you add them only to a nib file, they won't be available to an application until the nib file is loaded.

Images and sounds have their own Attributes displays. For images, this is mostly useful for images that are too large to show in the nib file window.



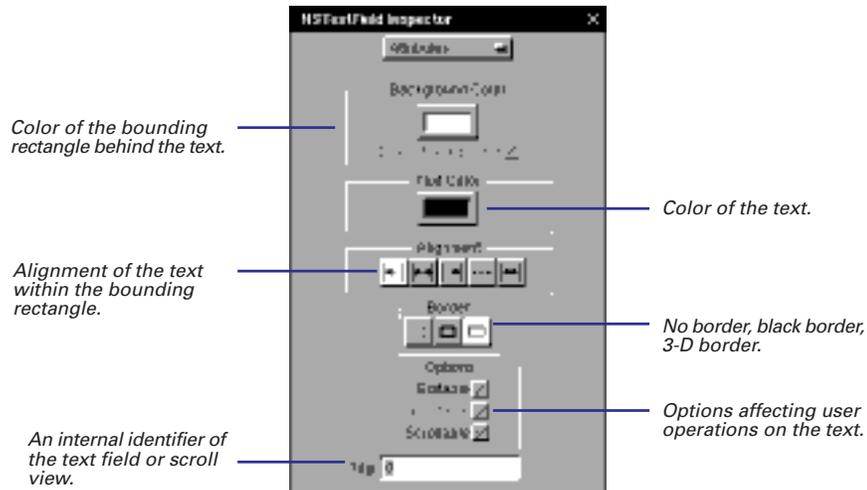
If your system has a microphone or some other input source connected, you can record new sounds. Click OK to save new sounds.



Customizing titles, text fields, and scroll views

- **Set background and text color, text alignment, border style, tag, and options affecting access to text.**

The objects that exist principally to display text—text fields, titles, and scroll views—have controls for initializing those objects with various characteristics. To see what certain effects look like, drag a text field onto a window and click the buttons on this display. The Title object is just a specially configured text field: non-selectable with transparent backgrounds and no borders.



A tag is an internal identifier of an object that you can use in your code. See “Using tags” in this chapter for more information.

For more information on the NSTextField, NSScrollView, NSScroller, NSText, and NSClipView classes, see the *Application Kit Reference*.

An NSScrollView object is a compound object consisting of one or two NSScroller objects and an NSClipView object, which has as its document view (subview) an NSText object in Interface Builder. The document view is what is scrolled. The NSScrollView object has a slightly different Attributes display: no text alignment buttons and a different set of options.

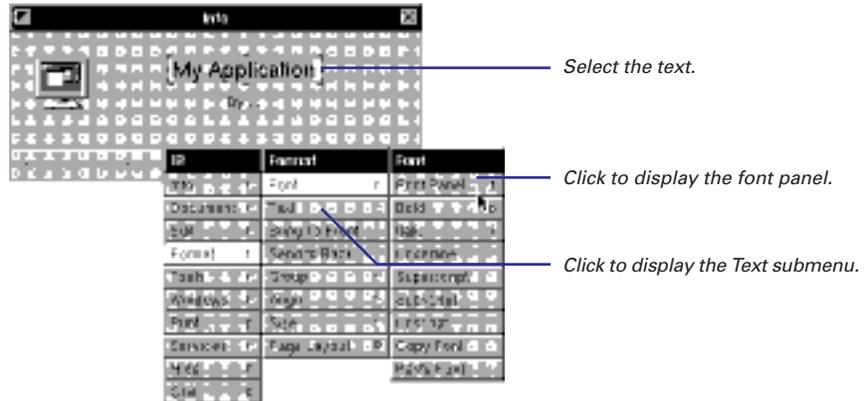
Text Field and Scroll View Options

Option	Description
Editable	Allows the user to edit text.
Selectable	Allows the user to select text.
Scrollable (NSTextField)	Text scrolls to the left if necessary.
Multiple fonts allowed (NSScrollView)	Text is in RTF format.
Graphics allowed (NSScrollView)	Text is in RTFD format (graphics can be inserted).

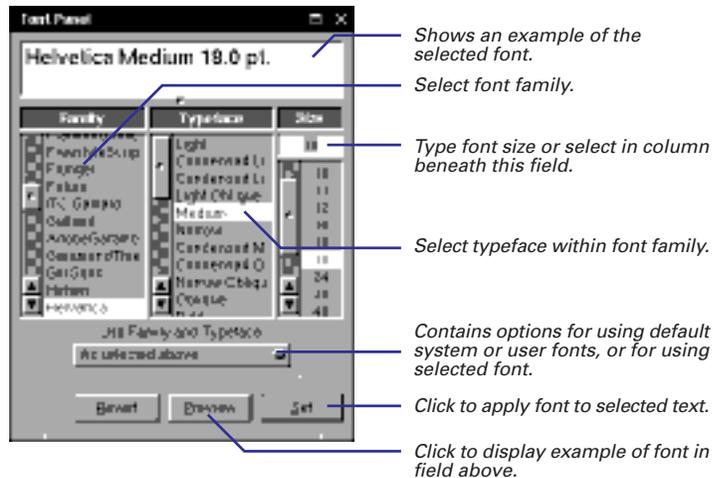
Setting textual attributes

- ▶ Set the font characteristics of selected text using the Font Panel.
- ▶ Set the alignment of selected text within its boundaries using commands on the Text menu.

Almost all palette objects—from buttons to browsers—can display text. You can set the font and alignment attributes of this text.



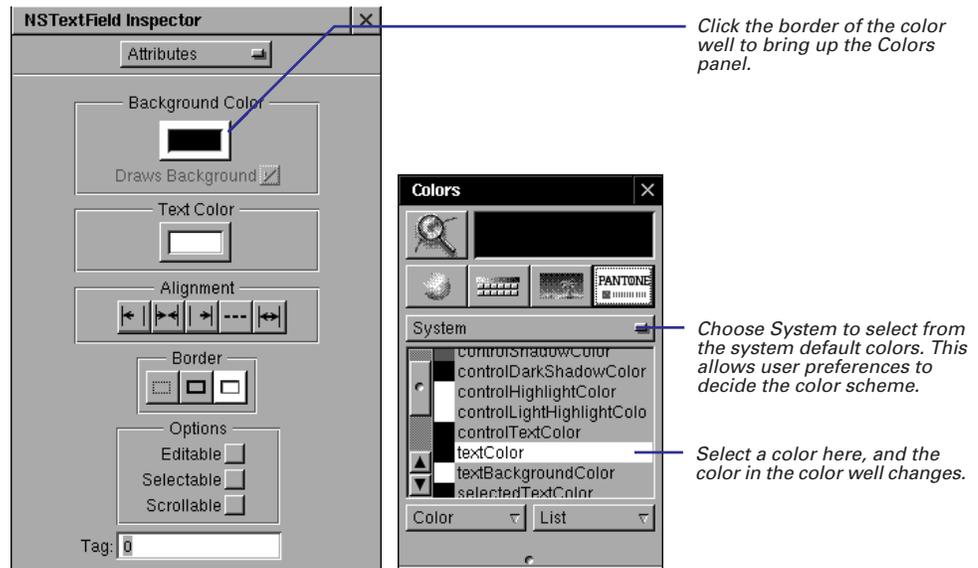
The Text submenu of the Format menu also has commands that affect selected text; it offers options for aligning text and for displaying, copying, and pasting the ruler in a Text object. With rulers you can set tabs and indentation. Note that rulers can only appear in NSText objects (for instance, inside a scroll view).



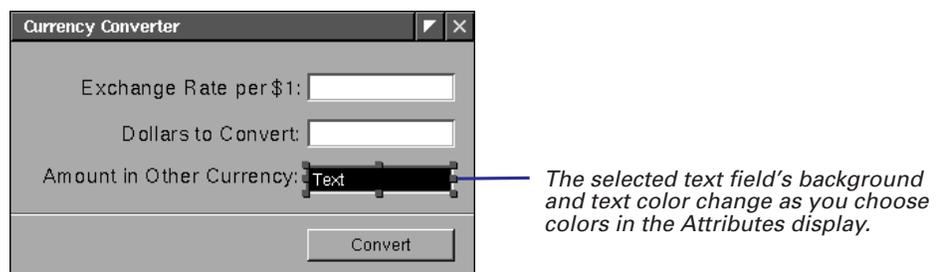
Setting text and background colors

- 1 Select an `NSTextField`, `NSScrollView`, or `NSMatrix` object.
- 2 In the Inspector panel, select the Background or Text color well.
- 3 In the Colors panel, choose a color.

You can set the color of any of the text objects (`NSTextField`, `NSScrollView`, and `NSMatrix`). You can specify both a color for the text and a color for the background.



As with setting fonts, you can either choose to enforce a specific color or allow the user to select the color. For example, if you choose Black for the Text Color from the NeXT list in the PANTONE view, the text will be black for all users. However, if you choose `textColor` from the System list, the text will be the color the user selects in the system default settings (which is black unless the user changes it).



As a shortcut, drag a color from the Colors panel directly to the object to color its text. If you hold down the Shift key when you drag, the background is colored instead.

Setting mnemonics

- ▶ **Alternate-double-click a letter to choose that as the mnemonic.**

You can set a mnemonic in any object that displays a title, such as a form, a text field, or a button. Users can select that object by holding down the Alternate key and pressing the mnemonic you've chosen for the object. In this way, you give your users the option of navigating through the interface through the keyboard instead of the mouse.



Hold down the Alternate key and double-click the letter to set the mnemonic. The letter is underlined.

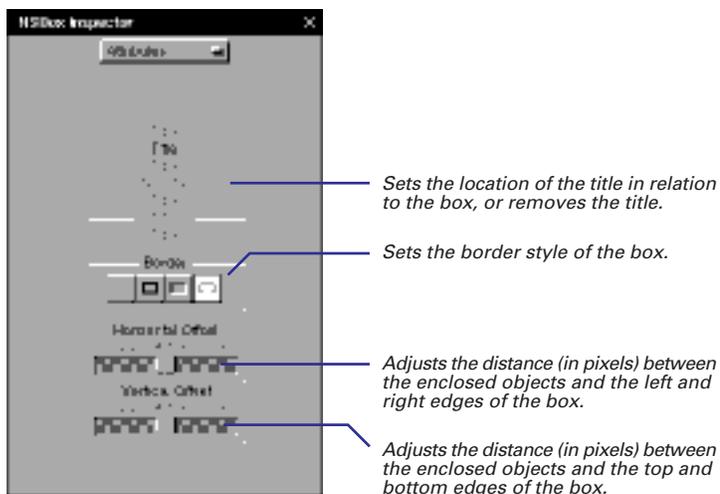
To delete a mnemonic, Alternate-double-click the letter again.

Mnemonics, together with inter-field tabbing, allow users to navigate through objects in a window using only the keyboard. Inter-field tabbing allows users to select objects by tabbing. See Chapter 4 for more information.

Setting box (group) attributes

- **Set title position, border style, and horizontal and vertical offsets.**

When you group a selection of objects in a box, that box (actually the box's content view) becomes the superview of the enclosed objects. In Interface Builder, you can move, copy, paste, and delete the group of objects as one. The box has several attributes that you can set.



Sets the location of the title in relation to the box, or removes the title.

Sets the border style of the box.

Adjusts the distance (in pixels) between the enclosed objects and the left and right edges of the box.

Adjusts the distance (in pixels) between the enclosed objects and the top and bottom edges of the box.

See Chapter 2, “Composing the Interface,” to learn how to group objects inside of a box.

For more information on box objects, see the `NSBox` class specification in the *Application Kit Reference*.

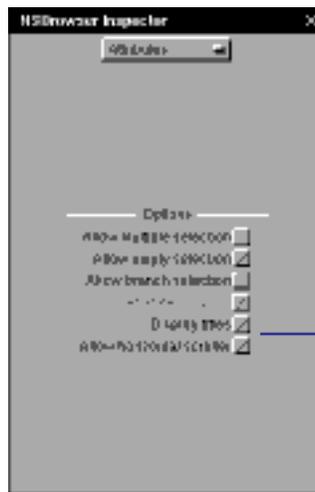
You can drag a box onto your interface and then programmatically replace its content view (blank by default) with another `NSView` object, or programmatically add subviews to the content view. You can also manipulate this box to make decorative rectangles and lines.

Tip: To make a line in an interface (such as a divider line between sections of a panel), drag a box onto the interface. Then switch off the title and make the box as narrow as possible in the required dimension (vertically or horizontally). Finally, set the offset (vertical or horizontal, whichever is applicable) to zero.

Customizing browsers

► **Select the browser options.**

Browsers display lists of data and allow users to select items from the list. They can hold one-dimensional lists or hierarchically organized lists of data such as directory paths. Browsers display these hierarchical levels in columns, which users can navigate using buttons or scrollers. An entry in a column can be either a *leaf node* or a *branch node*. Leaf nodes terminate a path; branch nodes, which have a right-arrow icon, lead into the next level in the hierarchy. A browser's attributes affect its navigation controls, methods of selection, and appearance.



Select options for browser.

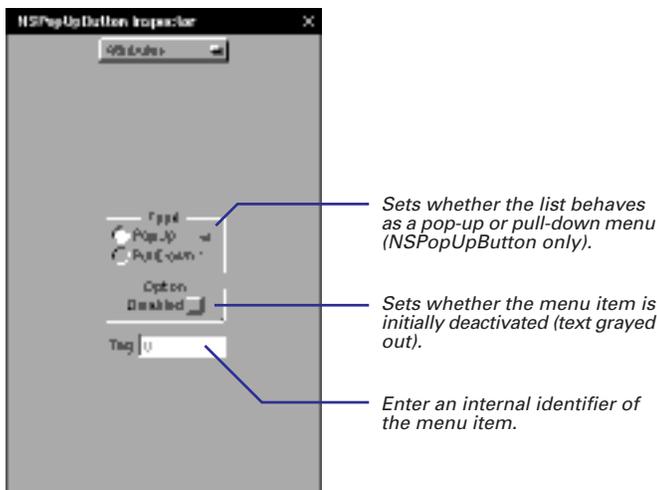
Browser Options

Option	Description
Allow multiple selection	The user can select more than one node at a time.
Allow empty selection	Makes it possible to have no cells selected; otherwise, the first cell in the column is selected by default.
Allow branch selection	The user can select branch nodes (such as directories).
Separate columns	Separates columns with a beveled bar (if not set, a black line appears).
Display titles	Titles are above columns and column divider is beveled bar.
Allow horizontal scroller	Allows users to scroll horizontally as well as vertically.

Setting attributes of menu items and pop-up lists

- ▶ Set whether the list is a pop-up or pull-down type (not applicable to menu items).
- ▶ Set whether the item is initially disabled.
- ▶ Assign a tag to the item.

Menus and pop-up or pull-down lists (NSPopUpButton instances) are compound objects containing objects that conform to the NSMenuItem protocol. The Attributes displays for menu items and NSPopUpButtons are almost identical. The following is the display for NSPopUpButtons.



If you choose Disabled, the menu item's text is gray at application launch. When the user clicks the item, no action message is sent. If conditions change to make the items's function relevant, your code must re-enable the item.

A tag is an internal identifier of an object that you can use in your code. See the task, "Using Tags," in this chapter for more information.

Once you expose a pop-up list's menu items, you can add more menu items to it from the Menu palette. See "Creating menus" in Chapter 2 for details.

Pop-Up Lists and Pull-Down Lists

An NSPopUpButton contains a trigger button and three menu items. Double-click the trigger button to see the menu items; you can initialize their titles or (in the Attributes display) disable them and assign them tags.



A *pop-up list's* trigger button always displays the item that was last selected. In a *pull-down list* the trigger button's title is fixed. A pull-down list is effective for selecting actions in a very specific context, like the "Operations" pull-down list in Interface Builder's Classes display.

Compound Objects

Most of the objects you can drag from the standard Interface Builder palettes are actually compound objects. They consist of two or more objects that work together in specific ways.

Controls and Cells

A control (an instance of an `NSControl` subclass) functions as an event translator. It translates a user event like a mouse click into an action message and directs that message to another object in the application (the target).

Controls supply the mechanism but not the content of the target/action paradigm. They need action cells (or instances of `NSActionCell` subclasses) to hold this information:

- **target** the object receiving the action message
- **action** the method that specifies what the target is to do

At least one of these cells occupies the same area as its control. Because it descends from `NSCell`, a cell also has content (text or image), which it draws upon request from the control.

This division of responsibility makes for greater efficiency because a control can have multiple cells and send a different action message to a different target for each of those cells. Because cells are lightweight objects, it is more efficient in some contexts to associate one control with many cells.

Matrices

A matrix (an instance of the `NSMatrix` class) is a control that manages more than one cell. It organizes its cells in rows and columns. The cells must be the same size and usually are of the same class (although a matrix can have instances of different subclasses of `NSCell`).

Each cell in a matrix can have its own action and target. A matrix also has its own action and target. If a cell doesn't have an action, the matrix's action is sent to its target. If a cell doesn't have a target, the matrix sends the cell's action to its own target.

In Interface Builder, you can convert a single-celled control (such as an `NSButton`, `NSSlider` or `NSTextField`) into a matrix by Alternate-dragging a resize handle of that control. The associated cell, whether an `NSButtonCell`, `NSSliderCell`, or `NSTextFieldCell`, is duplicated for each row and column of the matrix.

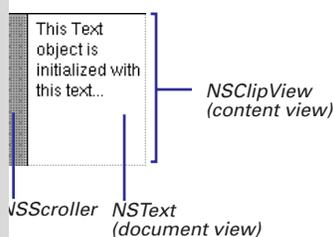
Forms are a special type of matrix (`NSForm` inherits from `NSMatrix`). They have special cells (`NSFormCell` instances) that compose both the form entry fields and the titles of those fields.

Special Compound Objects

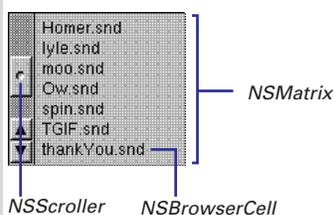
Some objects on Interface Builder's standard palettes are of a more complex composition.

- **Scroll View** This object coordinates the interaction between `NSScroller` objects and an `NSClipView` object to scroll a document. It consists of one or two `NSScrollers`, an `NSClipView`, and the document view, which is generally `NSText`.
- **Browser** This object has scroll bars for controls and columns to show hierarchically organized data. Each column is a matrix of `NSBrowserCell` objects.
- **Pop-Up List** This object has a trigger button and an array of objects that conform to the `NSMenuItem` protocol.
- **Menu** This object's content area contains an array of objects that conform to the `NSMenuItem` protocol.
- **Table View** See "Inside the `NSTableView` Object" in this chapter.

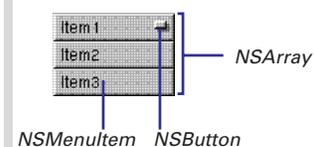
VSScrollView



NSBrowser



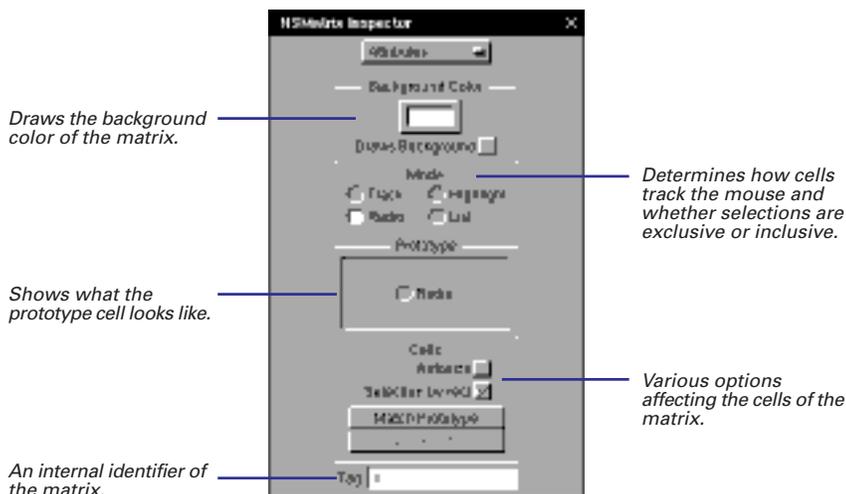
NSPopUpButton



Setting matrix attributes

- ▶ Set the background gray of the matrix.
- ▶ Set the matrix selection mode.
- ▶ Set autosizing behavior and other properties of cells.
- ▶ Inspect the cell prototype and change it, if necessary.

The Attributes display for matrices allows you to determine how a matrix and its cells look and behave.



Matrix Selection Mode

The selection modes specify how cells behave when a user is dragging a mouse within a matrix. They also determine if the user can select multiple items in the matrix—a column of switch buttons, for example, allows multiple selection.

- **Track:** The cells track the mouse when it is within their bounds but do not highlight themselves. This mode would be suitable for a “graphic equalizer” matrix of sliders. Dragging the mouse causes the sliders to move.
- **Radio:** Only one cell in the matrix can be selected at a time, as is the typical case with a matrix of radio buttons.
- **Highlight:** Each cell is highlighted while it tracks the mouse and is unhighlighted when done tracking. This mode allows multiple selections. A matrix of switch buttons commonly has this mode.
- **List:** Cells are highlighted as the mouse is dragged across them, but they do not track the mouse. In this mode, a matrix supports multiple selection, enabling a user, for instance, to select a range of text in a matrix of text objects.

Cells Options

Option	Description
Autosize	If set, the cells resize when the matrix is resized, keeping the space between cells constant. If not set, the space between cells changes.
Selection by rect	Allows users to select multiple cells by dragging the mouse around them.
Match Prototype	Applies the new prototype to the selected matrix's existing cells.
Tags = Position	Resequences the cell's tags if you've added cells to a previously created matrix. When you create a matrix, cells are assigned consecutive tags starting from zero. For two dimensional matrices, the progression is from left to right (row), then down (column). When you later add new cells, they all have tags of zero.

A tag is an internal identifier of an object that you can use in your code. See “Using tags” in this chapter for more information.

Changing the Prototype Cell

When a matrix creates its cells, it typically makes them by copying a prototype cell stored as an instance variable. (It can also instantiate its cells from their class.)

You can examine and alter this prototype cell's attributes through the Inspector's Prototype display. This display is only available when you select a matrix.

If you change the prototype, you must click the Match Prototype button on the Attributes display of the matrix for the existing cells to reflect the changes.



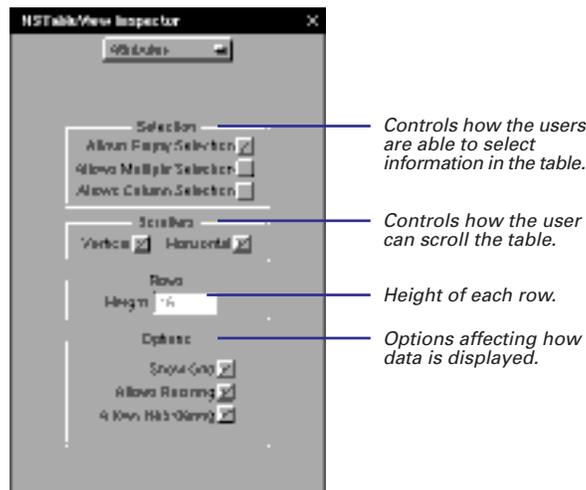
Choose Prototype here to display the prototype cell's attributes.

Setting up table views

- ▶ Set the type of selections the user can make.
- ▶ Set whether the table view should scroll vertically, horizontally, or both.
- ▶ Set the height for table rows.
- ▶ Set if the user can resize columns, reorder rows, and if cells are bordered.

A table view displays information in a table and allows the user to select and change that information. Table views contain rows and columns of information. When you create a table view in Interface Builder, you create the number of columns you want. Rows are added programmatically.

Tip: To add a column to a table view, select an existing column, then copy and paste.



The Selection options control how the user can select information in the table view. By default, the user can select rows in the table one at a time. Allows Column Selection means the user can select columns of information. Allows Multiple Selection means the user can select more than one row or column at a time.

Table View Options

Option	Description
Show Grid	Lines are drawn around each cell in the table.
Allows Resizing	The user can resize the table columns.
Allows Reordering	The user can rearrange the table rows.

Inside the NSTableView Object

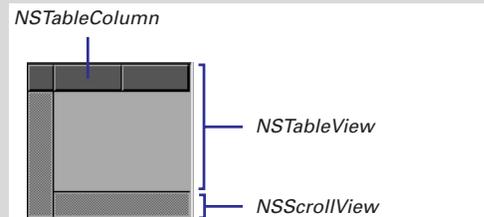
NSTableView used to be available only to people using the Enterprise Objects Framework or, before that, DBKit. Now, NSTableView is part of the Application Kit, so every application can take advantage of its features.

When you drag a table view from the TabulationViews palette to your interface, you're actually getting several objects. The NSTableView is nested inside of an NSScrollView. The NSTableView itself is made up of one NSTableColumn object for each column and an NSTableHeaderView, which displays the column headings.

Each NSTableColumn has an NSCell associated with it that is used to draw all of the cells in that column. The NSCell may have an NSFormatter associated with it that defines how the contents of that cell are formatted. You can associate an NSFormatter with the NSTableColumn's NSCell in Interface Builder by dragging one from the Formatters palette.

Also associated with an NSTableView is an object conforming to the NSTableDataSource protocol. You don't create this object in Interface Builder unless you're creating an application based on the EO Framework. The data source controls the display of data in the NSTableView. You implement methods defined by the protocol to retrieve values from the table, to change values in the table, or to add rows to the table.

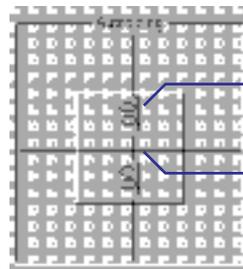
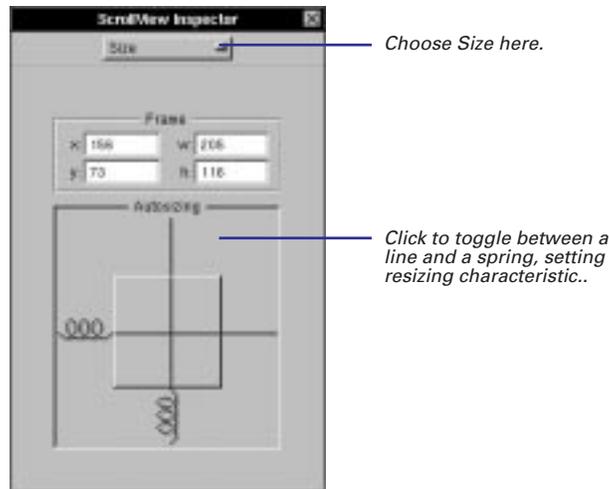
For more information about table views, see the NSTableView class specification in the *Application Kit Reference*.



Automatically resizing objects

- 1 **Select an object.**
- 2 **Choose the Size display of the Inspector panel.**
- 3 **In the Autosizing view of the display, click lines to make them springs or click springs to make them lines.**

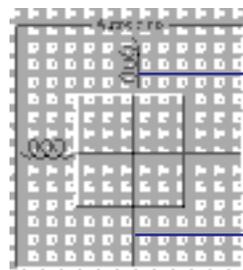
When you resize a window, the objects in the window must often adjust their size or the distances between themselves and other objects. The Size display of the Inspector panel lets you tell a selected object how to resize itself. The lines inside and outside the box affect different aspects of resizing behavior.



Inside the box

This spring indicates that, when the window orSuperview is resized vertically, the object resizes itself to maintain its distance from the top and bottom edges of the window orSuperview.

This straight line indicates that, when the window orSuperview is resized horizontally, the object maintains its initial size.



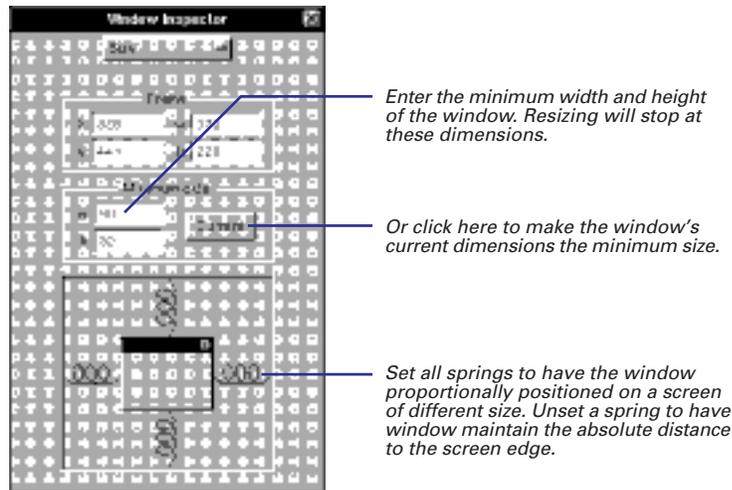
Outside the box

This spring indicates that, when the window orSuperview is resized vertically, the space between the top edge of the object and the top of the enclosing view or window is adjusted proportionally.

This straight line indicates that, when the window orSuperview is resized, the object maintains the initial distance between its bottom edge and the bottom of the enclosing view or window.

For examples of the effects of these “autosizing” characteristics on views within a resized window, see “Some Effects of Automatic Resizing.”

If you do not make a view resize itself when its superview or window resizes, some ugly behavior could result. For instance, if the user makes a window small, objects that don't resize themselves could become truncated by the resized window's borders. One recourse to this unwanted outcome is to specify a minimum size for the window.



Interface Builder includes a test mode that simulates the actual operation of the interface. In test mode, you can test the resizing behavior of your windows and views, see how connected objects communicate, play sounds associated with buttons, and do similar operations. See “Testing the Interface” in Chapter 4, “Making and Managing Connections,” for more information.

You might need to make several iterations in Interface Builder—setting resizing characteristics in objects and shrinking the window in test mode—to determine what the ideal minimum size should be.

When There Are Conflicts

You can create an impossible resizing relationship, such as specifying as fixed the object's dimensions and its distance from the window's edges. In cases of conflict, an object's fixed dimension takes precedence over its fixed distance from a border. If all dimensions are made resizable, adjustments to the window or superview's changed dimensions are made equally to the object and its distance from a border.

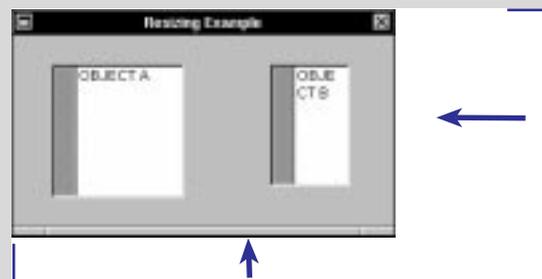
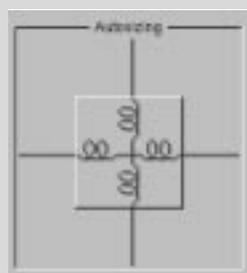
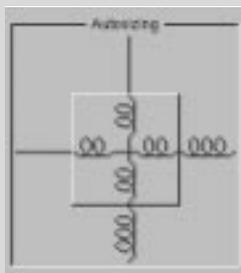
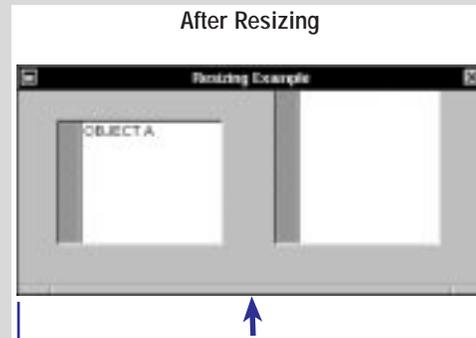
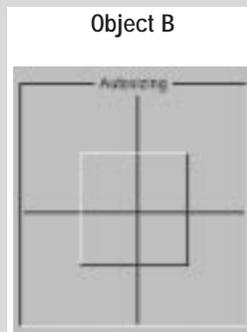
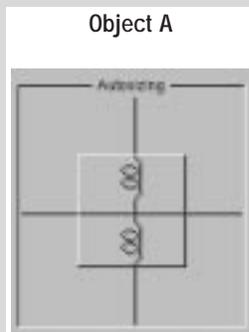
Some Effects of Automatic Resizing

The window below has two identical scroll view objects. Different autosizing “springs” are set in each, and then the window is resized in test mode. The screen shots under After Resizing show you the results.

In the first example, one object resizes vertically while the other doesn't (distances to borders are absolute for both). The result: the object that doesn't resize itself is truncated when the window is vertically shortened.

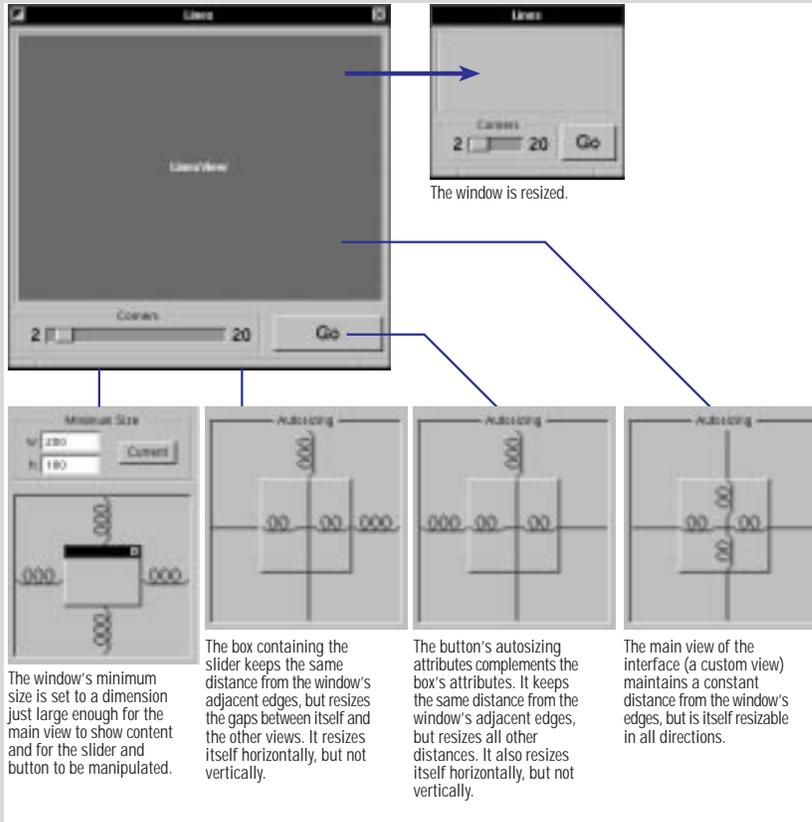
In the second example, both objects resize themselves, but Object B maintains its distance to surrounding objects. This causes Object B to be more severely resized than Object A.

To learn more about the effects of resizing, try some experiments on your own using different combinations of objects and autosizing attributes.



Automatic Resizing: An Example

This example interface incorporates autosizing attributes in such a combination that the window can shrink to a very small size and still be usable.



Using tags

- 1 In Interface Builder, specify the tag integers for objects.
- 2 If the integers are not intrinsically meaningful, define constants for them in your source code.
- 3 Send the tag message to a tagged object to get the integer.
- 4 Evaluate the integer and act upon it.

Tags are integers that you use in your code to identify objects. They offer a convenient alternative to such methods of object identification as fetching an object's title. (What if the object's title changes while the application is running, or the application is localized?) Tags can also carry useful information associated with an object, and thus make it easier to integrate that information into a program. Tags are commonly assigned to the cells contained by matrices.

You can specify tags in the Tag fields of most Attributes displays.



Enter a number to identify the object in your source code.

You can also set tags programmatically in most `NSView` objects by sending those objects the `setTag:` message.

The integers that you assign could have some intrinsic value; for instance, they could be numbers that are multiplication factors for a document-zoom feature, or numbers that correspond to the number of a keypad in a calculator application. If the tag numbers are not intrinsically meaningful (that is, they're arbitrary), it's prudent to define constants to express them.

```
typedef enum {
    LEFT = 1,
    RIGHT,
    BOTTOM,
    TOP,
    HORIZONTAL_CENTERS,
    VERTICAL_CENTERS,
    BASELINES
} AlignmentType;
```

When you need to identify a tagged objects in your code, use the **tag** method.

```
- (void)align:sender
{
    [self alignBy:(AlignmentType)[[sender selectedCell] tag]];
}
```

4

Making and Managing Connections

Connecting objects

Making connections in outline mode

Examining connections

Identifying objects in outline mode

Enabling inter-field tabbing

Disconnecting objects

Copying interconnected objects

Testing the interface

It could be said of me that in this book I have only made up a bunch of other men's flowers, providing of my own only the string that binds them together.

Montaigne, *Essais*

Let him look to his bond.

Shakespeare, *Merchant of Venice*

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.

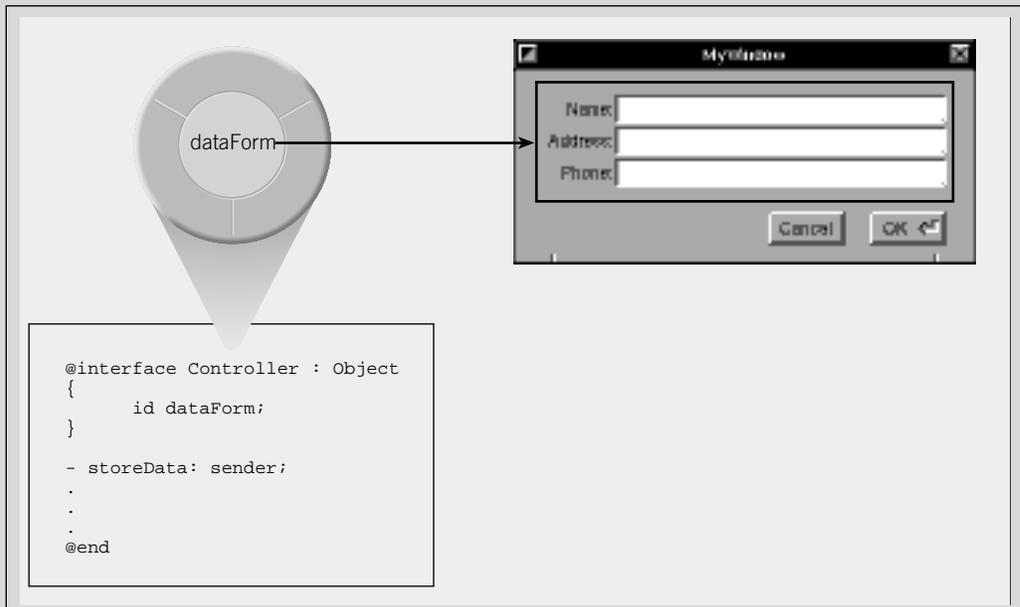
Targets

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.

Outlet



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 sends 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*.

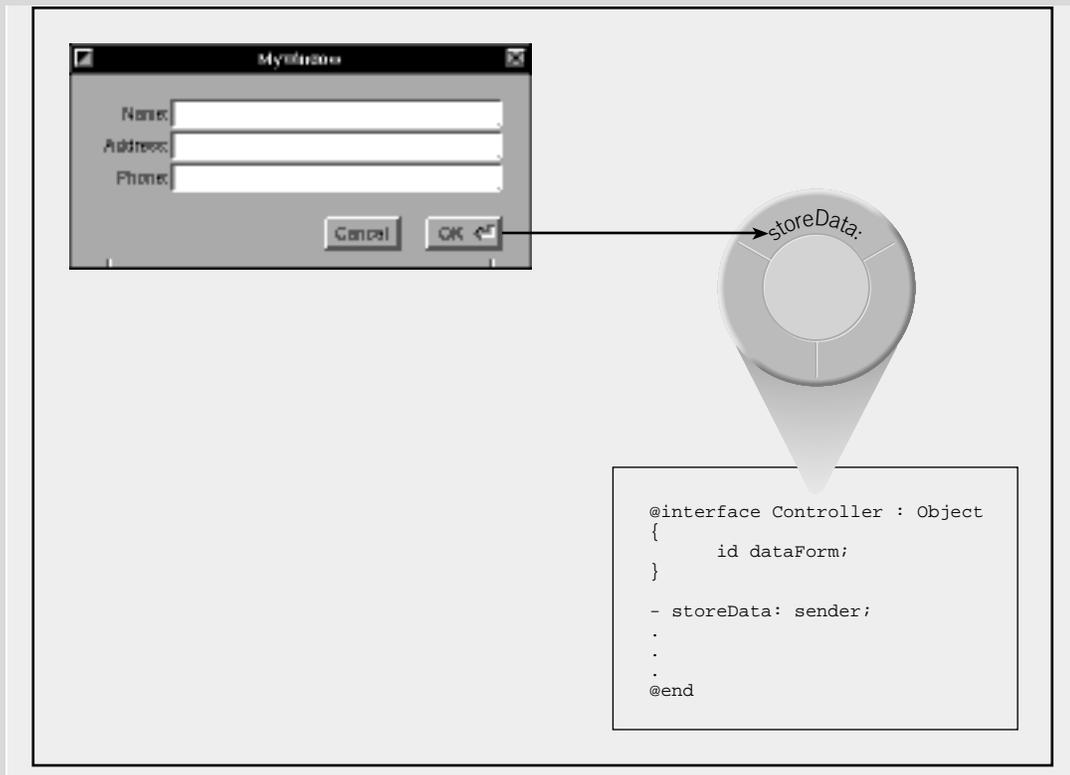
NSControl, an abstract class, defines for its many subclasses (such as NSButton, NSScroller, NSTextField, and NSForm) a paradigm for inter-object communication—action messages. But NSControl objects don't act alone: they always contain one or more NSActionCells (or one of its subclasses). The NSActionCell 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.

Action

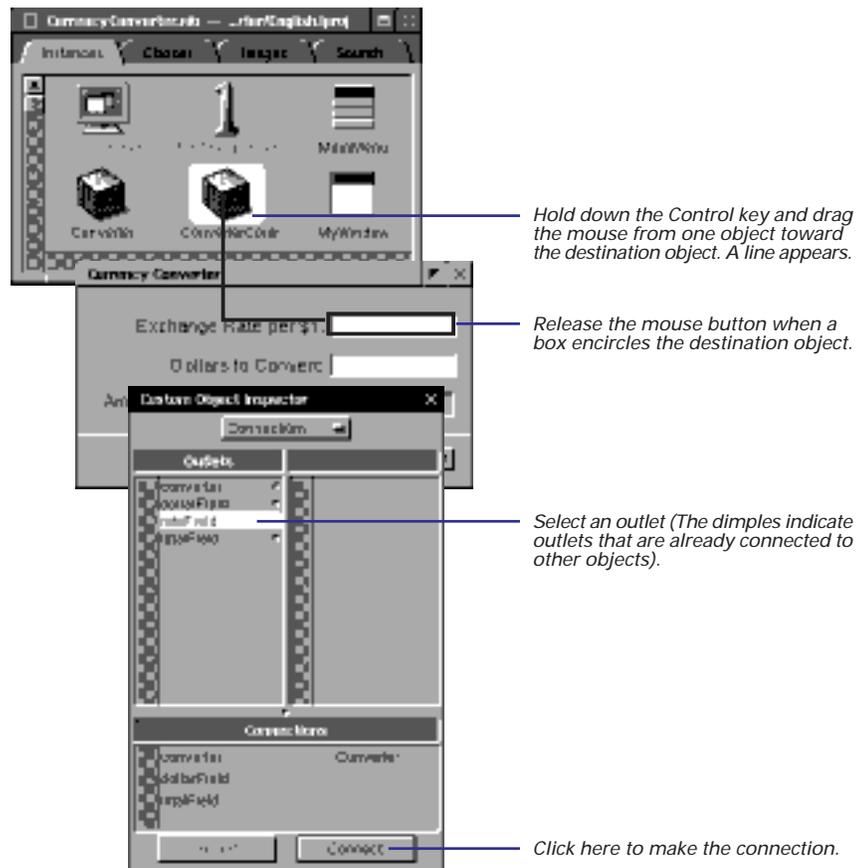


Connecting objects

- 1 **Select an object.**
- 2 **Control-drag a connection to another object.**
- 3 **In the Inspector panel's Connections display, select an outlet or action.**
- 4 **Click the Connect button.**

In an object-oriented application, isolated objects have little value; they need to send messages to each other to get the work of the application done. Interface Builder gives you a way to establish connections between objects.

When you Control-drag between two objects, the Inspector panel becomes the key window. Its Connections display shows the current and potential connections for the destination object.



If the Connect button doesn't become active when you select an outlet or action, you probably have connections locked. See "When You Don't Want to Disconnect" in this chapter.

Outlet Connections

In the previous example, the connection is made from a *controller* object—a custom object that manages the application—to a text field. The controller object (ConverterController) declares several *outlets*—identifiers of destination objects—as instance variables.

The example shows a connection between an object in the nib file window Instances display and an object in the interface. You can also make outlet connections between two objects in the Instances display.



When you make a connection between objects, the first column of the Connections display shows the source object's outlets ("source" meaning the object from which a connection line is drawn).

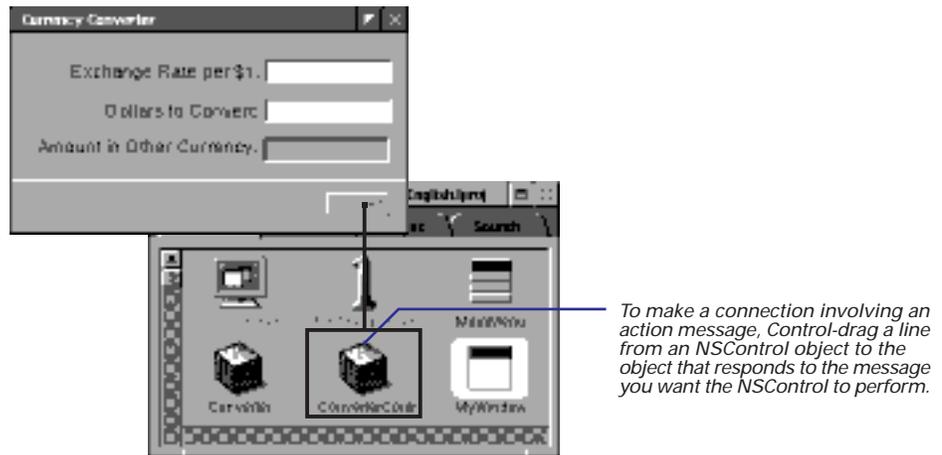
Action Connections

When you make a connection by dragging a line *from* an NSControl object in the interface—a button, slider, text field, menu command, pop-up list, or matrix—odds are that the destination object is a *target*

and that you can complete the connection by selecting an *action* method.

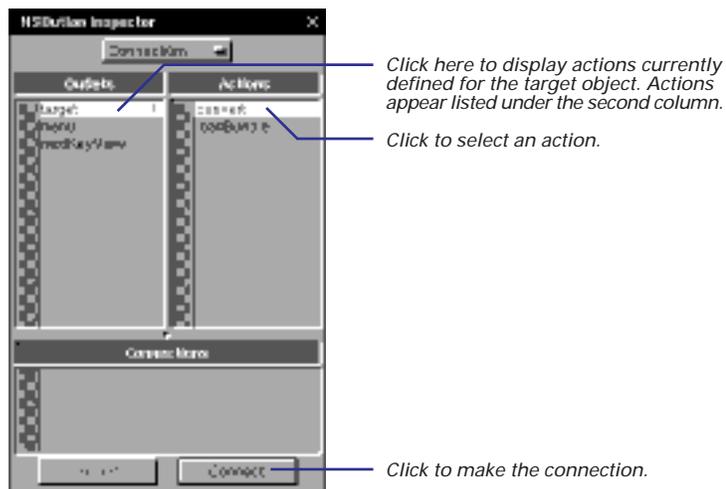
Outlets are destination objects specified as instance variables. Actions are methods that NSControl objects (such as buttons) invoke in another object. See "Communicating With Other Objects: Outlets and Actions" in this chapter for more information.

Chapter 6, "Subclassing," describes connecting the outlets and actions of custom objects in the context of creating a class.



The destination object in an action connection is frequently a custom object that manages the application or a particular window (controller object).

When you make a connection from an NSControl object, the Inspector panel shows the Connections display for the destination object.

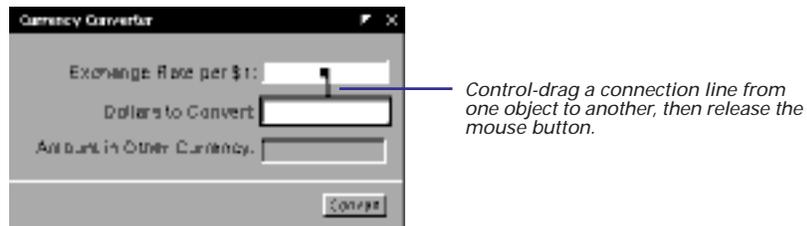


See “Compound Objects” in Chapter 3 for descriptions of the interaction between NSControl objects and NSCell objects, and of the role NSMatrix objects play.

When the user manipulates the NSControl object, such as clicking a button or moving a slider, the action message is sent to the destination object (the target).

Connections Within the Interface

Sometimes you can connect two objects on an interface. These connections can involve both outlets and actions.



Connections within an interface can also involve two Application Kit objects. Two examples are interconnecting text fields (so the user can tab from field to field), and connecting a menu command such as Print to an NSText object.

Tip: To enable printing of an NSText object, drag a connection line from the Print menu command (or other NSControl object that initiates printing) and select the **print:** action in the Connections display.

You can connect text fields and form fields so that when the user presses the Tab key, the cursor moves to another field. See “Enabling inter-field tabbing” in this chapter for information on this procedure.

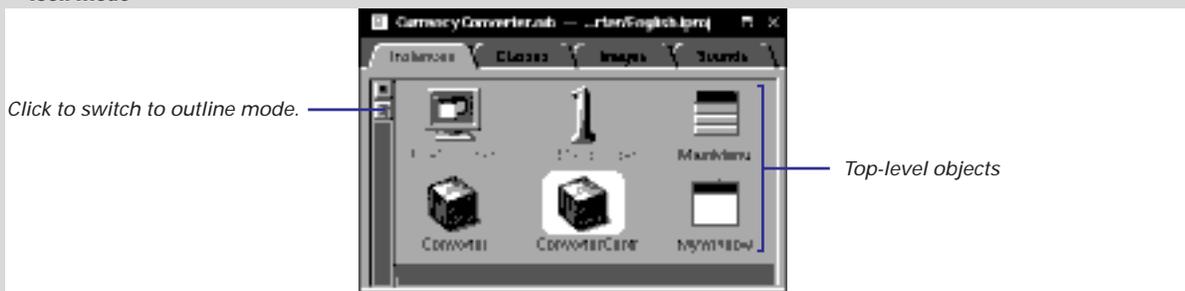
The Modes of the Instances Display

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*—those 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 mode—outline mode—that shows more detail about objects in the nib file, including their connections with each other.

Icon Mode



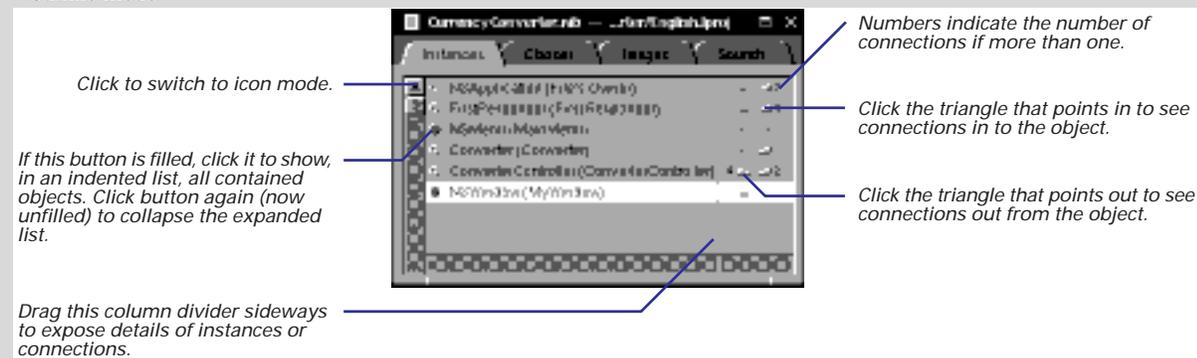
The most important advantage of the outline mode is that it shows *all* objects in the nib file, not merely the top-level objects. It also shows all connections, both connections into an object and connections from an object to other objects.

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.

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.

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

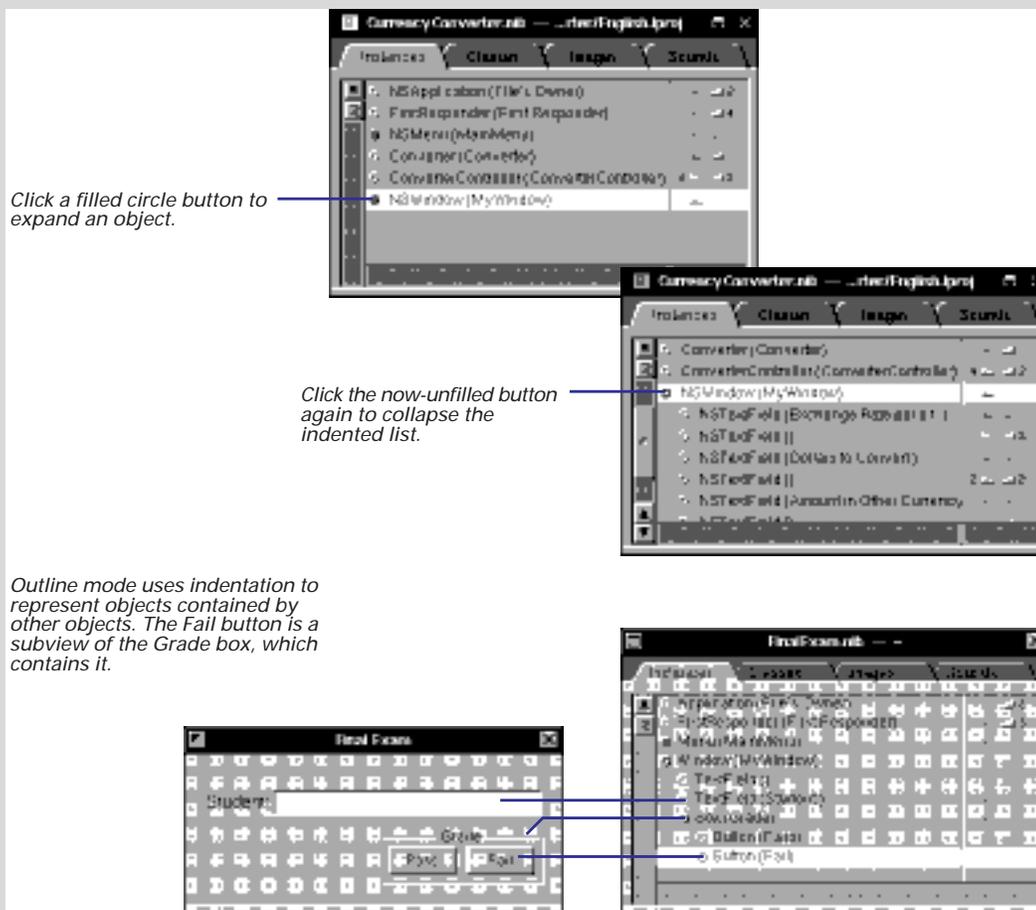


Expanding Objects in Outline Mode

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 “The View Hierarchy” in this chapter for a description of the relationship between superview and subview.

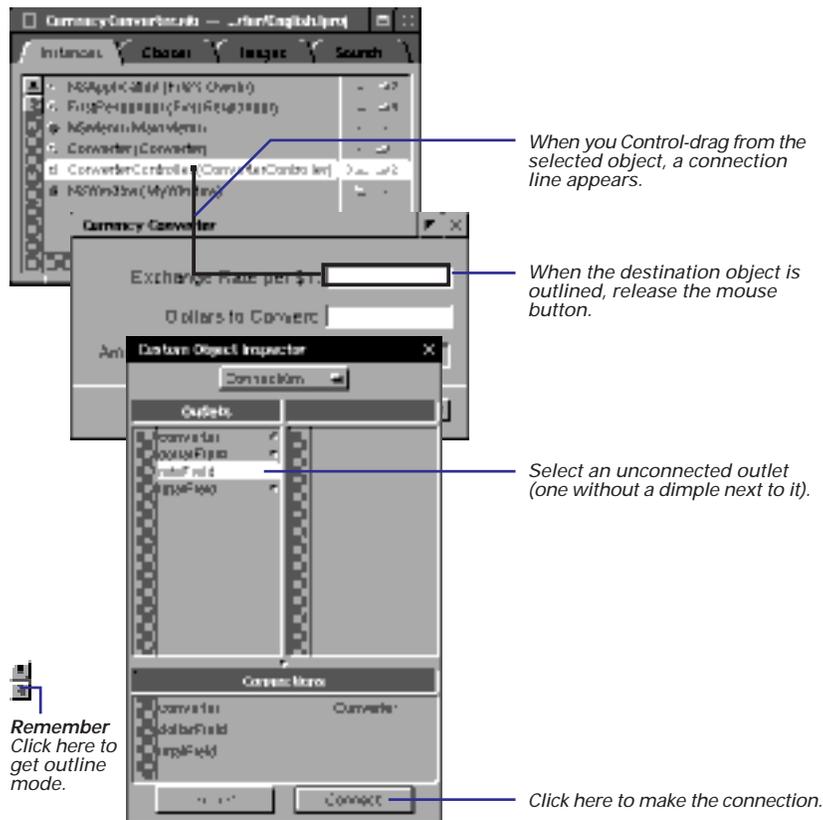


Making connections in outline mode

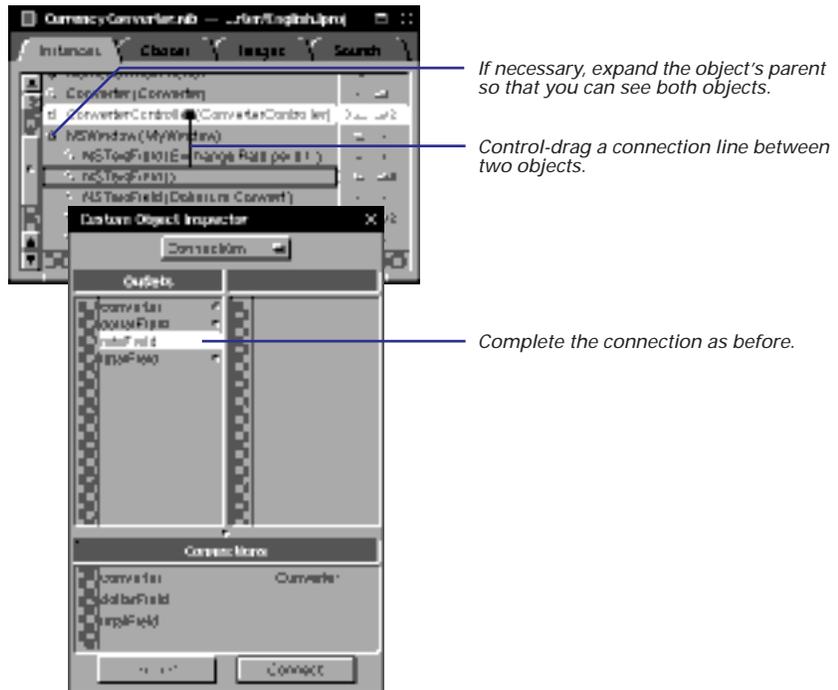
- 1 **Select an object.**
- 2 **Control-drag a connection to another object.**
- 3 **In the Inspector panel's Connections display, select an outlet or action.**
- 4 **Click the Connect button.**

You can make connections between objects in the outline mode of the Instances display as well as its icon mode. The connections can be between an object in the outline and an object in the interface or between two objects listed in the outline.

Before you make a connection involving an object in outline mode, make sure that the object is visible in the display. (You might have to expand the object's “parents” in outline mode to do this.)



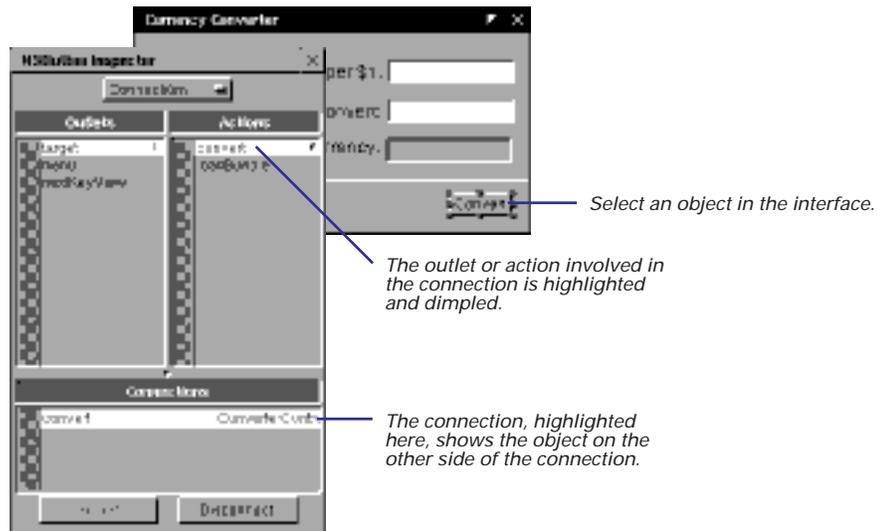
The outline mode offers a useful capability for making connections without leaving the nib file window. In this example, the same connection is made as in the previous example.



Examining connections

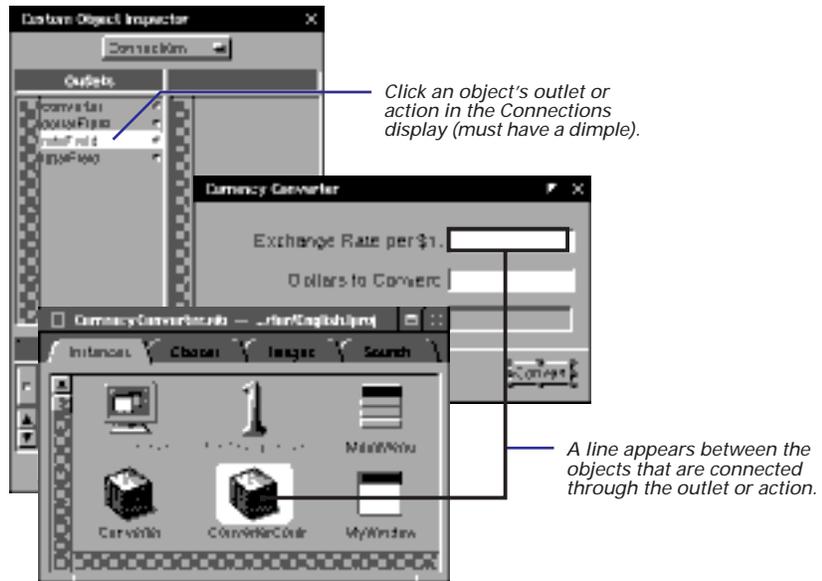
- ▶ **In the interface:**
Select an object and look at the Connections display of the Inspector panel.
- ▶ **In the Instances display:**
Select an object and look at the Connections display of the Inspector panel.
- ▶ **In the Connections display:**
Click a dimpled outlet to see the connection line drawn.
- ▶ **In outline mode:**
Click a triangle button in the column to the right of an object.

Interface Builder gives you many ways to examine and verify connections between objects. It makes it easy, for example, to discover what outlets and actions are associated with an object in the interface.

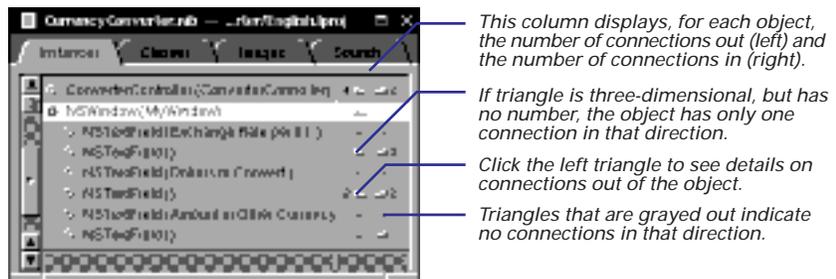


You can also select an object in the Instances display (in both icon and outline modes) and examine the Inspector panel as described above to find out what object it is connected to.

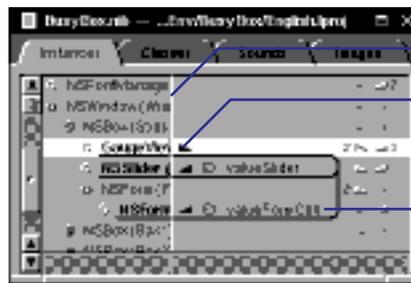
You can also examine object connections going in the other direction too, from the Connections display to the interface and the Instances display.



The Connections display allows you to see one connection at a time. The outline mode of the Instances display shows you *all* connections an object has, both connections into the object and connections from that object to other objects.



When you click a three-dimensional triangle, lines appear to show the connections between objects. The name and class of each connected object is highlighted in bold. Each connection is labelled with the name of an outlet or action.



To see more of a column, drag the column divider sideways.

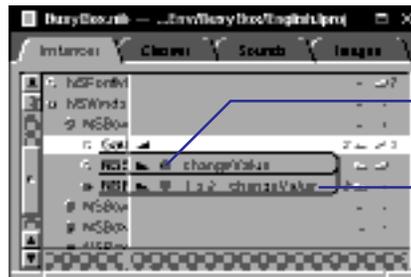
The right-pointing triangle indicates connection-out. Lines show you where the connections lead to.

The left-pointing triangle indicates connection-in. The electrical outlet icon represents an outlet; the name of the outlet follows.

⚙ Indicates action.

⚡ Indicates outlet.

Note that an object may have multiple connections with another object, both in and out, both outlets and actions. In these cases, the outline mode lets you toggle between the connections.



The cross-hairs icon represents a connection involving an action.

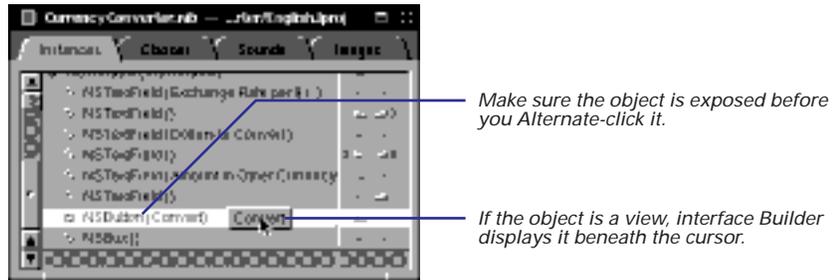
Connections with colon-separated numbers indicate multiple connections (here it means "1 of 2"). Click the colon to toggle between the connections.

To make the connection lines disappear, click the triangle button again.

Identifying objects in outline mode

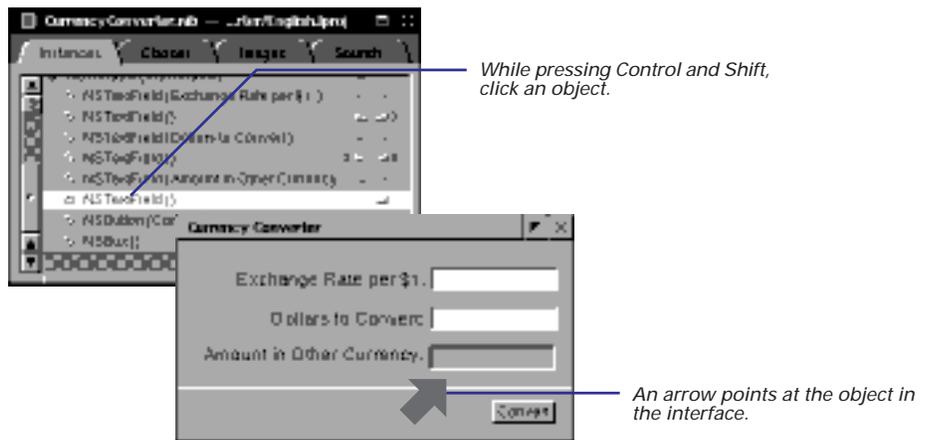
- ▶ To see a representation of an object, Alternate-click it in outline mode of the Instances display.
- ▶ To have an arrow point at the interface object, Control-Shift-click the object in outline mode.

In the outline mode of the Instances display, you might want to verify what an object is before connecting it to another object. You have two graphical ways to identify an interface object. One technique displays an image representing the selected object.



When you Alternate-click non-view objects in outline mode, the images that represent them in icon mode are displayed (cubes for custom objects, mini-windows for panels and windows). The File's Owner, First Responder, and Main Menu objects don't display icons.

The second technique locates an object in the interface with a large arrow.



See “The Modes of the Instances Display” in this chapter for an introduction to outline mode.

Control-Shift-Clicking the File's Owner, First Responder, and Main Menu objects has no effect.

Standard Objects in the Instances Display: File's Owner, First Responder, and Font Manager



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, "Dynamic Loading," 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.

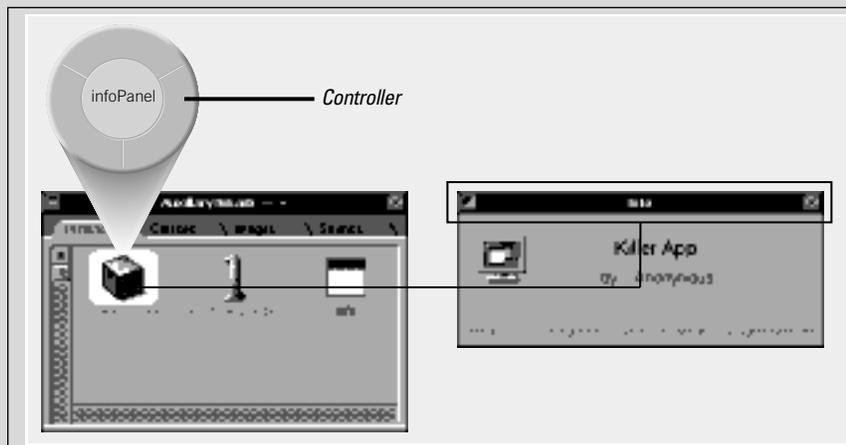


First Responder

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.

File's Owner



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'sSuperview, 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.

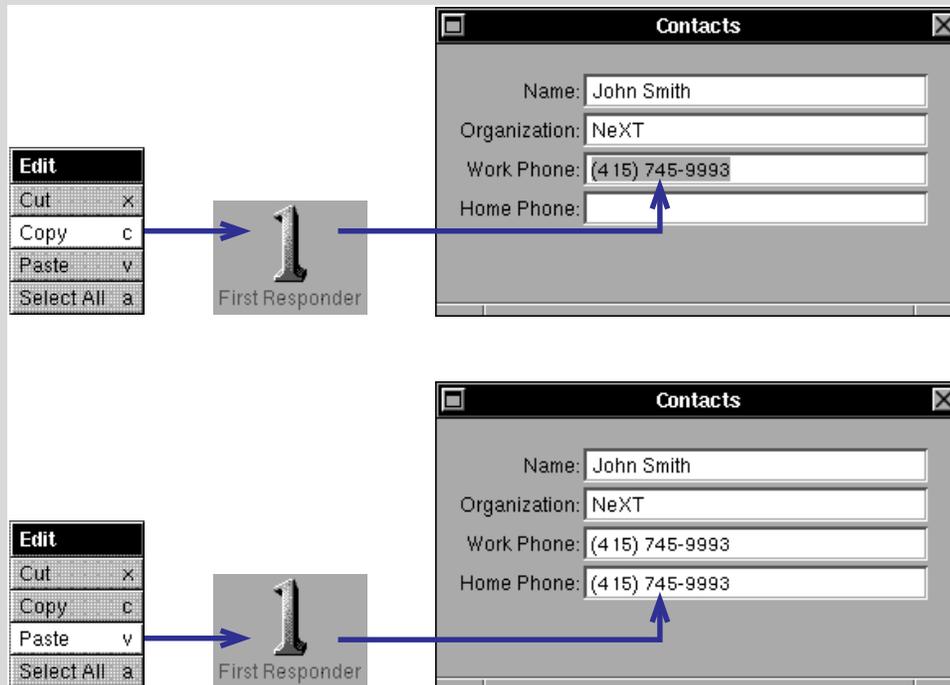
In Interface Builder you can connect an `NSControl` object in the interface to the First Responder icon. Thereafter, when the user manipulates this `NSControl` (say, by clicking a menu item entitled Copy) an action message (**copy:**) is sent to the object that is currently First Responder. If you examine 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



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.



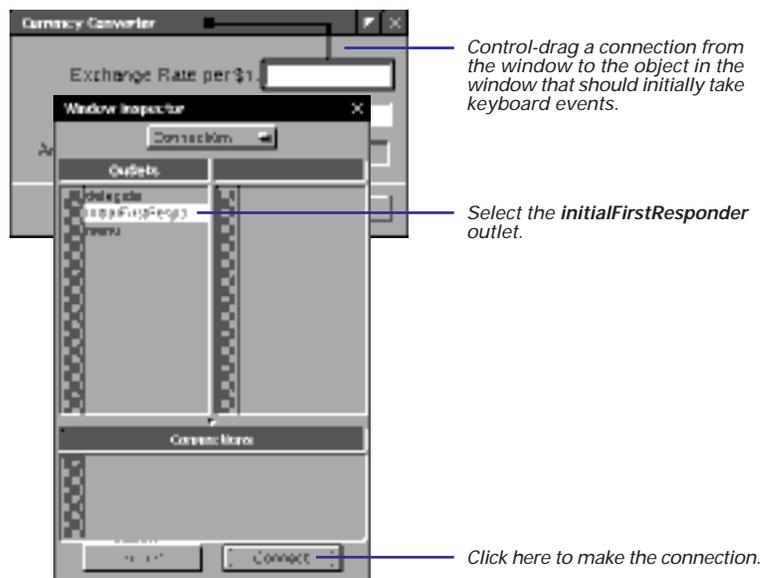
Enabling inter-field tabbing

- 1 Control-drag between the window and a view object.
- 2 In the Connections display of the Inspector panel, select `initialFirstResponder` and click **Connect**.
- 3 Control-drag between two view objects.
- 4 In the Connections display, select `nextKeyView` and click **Connect**.

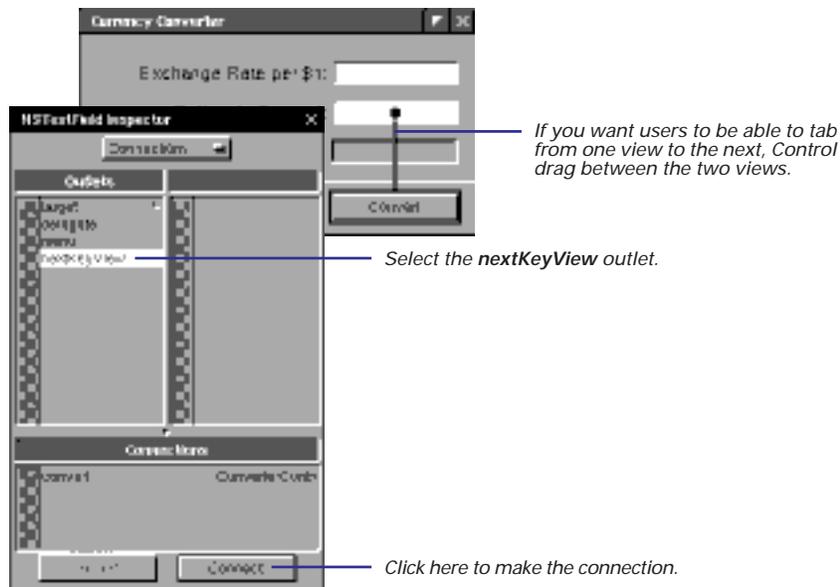
In OPENSTEP applications, users can navigate between fields and controls on the interface solely through use of the keyboard. Users can change the first responder by pressing the Tab key or Shift-Tab, can navigate through cells in a matrix by pressing the arrow key, and can change the state of a button or select a cell in a matrix by pressing the Spacebar.

You get most of this keyboard navigation feature in your application for free; you don't have to do anything special to allow users to navigate between cells in a matrix or fields in a form. However, you'll want to control what the Tab key does, that is, which view the cursor should go to next when the user presses the Tab key. You do this by connecting `NSView` objects to each other through the `nextKeyView` outlet.

First, decide which view should respond to keyboard events when the window becomes key, and connect the `NSWindow` `initialFirstResponder` outlet to that view.



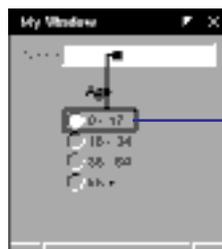
Next, use `NSView`'s `nextKeyView` outlet to connect view objects to each other.



Don't connect views that the user cannot select or edit. In the example above, we skip over the gray text field because it exists to show the result of the Convert button's action. The user cannot enter text into this text field, so it does not make sense to make a `nextKeyView` connection to it. You also should be careful not to connect to `NSCell` objects. For example, you shouldn't connect to an individual cell of a matrix or form; instead hook the preceding object to the entire matrix or form. The `NSMatrix` and `NSForm` objects determine the keyboard navigation between their own cells.

You should also assign key equivalents to buttons. The default button typically has a Return key equivalent, and the Cancel button typically has the Esc key equivalent. See Chapter 3 for more information.

If you don't make `nextKeyView` connections, default connections are made at run time. You can use Interface Builder's Test Interface command to see if these connections are satisfactory. See "Testing the interface" in this chapter.



WRONG:
Connecting an `NSView` to an `NSCell`.



RIGHT:
Connecting an `NSView` to an `NSView` (`NSMatrix` in this case).

Disconnecting objects

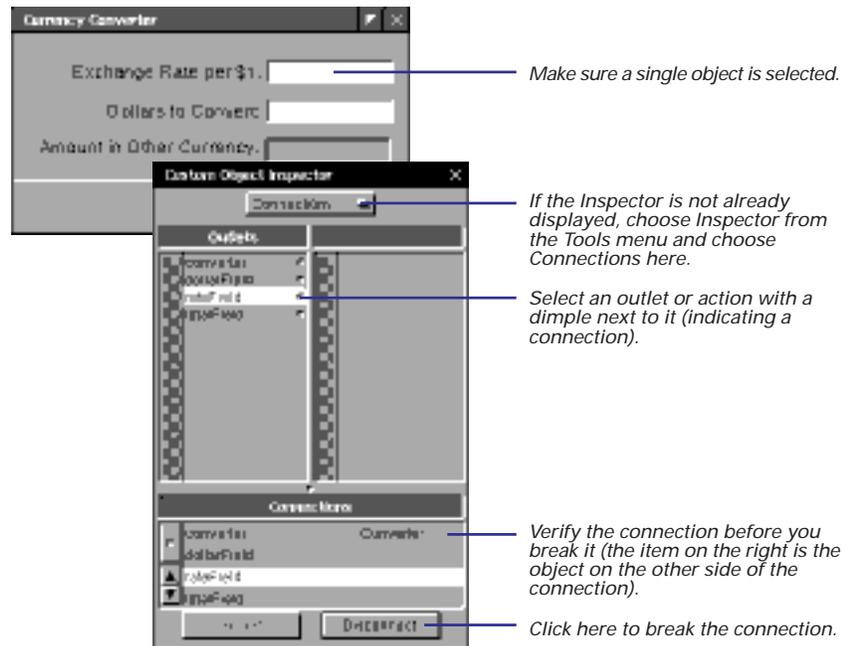
- 1 **Select an object in the interface.**
- 2 **In the Connections display of the Inspector panel, select a connection.**

- 3 **Click Disconnect.**

Or

- 1 **In the nib file window's outline mode, click a triangle button to display a connection.**
- 2 **Control-click the connection line.**

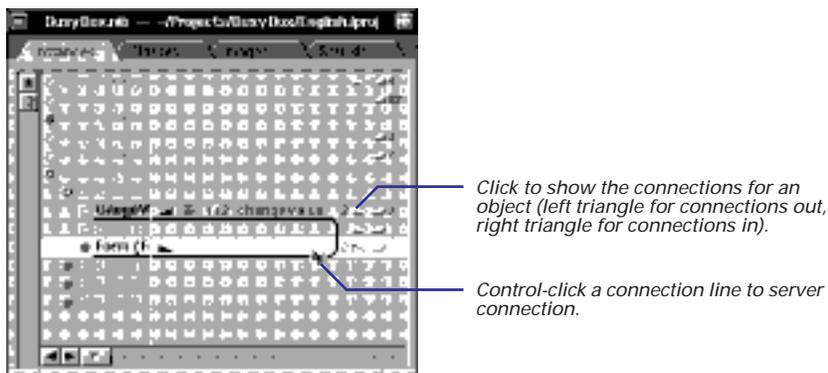
Interface Builder gives you two ways to break the connections between objects. The first method uses the Inspector panel.



You can also initiate this procedure by selecting objects in icon mode of the Instances display, and then disconnecting them in the Inspector panel as above.

See “Examining connections” in this chapter to learn how to use outline mode to display the connections between objects.

The alternative method for disconnecting objects allows you to perform the operation in one place: in the outline mode of the nib file window's Instances display. First show connections for an object by clicking a three-dimensional triangle button.



Click to show the connections for an object (left triangle for connections out, right triangle for connections in).

Control-click a connection line to sever connection.

You must Control-click on the *right* side of the column divider (nearest the connection-out and connection-in triangle buttons) to get the scissors to appear, and thus be able to break the connection. When you Control-click on the *left* side of the column divider, it begins a connection operation.

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,

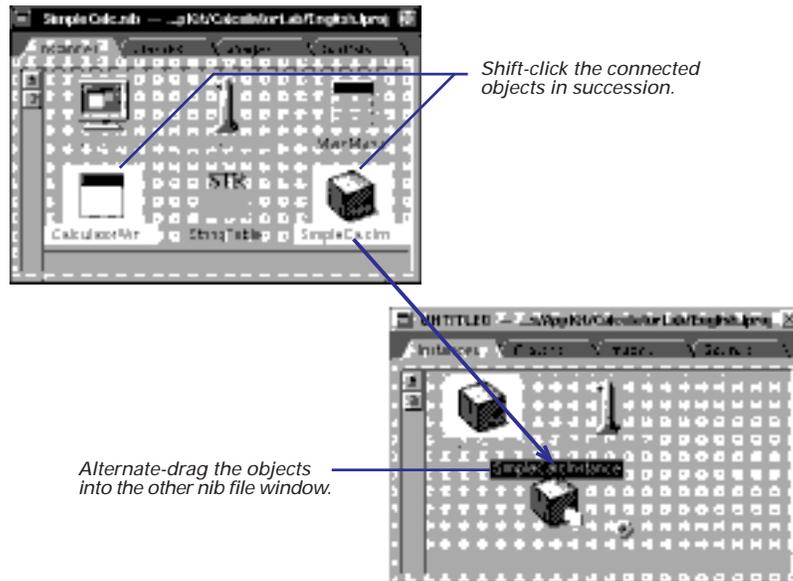
disconnect objects, or delete objects that have connections.

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.

Copying interconnected objects

- 1 **Select the objects that are connected.**
- 2 **Alternate-drag the objects into another nib file window or onto another window or panel.**

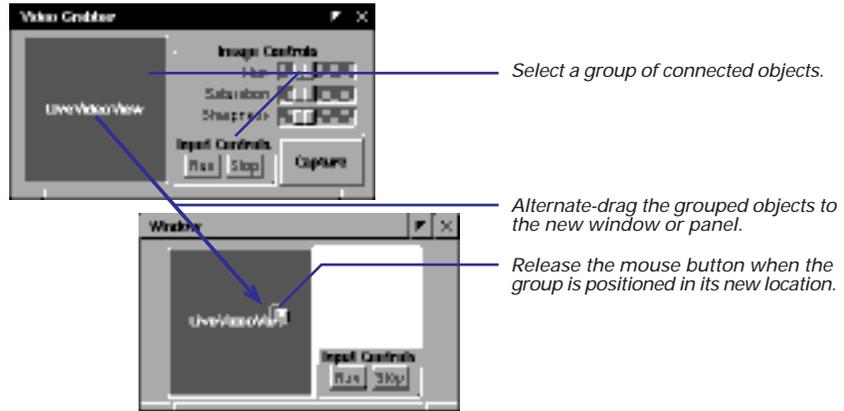
You can easily copy objects—with their connections—between nib files. You'll probably use this feature most often to copy a window and its views along with the custom object that manages those views.



Notice the icon representing the copied objects in the example above. Under the cursor is the icon representing the object that is actually dragged. The plus sign indicates that more than one object is involved in the operation. When the copying process completes, the new nib file window holds duplicates of the objects that include their connections to each other.

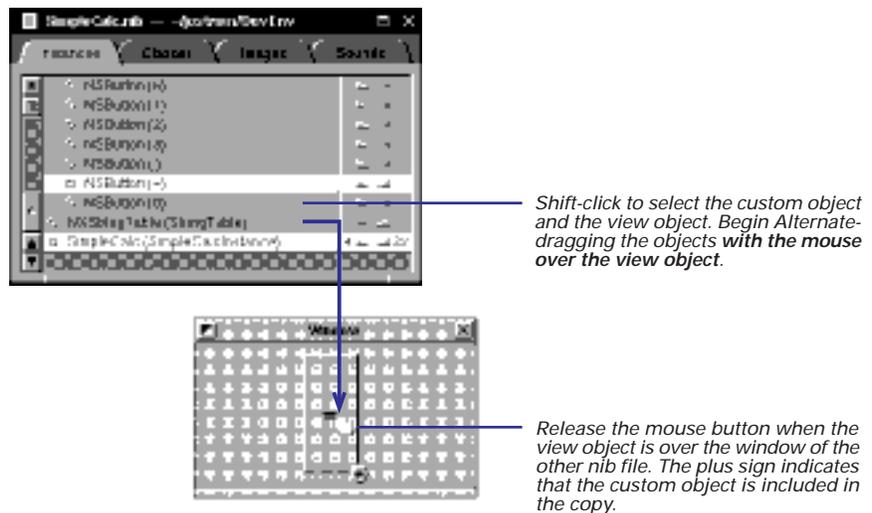
You can use the same basic technique to copy connected objects on an interface. In the next example, an instance of an `NSView` subclass is connected to the Run and Stop buttons. You can copy these objects and their connections by Alternate-dragging them onto a window in another nib file.

The various scenarios for copying objects and their connections between nib files is quite similar to the procedures for copying objects to dynamic palettes. See Chapter 5, “Using Dynamic Palettes,” for more information on this Interface Builder feature.



Another occasion for copying connected interface objects is when you want to make copies of text fields or form fields and preserve the connections between fields.

From the outline mode of the Instances display, you can copy an individual view object, a custom non-view object, and the connections between the two.



You can also copy interconnected interface objects to another window in the same nib file. See “Moving objects to other windows” in Chapter 2.

Testing the interface

- 1 **Choose Test Interface from the Document menu.**
- 2 **Check the functioning of OpenStep objects.**
- 3 **Choose Quit from the application menu or double-click the switch icon in the application dock.**

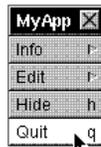
After you create an interface, Interface Builder lets you see how it works from the user's perspective.

Interface Builder's menu, windows, and panels disappear, leaving only the actual interface and (if you are testing the application's main nib file) the main menu. Give your interface a test ride. Here's some of the things you might try:

- Verify that the cursor moves from field to field when you press Tab.
- Verify that you can copy, cut, and paste text (First Responder actions).
- See if you can print (the Print menu item must be connected to an appropriate view object's `print:` action method).

Note: When you test your interface, the behavior provided by your custom classes is not called into play (with the exception of static, compiled palette objects). You can only test the behavior that OpenStep and static palette objects exhibit in themselves and when they send messages to each other. To test all components of your application, you must compile and run it.

When you are finished testing the interface, exit from test mode.



If testing the main nib file:

Click here to end test mode and return to Interface Builder.



If testing an auxiliary nib file:

Double-click the test mode icon in the application dock to exit test mode.

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.

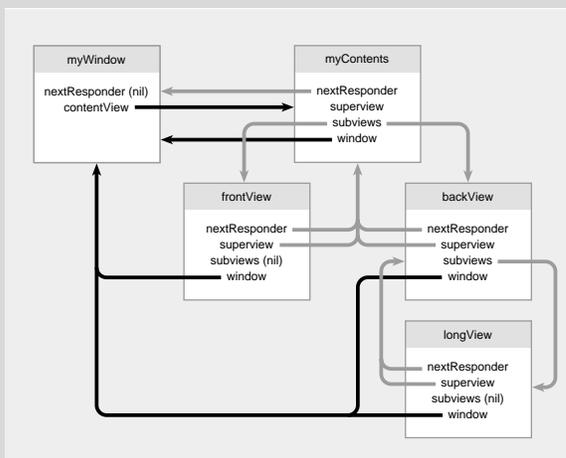
Within every window's content rectangle—the area enclosed by the border, title bar, and resize bar—is its *content view*. The content view is at the top of the view hierarchy. All other views of the window descend from it. Each view has one other view as its *superview* and can be the *superview* for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

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:

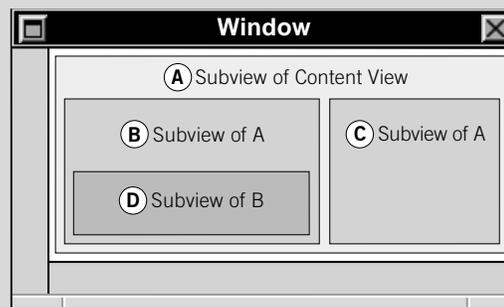
- **window** identifies the view's window (the window points to the content view)
- **superview** identifies the view's superview
- **subviews** a list of the view's subviews

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. The **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` and `NSResponder` classes in the *Application Kit Reference* for more information on the view hierarchy and event handling.

5

Using Dynamic Palettes

Creating and saving dynamic palettes

Storing view objects on dynamic palettes

Arranging objects on dynamic palettes

Storing top-level objects on dynamic palettes

Putting connected view and top-level objects on a dynamic palette

Managing palettes

Who hath not seen thee oft amid thy store?

Sometimes whoever seeks abroad may find
Thee sitting careless on a granary floor,
Thy hair soft-lifted by the winnowing wind...

John Keats, from *To Autumn*

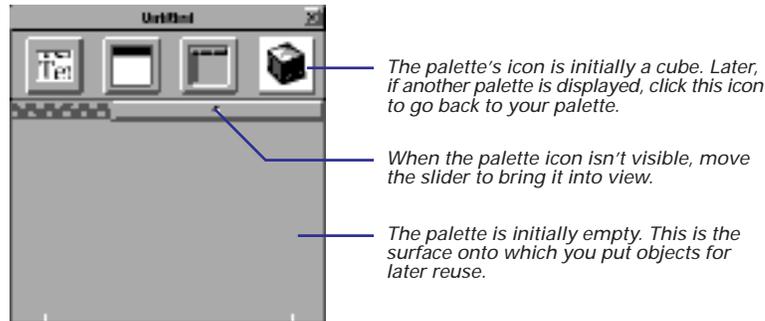
The superfluous is very necessary.

Voltaire

Creating and saving dynamic palettes

- ▶ To create a palette, choose **Tools ▶ Palettes ▶ New**.
- ▶ To save a palette, choose **Tools ▶ Palettes ▶ Save**.

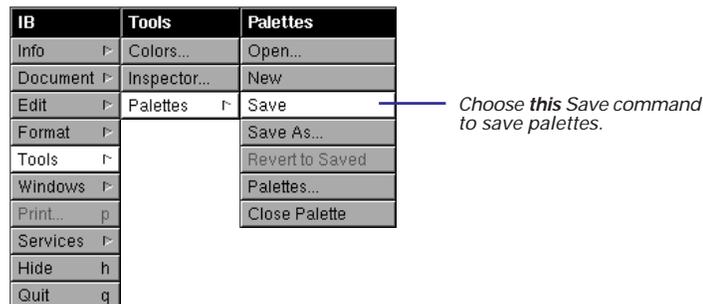
When you create a dynamic palette, an empty palette appears in the Palette window.



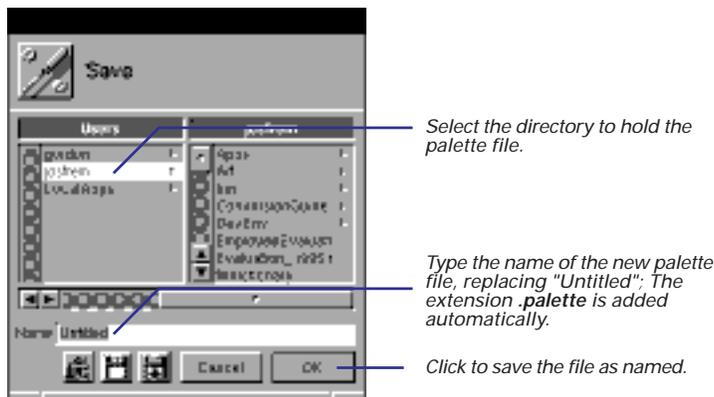
You can customize the icon for your dynamic palette. The task “Managing palettes” in this chapter tells you how to do this and also describes how to unload palettes in Interface Builder.

As with the standard Application Kit palettes, you can choose an existing dynamic palette by clicking its icon in the Palette window (when created, dynamic palettes have the generic cube icon). To use an object on a dynamic palette, follow the same procedure as for objects on the standard palettes: Drag the object from the palette and drop it onto an appropriate surface.

You must save your dynamic palettes. If you do not save a palette after you create it, you lose it when you quit Interface Builder. (Interface Builder prompts you if you try to quit without saving a palette.) Choose the Save command to save dynamic palettes, *but the Save command from the Palettes menu, not the Document menu*.



Interface Builder brings up the Save panel, allowing you to designate a name for the palette.



Tools for Interface Crafters: Static and Dynamic Palettes

A palette is a special display that holds one or more reusable objects. You can drag these objects from the palette to your application's interface. There are two types of palettes: static and dynamic. To the user, they seem identical, but the differences are many.

Static palettes are built as a project and have code defining their objects; dynamic palettes include no special code—they're unique configurations of (mostly) standard OpenStep objects. Consequently, static palettes must be compiled, but you can create dynamic palettes on the fly, without writing and compiling code. Objects on static palettes can have inspectors and editors, which dynamic-palette objects cannot have.

Creating static palettes (and their inspectors and editors) is a more complex process than creating dynamic palettes, but the resulting product has more value added to it. For example, if you want to store a button that has the title OK, you use a dynamic palette because the change involves only the Interface Builder Inspector panel. However, if you want to store a custom subclass of `NSButton`, you use a static palette. A static palette can store both the button and your custom code.

Dynamic palettes are a great convenience. You can save collections of your objects, with or without their interconnections,

to a dynamic palette at any time. You can save dynamic palettes and store them in the file system, just as you do with the traditional compiled palette. You can remove the palette from the Palette window and, when you need it again, just load it back into Interface Builder.

The possible practical uses of dynamic palettes are numerous. You can use them to:

- Store collections of often-used view objects configured with specific sizes and other attributes.
- Hold windows and panels that are replicated in your projects (such as Info panels).
- Store versions of interfaces.
- Keep interconnected objects as a template that you can later use as-is or modify for particular circumstances. For instance, you could store a group of text fields and their delegate, or a set of controls and their connections to a controller object.

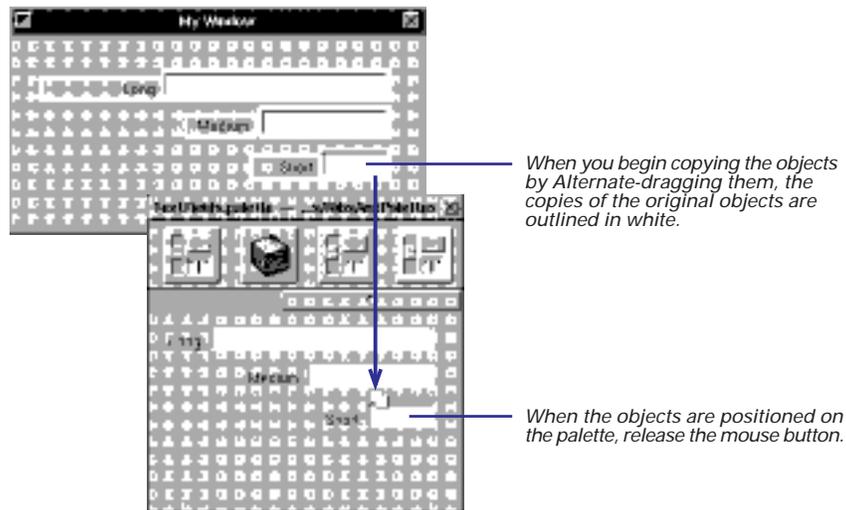
You can also use dynamic palettes for prototyping and group work. For instance, you could design an interface or a part of an interface, store the objects on a dynamic palette, and then mail the palette file to all interested parties.

Storing view objects on dynamic palettes

- 1 **Configure one or more view objects.**
- 2 **Select the objects.**
- 3 **Alternate-drag them to the palette.**

You can save any single view object or group of view objects to a dynamic palette so that you can use them again. The objects stored in this manner preserve the size and other attributes they have when you store them.

First, size each object to be stored and, through the Inspector panel, set its important attributes. If you are storing a group of objects, such as a set of controls within a box, make sure to position all objects in proper relation to each other.



When you begin copying the objects by Alternate-dragging them, the copies of the original objects are outlined in white.

When the objects are positioned on the palette, release the mouse button.

If there are several objects you want to store on a palette, you can drag each object onto the palette individually, or you can make a multiple selection in the interface. (Draw a selection rectangle around the objects, or Shift-click the objects in succession.) If you store a group of objects that are connected—say the three fields above were grouped in a box and connected through the **nextKeyView** outlet—their connections are copied also.

Tip: You can also copy view objects to a dynamic palette from the outline mode of the Instances display.

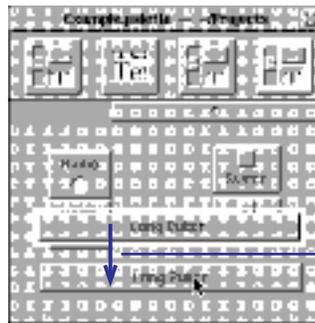
Once an object is on a palette, you can move it around the palette or remove it from the palette. See the next task, “Arranging objects on dynamic palettes” for details.

You can also store view objects that are connected to top-level objects. See “Storing top-level objects on dynamic palettes” for more information.

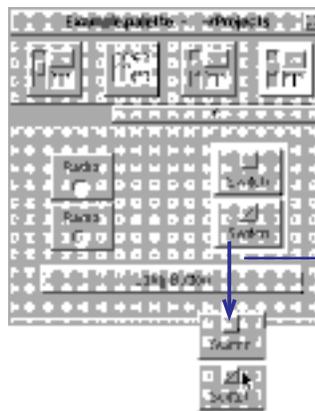
Arranging objects on dynamic palettes

- ▶ To position a palette object, Alternate-drag it within the palette.
- ▶ To remove a palette object, Alternate-drag it off the palette.

You can do two things to objects once they're on a dynamic palette: move them around the palette and delete them from the palette.



To reposition an object, Alternate-drag it to its new location on the palette.



To remove an object, Alternate-drag it until it is off the palette, then release the mouse button.

If you store an object on a dynamic palette and later discover that its size, connections, or attributes must be changed, you must:

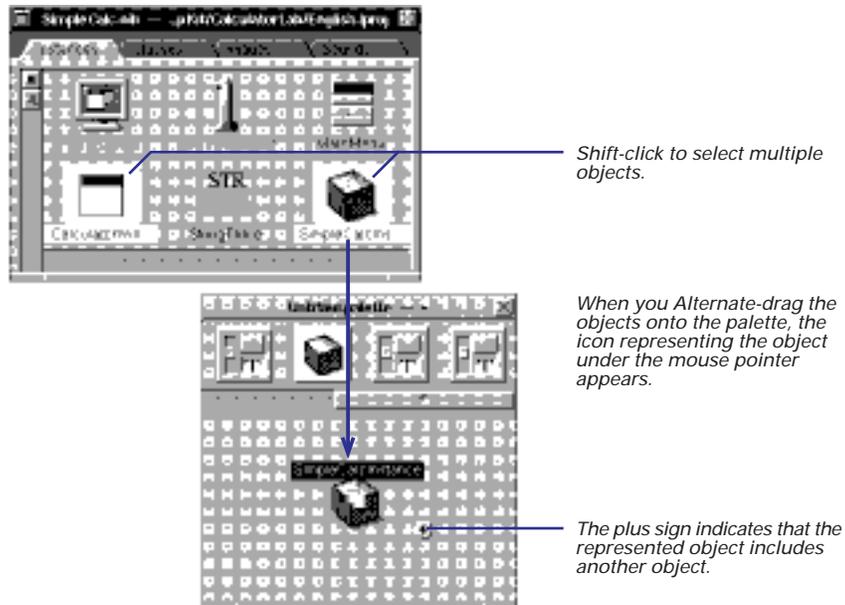
- Remove it from the palette.
- Resize it, reconnect it, or reset its attributes.
- Store it on the palette again (by Alternate-dragging it).

Storing top-level objects on dynamic palettes

- 1 In the Instances display, select one or more windows, panels, or custom objects.
- 2 Alternate-drag them and drop them on a dynamic palette.

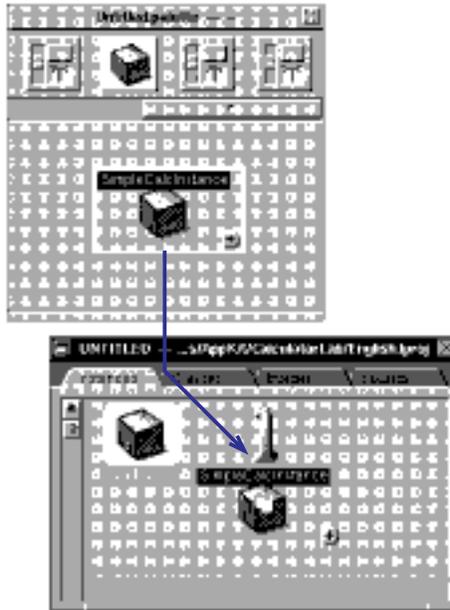
You can put custom objects, windows, and panels on dynamic palettes and reuse them again and again. You can store these top-level objects individually or as connected sets of objects. When you select a controller object and a window and store them together, the connections between them are also stored on the dynamic palette. In addition, all connections between a window or panel and its views are preserved as well as the connections among the views themselves.

To store a single top-level object on a dynamic palette, Alternate-drag it from the Instances view of the nib file window and drop it onto the palette. To store multiple, connected objects, make sure they're selected as a group first.



Tip: You can perform this same task whether in the outline or the icon mode of the Instances display.

When you drag the object or objects from the dynamic palette to add them to another nib file (or to duplicate them in the same nib file), make sure that the “surface” on which you drop the object (as represented by the icon) is compatible.



To reuse the connected objects, drag the object and drop it over a suitable surface.

You'll know which surface is compatible by the icon representing the object. If it's a cube (custom object), you must drop it over the nib file window. If it's a window or panel, you can drop it anywhere on the screen, including over the nib file window.

Putting connected view and top-level objects on a dynamic palette

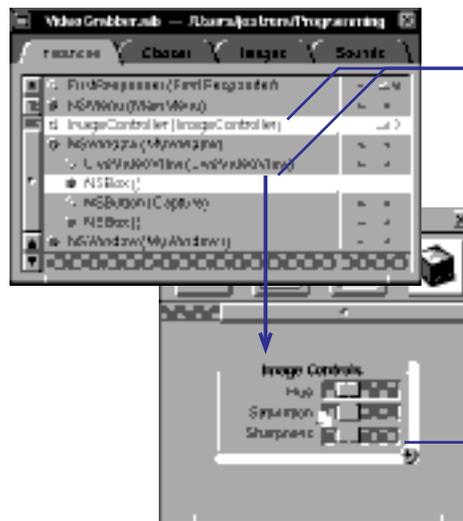
- 1 In outline mode, select a connected top-level object and a view object.
- 2 Alternate-drag the objects to the palette.

There might be situations when you don't want to store an entire window with the custom object that manages that window's views. You just want to store the custom object and some of the window's views, or you want to store the window and only some of its views. You can do this from Instances view outline mode.

For example, here's how three slider objects hooked up to a controller object (ImageController) look in outline mode when the connections are displayed.



Select the top-level object and view object by Shift-clicking them. You can select only one view object and one top-level object.



Select a top-level object and a view object connected to it. (In this case, the view is a box containing several subviews.)

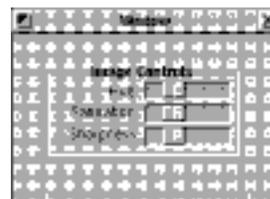
When you Alternate-drag the selected objects onto the dynamic palette, a representation of the object under the mouse pointer appears.

The plus-sign icon indicates that this palette object contains multiple objects, including all their connections with each other.

Because you can only store one view object per top-level object with this technique, you first might want to group all view objects you want stored (if they're not already grouped). To make a group, select all the objects and then choose **Format** ► **Group** ► **Group in Box**. If you don't want the enclosing box around the grouped objects, remove the **Bordered** option in the Inspector's **Attributes** display for boxes.

Tip: A useful selection technique is to first click a view object in the interface, then choose **Enter Selection** from the **Edit** menu. The view object becomes highlighted in the nib file window.

To verify the objects you stored, first drag the view object (with plus sign) that represents the combined objects from the dynamic palette and drop it over a new window or panel. Go to outline mode of the **Instances** view. The top-level and view objects you dragged off the palette are listed in the outline. Click the triangle button to verify that the connections are still there.



Managing palettes

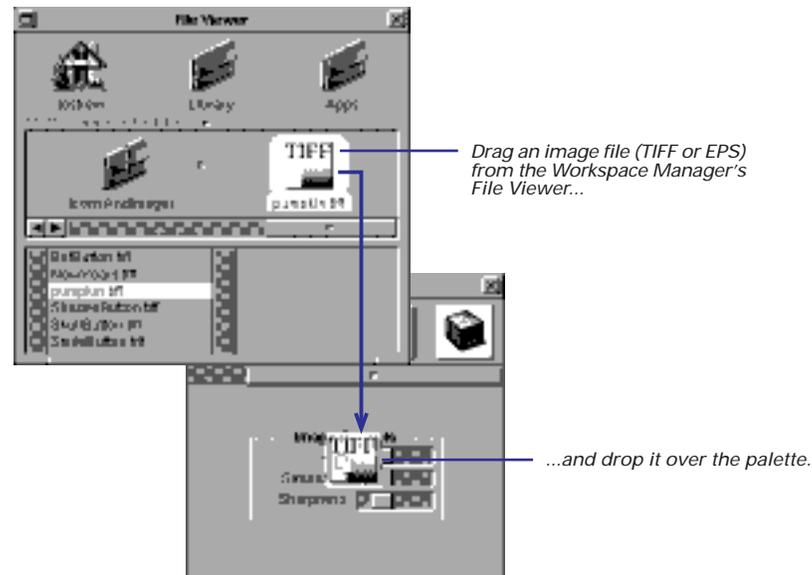
- ▶ **Customize the palette icon:**
Drag and drop an image file over the palette.
- ▶ **Install and uninstall palettes:**
Double-click a palette icon in the Palettes display of the Preferences panel.
- ▶ **Load a palette:**
Choose Tools ▶ Palettes ▶ Open and select a palette file.
- ▶ **Unload a palette:**
Alternate-drag a palette icon off the Palettes display of the Preferences panel.

Interface Builder gives you facilities for managing static and dynamic palettes. These management functions include:

- Customizing the palette icon (dynamic palettes only)
- Installing and uninstalling palettes on the Palette window
- Loading and unloading palettes

Customizing the Palette Icon

For dynamic palettes, Interface Builder gives you the option of specifying the icon that identifies the palette in the Palette window.

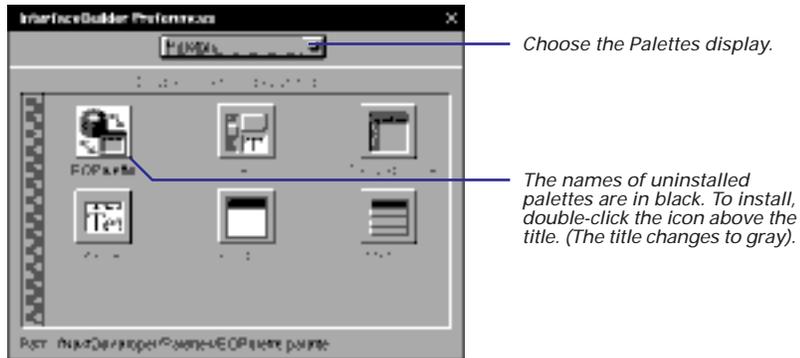


Tip: Since Interface Builder scales the image to fit into the icon rectangle, create or choose a TIFF or EPS image that is 35 X 35 pixels.

You can also customize the icons of static palettes, but you must do this programmatically, specifying the image file in a `palette.table` file.

Installing and Uninstalling Palettes

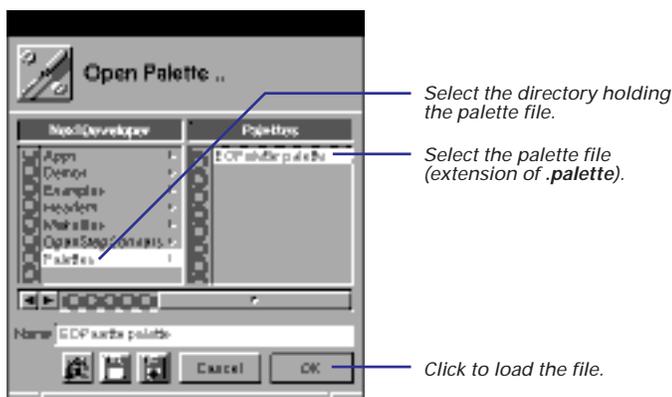
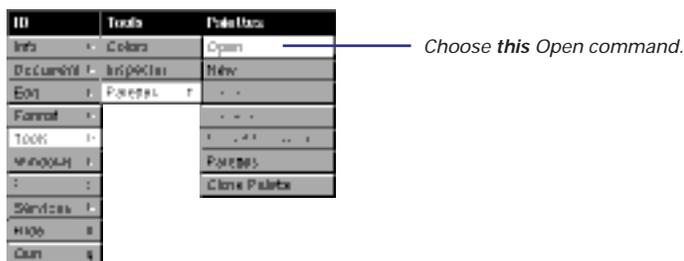
Palettes can be installed or uninstalled; installed palettes appear in the Palette window. Interface Builder stores as a user preference the names of the palettes you have previously installed and whether you currently want them installed. You can find out which palettes are uninstalled in the Palettes display of the Preferences panel. To see this display, choose Preferences from the Info menu.



To uninstall a palette, double-click its icon in the Preferences panel or use the Palette menu's Close Palette command.

Loading and Unloading Palettes

If the palette doesn't appear in the Preferences panel, open it using the Open command of the Palettes menu (not the Open command of the Document menu).



You can also open palettes by dragging them from the Workspace Manager's File Viewer onto the Palettes Preferences panel.

You unload palettes—thereby removing from your user preferences—by Alternate-dragging them off the Palettes display of the Preferences panel.

6

Subclassing

A roadmap to making or adding custom classes

Naming a new class

Specifying outlets and actions

Creating an instance of your class

Connecting your class's outlets

Connecting your class's actions

Generating source code files

Implementing a subclass of NSObject

Making your class a delegate

Implementing a subclass of NSView

Adding existing classes to your nib file

Updating a class definition

I inherited it brick, and left it marble.

Emperor Augustus

They rightly do inherit heaven's graces,
And husband nature's riches from expense.

Shakespeare, *Sonnets*

Observe how system into system runs,
What other planets circle other suns.

Alexander Pope, *An Essay on Man*

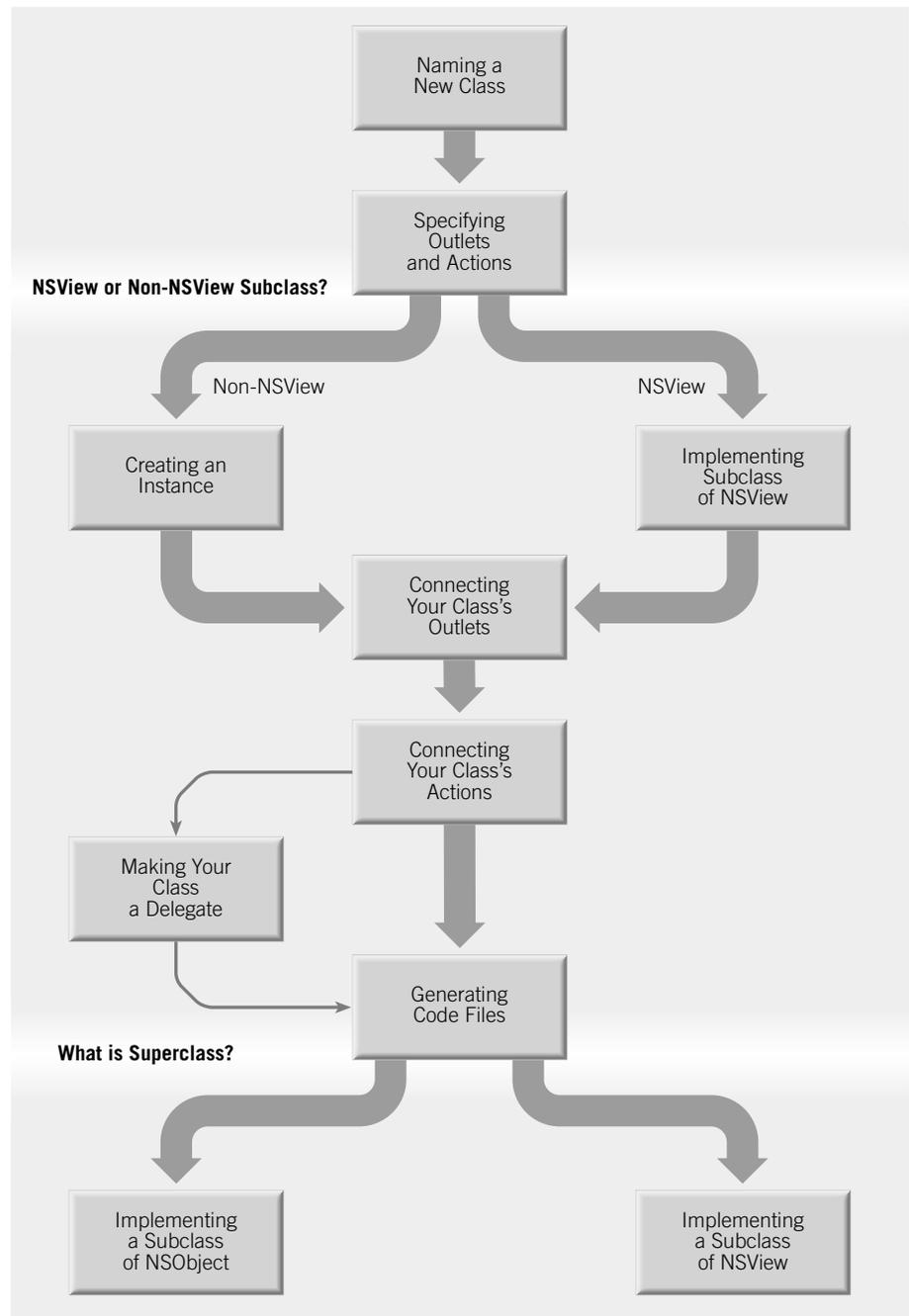
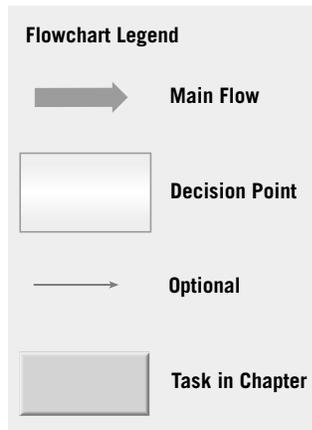
A roadmap to making or adding custom classes

- ▶ **Determine which flowchart applies to your situation.**
- ▶ **Follow the tasks in this chapter in the order specified by that flowchart.**

This chapter differs from the other chapters in this book because its subject is different. Creating a class (or adding an existing class) is not a set of discrete, modular tasks, but a process consisting of many interdependent tasks. The order of tasks in this chapter is therefore significant; with some exceptions, you need only follow the tasks sequentially, from first task to last task, and you'll end up with a useful class.

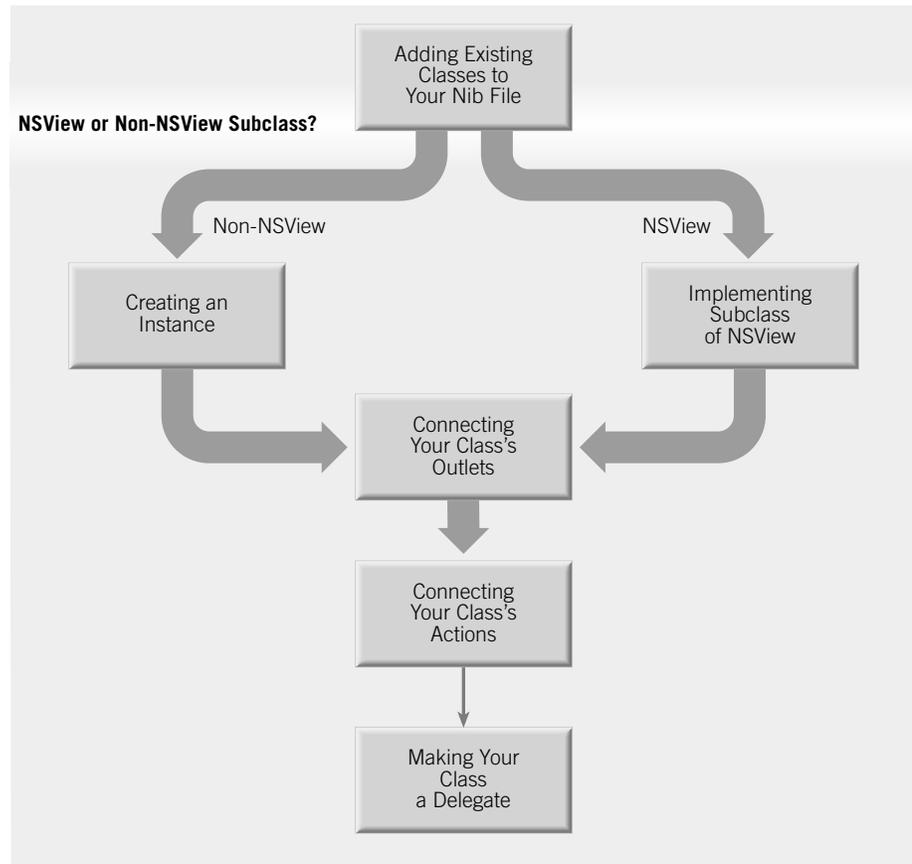
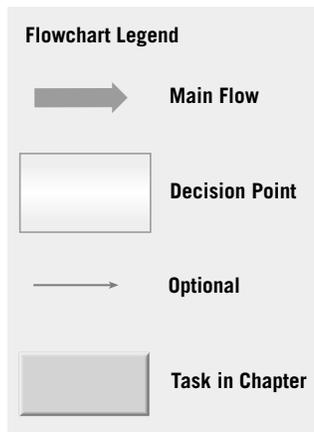
But those exceptions are significant, and so flowcharts are provided to point the way. The flowchart on the facing page guides you through the tasks required to define and implement a subclass of the NSObject class or of the NSView class. An additional flowchart identifies the tasks you must complete to integrate an existing class into an application.

This chapter also differs from other chapters in this book because it covers a topic that involves both Interface Builder and Project Builder. To start creating a class, you use Interface Builder. It helps you locate the class in the hierarchy, name it, connect an instance of it with other objects in an application, and generate template source files. When Interface Builder's role is done, you switch to Project Builder and provide the most important contribution, the source code that gives your class its distinctive behavior. (As an alternative, you can start creating a class in Project Builder then add it to Interface Builder and make the connections to other objects later.)



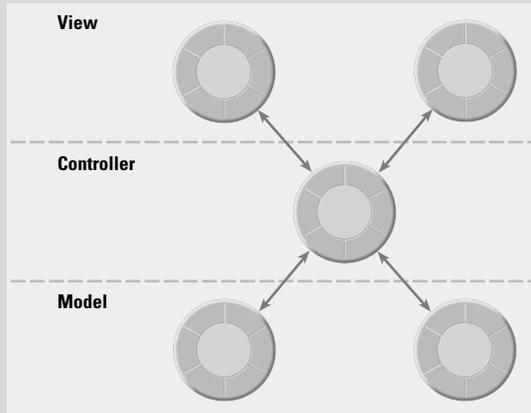
If you branch to “Implementing a subclass of NSView” after specifying outlets and actions, complete only the step “Making an Instance of an NSView Subclass” in that task for now, and go on to the next task. Do the rest of “Implementing a subclass of NSView” after you’ve generated code files.

After generating code files, you must switch over to Project Builder and open the header and implementation files.



The Model-View-Controller Paradigm

A common and useful paradigm for object-oriented applications, particularly business applications, is Model-View-Controller (MVC). MVC derives from Smalltalk-80; it proposes three types of objects in an application, separated by abstract boundaries and communicating with each other across those boundaries.



Model Objects

This type of object represents special knowledge and expertise. Model objects hold a company's data and define the logic that manipulates that data. For example, a Customer object, common in business applications, is a Model object. It holds data describing the salient facts of a customer and has access to algorithms that access and calculate new data from those facts. A more specialized Model class might be one in a meteorological system called Front; objects of this class would contain the data and intelligence to represent weather fronts. Model objects are not displayable. They often are reusable, distributed, and portable to a variety of platforms.

View Objects

A View object in the paradigm represents something visible on the user interface (a window, for example, or a button). A View object is "ignorant" of the data it displays. The Application Kit usually provides all the View objects you need: windows, text fields, scroll views, buttons, browsers, and so on. But you might want to create your own View objects to show or represent your data in a novel way (for example, a graph view). View objects, especially those in kits, tend to be very reusable and so provide consistency between applications.

Controller Object

Acting as a mediator between Model objects and View objects in an application is a Controller object. There is usually one per application or window. A Controller object communicates data back and forth between the Model objects and the View objects. It also performs all the application-specific chores, such as loading nib files and acting as window and application delegate. Since what a Controller does is very specific to an application, it is generally not reusable even though it often comprises much of an application's code.

Because of the Controller's central, mediating role, Model objects need not know about the state and events of the user interface, and View objects need not know about the programmatic interfaces of the Model objects. You can make your View and Model objects available to others from a palette in Interface Builder.

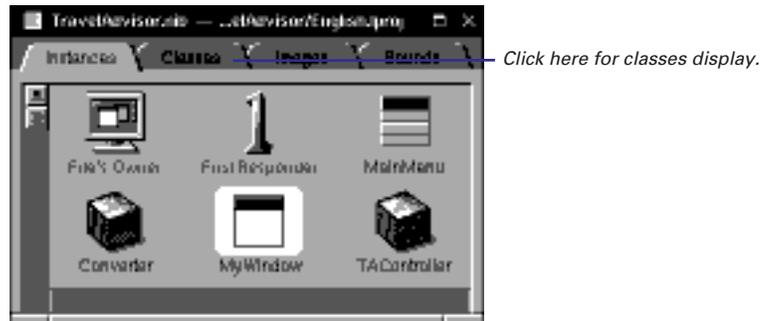
Hybrid Models

MVC, strictly observed, is not advisable in all circumstances. Sometimes its best to combine roles. For instance, in a graphics-intensive application, such as an arcade game, you might have several View objects that merge the roles of View and Model. In some applications, especially simple ones, you can combine the roles of Controller and Model; these objects join the special data structures and logic of Model objects with the Controller's hooks to the interface.

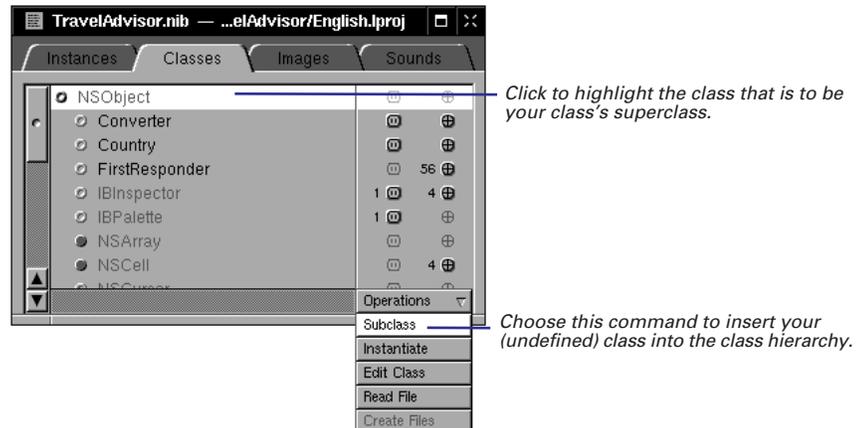
Naming a new class

- 1 In Interface Builder, display the **Classes** display of the nib file window.
- 2 Select the class you want your class to inherit from.
- 3 Choose **Subclass** from the **Operations** menu.
- 4 Type the name of your class over the highlighted “default” name.

When you create an application, you must create at least one subclass to do anything meaningful. The OpenStep frameworks do a lot of the work for you, but you must always supply, in one or more subclasses, the distinctive logical and computational flow of your application.



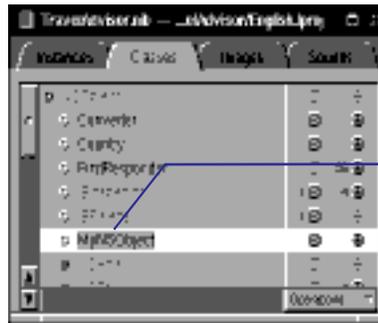
When you subclass, the first thing you must do is select your class's superclass. Ideally, the superclass of your class should behave much the way you want your class to behave. Your class merely adds the behavior you want to what the superclass offers, or modifies the superclass's behavior in some way. Often the behavior you want is so bound to resolving a particular problem that the proper choice of superclass is NSObject because it provides the most generic behavior.



See “A Short Practical Guide to Subclassing” in this chapter for more on the relation between superclasses and subclasses.

Tip: Pressing Return when a class is selected is equivalent to choosing the Subclass command.

The new class is listed under its superclass with a default name: the superclass name prefixed with “My” (such as “MyNSObject”). Replace this default name with the new name.



Type the name of your Class over the default name. Press Return.

Later, if you want to rename the class, first re-select the class name by double-clicking it. Then type the new name, replacing the selected text.

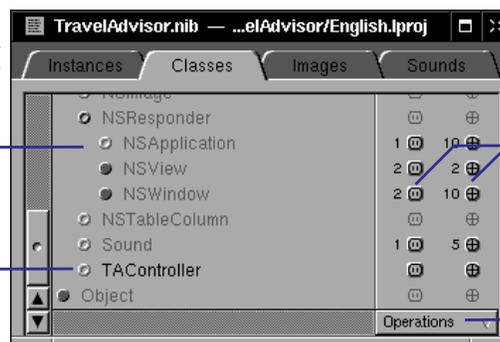
A Perspective on the Class Hierarchy

The Classes display of the nib file window shows the classes that the current nib file is aware of. The display lets you browse through both OpenStep classes and custom classes. The Classes display also depicts (by indentation) class-inheritance relationships and reveals the names of each class’s outlets and actions.

Keyboard Navigation Move up and down in the list of classes pressing the up arrow and the down arrow. When a class is highlighted, show its subclasses by pressing the right arrow; collapse an indented list by selecting the superclass and pressing the left arrow. If the nib file window is active, incremental search is active: just type the first few letters of a class until its name is highlighted.

The Classes display shows hierarchy by indentation (for example, `NSApplication` inherits from `NSResponder`). If the circle button is filled, the class has subclasses that are not shown. Click the button to display the subclasses.

If the class name is black, it is a custom class. If the class name is gray, the class is a NeXT-provided class.



Outlet (electrical-outlet icon) and target/action (cross-hairs icon) buttons. Click to display class outlets and actions.

A pull-down list of operations related to creating a class.

Specifying outlets and actions

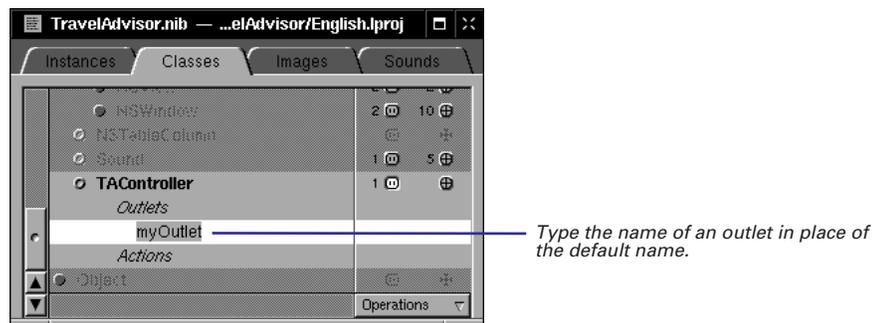
- 1 Click the button for an outlet or an action.
- 2 Select *Outlets* or *Actions*.
- 3 Choose the appropriate command from the Operations menu.
- 4 Enter the name of the outlet or action in place of the default name.

An object isolated from other objects is of little use. Interface Builder provides two ways for you to specify how objects of your class communicate with other objects: outlets and actions.

Before you begin this task, take a moment to consider what other objects you want instances of your class to send messages to, and the requests that instances of your class are apt to receive from other objects. The procedure itself is simple, and almost identical for outlets and actions.

Adding Outlets

Outlets are instance variables that identify other objects. In the Classes display, you access the outlets of a class by clicking the electrical-outlet button.



For background information on outlets and actions, see “Communicating With Other Objects: Outlets and Actions” in Chapter 4, “Making and Managing Connections.”

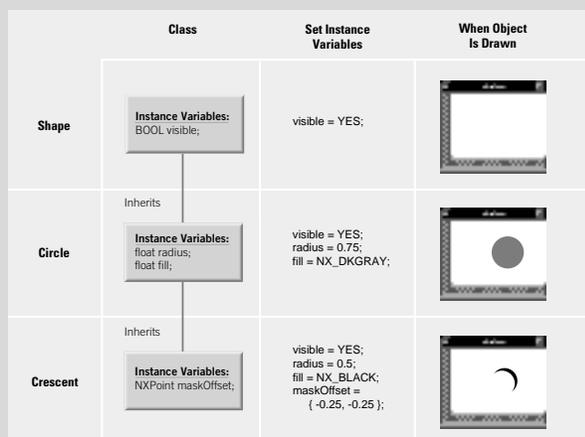
When you press Return, the outlet is renamed and Interface Builder highlights the new outlet. If you have another outlet to specify, choose Add Outlet again from the Operations menu and type the outlet’s name over the default name.

A Short Practical Guide to Subclassing

Subclassing is not an esoteric art but one of the most common and essential tasks in object-oriented programming. But it doesn't need to be a difficult chore, especially if you take the time to learn what's in the class hierarchy.

What is Subclassing?

The principal notion behind subclassing is inheritance. Classes stand in relation to other classes as child to parent or parent to child. A class might have many child classes (or subclasses), but always has only one parent class (superclass). At the head of this class hierarchy is the root class.



The attributes (instance variables) and behavior (methods) defined by a class are shared by all descendents of that class. To put it another way, each new class is the accumulation of all class definitions in its inheritance chain.

For example, the `NSView` class defines two instance variables for location and size (**frame** for the superview orientation, and **bounds** for within the view) from which all instances of its numerous subclasses derive their own basic position and dimensions. The `NSView` class also defines several methods for setting and getting these instance variables; again, all subclasses of `NSView` inherit the behavior defined by these methods. You can send the same messages to any instance of an `NSView` subclass to have it resize itself.

So subclassing is usually the extension and specialization of the inheritance chain. When you define a class that inherits from another class, you are specifying how it differs from that superclass.

But there are reasons for creating a subclass—or a “branch” of subclasses—other than getting different behavior. You may want to define a class that dispenses generic functionality to its subclasses, such as an `Output` class that performs tasks common to both a `Printer` class and a `Fax` class. You might want a class to declare methods (perhaps unimplemented) that set up a protocol that future subclasses can implement. Code reusability is an additional motive: the behavioral elements shared among classes can go into a single superclass for those classes.

Analyzing the Inheritance Chain

As the first step in subclassing you should analyze the inheritance chain. This point may seem obvious, but it is important enough to emphasize. You should do more than just identify the most suitable superclass; you want to understand exactly what it does and how it interacts with other classes.

Carefully read the specifications. Note which methods are available. Determine what the methods do and how they are related to each other; identify the accessor methods, those that get and set the instance variables; identify the interfaces to instances of other classes (such as outlets).

If the behavior you want for your class is targeted at a special problem, even if that problem is managing an application or window, it might make the most sense to subclass `NSObject`. These kind of subclasses, often called controller or model classes, are common in OpenStep applications. See “Implementing a subclass of `NSObject`” for details on creating typical controller classes. Also, see “The Model-View-Controller Paradigm” in this chapter for a description of the distinguishing characteristics of controller and model types of classes.

Instance Variables: To Add or Not to Add

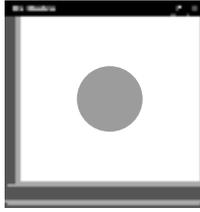
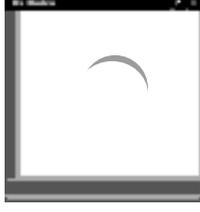
Instance variables represent an object's attributes and hold pointers to other objects (outlets). If instances of your class require special attributes or outlets, add them.

But, as a general rule, avoid adding instance variables unless they are absolutely necessary. Instance variables add weight to objects. You sometimes generate certain objects (for example, cells in a file-system browser) in large numbers. The more data these objects carry, the more memory gets consumed.

Often you can compute values from other values. Sometimes you can get pointers to other objects without having to specify outlets. Or you can represent attributes in lightweight fashion, especially if they are Boolean in nature, by encoding them as bits in an integer.

If you do not want to give subclasses of your class access to its instance variables, put the **@private** directive before the declarations of the instance variables you want to conceal. (Many instance variables are private in OpenStep classes.)

This example illustrates the effects of polymorphism and inheritance in a hypothetical class hierarchy. The Shape class provides basic functionality and a single instance variable. The Circle class, a subclass of Shape, adds more instance data and actually implements drawing. The Crescent class supplements its superclass (Circle) with more specialized behavior and data.

	Class	Set Instance Variables	When Object Is Drawn
Shape	<p>Instance Variables: BOOL visible;</p>	visible = YES;	
Circle	<p>Inherits</p> <p>Instance Variables: float radius; float fill;</p>	visible = YES; radius = 0.75; fill = NSDarkGray	
Crescent	<p>Inherits</p> <p>Instance Variables: NSPoint maskOffset;</p>	visible = YES; radius = 0.5; fill = NSBlack; maskOffset = {-0.25, -0.25};	

A Short Practical Guide to Subclassing (continued)

Determining Your Class's Methods

Look at your class from the perspective of potential clients. What will they want it to do? What information will they expect back? The answers to these and similar questions will lead to the set of methods for your class. Based on relation to superclass, methods generally come in three types:

- **Added methods** These new methods extend the class definition. They include accessor methods for new instance variables.
- **Replacement methods** These types of methods completely override the superclass method of the same name. They can also, by being a “null” implementation, block the invocation of the superclass method.
- **Extended methods** These methods also override a superclass method, but then in the implementation invoke the superclass method by calling **super**. This is a common technique for adding behavior or getting cumulative behavior (such as archiving) across the inheritance chain in response to a single message (such as **encodeWithCoder:**).

What is Public, What is Private?

When designing your subclass, also identify the code that is part of the interface and code that is private to the class.

- **Public methods** These implement your class's interface. External objects invoke these methods by sending messages to instances of your class. Among these types of methods are accessor methods, which mediate client access to instance variables. You declare public methods in the header file for your class.
- **Private methods** These methods can be invoked by objects within a project but are invisible to external objects. You usually declare them in a private header file and prefix the method name with an underscore character.
- **Functions** Non-library static C functions are also private to your class. They are marginally faster than methods because they don't involve the overhead of the run-time object system.

Use a method if you're accessing instance variables, and use a public method if that method is part of your public interface.

Alternatives to Subclassing

Sometimes you can get particular behavior without additional subclassing. OpenStep and the Objective-C language give you

many ways to merge and synchronize your class's behavior with the behavior of OpenStep classes and even other custom classes.

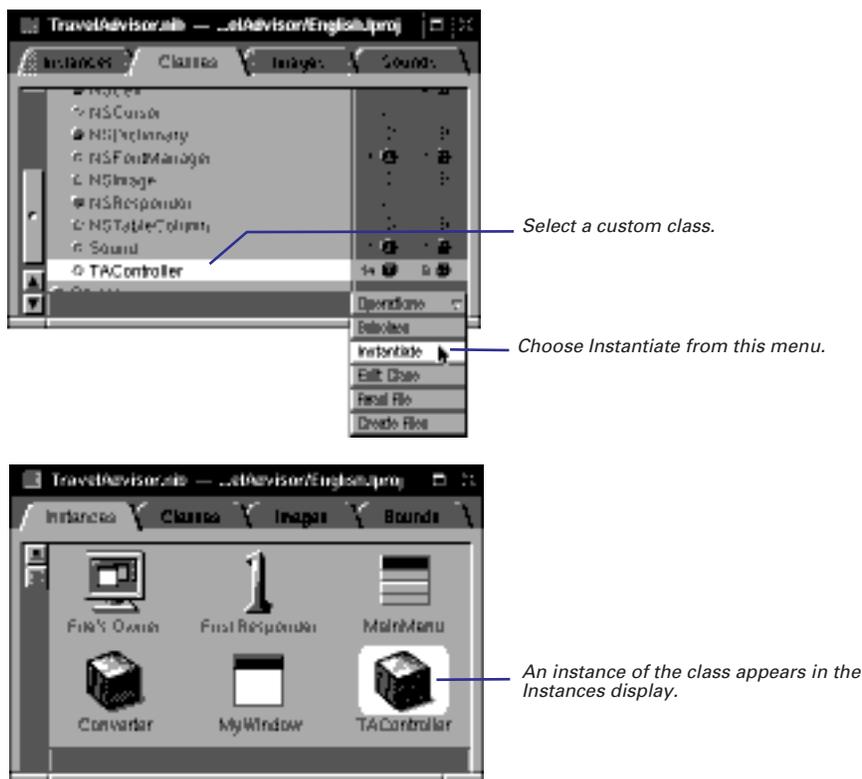
- **Delegation** An object can send, on specific occasions, messages to another object registered as its delegate. If the delegate implements the methods so invoked, it can participate in the work of the object. For example, an `NSBrowser` object sends messages to its delegate requesting cells to insert into a column. Other major Application Kit classes with delegation protocols are `NSApplication`, `NSWindow`, and `NSText`.
- **Notifications** Many objects post notifications to all interested observers when a particular event takes place or is about to take place. Notifications allow observing objects to coordinate related activities and sometimes give them a chance to veto the event. This can be better than delegation because an object can have many observers but only one delegate. See the specification for `NSNotificationCenter` (a Foundation Framework class) for details on adding an observer object and on responding to notifications.
- **Protocols** A protocol is a list of method declarations associated with a particular purpose but unattached to a class definition. By adopting the protocol and implementing the methods, your class can interact with OpenStep classes and accomplish that purpose. OpenStep publishes many protocols, including those for copying objects and encoding objects for archiving.
- **Categories** These are Objective-C constructs that enable you to add methods to a class without having to subclass it. The methods become part of the class, inherited by all of its subclasses. The only major drawback is that you cannot declare new instance variables (however, you can access all existing instance variables). Besides extending a class definition, you use categories to group, manage, and configure methods in large classes.

Creating an instance of your class

- 1 **Select your class in the Classes display.**
- 2 **Choose Instantiate from the Operations menu.**

You cannot connect classes to other classes. Only instances of classes—objects—can really communicate with each other. Interface Builder requires a real instance of your class to enable the connection of your object to other objects.

The procedure for generating instances of non-NSView classes in Interface Builder is simple. *This procedure applies only to classes that don't inherit from the NSView class.*



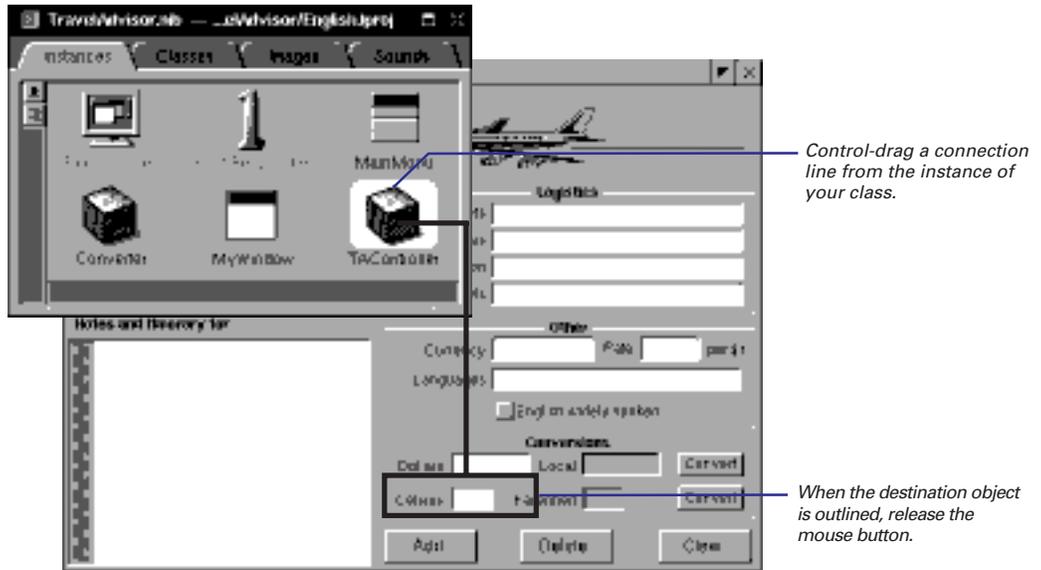
For details on creating an instance of an NSView subclass, see “Implementing a subclass of NSView” later in this chapter.

When the new instance appears in the Instances display, it takes the same name as the class. Rename it, if you want, to something more indicative of an object. (Double-click the text to select it, then type the new name.) For example, ApplicationController could become ApplicationControllerObject. Be aware, however, that this name is merely a convenient way to identify the object in Interface Builder; it does not create an identifier that you can reference in code.

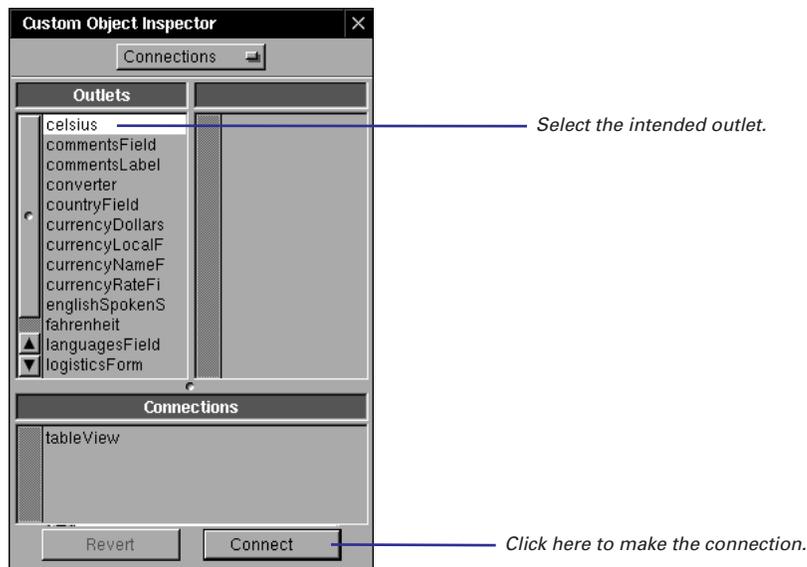
Connecting your class's outlets

- 1 **Control-drag a connection line from the instance to another object.**
- 2 **In the Inspector's Connections display, select the outlet that identifies the destination object.**
- 3 **Click the Connect button.**

You initialize an outlet in Interface Builder by making a connection from your instance to another object.



When you establish the line connection, the Inspector panel for the destination object becomes the key window. Specify the outlet identifier for this object.



This task and the next one “Connecting your class’s actions,” summarize information more fully presented in Chapter 4, “Making and Managing Connections.”

Connecting your class's actions

- 1 Control-drag a connection line from a Control object to your class's instance.
- 2 In the Inspector's Connections display, select the appropriate action.
- 3 Click the Connect button.

Action connections go from an NSControl object to your class's instance.

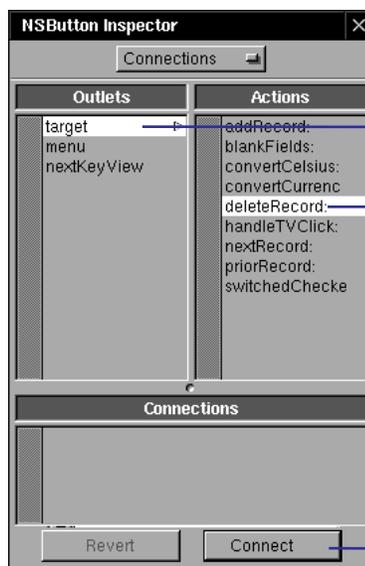


Locate an NSControl object and Control-drag a connection line from it.



The destination object is usually a custom object whose class has defined action methods. Release the mouse button when this object is outlined.

When the line is set between the objects, the second column of the Connections display shows the action methods that the target object (your instance) has declared. Select the action for this NSControl object.



Select the target, which is your instance.

Select an action defined for the class.

A dimple indicates that a connection already exists for the action.

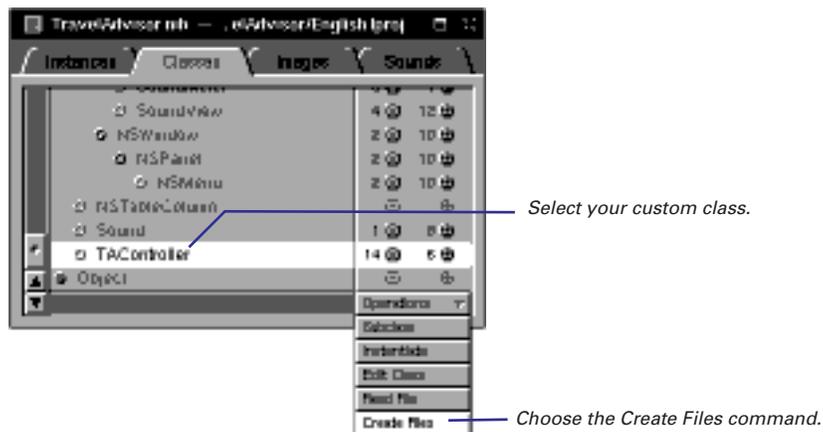
Click here to make the connection.

You can make connections between objects entirely within the outline mode of the Instances display. For more information on the outline mode, see Chapter 4, “Making and Managing Connections.”

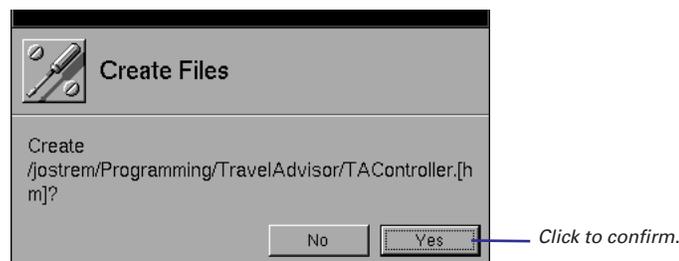
Generating source code files

- 1 **Select your class in the Classes display.**
- 2 **Choose Create Files from the Operations pull-down list.**
- 3 **Click Yes in the subsequent attention panels/message boxes.**

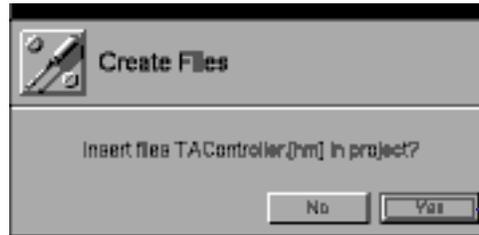
Before you begin specifying the behavior of your class in code, you typically generate template source code files for your class from the information contained in the nib file. The header file (*MyClass.h*) created by Interface Builder declares the outlets you specified as instance variables (of type *id*) and declares the actions as instance methods of the form *methodName:sender*. The implementation file (*MyClass.m*) contains empty function blocks for each of these methods.



When you generate source code files, Interface Builder displays an attention panel/message box to confirm creation of the files.



If you confirm creation and the nib file is associated with a project, another attention panel/message box subsequently asks if you wish to add the template code files to the project. Click Yes to add the files to the project.



And then they appear in Project Builder.



Implementing a subclass of NSObject

- ▶ Import header files.
- ▶ Declare new instance variables.
- ▶ Implement accessor methods.
- ▶ Define target/action behavior.
- ▶ Define initialization and deallocation behavior.
- ▶ Define how objects are copied.
- ▶ Define how objects are compared.
- ▶ Implement archiving and unarchiving.
- ▶ Define special behavior for your class.

This task summarizes the steps that you must complete—and can optionally complete—to implement a subclass of NSObject. With this kind of subclass, the subtleties arising from inherited behavior are simplified. Still, the interaction of your class with the root class is very important and applies to all subclasses.

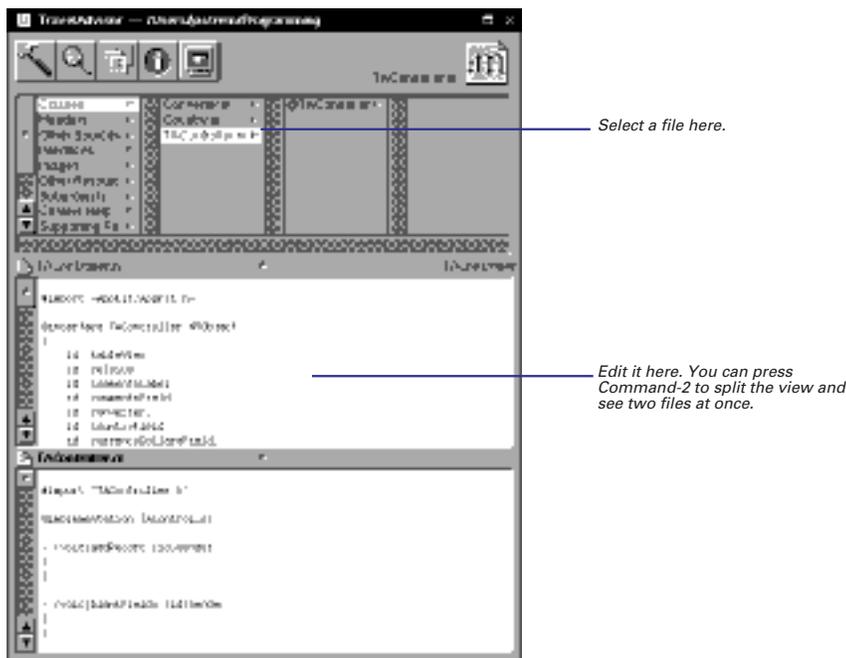
The task assumes that you have completed the following prerequisites in Interface Builder, presented earlier in this chapter:

- Naming a class, positioning it in the class hierarchy
- Specifying outlets and actions for the class
- Creating an instance of the class
- Connecting the instance to other objects through the outlets and actions
- Generating code files from the nib file

When you have generated code files in Interface Builder, switch over to the Project Builder application and open your project. Open your class's header file (*ClassName.h*) and implementation file (*ClassName.m*).

For more on the NSObject class, see its description in the *Foundation Framework Reference*.

The book *Object-Oriented Programming and the Objective-C Language* describes in detail many topics related to the NSObject class and class creation.



Importing Header Files

This step is little different from what you must do in regular C programming: At the beginning of your implementation file include the header files declaring all types and functions that your code is using, as well as the header files for all referenced classes, protocols, and methods. Instead of `#include`, however, use the `#import` directive; `#import` ensures that the file is included only once.

Remember to import your class's header file. By doing so you include the interface files for all inherited classes. To include the Application Kit classes, all you need to do is `#import <AppKit/AppKit.h>`. (Interface Builder imports both `AppKit.h` and your class header files for you automatically).

```
/* TAController.h */
#import <AppKit/AppKit.h>
#import "Country.h"
```

```
/* TAController.m (implementation file) */
#import "TAController.h"
#import "Converter.h"/* Needed in implementation, not interface */
```

Declaring New Instance Variables

The header file that Interface Builder generates declares outlets as instance variables. You might want to add new instance variables for your class to this list. All instance variables should be data that is essential to an instance of your class. They can be strings, integers, floating-point values, and other objects.

```
@interface TAController:NSObject
{
    id tableView;
    ...
    NSMutableDictionary *countryDict;
}
```

Notes on the code: In this example, the instance variable `tableView` derives from an outlet specified in Interface Builder. It is written to the header file when template files are generated. The instance variable `countryDict` has been added to identify an instance of the Foundation class `NSMutableDictionary`. Explicit typing is recommended.

Implementing Accessor Methods

Accessor methods retrieve and set the values of instance variables. They provide the encapsulation of an object's data, which only the object itself (and usually instances of subclasses) can directly access. Accessor methods mediate access to instance variables, allowing client objects to get and set values through an object's interface—that is, by sending messages.

Accessor methods that *retrieve* the value of an instance variable by convention take the same name as the instance variable. They usually have a single statement that returns the value of the instance variable. Methods that *set* the value of an instance variable by convention take the name of the instance variable (first letter capitalized) prefixed with “set.” Set methods often test passed-in values for validity before assigning them.

```
- (NSString *)name
{
    return name;
}

- (void)setName:(NSString *)str
{
    [name autorelease];
    name = [str copy];
}
```

Notes on the code: The **name** method retrieves the value of the instance variable **name**; it simply returns the value. The **setName:** method sets the value of the instance variable **name**. Because **name** is an object, it releases the instance variable before assigning it the new value. Again because **name** is an object, the new value is copied to make sure that it remains valid.

Your class might not need to implement accessor methods if it has no need for client objects to set or retrieve the values of its objects' instance variables.

Defining Target/Action Behavior

When you defined your class in Interface Builder, you specified certain methods (*actions*) that NSControl objects in the interface invoke in your object (the *target*) when an certain user event occurs. In implementing your class, you must specify the behavior of these methods. The sole argument of action methods is **sender**, the object sending the message.

```
- (void)handleTVClick:(id)sender
{
    Country *newerRec;
    int index = [sender selectedRow];

    if (index >= 0 && index < [countryKeys count]) {
        newerRec = [countryDict objectForKey:[countryKeys
            objectAtIndex:index]];
        [self populateFields:newerRec];
        [commentsLabel setStringValue:[NSString stringWithFormat:
            @"Notes and Itinerary for %@",
            [countryField stringValue]]];
        recordNeedsSaving=NO;
        [tableView tile];
    }
    return;
}
```

Notes on the code: This method updates other fields in a window with information from an NSDictionary when the user selects a row in a table view. It uses **sender**, which identifies the NSControl object sending the message, to find out which key to use when retrieving the information from the NSDictionary.

handleTVClick: is an abbreviated version of a method you implemented if you worked through the TravelAdvisor tutorial in *Discovering OPENSTEP Programming*.

Defining Initialization and Deallocation Behavior

The NSObject class defines methods that subclasses must override to initialize their instances and to deallocate them. These methods are invoked at the start and end of an object's life. Initialization sets the initial values of instance variables and dynamically allocates and initializes variables. Deallocation frees the memory allocated to these variables.

Subclasses of NSObject almost always need to override **init** and **dealloc**. (An exception is a subclass that has no instance variables; in this case, it can rely on NSObject's implementation of **init**, which simply returns **self**.) You can define other initialization methods for your class that take arguments and perform more specialized initializations. However, a subclass of NSObject must always implement **init**, even if **init** only invokes one of these specialized initializers, passing in a default value.

Designated Initializer One of a subclass's initialization methods must be the *designated initializer*. The designated initializer invokes its superclass's designated initializer (in NSObject's case, **init**), performs most of the work, and returns **self**. The other initialization methods in a class eventually end up invoking the designated initializer.

Invoking super's Initializer Since an object's full complement of attributes includes those instance variables declared and initialized by superclasses, initialization should cascade down the inheritance chain, starting with the NSObject class. This means that initialization should almost always *begin* with the invocation of the superclass's designated initializer. For the same reason, deallocation should almost always *end* by invoking the superclass's **dealloc** method, after deallocating its own dynamically allocated instance variables. If your **dealloc** method invokes **super's dealloc** first, the object will be deallocated before it has had a chance to free its own allocated storage.

For more on designated initializers, see the description of the **init** method in the NSObject class specification in the *Foundation Framework Reference* or see *Object-Oriented Programming and the Objective-C Language*.

```
- (id)init
{
    [super init];

    name=@" ";
    airports=@" ";
    airlines=@" ";
    transportation=@" ";
    hotels=@" ";
    languages=@" ";
    currencyName=@" ";
    comments=@" ";

    return self;
}

- (void)dealloc
{
    [name release];
    [airports release];
    [airlines release];
    [transportation release];
    [hotels release];
    [languages release];
    [currencyName release];
    [comments release];

    [super dealloc];
}
```

Remember, if you create an object (such as an instance of `NSString`) in your initialization code or elsewhere, you are responsible for its deallocation (with **autorelease** or **release**). If you create an object in an initialization method, the proper place for releasing it is in **dealloc**.

See “Creating and Deallocating Different Types of Objects” in this chapter for some background. For complete details, read the introduction to the *Foundation Framework Reference*.

Notes on the code: This example shows the **init** method (which is also the designated initializer in this case) starting off by sending **init** to **super** to have its superclass (`NSObject`) complete its initializations first. It then sets the object’s instance variables to initial values (empty strings here) and returns **self**. Until it returns **self**, the object is in an unusable state. The **dealloc** method mirrors the **init** method. It releases all dynamically allocated instance variables. The **release** method decrements an object’s reference count and, if the count afterwards is zero, **dealloc** is invoked and the object is deallocated. It then invokes **super’s dealloc** method to have the superclass deallocate its own instance variables.

Defining How Objects Are Copied

If you expect that objects of your class will be copied, adopt the `NSCopying` protocol; if your class can create mutable versions of an object, also adopt the `NSMutableCopying` protocol.

```
@interface MyClass : NSObject <NSCopying, NSMutableCopying>
```

Next implement the protocol methods, `copyWithZone:` and `mutableCopyWithZone:`. These are simple implementations of these methods:

```
- (id)copyWithZone:(NSZone *)zone {  
    return [[MyClass allocWithZone:zone] init];  
}  
  
- (id)mutableCopyWithZone:(NSZone *)zone {  
    return [[MyMutableClass allocWithZone:zone] init];  
}
```

Defining How Objects are Compared

A problem similar to copying objects is comparing objects. `NSObject`'s default behavior, in the `isEqual:` method, is to compare the identifiers of objects (their `ids`). If the `ids` of the receiving and argument objects are equal, the objects are considered equal. You might find this behavior acceptable for instances of your class, but if you don't, override `isEqual:`.

Suppose you have a class named `Color`, and this class has one instance variable, an integer which holds an industry-accepted identifier of a color. What is important in demonstrating equality of objects in this case is not the equality of `ids`, but of the values of their color instance variables.

Implementing Archiving and Unarchiving

When an object of your class has been around for awhile, responding to events and to messages from other objects, its state—the values of its instance variables—is likely to change. “Off” might change to “on,” true to false, red to green. When the user quits the application owning your object, you want to save the important parts of that object’s state and then restore them the next time the application runs. This is called archiving.

The mechanism for archiving and unarchiving objects is implemented using the classes NSCoder, NSArchiver, and NSUnarchiver and the protocol NSCodering. It encodes an application’s object in a way that enhances their persistency and distributability. The repository of this encoded object information can be a file or an NSData object. You should archive any instance variables or other data critical to an object’s state.

When a class adopts the NSCodering protocol, it receives a message requesting that it encode itself and a message asking that it decode and initialize itself. You implement two NSCodering methods to intercept these messages:

encodeWithCoder: and **initWithCoder:**.

Both **encodeWithCoder:** and **initWithCoder:** should begin by invoking the corresponding superclass method so that the superclass archives or unarchives its instance variables first. (If the class inherits directly from NSObject or any other class that does not adopt NSCodering, however, these methods should not invoke the superclass method.) The invocation of **super’s initWithCoder:** returns the partially initialized object (**self**). End **initWithCoder:** by returning **self**.

NSArchiver and NSUnarchiver provide methods that write data to and read data from the archive. Among these are **encodeObject:**, **encodeValuesOfObjCTypes:**, **decodeObject:**, and **decodeValuesOfObjCTypes:**. You send the message **encodeRootObject:** or **archiveRootObjectToFile:** to the NSArchiver class to invoke an **encodeWithCoder:** method. To invoke an **initWithCoder:** method, you send the message **unarchiveObjectWithFile:** or **decodeObject** to the NSUnarchiver class. You never invoke **encodeWithCoder:** or **initWithCoder:** directly.

```

- (void)encodeWithCoder:(NSCoder *)coder
{
    [coder encodeObject:name];
    [coder encodeObject:transportation];
    [coder encodeObject:hotels];
    [coder encodeObject:languages];
    [coder encodeValueOfObjCType:"s" at:&englishSpoken];
    [coder encodeObject:currencyName];
    [coder encodeValueOfObjCType:"f" at:&currencyRate];
    return;
}

- (id)initWithCoder:(NSCoder *)coder
{
    name = [[coder decodeObject] copy];
    transportation = [[coder decodeObject] copy];
    hotels = [[coder decodeObject] copy];
    languages = [[coder decodeObject] copy];
    [coder decodeValueOfObjCType:"s" at:&englishSpoken];
    currencyName = [[coder decodeObject] copy];
    [coder decodeValueOfObjCType:"f" at:&currencyRate];
    return self;
}

```

Notes on the code: NSCoder defines matching sets of methods for encoding and decoding objects of different types. In this example, several objects are encoded using the **encodeObject:** method and decoded using the **decodeObject:** method. One Boolean and one float variable are encoded and decoded using **encodeValueOfObjCType:** and **decodeValueOfObjCType:**, respectively. Note that the data, by type, must be decoded in the same sequence as it was encoded. The superclass method is not invoked because the class inherits directly from NSObject, which does not conform to NSCoder.

You don't need to archive every instance variable of your class. Some of these values you can re-create from scratch and others are transitory and hence unimportant (such as a seconds variable used for timing the period since a certain event). Application Kit objects configured in Interface Builder are automatically unarchived from their nib file, but only as you originally initialized them. If you want to retain some changed attribute of these objects, you should archive the attribute and then initialize the object with the unarchived attribute in the **awakeFromNib** method. (An **awakeFromNib** message is sent to each of the objects unarchived from a nib file after all of the objects in the nib file have been unarchived and all of the outlets are set.)

```
- (void)awakeFromNib
{
    [countryField selectText:self];
    ...
    [commentsField setDelegate:self];
    ...
    [currencyRateField setDelegate:self];
}
```

Notes on the code: In this implementation of `awakeFromNib`, the object must communicate with fields on its interface through the outlets `countryField`, `commentsField`, and `currencyField`. It places the cursor inside `countryField` and makes itself the delegate of the fields `commentsField` and `currencyRateField`. These initializations are done here and not in `init` because the connection between the objects must be unarchived from the nib file first.

Defining Special Behavior

The final step in implementing a subclass of NSObject is writing the methods that are special to your class, those methods that give it its distinctive behavior. This step is all up to you. If you want examples that you can use as models, look in `($NEXTROOT)/NextDeveloper/Examples`.

Other NSObject Methods You Could Override

There are several other NSObject methods that you might want to implement:

description Implement this method to return a descriptive debugging message as an NSString object. When you're debugging, `gdb` displays your message when you use the `po` command.

awakeAfterUsingCoder: Implement this method to re-initialize the object, providing it one last chance to propose another object in its place.

replacementObjectForCoder: Implement this method to substitute another object for your object during encoding.

initialize Implement this class method if you want to initialize your class before it receives its first message. This is a good place to set the version of your class (`setVersion:`).

forwardInvocation: Implement this method if you want to forward messages with unrecognized selectors to another object that can handle the message.

The Structure of Header Files and Implementation Files

```
#import <UIKit/UIKit.h>

@interface TAController:NSObject
{
    id tableView;
    ...
    BOOL recordNeedsSaving;
}

/* target/action */
- (void)addRecord:(id)sender;
- (void)deleteRecord:(id)sender;
/* housekeeping methods */
- (id)init;
- (void)awakeFromNib;
- (void)dealloc;
...
@end
```

```
#import "TAController.h"

@implementation TAController

- (void)addRecord:(id)sender
{
    /* some code here */
    return;
}

- (id)init
{
    /* some code here */
    return self;
}

/* ... */
@end
```

Header File

- Begin by importing header files for declaration types (**#import**).
- **@interface** begins class interface declaration. Class name precedes superclass, separated by a colon.
- Put the declarations of instance variables within curly braces.
- After right curly brace declare your methods.
- Action methods take the argument **sender**.
- End class interface declaration with **@end**.

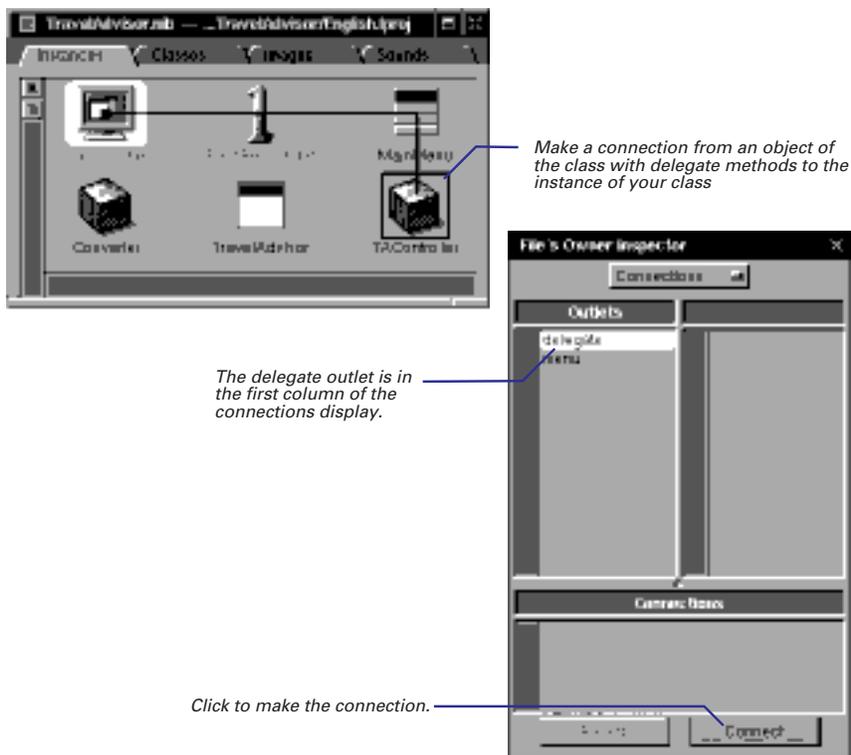
Implementation File

- Begin by importing relevant header files, especially the class header file.
- **@implementation** followed by class name begins implementation section.
- Implement all methods.
- End implementation section with **@end**.

Making your class a delegate

- 1 **Connect your instance to an object that has delegates.**
- 2 **Select the delegate outlet in the Connections inspector.**
- 3 **Click Connect.**
- 4 **Implement the delegate methods.**

Several OpenStep classes allow you to register an object as their delegate. As certain events occur, the objects send messages to their delegates, giving them the opportunity to participate in processing. In Interface Builder, you can easily designate your class's instance as a delegate.



Next, implement the delegate methods you want your class to respond to. In this example, the object acting as delegate archives itself before the application terminates.

Messages to delegates sometimes notify them of impending or just-transpired events, and sometimes request them to complete some work. Major classes with delegate methods are `NSApplication`, `NSWindow`, `NSText`, and `NSBrowser`. See the *Application Kit Reference* for details on delegate methods.

```
- (void)applicationWillTerminate:(NSNotification *)note
{
    [NSArchiver archiveRootObject:self toFile:@"TravelData"];
}
```

Tip: You can programmatically set an object's delegate by sending it the `setDelegate:` method.

Implementing a subclass of NSView

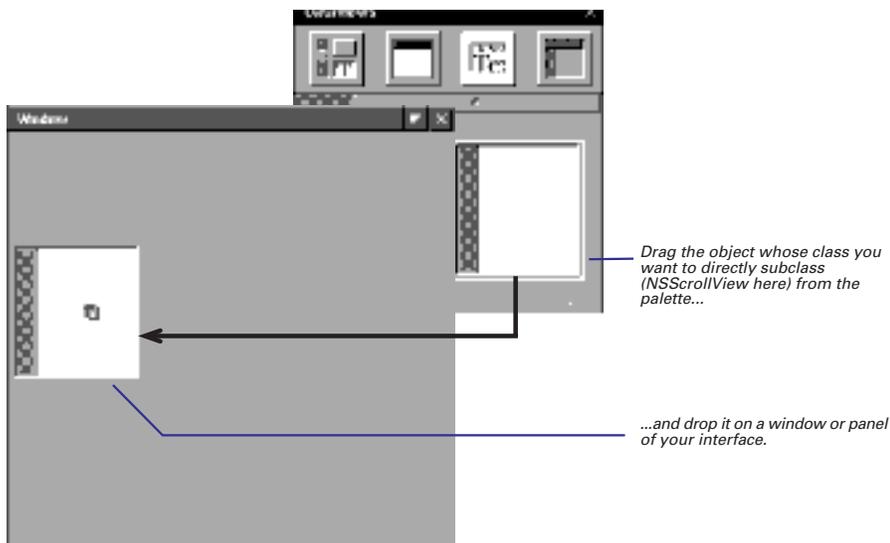
- 1 Identify the class and its outlets and actions.
- 2 Place and resize an object from the Views palette on a window or panel.
- 3 Assign your class as the class of the object.
- 4 Connect the instance to other objects in the interface.
- 5 Generate code files.
- 6 Complete programming tasks necessary for any object.
- 7 Complete programming tasks specific to NSView objects:
 - ▶ Initialize an NSView object.
 - ▶ Draw an NSView object.
 - ▶ If necessary, handle events.

Making a subclass of the NSView class is a procedure that differs from making a subclass of the NSObject class. But it starts out the same. In the Classes display of Interface Builder, choose Subclass from the Operations menu while NSView or one of its subclasses is highlighted in the browser. Then name your class and add outlets and actions.

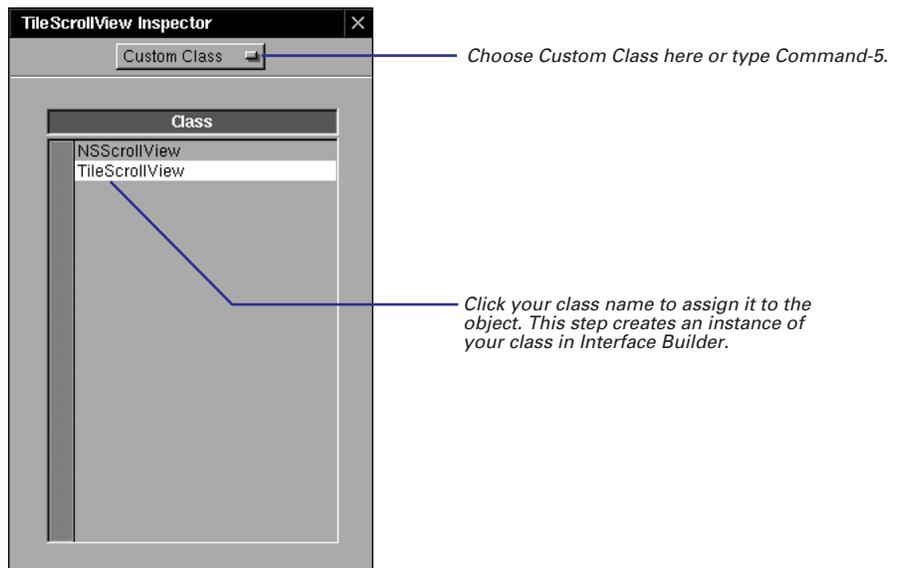


Making an Instance of an NSView Subclass

Place an instance of your class on your interface. If you're subclassing an NSView subclass (such as NSButton or NSTextField), drag the object that represents that class (that is, the button or the text field) from the palette window into your interface's window. If you're subclassing NSView directly, use the CustomView object on the Views palette.



Position and resize the object, and while it's still selected, bring up the Custom Class display of the Inspector panel by typing Command/Control-5. Assign a class name to the object; this creates an instance of your NSView subclass.



Tip: Make sure you choose the appropriate superclass. If you subclass an NSView subclass, rather than subclassing NSView directly, you can still set that class's attributes for your object using the Inspector panel's Attributes display. If you subclass NSView directly, you lose the ability to set attributes using the Inspector panel.

The next three steps that you must complete are the same tasks that follow the instantiation of NSObject subclasses:

- Connect the instance to other objects in the interface (“Connecting your class’s outlets” and “Connecting your class’s actions”). But now the instance appears as part of the interface, and not as an icon in the Instances display of the nib file window.
- Generate code files and have them inserted in your project (“Generating source code files”).
- Switch over to the project in Project Builder that contains the nib file, and open your class’s code files.

Since NSView inherits from NSObject, next complete some of the same programming tasks recommended for subclasses of NSObject:

- Declaring new instance variables
- Implementing accessor methods
- Implementing target/action methods
- Archiving and unarchiving

To create a functional subclass of NSView, you must complete two additional steps (and might want to complete another), which are described on the following pages.

Initializing NSView Objects

Subclasses of NSView override **initWithFrame:** instead of **init**. In **initWithFrame:** (NSView's designated initializer) you initialize a just-allocated instance of your class, setting its attributes to an initial state. The method's sole argument is the rectangle in which drawing is to occur (usually the frame of the view).

In this example, **initWithFrame:** initializes instance variables of varying types and performs other housekeeping chores.

```
- (id)initWithFrame:(NSRect)frameRect {
    [super initWithFrame:frameRect];
    glist = [[NSMutableArray allocWithZone:[self zone]]
            init];
    slist = [[NSMutableArray allocWithZone:[self zone]]
            init];
    cacheImage = [self createCacheWithSize:[self bounds].size];
    [self cache:[self bounds] andUpdateLinks:NO];
    gvFlags.grid = 10;
    gvFlags.gridDisabled = 1;
    [self allocateGState];
    gridGray = DEFAULT_GRID_GRAY;
    PSInit();
    currentGraphic = [Rectangle class];      /* default graphic */
    /* trick to allow NSApp to control currentGraphic */
    currentGraphic = [self currentGraphic];
    editView = [self createEditView];
    [[self class] initClassVars];
    [self registerForDragging];
    spellDocTag = 0;
    return self;
}
```

The NSView class offers your subclass a wealth of inherent functionality. It includes methods for managing the view hierarchy, for converting coordinates and modifying the coordinate system, for managing cursors and events, and for focusing, clipping, scrolling, dragging, and printing. See the description of the NSView class in the *Application Kit Reference*.

Notes on the code: The implementation of an **initWithFrame:** method begins by invoking **super's** **initWithFrame:** method, ends by returning **self**, and in between sets the instance variables to initial values. Often the attributes set have a visual aspect, and affect how the view is drawn.

As with NSObject subclasses, you might have to implement the **dealloc** method to deallocate dynamically allocated storage.

Drawing NSView Objects

An NSView object draws itself with the **drawRect:** method. To invoke **drawRect:**, another object must send **display** to the NSView object. The **drawRect:** method is also invoked automatically when windows are resized and exposed, when NSViews are scrolled, and when similar events happen. The NSRect argument passed to **drawRect:** indicates how much of the NSView needs to be drawn.

```

- (void)drawRect:(NSRect)rect
{
    int grid;
    float gray;

    grid = [spacing intValue];
    grid = MAX(grid, 0.0);
    PSsetgray(NSWhite);
    NSRectFill(rect);
    if (grid >= 4) {
        gray = [grayField floatValue];
        gray = MIN(gray, 1.0);
        gray = MAX(gray, 0.0);
        PSsetgray(gray);
        PSsetlinewidth(0.0);
        [self drawGrid:grid];
    }
    PSsetgray(NSBlack);
    NSFrameRect([self bounds]);
}

```

The PostScript functions and operators available for use are described in *DPSClientLibrary Reference*.

pswrap is a program that creates a C function to correspond to a sequence of PostScript code. Note that your custom **pswrap** code (extension **.psw**) must go in Project Builder under Other Sources. **pswrap** is described in detail in Adobe Systems' *pswrap Reference Manual*.

Notes on the code: The example above shows **drawRect:**. This example fills in the view with a white background, draws a grid using a user-selectable gray value, then uses **NSFrameRect()** to draw a black border around the view.

In implementing **drawRect:**, write whatever code helps to draw your NSView. You can call **pswrap**-generated functions to send PostScript code to the Window Server. You can send messages to bitmap objects, requesting them to composite source images stored in off-screen windows. You can change font styles and text colors. If your NSView uses an NSCell to do any of its drawing, you can send **drawWithFrame:inView:** or **drawInteriorWithFrame:inView:** to the NSCell within **drawRect:**.

The **drawRect:** method defines an NSView's static appearance on the screen. Your subclass can also add other methods for dynamic drawing in response to user events. In these methods you might highlight the NSView, drag it from one place to another, or animate it. The Application Kit locks focus automatically when **drawRect:** is invoked. In dynamic-drawing contexts you must lock and unlock focus yourself when drawing.

If you want your view to respond to mouse clicks, key presses, or other user events, you must do at least two things:

- Re-implement NSView's **acceptsFirstResponder** method to return YES.
- Decide which event types you want to respond to and implement the appropriate methods: **mouseUp:**, **mouseDown:**, **keyDown:**, **mouseEntered:**, and so on.

The event methods are defined in the NSResponder class, where the default implementation is to forward the event message to the next responder.

When it invokes an event method, the input system passes in an NSEvent object. This object holds details related to the event: the type of event, the mouse's location (in the window's base coordinates), the window number, a time value associated with the event, flags indicating modifier keys and mouse buttons, and supplementary data.

You can find or derive much of the information required for handling an event in the NSEvent parameter. For instance, you can convert the NSEvent mouse location to your NSView's base coordinate system with **convertPoint:fromView:**. You can check for modifier keys or mouse buttons using the keyboard-state flags masks.

The following example illustrates several of these techniques.

The NSEvent class is described in the *Application Kit Reference*.

```
- (void)mouseDown:(NSEvent *)event
{
    NSPoint p, start;
    int grid, gridCount;

    start = [event locationInWindow];
    start = [self convertPoint:start fromView:nil];
    grid = MAX([spacing intValue], 1.0);
    gridCount = (int)MAX(start.x, start.y) / grid;
    gridCount = MAX(gridCount, 1.0);

    event = [[self window] nextEventMatchingMask:
             NSLeftMouseDownMask|NSLeftMouseUpMask];
    while ([event type] != NSLeftMouseUp) {
        p = [event locationInWindow];
        p = [self convertPoint:p fromView:nil];
        grid = (int)MAX(p.x, p.y) / gridCount;
        grid = MAX(grid, 1.0);
        if (grid != [spacing intValue]) {
            [form abortEditing];
            [spacing setIntValue:grid];
            [self display];
        }
        event = [[self window] nextEventMatchingMask:
                NSLeftMouseDownMask|NSLeftMouseUpMask];
    }
}
```

Tip: If you want your `NSView` to handle target/action messages sent to the First Responder (for example, copy and paste), be sure to override `acceptsFirstResponder` to return YES, and then implement the appropriate methods (`copy:` and `paste:`).

Creating and Deallocating Different Types of Objects

As you create objects, you need to make sure that they are going to be deallocated eventually, and you also need to make this doesn't happen until you don't need the object anymore. You do this by sending messages that increment and decrement the object's reference count, a count of how many objects refer to it. When and how you should do this depends on when and how you create the object.

The Autorelease Pool

OpenStep uses an autorelease pool to automatically deallocate objects. When you send an **autorelease** message to an object, it adds the object to the autorelease pool. At the top of the event loop, the pool sends every object in it the **release** message. **release** decrements the reference count. If the reference count becomes 0, it deallocates the object (by sending **dealloc**).

Application projects automatically have an autorelease pool, just as they automatically have an event loop. If you're working on a non-Application project, you can create an autorelease pool by creating an instance of the Foundation `NSAutoreleasePool` class. (See its specification in the *Foundation Framework Reference*.)

Temporary Objects

If you create an object inside a method and you want that object to go away after the method has finished executing, use a **+classname** method (so called because their names begin with the name of the class minus the NS prefix) to create the object. These methods allocate the object (which increments the reference count), initialize it, and send it an **autorelease** message so that it is deallocated at the top of the event loop. For example, this `NSNumber` object will exist only for one event cycle:

```
NSNumber *intObject = [NSNumber
    numberWithInt:anInt];
```

The methods **alloc**, **copy**, and **mutableCopy** increment an object's reference count, so if you use one of these to create a temporary object, be sure to send that same object an **autorelease** message.

Instance Variables

Objects that are instance variables should be created when an object is initialized and not go away until that object is deallocated. If you use a **+classname** method to create an instance variable, it will be deallocated at the top of the event loop. To prevent this, send **retain** to the object immediately after you create it. **retain** increments the reference count. Another way to make sure that an instance variable is not deallocated is to use the **alloc** method directly (or **copy** or **mutableCopy**) to create it.

No matter which method you use to create the instance variable, send it a **release** message in your object's **dealloc** method to indicate that you're done with it.

Sometimes you have two objects with instance variables that refer to each other. In this case, only one of the objects should retain the other. For example, an `NSView` object has a `Superview` and one or more `Subviews`, each pointing to other `NSView` objects. If an `NSView` object retained both its `Superview` and its `Subviews`, no `NSView` would ever be deallocated. The `Superview` won't release its `Subview` instance variables until it is deallocated, and it can't be deallocated because the `Subviews` don't release the `Superview` until they are deallocated. For this reason, `NSView` objects retain their `Subviews`, but not their `Superviews`.

As a rule of thumb, if your application has a similar object hierarchy, the "parent" object should retain its "children," but the children should not retain their parents.

Custom Objects Created in Interface Builder

If you create a custom object that does not inherit from `NSView` or `NSWindow` in Interface Builder, send it a **release** message in your object's **dealloc** method. Custom objects have a retain count of 1 when they're unarchived from the nib file.

NSView Objects Created in Interface Builder

Views created in Interface Builder are retained and released automatically. `Superviews` retain all `Subviews` as they are added to the hierarchy and release them as they are removed. If you swap views in and out of the hierarchy or move views from one window to another, you should **retain** the views that are not in the hierarchy (and **release** them either after you add them to the hierarchy or in **dealloc**).

NSWindow Objects Created in Interface Builder

Windows created in Interface Builder are not released until the user quits the application. If you want a window to be released when the user closes it, set the "Release when closed" attribute in Interface Builder.

For more on this topic, see the introduction to the *Foundation Framework Reference*.

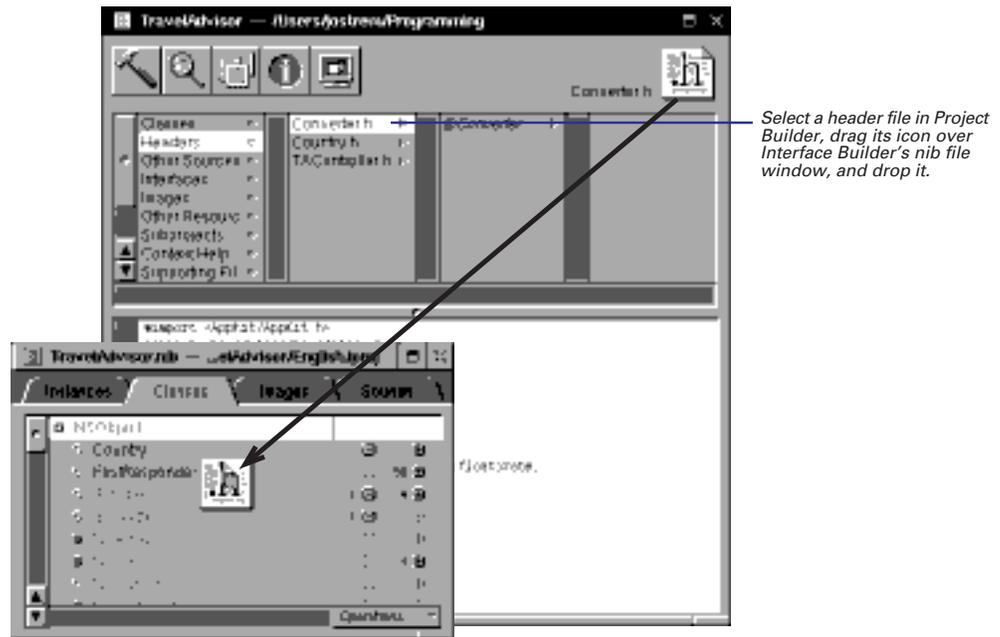
Adding existing classes to your nib file

- ▶ **Drag the header file from the File Viewer/Desktop, File Manager, or Project Builder into the nib file window.**

Or

- ▶ **Copy a class in one nib file and paste it in another.**

The easiest way to add a class to your nib file is to drag the header file for an existing custom class from the Workspace Manager's File Viewer/Desktop or File Manager or from Project Builder's main window into Interface Builder.



The new class appears in the Classes display under its subclass and with its outlets and actions defined. After adding the class, you must still connect it to other objects through its outlets and actions. To do this, complete these steps:

- Make an instance of the class (for `NSView` subclasses, that means assigning your class to a view object).
- Connect the instance's outlets and actions to other objects in the nib file.

If you are going to write a header file and then drag the file into Interface Builder, follow the conventions for header files described in “The Structure of Header Files and Implementation Files,” in this chapter.

Creating Classes in Project Builder

Instead of defining a class in Interface Builder and using Create Files to create the source code, you can create the source code in Project Builder first and add the class to Interface Builder later. To create a class in Project Builder, use the File ► New in Project command to create template source code files, write your code, then drag the header file into the nib file window. When you create a class in this manner, any method you've defined that takes a single argument named **sender** and that returns **id** or **void** is considered an action. Any instance variable that is type **id** or has the **IBOutlet** keyword prefixed to its declaration is an outlet.

```
#import <AppKit/AppKit.h>
#define IBOutlet          /* Needed to avoid compiler errors. */

@interface TAController:NSObject
{
    IBOutlet NSTableView *tableView;          /* an outlet */
    id commentsLabel;          /* another outlet */
    ...
    NSMutableDictionary *countryDict;      /* not an outlet. */
    ...
}

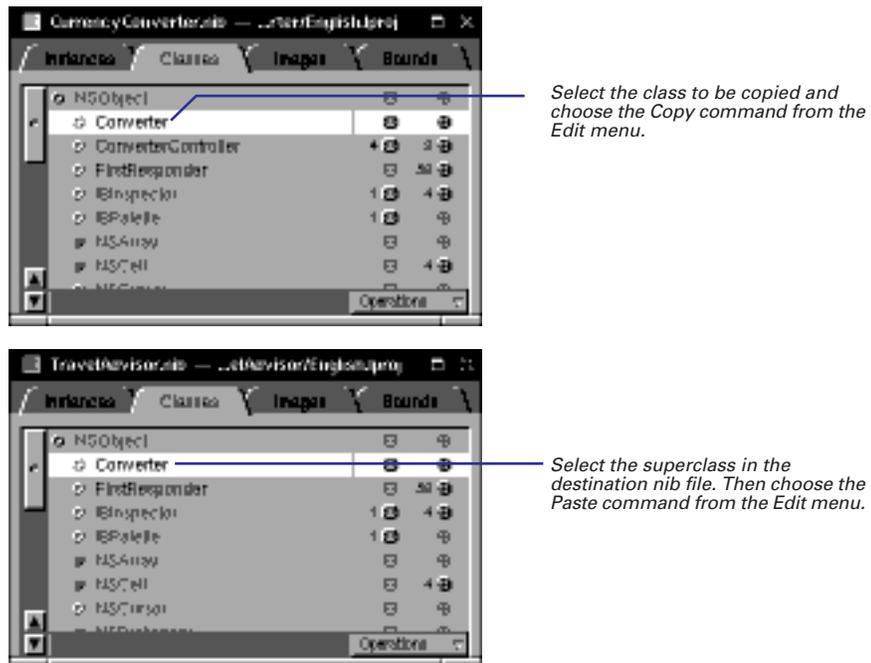
/* target/action */
- (void)addRecord:(NSButton *)sender;          /* an action */
- (void)convertCelsius:(id)sender;          /* another action */

/* Data read and write methods */
- (void)populateFields:(Country *)aRec;      /* not an action */

@end
```

Copying Classes Between Nib Files

You can copy class definitions between nib files, in the same or different projects, by copying a class in one nib file and pasting it into another nib file.

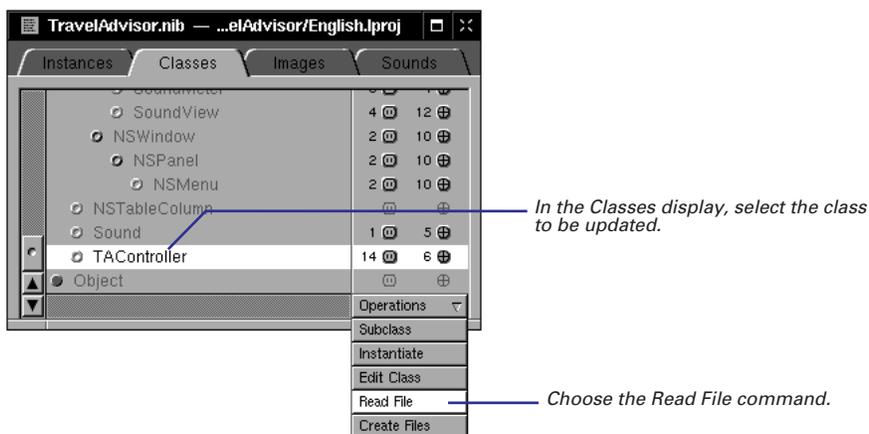


A duplicate of the original class appears in the Classes display of the destination nib file. Generate an instance of the class in the destination nib file and connect it to other objects in the nib file through its outlets and actions.

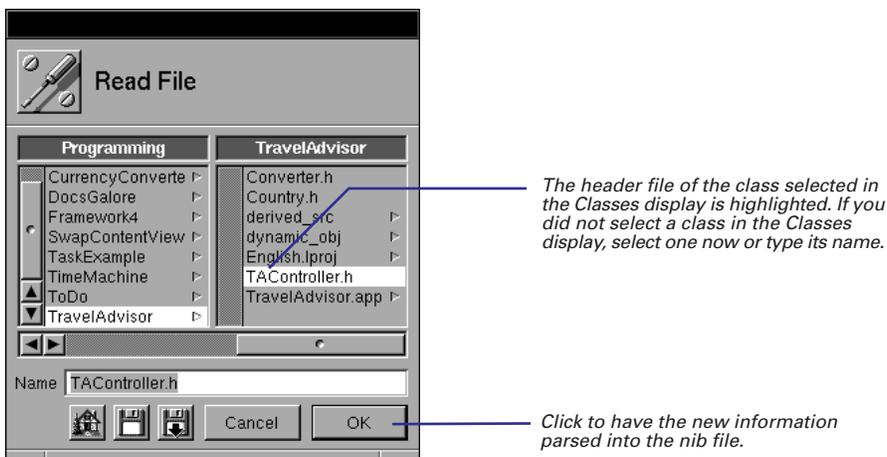
Updating a class definition

- Choose the Read File command and select a header file in the Open panel/dialog.

If you later add outlets and actions to the header file, or delete them from it, Interface Builder allows you to update the nib file with this new information.



Interface Builder brings up an Open panel/dialog for you to confirm (or select) the class definition to update.



If there are any new outlets and actions, remember to connect these outlets and actions to other objects in the nib file.

Tip: You can also use the Read File command to add an existing class to a nib file, or you create a header file in Project Builder using the New in Project command and read it into a nib file using this command.

7

Editing Code

Moving, copying, deleting, and replacing code

Checking delimiters

Indenting code

Navigating within code files

Navigating between code files

Using name completion in editing

Displaying multiple views of code

Undoing and redoing changes

Formatting and inserting graphics in RTF text

Then, arising with Aurora's light,
The Muse invoked, sit down to write;
Blot out, correct, insert, refine,
Enlarge, diminish, interline.

On Poetry

Jonathan Swift

I write until beer o'clock.

Stephen King

Moving, copying, deleting, and replacing code

1 Select the code to be copied or moved.

2 Choose the appropriate command from the Edit menu:

► Copy

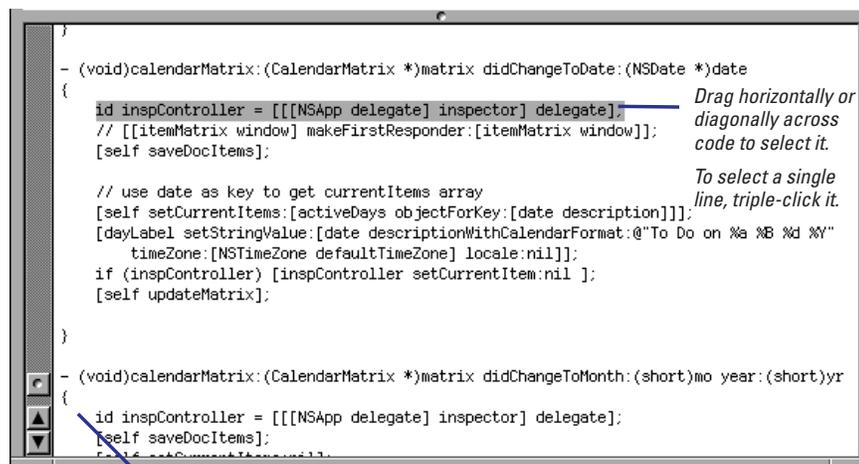
Or

► Cut

3 Insert the cursor where you want the code to go.

4 Choose Paste from the Edit menu.

As you can in other OpenStep applications, you can use the Copy, Move, and Paste commands to delete, copy, and replace code.



The cursor marks the point where pasted code is inserted.

When you choose the Copy or Cut commands, the selected code goes into a *kill buffer*. The Paste command puts the contents of this buffer into the stream of characters at the location marked by the cursor. You can issue multiple Paste commands to copy the same contents multiple times. Of course, if you only want to delete code, don't follow the Cut command with Paste.

Several techniques can help you move and copy code:

- If the “Indent pasted lines” option is checked in the Indentation preferences, pasted code will be automatically indented.
- To replace code rather than inserting it, select the destination code before pasting the new code in place of it.
- To delete code without copying it to the kill buffer, select the code and then press the Delete key.

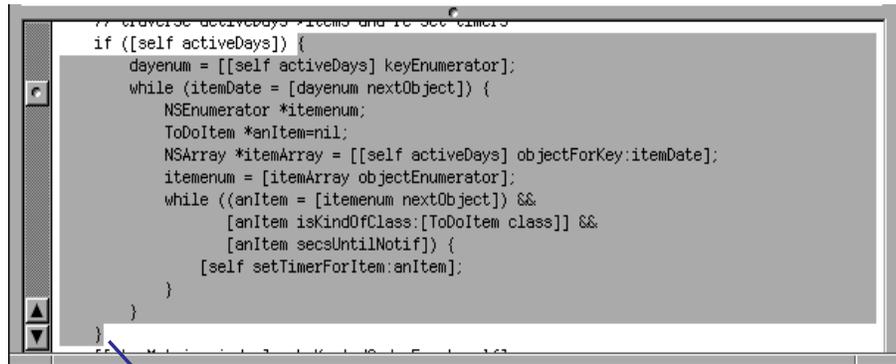
You can use the delimiter-checking feature as a shortcut for selecting messages or blocks of code prior to copying, moving, or replacing them. See “Checking delimiters” on page 158.

Several Emacs commands also allow you to copy, cut, and paste code. See “Emacs Key Bindings” on page 162 for details.

Checking delimiters

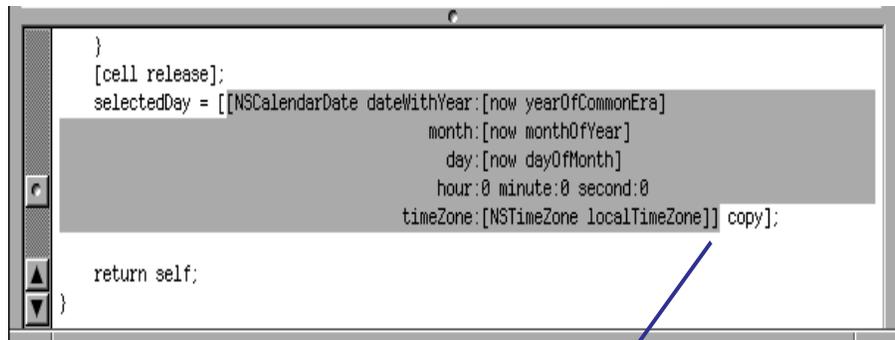
- ▶ Double-click a brace to highlight the block it delimits.
- ▶ Double-click a square bracket to highlight the message it delimits.
- ▶ Double-click a parenthesis to highlight the function arguments or C expression delimited.

One common source of compiler errors is mismatched code delimiters: braces for blocks of code, brackets for message expressions, and parentheses for C expressions and function arguments. Project Builder's code editor allows you to check which delimiters match just by double-clicking.



You can double-click either the starting delimiter or the ending delimiter to highlight code.

Put the cursor as close as possible over the delimiter



This is a good way to check for nested messages.

You can also enable your right mouse button so that when you click anywhere in a type or message expression, that type or message is selected and *then* the Project Find panel performs a definition lookup. See <<x-ref?>> for more on this feature.

In addition to checking for missing or extra delimiters, you can use this technique for selecting the code in between (and including) the delimiters.

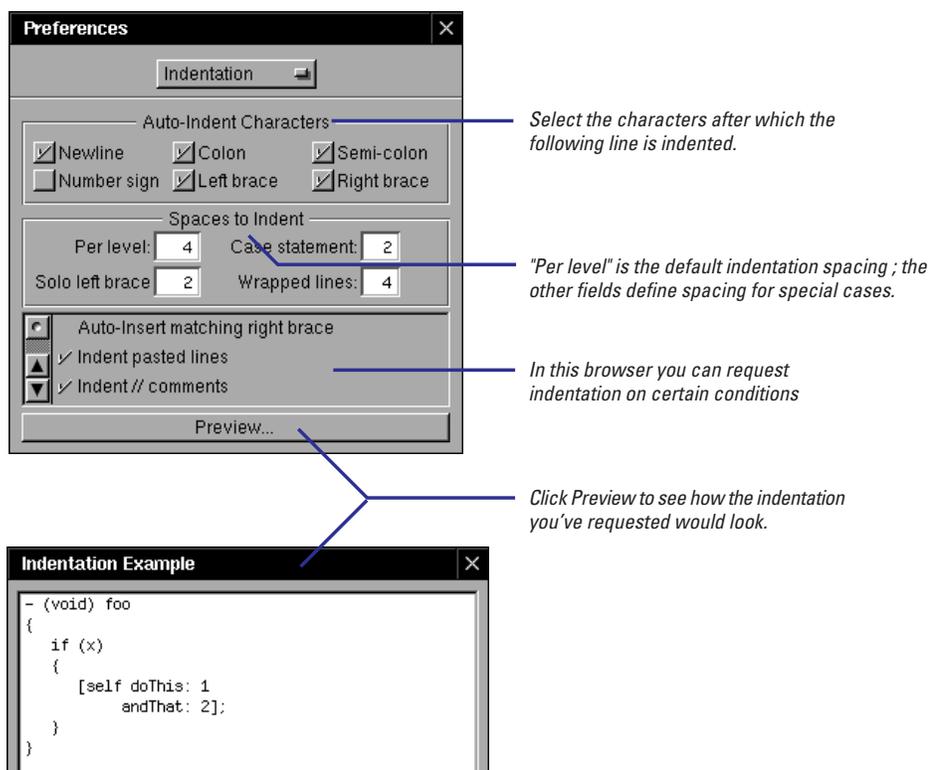
If a matching delimiter is missing, the code editor displays the error in the status area in its upper-right corner. Project Builder also checks for delimiters as you type code. When you type a delimiter of any type, the cursor briefly jumps to the opposite delimiter. If the opposite delimiter is out of view, the line it's on appears in the status area.

Indenting code

- ▶ To customize automatic code indentation, choose **Preferences** from the **Info** menu and select the desired **Indentation** preferences.
- ▶ To indent a line of code, click anywhere in the line and choose **Edit** ▶ **Indentation** ▶ **Indent**.
- ▶ To force indentation left or right, select code and then chose **Shift Left** or **Shift Right** from the **Indentation** menu.
- ▶ To format messages with many arguments, choose **Expand Message Expression** from the **Indentation** menu.

One of the more tedious tasks programmers must do is indenting code: aligning statements and blocks of code with the same scope on the same tab stop. Project Builder alleviates much of this tedium by providing options for both automatic and manual indentation.

In the Indentation preferences display, you can tell the code editor when to indent each line of code based on the final character of the previous line. You can also specify how many spaces lines should be indented, based on certain conditions.



Tip: The Key Bindings preference display has options related to using the Tab key for indentation: “Indent only at beginning of line” and “Indent always.” Use the second option if you want to use Tab to indent a line when the cursor is anywhere in that line.

For both single lines and ranges of lines, you can issue commands to indent according to the indentation characteristics specified in Preferences; or you can give commands to force a certain indentation.

```

}
if ([doc isKindOfClass:[ToDoDoc class]]) {
    result = NSRunAlertPanel(@"To Do", @"Save %@", @"Save",
@"Don't Save", @"Cancel", [[NSApp keyWindow] representedFilename]);
    switch(result) {
case NSAlert:DefaultReturn:

```

When indenting single lines of code, you can insert the cursor anywhere.

```

}
if ([doc isKindOfClass:[ToDoDoc class]]) {
    result = NSRunAlertPanel(@"To Do", @"Save %@", @"Save",
@"Don't Save", @"Cancel", [[NSApp keyWindow] representedFilename]);
    switch(result) {
        case NSAlert:DefaultReturn:

```

Choose Indent from the Indentation menu to indent the line according to preferences

```

}
// Here are multiple lines of comments that you want to
// indent eight spaces. Since they fall outside of any block of
// code they're considered to be unindented
- (void)dealloc
{

```

Select multiple lines of code to indent them as a group.

This example shows multiple lines force-indented right after Shift Right is chosen twice from the Indentation menu.

Messages with multiple arguments can be hard to read. Project Builder gives you a command on the Indentation menu with which to format them.

```

// compose date
selDate = [NSDate dateWithYear:[thisDate yearOfCommonEra]
month:[thisDate monthOfYear] day:selDay hour:0 minute:0 second:0
timeZone:[NSTimeZone localTimezone]];

```

Select a message expression, preferably one with many arguments.

```

// compose date
selDate = [NSDate dateWithYear:[thisDate yearOfCommonEra]
          month:[thisDate monthOfYear]
          day:selDay
          hour:0
          minute:0
          second:0
          timeZone:[NSTimeZone localTimezone]];

```

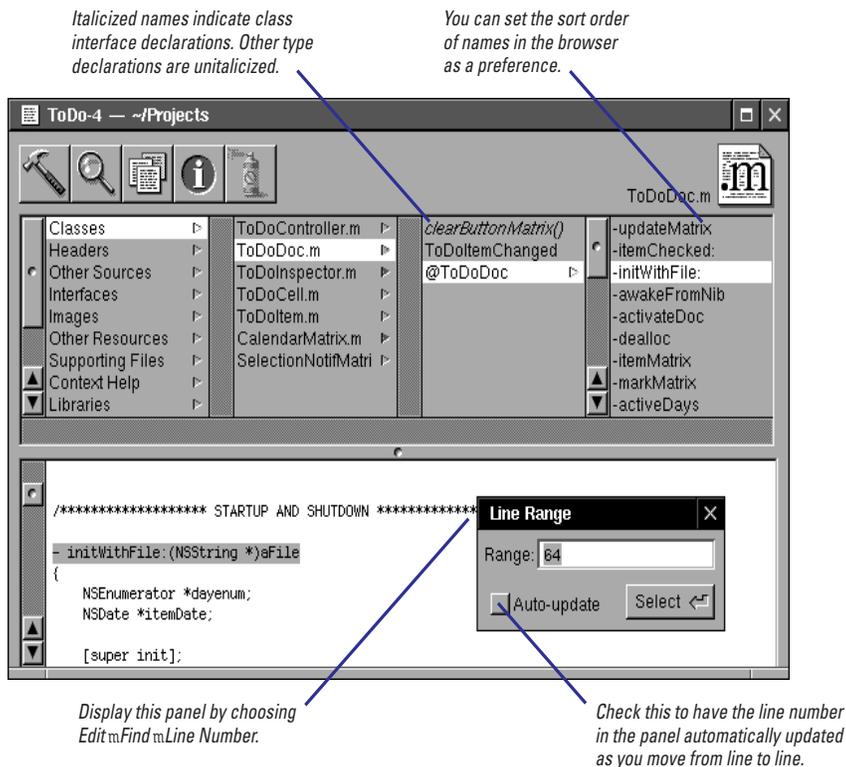
When you choose the Expand Message Expression command, the message is formatted so arguments are aligned.

To return a formatted message expression to an unformatted message expression (in other words, reverse the sequence in the above example), select the expression and choose Edit ► Indentation ► Compress White space.

Navigating within code files

- ▶ Go directly to methods and functions by selecting their names in the project browser.
- ▶ Enter a line number in the Line Range panel to go that line.

You can, of course, go from one place in a source-code file to another place in the same file by scrolling the code editor. Although this mode of navigation is sometimes inescapable, you have other navigation techniques at your disposal.



You can use Emacs key bindings to move around in code without ever touching the mouse. See “Emacs Key Bindings” on page 162 for a list of enabled Emacs commands.

The incremental-search feature (Emacs binding Control-s) is a powerful code-navigation tool for locating text strings within a file. You can also use the Find panel and especially the Project Find panel to find (and replace) specific definitions, reference, and text strings. See the chapter “Finding Information” for details on all of these search features.

Methods, functions, and types appear in the project browser only if the project has been indexed. In the Indexing preferences, you can specify how items in the browser should be sorted: by position in the file, by symbol name (that is, alphabetically), or by symbol name within type.

As its name suggests, you can use the Line Range panel not only to navigate to specific line numbers, but to select ranges of text by specifying colon-separated line numbers. The panel is also a useful tool for learning the current line number. One place where this might be useful is `gdb` (run from the command line) where you can set breakpoints within methods by *file:line number*.

Tip: You can “visit” a line of code and return directly to your original location with a couple of Emacs commands. First set a mark by pressing Escape, then the spacebar. Navigate to the other line, view it (or copy it, or whatever), and press Control-x Control-x to return the marked line.

Emacs Key Bindings

Emacs is an interactive, customizable, richly featured code editor that is popular among many programmers, especially UNIX programmers. Project Builder's code editor incorporates many common Emacs commands.

You issue Emacs commands with the Control key (Ctl) or the Escape (Esc) key, but how you give the command differs with each key. You press the Control key just before the character key, and keep pressing them together. For Escape commands, press the Escape key first, then press the character key. Also, some commands begin with Control-x and are followed by a separate key press. The separation of the final character in Control-x and Escape commands is represented by a space.

Notes:

- On OpenStep for Windows, many Emacs key bindings conflict with the standard bindings for Control keys on Windows applications (for example, Ctl-v is scroll forward in Emacs but is Paste in Windows). Windows key binding override any corresponding Emacs key bindings.
- To use the Emacs commands that begin with the Escape key (Esc), you must select the "Act as Emacs Meta key" option in the Key Bindings preferences display.

Moving Around

Command	What It Does
Ctl-f	Move forward one character
Ctl-b	Move backward one character
Esc f	Move forward one word
Esc b	Move backward one word
Ctl-n	Move to the next line
Ctl-p	Move to the previous line
Ctl-e	Move to the end of the line
Ctl-a	Move to the beginning of the line
Ctl-v	Scroll forward a "page"
Esc v	Scroll backward a "page"
Esc >	Go to the end of the edited file
Esc <	Go to the beginning of the edited file
Ctl-l	Center cursor in middle of displayed code
Ctl-x Ctl-x	Exchange point and mark (return to mark)
Ctl-s	Search forward incrementally
Ctl-r	Search backward incrementally

Editing, Deleting, and Copying

Command	What It Does
Ctl-d	Delete character under cursor
Ctl-k	Delete (kill) to end of line
Ctl-y	Paste (yank) contents of kill buffer
Esc-d	Delete next word or to end of current word
Esc-Del	Delete previous word
Ctl-x u	Undo last change (applies successive undos)
Ctl-i	Indent line
Esc w	Copy region
Esc y	Yank-pop: paste previously cut text in kill buffer

Files and Views (Buffers)

Command	What It Does
Ctl-x 2	Split current view into two views (Split command)
Ctl-x 1	Make one view (Maximize command)
Ctl-x o	Edit in other view
Ctl-x Ctl-b	Open Loaded Files panel
Ctl-x b	Next loaded file
Ctl-x Ctl-f	Display Open Quickly panel
Ctl-x Ctl-s	Save current view to file
Ctl-x Ctl-w	Write to file (Save As)
Ctl-x s	Save all loaded files
Ctl-x i	Insert file
Ctl-x k	Close current file

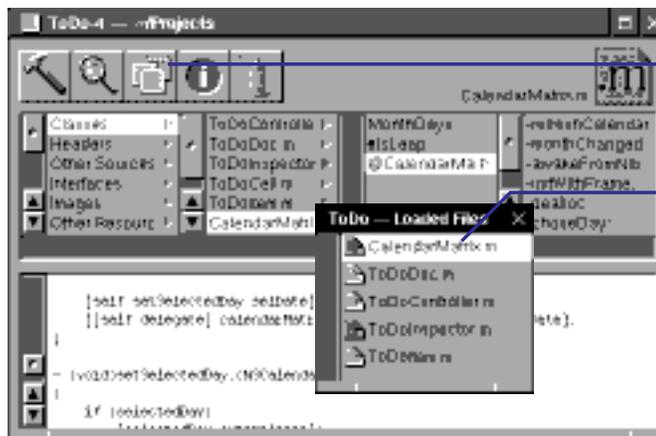
Miscellaneous

Command	What It Does
Ctl-x space	Sets a mark which, with point, marks a region.
Ctl-x '	Go to next error (as displayed in Build panel exception browser)
Ctl-x p	Go to previous error
Esc .	Find definition of current symbol using Project Find panel (as identified by location of cursor)
Ctl-q	Quote next character (for example, a control sequence)
Ctl-g	Quit current command

Navigating between code files

- ▶ Select source-code files in the project browser.
- ▶ Use the Loaded Files browser to navigate among opened files
- ▶ Use the “Open File or Project” panel to locate and open project or non-project files.

When you select header files, Objective-C implementation files, and other source-code files in the project browser, those files are displayed in the code editor. Although this is a useful feature, it can sometimes require complicated mouse work, especially if you have many project files spread across many categories. The Loaded Files browser provides a navigational focus for the set of files you’re most interested in—the files that you’ve already opened.



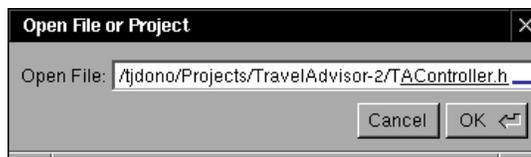
To display the Loaded Files browser, click here or choose `Tools ▸ Loaded Files`.

Click a file to redisplay it in the code editor.

The default sort order is by time visited. You can change the order to alphabetical by choosing `Tools ▸ Loaded Files ▸ Sort by Name`.

To remove a file from the Loaded Files browser, select it and choose Close from the Edit menu.

A quick way to locate files in the file system, especially good for non-project files, is to use the “Open File or Project” panel. To display this panel, choose Open Quickly from the File menu.



Press the spacebar or Escape to invoke name completion. The next possibly matching file or directory is underlined.

Tip: You can use several Emacs commands to edit the path in the Open File field: Control-a (beginning of line), Control-e (end of line), Control-k (delete to end of line), Control-f (forward character), Escape b (backward “word”), and so on.

You can also drag document icons from the File Viewer and drop them over the code editor to open and display them.

Using name completion in editing

- ▶ To cycle through the symbols *local* to a given scope and having the same prefix, type the prefix and press Escape repeatedly.
- ▶ To cycle through *all* project symbols with the same prefix, type the prefix and press Alternate-Escape repeatedly
- ▶ To display a list of all global symbols with the same prefix, type the prefix and press Alternate-I.

Name completion is a feature that displays all completions of a partial symbol name, including classes, methods, functions, constants, structures, and even local variables. It has several uses in code editing: It allows you to locate symbols that are only vaguely familiar; it also helps to prevent compilation errors due to misspellings; and it simply a convenient way to insert symbols without having to type them. With name completion, you can obtain symbols local to a file or global to the project.



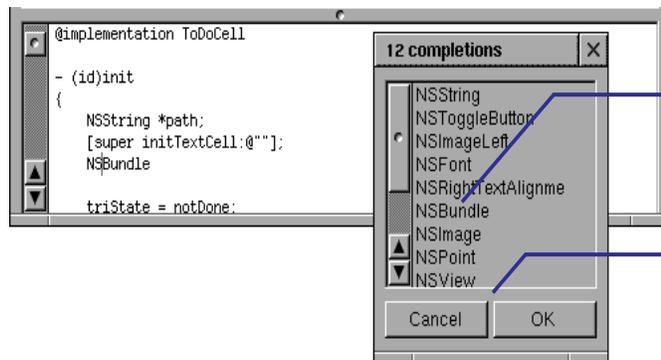
The part of the symbol that completes the prefix is underlined.

Choose a name by clicking the mouse or pressing any key other than Escape. Cancel by pressing the Backspace key.

You must type at least a one-character prefix or insert the cursor within an existing symbol.

If you insert the cursor within an existing symbol, any symbol chosen from name completion will be inserted before the existing symbol.

If you prefer not to cycle through all symbols with a given prefix, you can display a panel that lists possible completions from among the project's global symbols.



When you select a symbol name, it's immediately inserted.

Click Cancel to undo the insertion, click OK to confirm it.

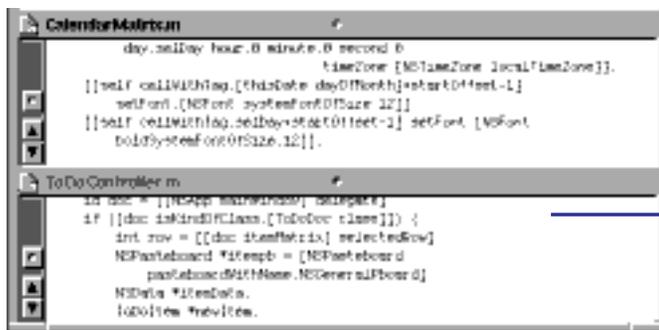
Name completion is available in many other contexts, including the fields of the Project Find and Find panels, and in some fields of the Project Inspector panel. In addition, you can use name completion to complete file names and pathnames in all Open and Save panels and in the Open Quickly file (where the space bar rather than Escape is used). See chapter 9, “Finding Information,” to learn how name completion is used in find and replace operations.

To use name completion, the project must be indexed. When a project is indexed, it “knows” about all symbols—both those that the project internally defines and those that it imports.

Displaying multiple views of code

- ▶ To open a new view in the code editor, chose File ▶ View ▶ Split.
- ▶ To tear off a window from the code editor, chose File ▶ View ▶ Tear Off.

You can edit code in multiple views in the code editor. The views can display different areas of the same file or different files. The multiple-view feature permits you to view and edit related sections of code—like method declarations in a header file and their implementations in the .m file—without have to navigate among files, and lose context in the process.



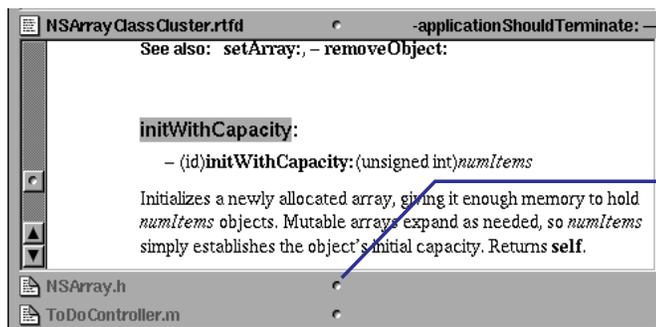
When you split a view, the new view occupies the lower half of the previous view and displays the same general contents.

Click in a view to select it. Open a file or select it from the project or Loaded Files browser to display it.

You can split views repeatedly, with each split view being halved. As you edit in one view, your changes are reflected in all other views of that same file.

Tip: Press Control-x o to cycle through the current views, selecting each one in turn.

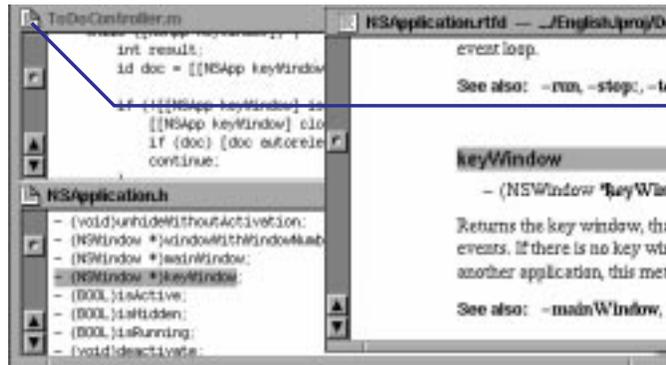
You can enlarge the editing area temporarily by “closing” views. To do this, move a divider (a bar with a dimple in its center) to the top or bottom of the code editor, or to an adjacent divider.



These two views are dragged “closed,” enlarging the editing area. To restore them, you’d drag the dividers upward.

To remove a specific view, select it and choose File ▶ View ▶ Close (the “parent” view is automatically selected next). To remove all views except the one you’re working in, select that view and choose File ▶ View ▶ Maximize.

Instead of cluttering the code editor with views that become smaller and smaller as you add each subsequent view, you can “tear off” a view and put it in its own window. To tear off a view, select it and choose File ► View ► Tear Off.



You can also create a tear-off window by Alternate-dragging the file icon off the Project Builder main window.

The window of the tear-off view behaves like any other window, except that editing of its contents is synchronized with any other view displaying the same file. To close a tear-off window, click the close button. Do not choose Close from the View menu.

Note: You cannot display files in tear-off windows by dragging file icons into them and you cannot split tear-off windows. You also cannot use the Emacs command Control-x o to jump to other tear-off windows or views in the code editor.

Undoing and redoing changes

- ▶ To undo a change, choose **Edit ▶ Undo ▶ Undo**.
- ▶ To redo an undone change, choose **Edit ▶ Undo ▶ Redo**.

Project Builder saves every editing change you make to a *kill buffer*. If you make a mistake, or decide that a modification you made earlier is not what you want, you can undo the change. Because the kill buffer is a stack, when you give the Undo command, you're undoing the most recent change; the next Undo command (without any other intervening command) undoes the previous modification, and so on until the beginning of the editing session (that is, when the file last had no unsaved modifications).

```

NSMutableDictionary *firstOfMonth, *secondDate = [self selectedDay], *now =
NSMutableDictionary *date];
int i, j; currentMonth = [secondDate monthOfYear];
unsigned int, currentYear = [secondDate yearOfComponent];
short daysInMonth;
id cell;

```

You delete these variables and their initializations.

```

NSMutableDictionary *firstOfMonth, *secondDate = [self selectedDay], *now =
NSMutableDictionary *date];
int i, j;

short daysInMonth;
id cell;

```

```

NSMutableDictionary *firstOfMonth, *secondDate = [self selectedDay], *now =
NSMutableDictionary *date];
int i, j;
unsigned int currentYear = [secondDate yearOfComponent];
short daysInMonth;
id cell;

```

Choose the Undo command once.

After deselecting the restored text, choose Undo again.

```

NSMutableDictionary *firstOfMonth, *secondDate = [self selectedDay], *now =
NSMutableDictionary *date];
int i, j; currentMonth = [secondDate monthOfYear];
unsigned int currentYear = [secondDate yearOfComponent];
short daysInMonth;
id cell;

```

If you leave the text selected, the change is left undone when you perform the next Undo.

In undoing changes, deselect code if you want to retain it; keep the code selected to continue cycling through Undos.

To reinstate a change that you've just undone, give the Redo command. For example, if you decide that you don't want the **currentMonth** variable you've just restored by undoing (last illustration, above), choosing Redo will yield this:

```

NSMutableDictionary *firstOfMonth, *secondDate = [self selectedDay], *now =
NSMutableDictionary *date];
int i, j;
unsigned int currentYear = [secondDate yearOfComponent];
short daysInMonth;
id cell;

```

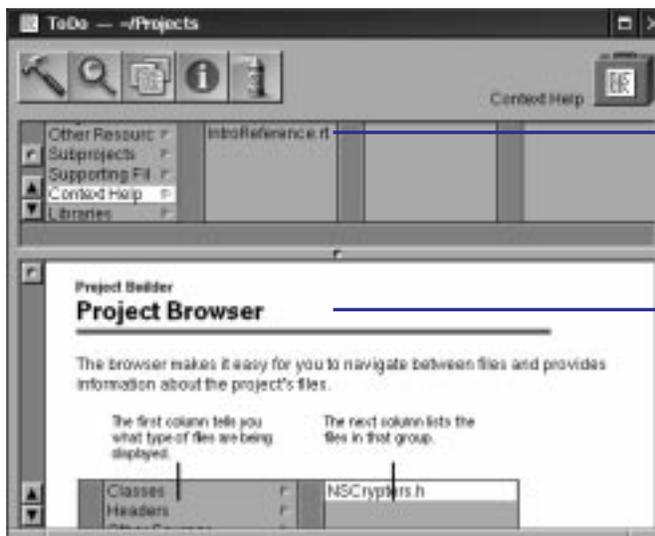
Successive Redo commands reverse the effects of each successive Undo.

Instead of undoing changes in succession by repeatedly choosing the Undo command, you can simultaneously undo all changes made to a region of code since the beginning of the last editing session. Simply select the region and choose **Edit ▶ Undo ▶ Undo Region**.

Formatting and adding graphics to RTF text

- 1 **Create or add an RTF or RTFD file as a project or non-project file.**
- 2 **Format the text using the commands on the Format menu or on one of its submenus.**
- 3 **If you want to add graphics, drag and image from the File Manager or from the Images suitcase and drop it over the code editor.**

You can create and format RTF (rich text) and RTFD files in Project Builder. These files can be project or non-project files. Typical RTF project files are context-sensitive help files and, for framework projects, reference documentation. A common non-project RTF file might be a README file.



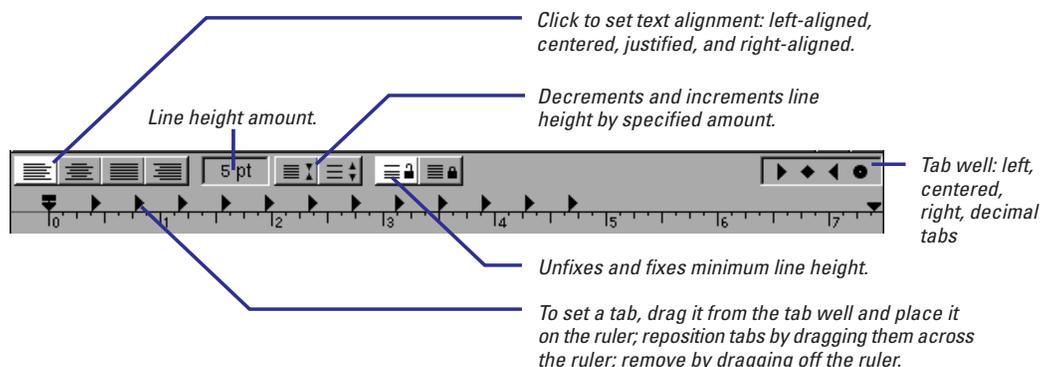
Create or add context help files here. Non-project files go in the Non Project Files "suitcase" of the project browser.

Text can have various attributes, including font, size, style, and color. It can also include graphical images.

You can find RTF editing commands on the Format menu and its submenus. Possibly the most important of these is the one that displays the Font panel (Format ► Font ► Font Panel). The Font menu also has submenus of commands for performing fairly sophisticated typographical operations

Font Submenus	Description
Kern	Adjusts the spacing between selected letters.
Ligature	Use All or Use Default joins certain combinations of characters, such as "fl".
Baseline	Superscript and Subscript commands lower and raise the selected text's baseline by a set amount. The Raise and Lower commands adjust the baseline incrementally.

You can display a ruler above RTF text by choosing **Format ▶ Text ▶ Show Ruler**. This ruler enables you to set margins and tabs in selected text, to adjust line height (spacing between lines), to set alignment, and to lock changes.



Note: The unfixed (open lock icon) and fixed (closed lock icon) buttons affect the minimum line height. If the fixed button is selected, users can decrement the line height below the limit set by the highest character. If the unfixed button is selected, the line height cannot be adjusted below that limit.

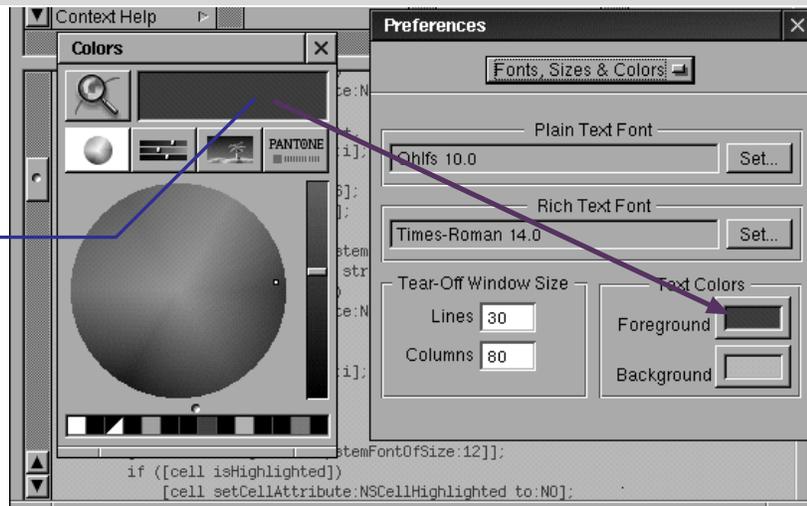
Code files are ASCII files by default. You can, however, convert them to RTF by inserting the cursor anywhere in the file and choosing **Make Rich Text** from the **Format** menu. You can then give portions of the code their own attributes; for instance, comments can be in blue and recently modified text can be in bold face. Code with RTF attributes can be safely compiled.

Customizing the Editing Environment

In the **Fonts, Sizes & Colors** preferences panel, you can customize the default attributes of the code editor, including text color and font, background color, and the size of tear-off windows. Attributes take effect when the next file is opened or when you create the next tear-off window.

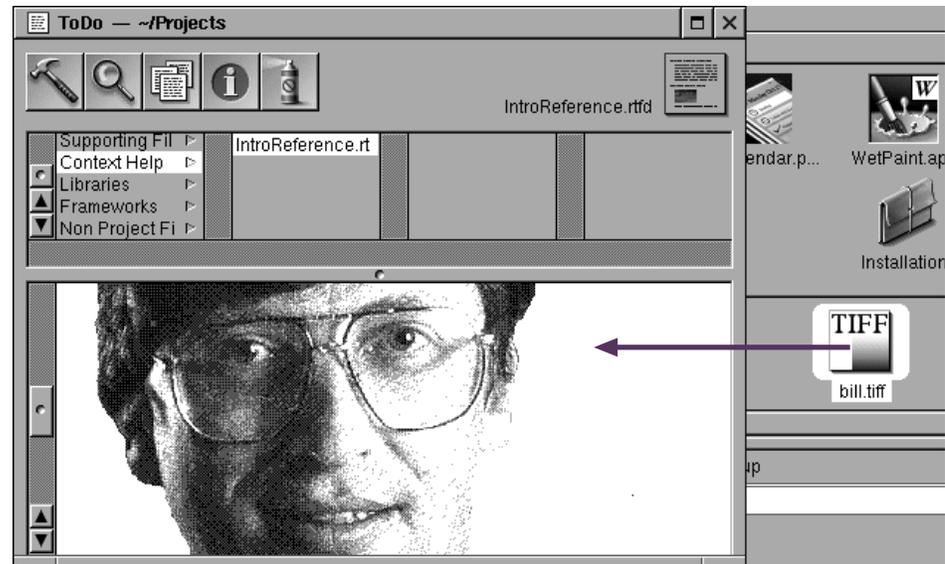
Drag a color into one of the Text Colors color wells to set the default foreground (text) and background colors.

The plain text font should be a fixed-pitch font to ensure that indented lines are aligned properly.



RTFD files are Rich Text Format files that can display graphical images. You can easily add EPS and TIFF images to RTF text displayed in Project Builder.

You add images by dragging them from the File Viewer or Project Builder and dropping them in the code editor; they're inserted where the cursor is. (These files become RTFD files in the process, if they're not already.).



8

Finding Information

Using the project find panel

Searching for project symbols

Searching for any text

Looking up reference
documentation

Looking up man pages

Replacing code all at once

Replacing code sequentially

Attempt the end, and never stand to doubt;
Nothing's so hard, but search will find it out.

Robert Herrick

As the power of endurance weakens with age, the urgency
of the pursuit grows more intense...And research is always
incomplete.

Mark Pattison

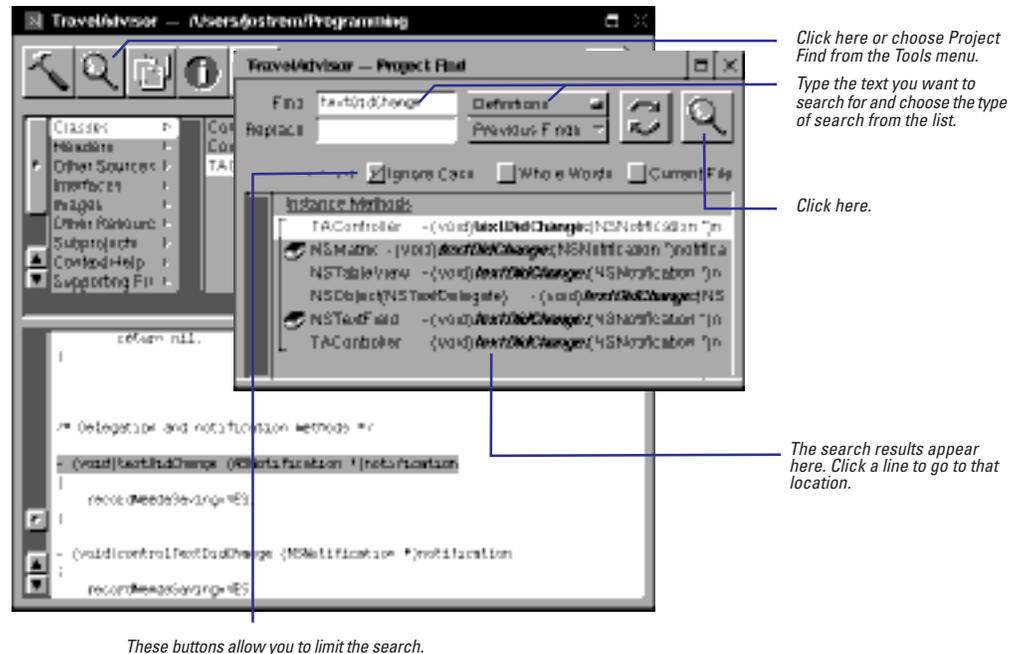
Where ask is have, where seek is find,
Where knock is open wide.

Christopher Smart

Using the project find panel

- 1 Click the Project Find button in Project Builder's main window.
- 2 Enter text to search for in the Find field.
- 3 Choose the type of search from the pop-up list.
- 4 Click the Find button.

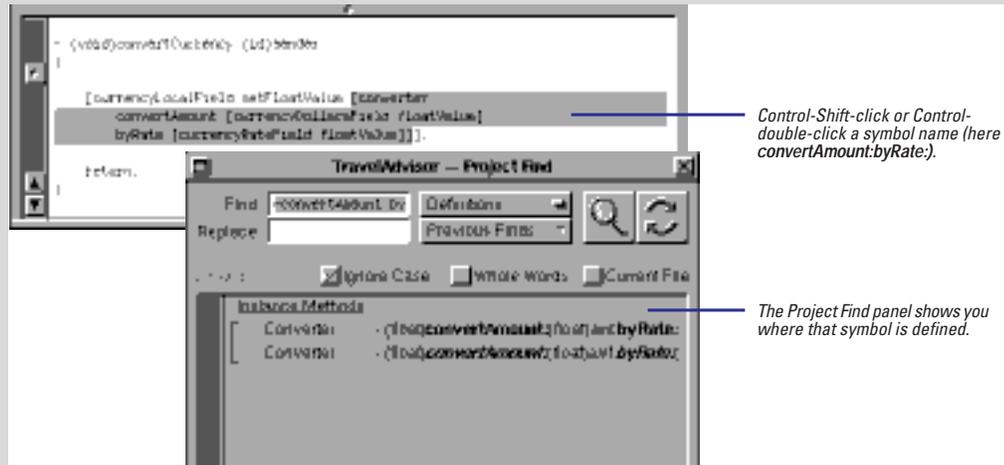
The Project Find panel isn't just a find panel—it's an information source. It searches your entire project, including the frameworks. It can tell you what type of symbol (such as class, method, or variable) the string you entered is, how that symbol is defined, where that symbol is used, and if there's documentation. Then, like any good find panel, it takes you right where you want to go.



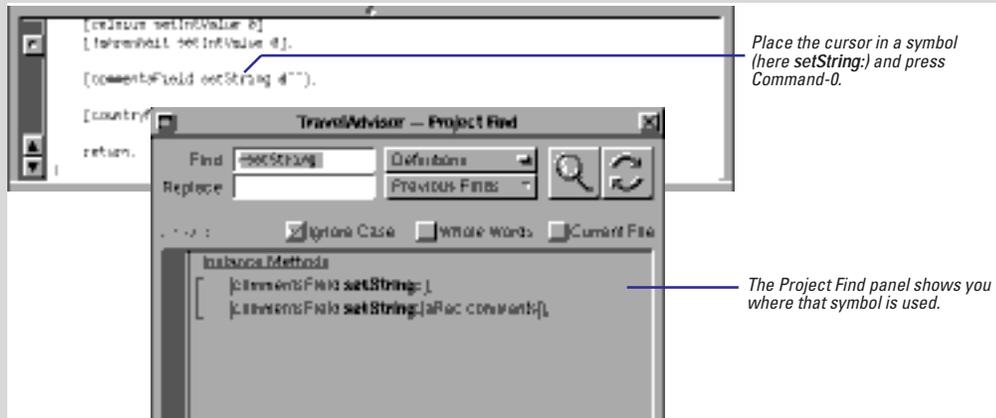
The Project Find panel makes use of the project index. If your project is not indexed, you can only perform textual searches. For more information on the project's index, see Chapter 1.

Project Find Shortcuts

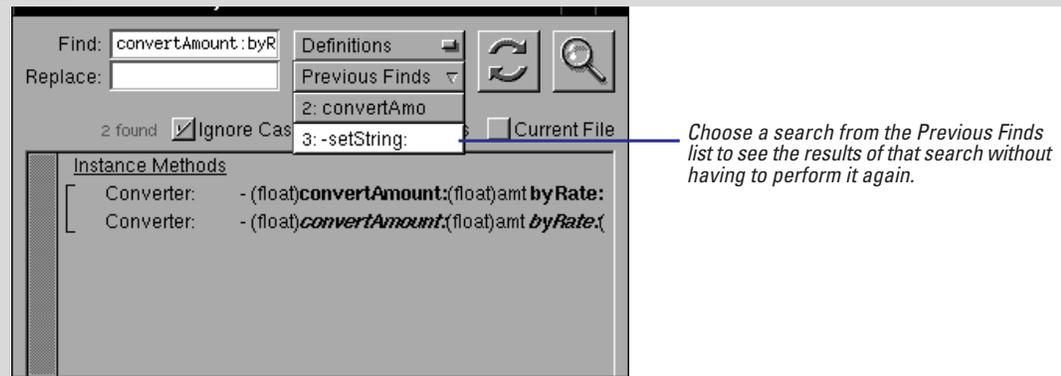
Definition Search Shortcut



Reference Search Shortcut



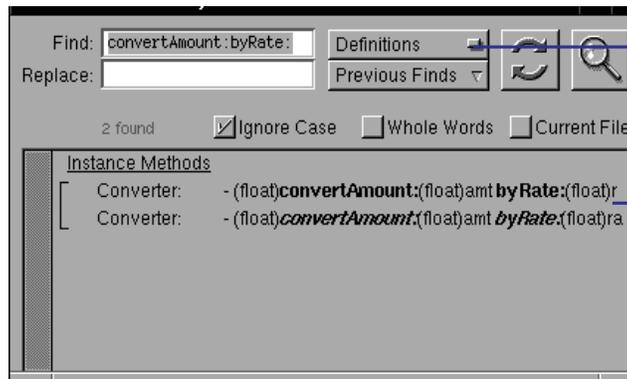
Previous Finds



Searching for project symbols

- ▶ To find out how and where a symbol is defined, choose **Definition search** in the Project Find pop-up list.
- ▶ To find where a symbol is used, choose **Reference search** in the pop-up list.

If you need to look up the syntax of a method or the type of a variable, perform a Definitions search in the Project Find panel. Project Find looks for any and all symbols named with the specified string and then shows both the line where the symbol is declared and where it is defined.



Choose Definitions from the pop-up list to find the symbol's definition.

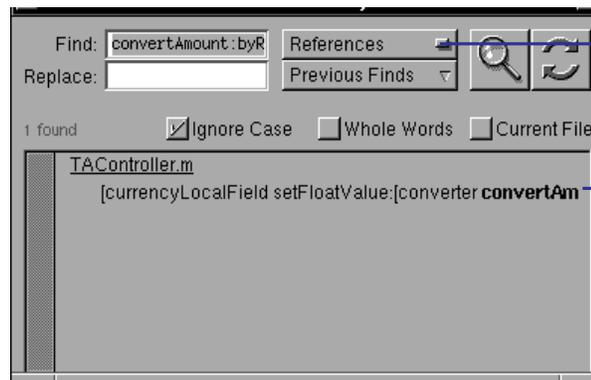
The results are classified by symbol type and contain the locations of both declarations and implementations.

Quick Find

If you know the name of a symbol, you can quickly and directly ascertain where it is defined or referenced. Type a known class, protocol, method, function, macro, or type in the Find field using a special syntax described below. Then click the Find button to display definitions.

Class	@class
Protocol	<protocol>
Instance method	-method
Class method	+method
Method in particular class	[class method]
Function	function()
Macro	#macro

Sometimes, you want to know where a symbol is used rather than where it is defined. In these cases, you should choose References from the panel's pop-up list. The Reference search will show you each line of code that references the specified symbol. Click one of the lines to jump to that location.



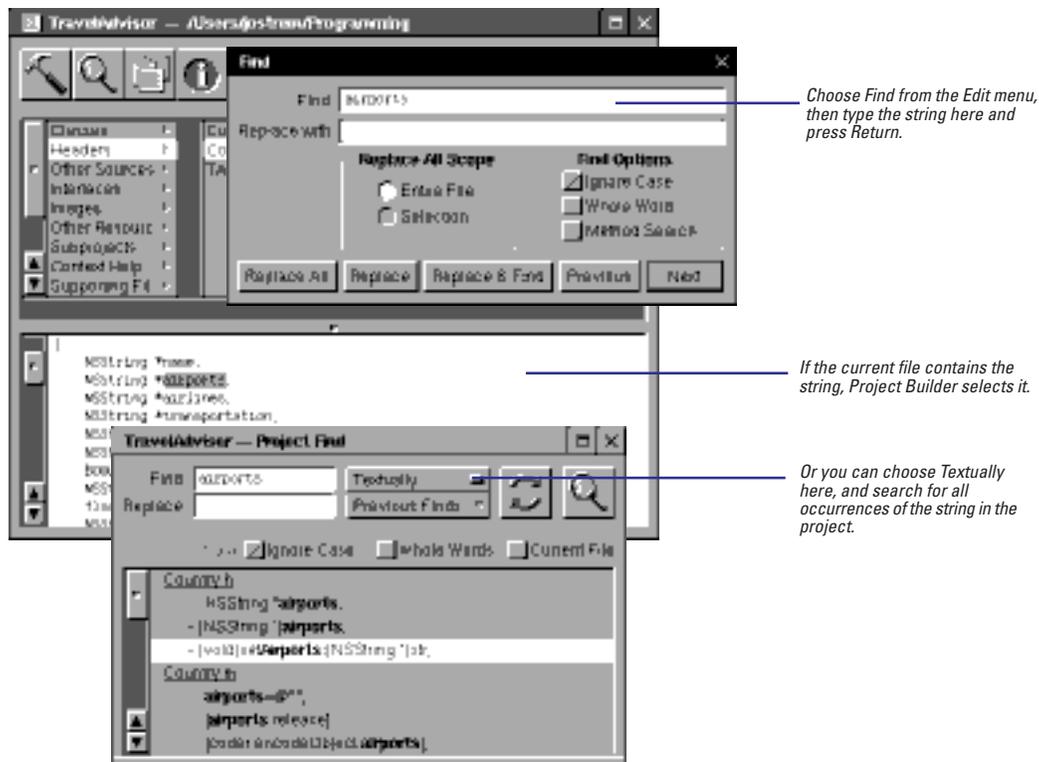
Choose References from the pop-up list to find where the symbol is referenced.

The results show all uses of the symbol within your source code.

Searching for any text

- ▶ To search for a string in a particular file, choose Find from the Edit menu.
- ▶ To search for a partial string within a file, type Control-s to use the incremental search facility.
- ▶ To search for a string project wide, use the Project Find panel and choose searching Textually.

Sometimes it's better to perform a text-based search than a language-based search. For example, if you know you're in the correct file and you just want to go to a different place in that file, you can use the text-based search within a file. Just choose Find from the Edit menu or type Command/Control-f, and the standard OpenStep find panel is displayed.



Choose Find from the Edit menu, then type the string here and press Return.

If the current file contains the string, Project Builder selects it.

Or you can choose Textually here, and search for all occurrences of the string in the project.

Customizing Searching

Project Builder's Preferences panel allows you some control over incremental searching and over the Project Find panel. To see these preferences, choose Preferences from the Info menu and choose Find from the pop-up list.

The first two groups of preferences control incremental searching. Turn on the wrapping preferences if you want incremental searching to wrap around to the beginning of the file when it reaches the end.

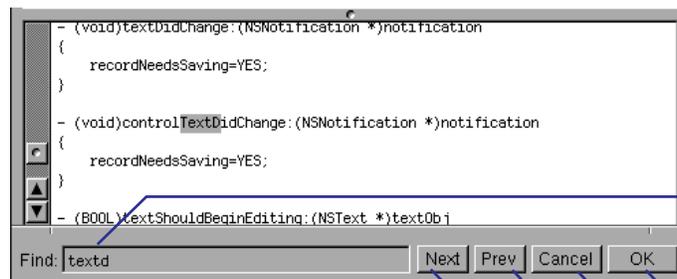
Allow searching for newlines lets you enter a newline in the search string. By default, typing a newline will exit the incremental search window.

If Ignore case is set, incremental searching ignores the case of the characters you type, instead searching for both the upper and lower-case version of the letter. If you want a case-sensitive search, turn off this preference.

The Language-based find preferences control the searches you perform from the Project Find panel. If you normally use the right mouse button to display the menu in OPENSTEP applications, you can enable the right mouse button in Project Builder as an alternative to the Control-Shift-clicking a symbol. (It performs a Definition search on that symbol.) The Order results by type options groups definition searches by the file in which they were found rather than by the Objective-C symbol type.

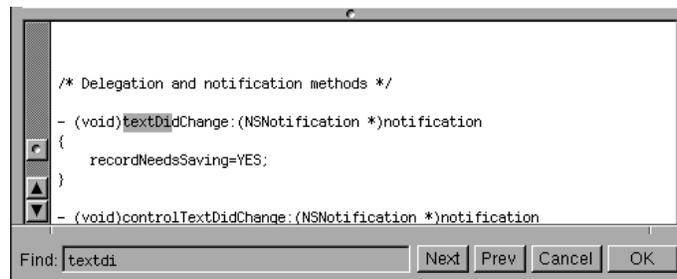
Note: When you choose Textually in the Project Find panel, only your project's files are searched. Frameworks and documentation aren't searched.

Another nifty way to get to where you want to go is to use the incremental search and replace facility. Incremental searching is a quick way to search if you know you're in the right file and you only know part of the symbol's name (or you want to type as few characters as possible).



Press Control-s, then begin typing a string.

These buttons allow you to search forward or backward, to cancel the search, or to end it.



As you continue typing, the selection moves. When you want to stop, press Return or click the mouse anywhere.

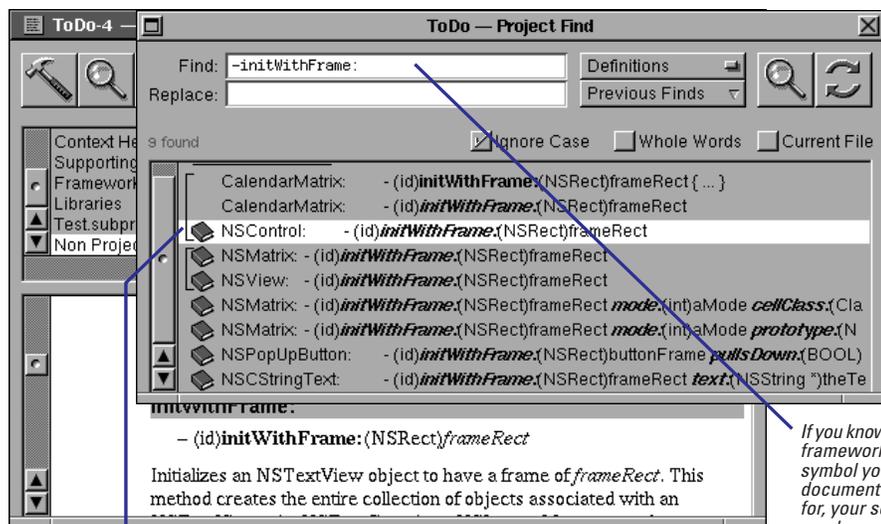
Tip: You can also use incremental search to navigate through the hits in the Project Find panel. Just type Control-s while the Project Find window is key.

In addition to the file-based Find panel and the incremental search facility, the Project Find panel can also perform text-based searches. This might be useful when you're trying to find text inside a comment and you don't know which file contains the comment.

Looking up reference documentation

- 1 **Open the Project Find panel.**
- 2 **Perform a search for definitions of project symbols.**
- 3 **Click the book icon to the left of a search-result item.**
- 4 **Examine the documentation displayed in the code editor.**

The Project Find gives you access to documentation on the OpenStep frameworks imported by your projects. For applications, these frameworks always include the Application Kit and Foundation. By linking descriptions in the documentation to individual results of searches for symbol definitions, Project Find enables you to navigate quickly to a particular description.



The book icon indicates that reference documentation on the class, protocol, method, function, or other type is available.

If you know which framework symbol you want documentation for, your search can be narrow and explicit.

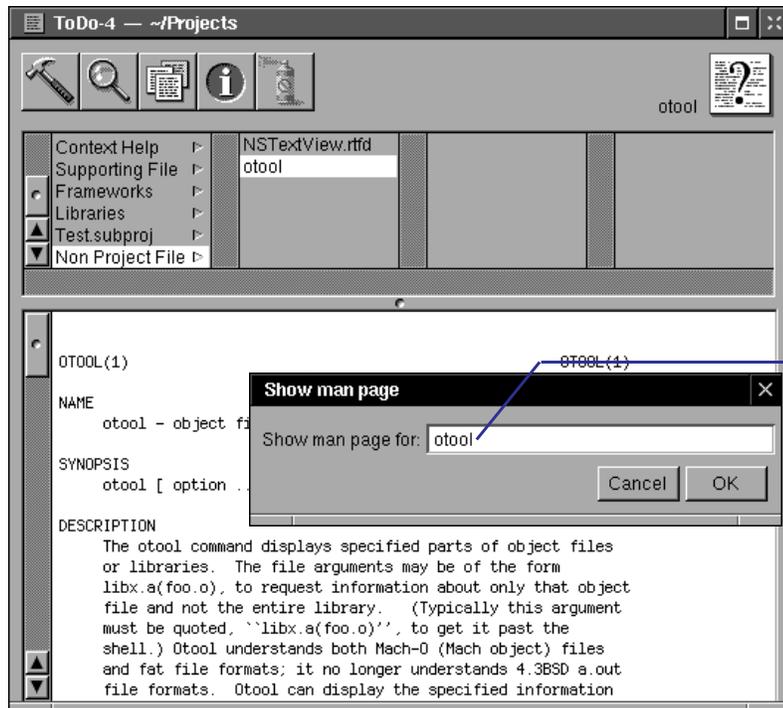
The related passage in the documentation appears in the code editor. From here you can browse through the rest of the file; if the file you're viewing is a class or protocol specification, you can go to the beginning of the file to read an overall description of the class (or protocol) and its methods. Project Builder puts documentation files in Non Project Files so you can access and examine these files throughout a coding session. As with code files, you can display documentation in a separate tear-off window or in an additional view in the code editor.

Looking up man pages

- 1 Choose **Edit ► Find ► Man Page**.
- 2 In the “**Show man page**” panel, **type the name of a standard C library routine, or command-line program**.
- 3 Click the **OK** button.
- 4 Examine the man page in **Non Project Files**.

For access to OPENSTEP documentation other than framework reference, access the Digital Librarian bookshelf **NextDeveloper.bshlf**. This bookshelf also provides instructions for creating your own bookshelf and customizing it with documentation targets.

Rhapsody includes the man pages facility, which displays information on standard C library routines, command-line programs, and (for Mach), custom programming tools provided by NeXT. Project Builder gives you access to this facility, eliminating the need to switch contexts in order to issue the **man** command at a Terminal shell.



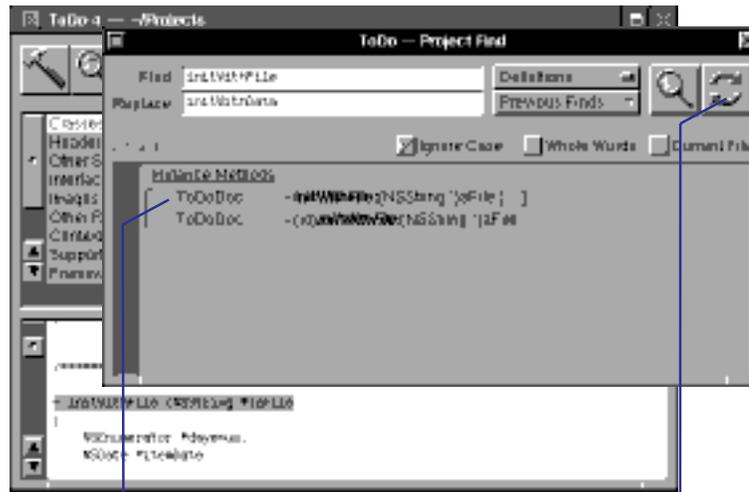
You must type the complete name of the routine, tool, or utility. Name completion is disabled in this field because these names are not part of the project.

The related man page appears in the code editor. Project Builder puts referenced man pages in Non Project Files so you can access and examine these files throughout a coding session. As with code files, you can display man pages in a separate tear-off window or in an additional view in the code editor.

Replacing code all at once

- 1 **Open the Project Find panel.**
- 2 **Perform a search for symbol definitions or references, or for text strings.**
- 3 **Type in the Replace field the text to replace the entry in the Find field.**
- 4 **Replace the text globally or selectively:**
 - ▶ **Globally:** Click the Replace button.
 - Or*
 - ▶ **Selectively:** Select the items in the search results to replace, then click the Replace button.
- 5 **Confirm the replacement.**

The Project Find panel allows you to replace symbols in your code or other text at one time. Use the Project Find panel when you prefer simultaneous replacement project-wide rather than sequential replacement within a file.

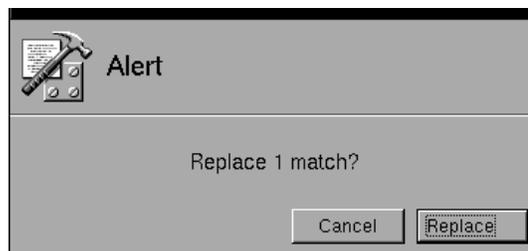


You can select items and review them to determine if they should be replaced. Shift-click to unselect items.

The Replace button

Select contiguous multiple items by dragging across them; select non-contiguous items by Shift-clicking. If you want the replacement made to every item in the search results, leave all items unselected.

When you click the Replace button, Project Builder prompts you for confirmation before performing the replacement.



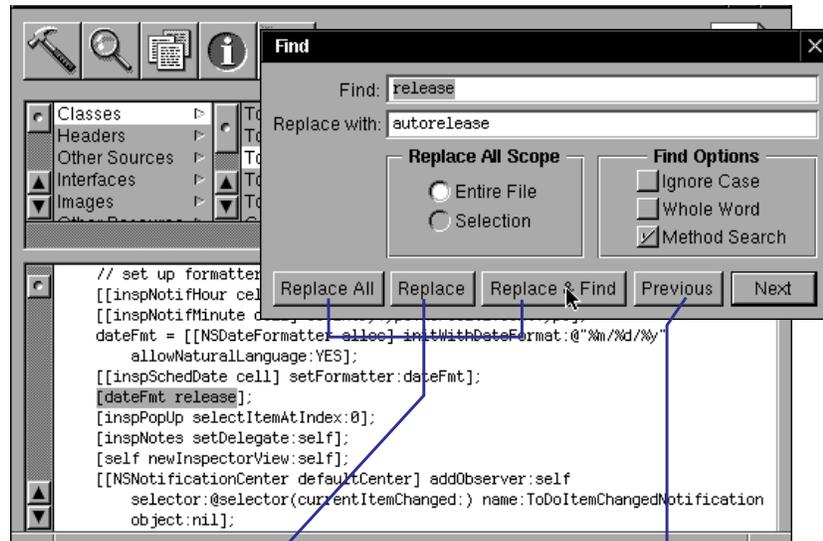
See the following task, “Replacing code sequentially,” for this alternate search-and-replace technique.

You can undo replacement changes with successive Edit ▶ Undo ▶ Undo commands.

Replacing code sequentially

- 1 **Choose Edit ► Find ► Find Panel.**
- 2 **Type the string of text to find in the Find field.**
- 3 **In the Replace field type the string of text to replace what's in the Find field.**
- 4 **Click Next.**
- 5 **Review the highlighted text in context, replace (or don't replace) the text, and:**
 - **Go on to the next occurrence.**
Or
 - **Conclude the find session.**
Or
 - **Replace all occurrences to the end of the file.**

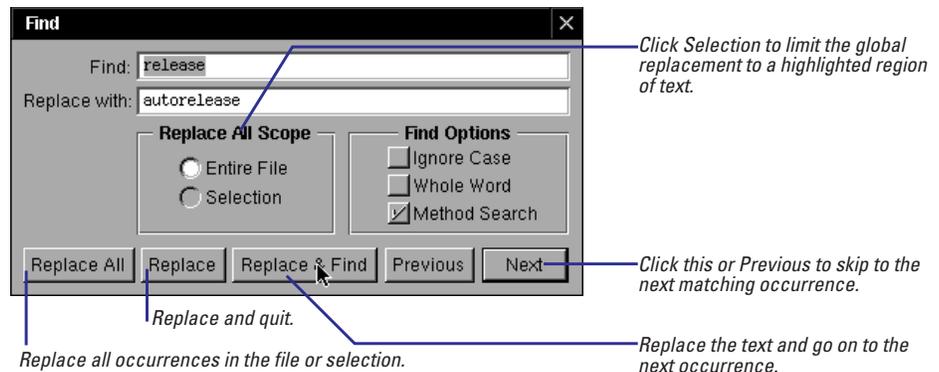
The Find panel differs from the Project Find in a couple of significant ways. It limits the text that it replaces to the file currently displayed in the code editor (or to a selected region). And it allows you to replace code or other text sequentially by highlighting each successive find in the code editor, thereby giving you the opportunity to evaluate the change in context before making it.



To replace code, click one of these buttons.

Click to begin the search in reverse direction.

Once you begin a search, you have several replacement options at each matching occurrence of text:



You can undo replacement changes with successive Edit ► Undo ► Undo commands.

9

Building

Building the program

Building for multiple architectures

Building on a remote computer

Using build targets

Creating your own build targets

Setting search paths

Setting compiler and linker options

Creating a precompiled header

Customizing your makefiles

When we build, let us think that we build for ever.

John Ruskin

Well building hath three Conditions: *Commodity, Firmness, and Delight.*

Sir Henry Wotton, *Elements of Architecture*

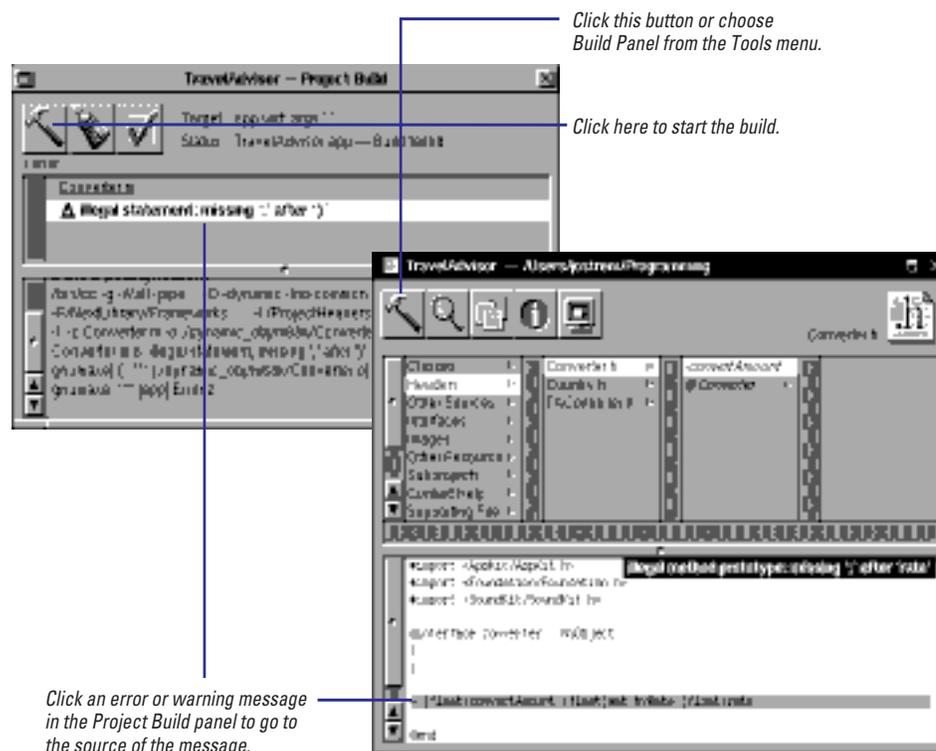
Burrow awhile and build, broad on the roots of things.

Robert Browning, *Abt Vogler*

Building the program

- 1 Click the **Build** button to bring up the **Project Build** panel.
- 2 In the **Project Build** panel, click the **Build** button to start the build.

When you build a program in Project Builder, you are really invoking the **make** utility to create an executable. The **make** utility invokes the compiler and linker using information from the project makefile.



Before the build begins, Project Builder prompts you to save any unsaved source files, nib files, or the project itself. In this way, Project Builder ensures that you are always building the latest version of the project.

Before you build, you might want to set specific build options for the project or modify its makefiles. The rest of this chapter describes these tasks.

If the build fails because of link errors, chances are you aren't linking against the correct library or framework. See "Some OPENSTEP Libraries" in this chapter.

If the build fails because of link errors, look in the bottom browser for more information. The bottom browser shows the exact compiler and linker commands being executed, and it shows all messages produced by these commands. If the Project Build panel appears to only have one browser, drag the split view knob up until you can see the bottom browser.

All About make and gnumake

make is a standard command-line program that builds programs. Its main purpose is to make it easy for you to perform incremental builds. Project Builder uses **gnumake**, a version of **make** written by the Free Software Foundation.

A make Terminology Primer

You tell **make** how to build a program by creating a *makefile*. A makefile consists of *rules*, which in turn are made up of *build targets*, a list of *dependencies*, and one or more *commands*.

Build targets are the targets of the **make** command, that is, what you want to build. In Project Builder, there are several provided targets that build the project in different ways. The default is to build an optimized, debuggable executable. You can select a different target to turn optimizations off, to generate profiling information, and to install the project in its end location. (One special target is **clean**, which removes all object files and executables, forcing a full build the next time around.)

Dependencies are the input files used to create the target. For example, an executable depends on the object files linked together to create it. When **make** is asked to build an executable, it checks all object files listed as dependencies. If the object file is out of date or does not exist yet, it creates that object file by compiling the source file. Once all object files listed as dependencies are created and are up-to-date, **make** can link them together to create the executable.

Commands are the commands used to create the target. For example, to create an application, you need to specify commands that compile all of the source code files into object files, link the object files into an executable, create the application's wrapper directory, and copy the executable and the application's resources into that directory.

A makefile can also contain *macros*, which make it easier to write consistent makefile rules. Makefile macros serve the same purpose as **#define** macros in a C program. They make it easier to update the makefile. For example, you can define a macro **CFLAGS** to be all of the flags you usually pass to the Objective-C compiler and use it everywhere you specify a compile command. Then, if you want to change one of these options, you only need to change it in one place, in the definition of **CFLAGS**.

If you need to update makefiles at all in Project Builder, you usually only need to redefine some provided macros. If you want to, you can also create your own targets. This chapter describes how to perform these tasks.

gnumake

gnumake, from the Free Software Foundation, is now the default **make** utility for OPENSTEP. **gnumake** has many features that aren't available in other **make** utilities.

Some of the unique things you can do if you use **gnumake** are:

- Perform parallel compiles so that your project builds faster.
- Use conditional statements in a rule. You can use this feature to specify in a single rule different compiler arguments based on which type of compiler you are using. In other **make** utilities, you would have to define two different rules.
- Use a standard set of functions to manipulate strings and filenames used in the makefile. For example, **gnumake** provides functions that return a specified file's extension, its basename, and its directory.
- Define a macro based on its own previous definition. For example, **gnumake** allows you to say `'CFLAGS := $(CFLAGS) -O'` which assigns to the macro **CFLAGS** its previous definition with **-O** appended to it.
- Define a macro that contains a newline using the **define** directive.
- Use **MAKELEVEL** to keep track of recursive use of **make**.
- Declare a phony target with **.PHONY**.
- Specify a search path for included makefiles and specify extra makefiles to be read with an environment variable.
- Use **vpath** to specify a search path for files with a particular extension.
- Use a special search mechanism for libraries by specifying **-lname** as a dependency. This causes **gnumake** to search for the library in the **VPATH**, then in **vpath**, then in **/lib**, **/usr/lib**, and **/usr/local/lib**.

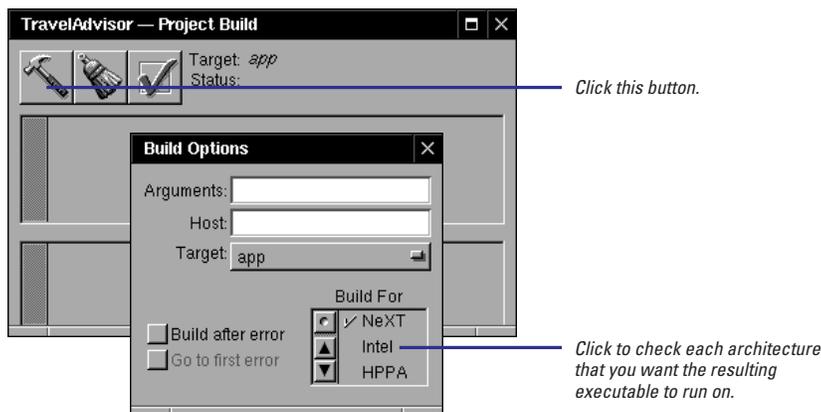
For More Information

A number of books describe **make** in general terms. If you need to learn more about **gnumake**, see the document "GNU Make" provided by the Free Software Foundation.

Building for multiple architectures

- 1 Click the **Build** button to bring up the **Project Build** panel.
- 2 In the **Project Build** panel, click the check-mark button.
- 3 In the **Build Options** panel, select the architectures you want to build for.

Usually when you build, you create an executable that runs only on the type of computer you used. If you build on an Intel computer, the executable will run only on Intel computers. If you want the executable to run on more than one type of computer, you need to set this up in the Build Options panel. You can build executables for any architecture that OPENSTEP currently supports: Intel, NeXT, and SPARC. You must have the libraries for that architecture installed on your computer.



Portability Do's and Don'ts

If you build multiple architecture ("fat") binaries or you build code on one architecture to run on another, make sure you're writing portable code. If you use the OpenStep libraries and avoid hard-wired data values, your application will probably be portable. Here's a list of some specific do's and don'ts for writing portable code.

- *Do* use relative values when positioning windows on the screen. *Don't* use absolute positions.
- *Do* use the `NSEvent characters` method to find out what key was pressed. *Don't* use `keyCode`.
- *Do* use `sizeof` when passing the size value to `malloc()`. *Don't* use a constant.
- *Do* refer to a structure's fields and a function's parameters by name. *Don't* try to deconstruct data formats, such as `float` or `struct`, or a function's argument list yourself.
- *Do* use the OpenStep objects `NSData`, `NSString`, and `NSDictionary` to read and write external data. *Don't* rely on a particular byte order or alignment when reading and writing external data.

Building a Multiple Architecture ("Fat") Binary

The following sequence of events occurs when you build a fat binary:

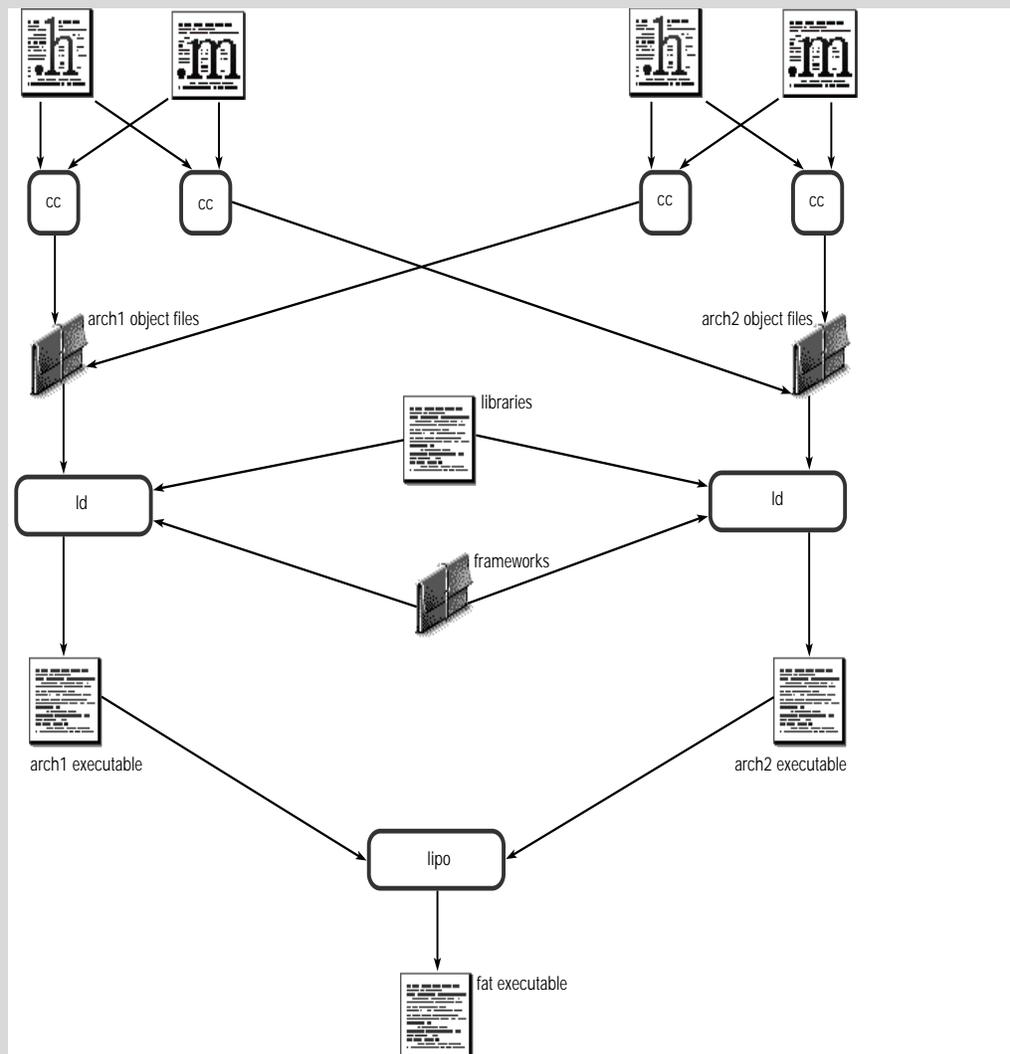
- Each source file is compiled once for each architecture to produce thin object files.

The object files are stored in subdirectories under `dynamic_obj` (or `dynamic_debug_obj`) that are named for the processor. For example, if you build for both Intel and NeXT, there are two directories under `dynamic_obj`: `i386`, containing object files for Intel, and `m68k`, containing object files for NeXT.

- After all of the source files have been compiled, the linker is invoked once for each architecture to produce thin executables.

Each executable has an extension that describes the type of processor it runs on, for example, `MyProject.m68k`.

- After an executable has been built for each processor, the `lipo` command is invoked to combine the executables into one binary file named `MyProject`.



Three Ways to Set Build Options

You can set build options in three different places: in the Preferences panel, in the Project Inspector panel, and in the Build Options panel. Each panel has a unique purpose.

- Use the Build Preferences panel to set options you're always going to use, no matter which project you're working on.

For example, you may want to change the sound you hear upon each successful build. To open the Build Preferences panel, choose **Info** ► **Preferences**, then choose **Build** from the pop-up list in the Preferences panel.

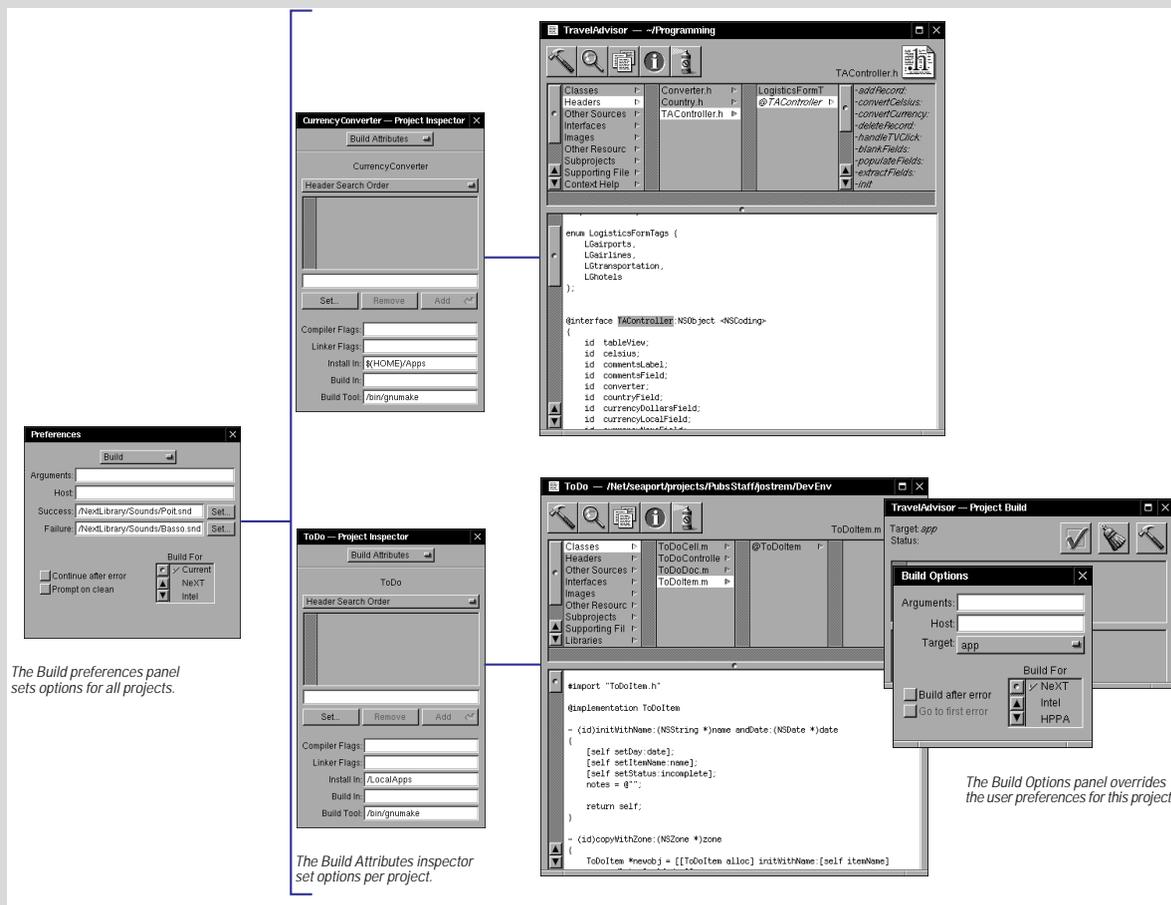
- Use the Build Attributes inspector to set options that apply to a specific project, no matter which user is working on the project.

For example, you might define a build target specific to one

project. Or you might want to use a specific compiler option for one project. To bring up the Build Attributes inspector, choose **Tools** ► **Inspector** ► **Show Panel**, then choose **Build Attributes** from the pop-up list in the Project Inspector panel.

- Use the Build Options panel to set your preferences for a specific project. Options on this panel apply only when you yourself are building the project.

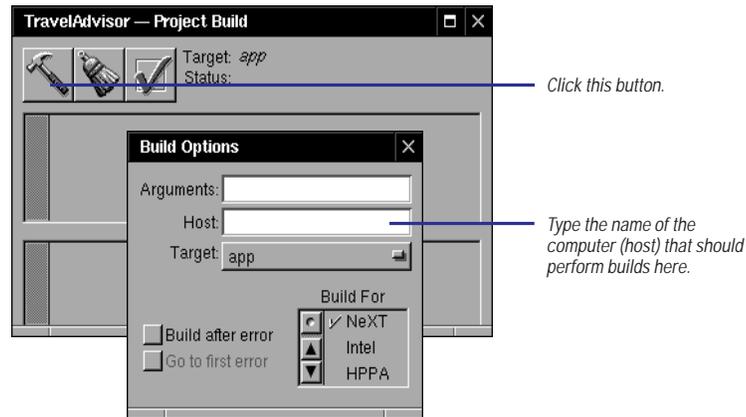
To bring up this panel, click the check mark button on the Project Build panel. The options on the Build Options panel remain set even after you quit Project Builder.



Building on a remote computer

- 1 Click the Build button to bring up the Project Build panel.
- 2 In the Project Build panel, click the check-mark button.
- 3 Type the host name of the computer that should perform the build in the Host field.
- 4 Build the program.

A build takes up a lot of your computer's CPU time and disk resources. You can still perform other tasks while the build is running, but these other tasks may run slower. If this happens, you may choose to build remotely on another computer on your network. This way, the CPU on your computer can be dedicated to the other tasks you are performing.



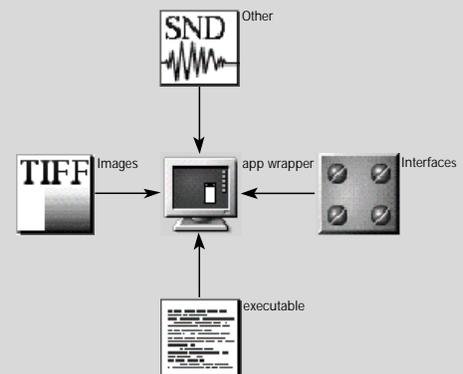
There's no difference in what you see when you build on another host; the Project Build panel still displays the status of the build and updates as the build status changes.

Note: Be sure you know what version of OPENSTEP the host is running before you use it to build your project.

The App Wrapper and Other Bundles

Some project types (namely Application, Loadable Bundle, and Framework) don't just produce an executable when you build them. They create the executable, and then they create a directory containing the executable and the project's resources (the files under Images, Interfaces, and Other Resources in the project browser).

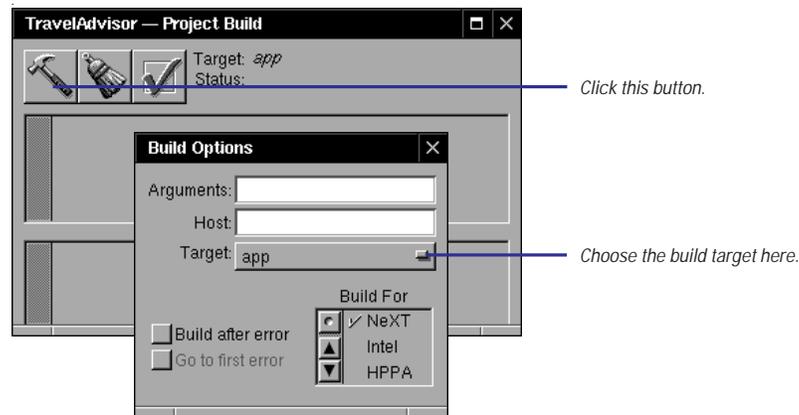
The generic term for a directory containing such items is "bundle." When the bundle contains an application, it is often called an "app wrapper" because it wraps up all of the things an application needs into a single unit.



Using build targets

- 1 Click the **Build** button to bring up the **Project Build** panel.
- 2 In the **Project Build** panel, click the **check-mark** button.
- 3 In the **Build Options** panel, choose a target from the **Target** pop-up list.

A *build target* is an argument passed to the **make** utility that tells it which makefile rules to use when building. The default build target, which is named for the project type, produces an optimized, debuggable executable and places it in the project directory. This target is often suitable, so in many cases you don't have to worry about the build target. If you need a different target, choose it from the **Build Options** panel before you build the project



To do a **make clean**, click the broom button on the **Project Build** panel. As described in “All About **make** and **gnumake**,” **make clean** is a special target that deletes all object and executable files.

If you're running **OPENSTEP** on a **RISC** architecture and you need to debug, you may want to choose a target that does not optimize your code.

Other Build Targets

Besides the default target, the one named for the project type, the other available targets are:

debug Compiles with **-DDEBUG** on and optimizations off. Use this target if your program uses the **DEBUG** macro to provide more debugging information or if you want to make sure local variables do not get optimized while you are debugging.

install Places the executable in the installation directory specified in the **Build Attributes** inspector.

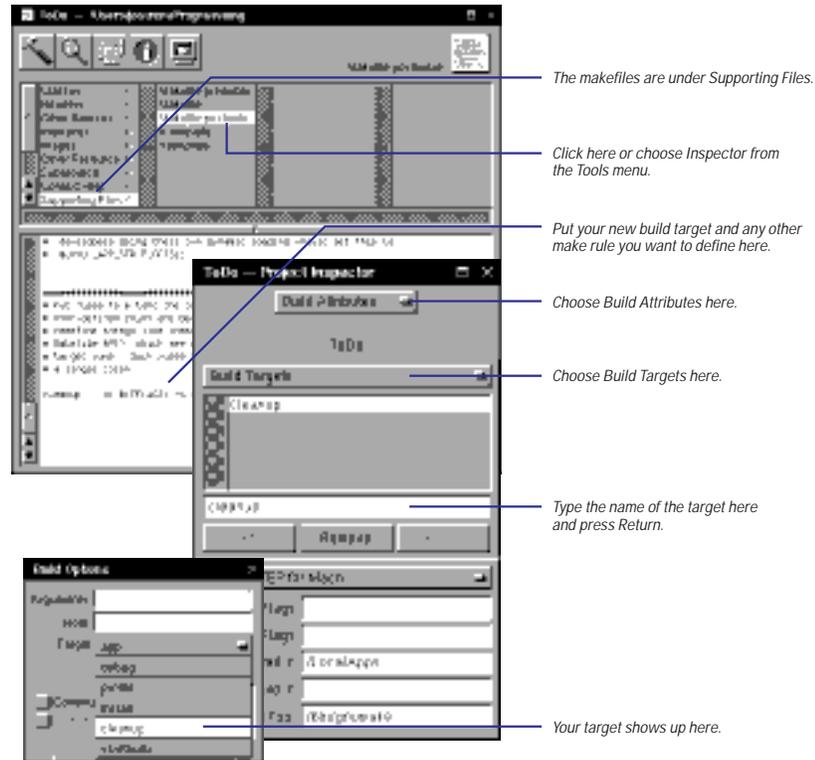
profile Generates (with **-DPROFILE**, **-pg**, and all warnings on) a file containing code to generate a **gprof** report. Use this target when you are tuning the performance of an application. See the **gprof** man page for details.

<default> Uses the first target listed in the makefile. **Warning:** If you place a target at the end of the **Makefile.preamble** file, it becomes the default target.

Creating your own build targets

- 1 Define a new target in the **Makefile.postamble** file and save the file.
- 2 Click the **Project Inspector** button and choose **Build Attributes** in the **Project Inspector** panel.
- 3 Type the name of the target in the **Build Targets** list.

If the targets provided by Project Builder don't meet your needs, you can define your own target in the file **Makefile.postamble**.



Project Builder uses the information you specify in the Build Attributes inspector to index the project and to update the project makefile. You'll read about the other fields in the Build Attributes inspector later in this chapter.

If you refer to the executable name in your target, use the **EXECUTABLE_EXT** makefile macro to give it the correct extension. **EXECUTABLE_EXT** is **.exe** in Windows environments and nothing on Rhapsody and supported UNIX environments. For other makefile macros, see the section "Customizing your makefiles" in this chapter.

Caution: Don't use **Makefile.postamble** to redefine targets that are already defined (debug, install, profile, app, and so on). If you do, the results are unpredictable.

After you define a target, you need to let Project Builder know about it so that it appears in the Target pop-up list in the Build Options panel. You do this using the Build Attributes inspector.

Setting search paths

- 1 Click the **Project Inspector** button and choose **Build Attributes** in the **Project Inspector** panel.
- 2 Add the library's location to the **Library Search Order** list.
- 3 Add the location of its header files to the **Header Search Order** list.

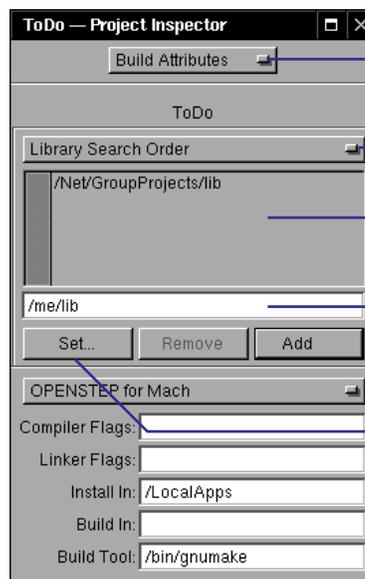
Or

- 2 Add the framework's location to the **Framework Search Order** list.

The compiler and linker search a standard set of directories for library executables and header files. If you link with a library or framework that is not stored in one of the standard locations, you need to add its location to the search path. You do this from the Build Options inspector.



Click here or choose *Inspector* from the *Tools* menu.



Choose *Build Attributes*.

Choose the appropriate search order list.

Nonstandard directories that are already a part of the search path are displayed here.

Type the directory to be searched here and click *Add*.

Or click here and choose the directory to be searched from the panel that appears.

The standard search paths are:

Type of file	Search path
Frameworks	/LocalLibrary/Frameworks /NextLibrary/Frameworks
Header files	the project directory /LocalDeveloper/Headers /NextDeveloper/Headers
Libraries	/lib /usr/lib /usr/local/lib

For libraries in nonstandard locations, add the library location to Library Search Order and the header file location to Header Search Order. For frameworks, just add the framework location to Framework Search Order; Project Builder already knows to look inside of a framework for its header files.

Some OPENSTEP Libraries

Most of the OPENSTEP libraries are now delivered as frameworks. Because all of the files associated with a framework are in one location, it's pretty easy to find out which framework you need to link with if you import one of its headers.

There are still a few old-style libraries delivered with OPENSTEP. Here's a list of the more commonly used ones and when you would link against them:

- **/usr/lib/libcurses** Contains cursor control functions. Link with this library if you import the header file **curses.h**.
- **/usr/lib/libdbm** Contains database subroutines. Link with this library if you import the header file **dbm.h**.
- **/usr/lib/libDriver** Link with this library if your program interacts with a device driver.
- **/usr/lib/libg++** Contains the C++ libraries. Link with this library for projects that contain C++ code.
- **/usr/lib/libiostream** Contains C++ I/O streams support.
- **/usr/lib/libMallocDebug** Contains a special implementation of **malloc**. Link with this library if you want to use the **MallocDebug** application to examine your application's memory usage.

Setting compiler and linker options

- 1 Click the **Project Inspector** button and choose **Build Attributes** in the **Project Inspector** panel.
- 2 Type compiler options in the **Compiler Flags** field.
- 3 Type linker options in the **Linker Flags** field.

The `make` utility passes the same options to the compiler and linker every time you build a project. You can add to these options using the Build Attributes inspector. The compiler and linker options you specify here are also added to the compiler and linker options used when building the project's subprojects.



Click here or choose **Inspector** from the **Tools** menu.



Choose **Build Attributes**.

Type compiler and linker options in these fields. For library projects, the linker options are passed to `libtool`.

The pop-up list above the compiler flags controls the target platform for those flags. You can specify different values for the bottom five options depending on what platform you are building for—Mach, Windows, or a PDO platform. For more information, see “The Platform Pop-Up’s Purpose.”

Interesting Compiler and Linker Options

Here are some interesting compiler and linker options you may want to try. For more options, see the `cc(1)` and `ld(1)` man pages.

Compiler Options

- ansi** Use strict ANSI C definition.
- traditional** Use the traditional Kernigan & Ritchie C definition.
- bsd** Use strict BSD semantics.
- Wpointer-arith** Print a warning if pointer arithmetic is used on a void pointer or a function pointer.
- finline-functions** Make all simple functions inline.
- pipe** Use pipes in place of temporary intermediate files.

Linker Options

- sectorder** Order the blocks in a specified section.
- undefined** Specify how undefined symbols are treated: as errors, warnings, or ignored.
- whyload** Indicate why each member of a library is loaded.
- ysym** For a given symbol, list files that referenced it.
- Yn** For the first *n* undefined symbols, lists the file that referenced the symbol.

Dynamic Linking

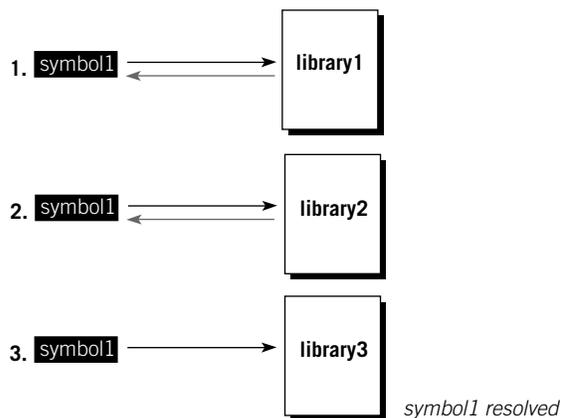
OPENSTEP 4.0 introduces dynamic linking. When you use dynamic linking, references are resolved at run time instead of at link time. This means you don't have to relink your application every time a definition in a dynamic library changes. You get the benefit of the changes without having to perform a build.

Dynamic linking is the default, and you must use it if you link with a dynamic shared library. All frameworks are dynamic shared libraries. (If you want to create your own dynamic shared library, see Chapter 12, "Creating Frameworks and Dynamic Shared Libraries.")

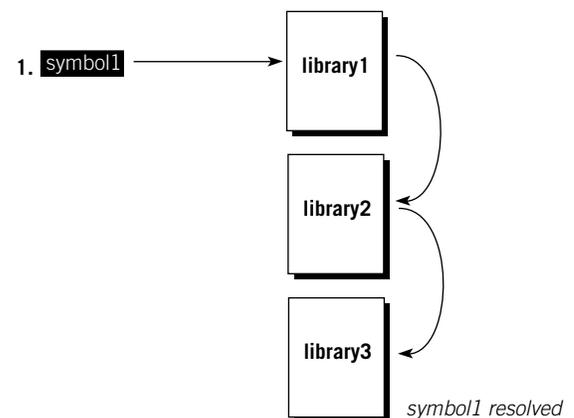
The main difference between static and dynamic linking is in how libraries are searched for unresolved references. When you use static linking, each library is searched for unresolved references exactly once.

When you use dynamic linking, the static linker must simulate the dynamic link editor to see if there are any unresolved references. It places each library in a search list. Then, whenever an unresolved reference is encountered, it searches each library in the search list in order until it can resolve the symbol. With dynamic linking, a library might be searched several times.

Static Linking



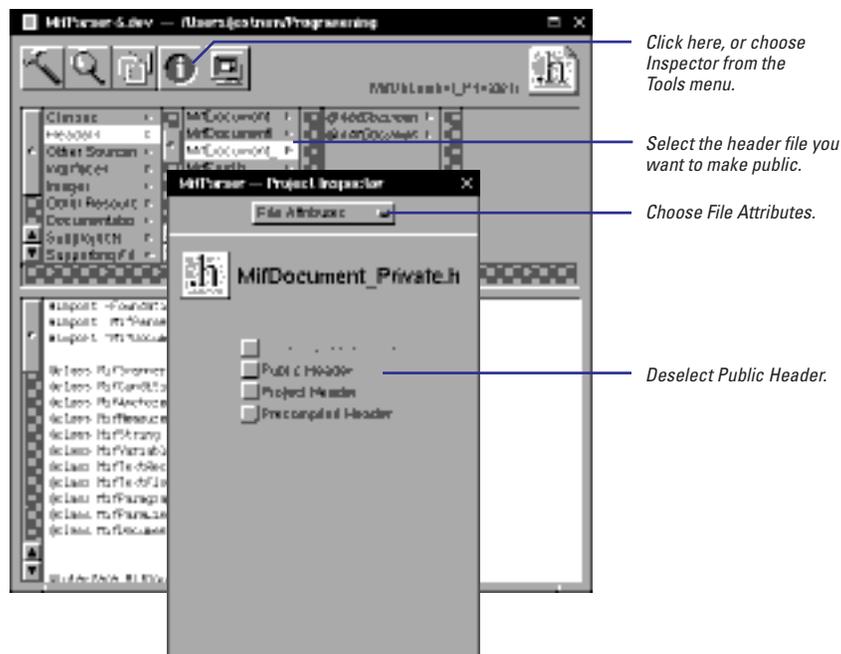
Dynamic Linking



Creating a precompiled header

- 1 Select the header file in the project browser.
- 2 Click the Inspector button.
- 3 Choose File Attributes in the Inspector panel.
- 4 Select Precompiled Header in the inspector.

A precompiled header is a header file that has been parsed and preprocessed, thereby improving compile time and reducing symbol table size. Only those macros and external declarations needed to compile the file are read from the precompiled header. Precompiled headers have a `.p` extension. You can define one for any type of project.



When you create a header file that you intend to precompile, follow these guidelines:

- Make sure that you import files in the proper order to avoid undefined symbol errors. If ClassA defines an instance variable of the ClassB, ClassB's header should be listed before ClassA's header in the precompiled header.
- Only import the system headers that are necessary for the interface.

A system header imports many other headers. The more headers you import into your precompiled header, the greater the risk of having a name conflict. For example, a system file might define a public struct that conflicts with a private struct declared in your project's headers. If a name conflict occurs, the compiler won't use the precompiled header.

If you select both Public Header and Precompiled Header in the File Attributes inspector, the `.h` file is installed, not the `.p` file. If you want the `.p` file installed (for example, if you're building a Framework or Library, see "Installing a precompiled header" in Chapter 12, "Creating Frameworks and Dynamic Shared Libraries/DLLs.")

To avoid name conflicts, import all of the project's header files into one precompiled header, and import all other header files separately. Make sure that system files that aren't necessary for the interface aren't imported indirectly either.

For example, suppose you have a project with a precompiled header named **Precomp.h** that imports **ClassA.h** along with all other header files in the project. **ClassA** defines an object that uses Foundation API in its interface declaration, and the implementation of **ClassA** uses functions from **libc**. **ClassA.h** should import the Foundation's precompiled header and **Precomp.h**. It should not import **libc.h** because the interface declaration doesn't need it and importing it increases the risk of a name conflict. Instead, **libc.h** should be imported in the implementation file, **ClassA.m**, because it is necessary for the implementation.

ClassA.h

```
#import <Foundation/Foundation.h>
#import "Precomp.h" /* use this for faster compiles */

@interface ClassA : NSObject
{
    NSString *aString;
}
- (NSString *)aString;
...
@end
```

ClassA.m

```
#import "ClassA.h"
#import <libc/libc.h>

@implementation ClassA
...
@end
```

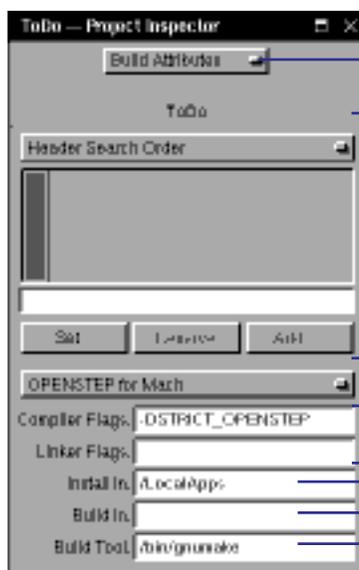
Customizing your makefiles

- ▶ Set information in the Build Attributes inspector.
- Or*
- ▶ Edit the files **Makefile.preamble** and **Makefile.postamble**.

Sometimes it's necessary to alter the standard build process as defined by the project makefile. If you need to do this, first look for the options you need to change in the Build Attributes inspector. Project Builder uses the information you set in the Build Attributes inspector to index the project. So to make sure the project index is correct, you should use this inspector instead of editing the makefile directly. (Project Builder updates the makefile for you.)



Click here or choose Inspector from the Tools menu.



Choose Build Attributes.

Search paths for files in nonstandard locations.

Options for the `cc`, `ld`, and `libtool` commands.

Where the executable lives after it is installed.

Where intermediate files (such as `.o` files) are placed.

Path for the `make` utility.

If the inspector does not have an option for what you need to set, edit the files **Makefile.preamble** and **Makefile.postamble**. Both files contain makefile variable definitions, and the comments in these files describe what each macro defines.

“Creating your own build targets,” “Setting search paths,” and “Setting compiler and linker options” in this chapter describe some of the fields in the Build Attributes inspector.

In general, **Makefile.preamble** contains macros that add to the standard makefile definitions, and **Makefile.postamble** contains macros that override the standard definitions. For example, the **LIBS** macro defines the libraries that your program should link with. The standard makefile sets this macro to the libraries in the project. If you want to change this definition, you uncomment the **LIBS** definition in **Makefile.postamble** and change the definition. However, if you want to link with more libraries than those added to the project, set **OTHER_LIBS** in **Makefile.preamble** to the additional library's name.

Reducing Compile Time

Each build begins by exporting any public headers to a location where they are visible to the rest of the project (for example, headers in subprojects are exported to the `derived_src` directory if you mark them as Project Headers in the File Attributes inspector). If you're building a project that does not export any headers (no boxes are turned on for header files in the File Attributes inspector), such as an application or tool with no distributed objects or library API, you can omit this step by setting this macro in **Makefile.preamble**:

Preamble Macro	Description
<code>SKIP_EXPORTING_HEADERS</code>	Skips the exporting headers step of the build.

Adding Make Dependencies

If you add dependencies or targets to the makefile, set these macros in **Makefile.preamble**.

Preamble Macro	Description
<code>OTHER_PRODUCT_DEPENDS</code>	Dependencies you defined that should be used in all builds.
<code>OTHER_INITIAL_TARGETS</code>	Targets you defined that should be built before subprojects.
<code>OTHER_INSTALL_DEPENDS</code>	Dependencies you defined that should be used for the install target.

Setting Up The Install Target

To set up the install target to work the way you want, set these macros in **Makefile.preamble**.

Preamble Macro	Description
<code>DSTROOT</code>	Path to prepend to the installation path specified in the Build Attributes inspector. The default is <code>/</code> .

If you're building a library or framework, see Chapter 12, "Creating Frameworks and Dynamic Shared Libraries," for more information on setting up the makefiles.

And these macros in **Makefile.postamble**.

Postamble Macro	Description
INSTALL_AS_USER	The owner of the installed files. The default is root.
INSTALL_AS_GROUP	The group for the installed files. The default is wheel.
INSTALL_PERMISSIONS	The installed files' permissions. The default is read and execute permissions turned on for all users.
APP_STRIP_OPTS	Stripping options to pass to the strip tool. The default is no options, which strips debugging symbols out of the executable. Only set options here if the application loads other bundles.
APP_WRAPPER_EXTENSION	The extension to use for application's output. The default is .app .

Setting Up Make Clean

To set up **make clean** to work the way you want, set these macros in **Makefile.preamble**.

Preamble Macro	Description
OTHER_GARBAGE	Files that should be deleted in addition to object files and executables.
CLEAN_ALL_SUBPROJECTS	If defined, make clean cleans subprojects as well. This macro is defined by default. To undefine it, comment it out in Makefile.preamble .

The Platform Pop-Up's Purpose

Do you want your project to run on multiple platforms? For example, are you writing an application that you plan to have run on both Mach and Windows? Or maybe you're writing a framework that you also want to build on one of the Portable Distributed Objects (PDO) platforms.

If this is your situation, the platform pop-up list is for you. (This is the list that appears just above the compiler options on the Build Attributes inspector.) Different platforms have different requirements. For example, you might install the application in a different location on a Windows platform than you would a Mach platform. You'll need to use different linker options because the platforms each use a native linker. Set the options as you want

them for one platform, then change the pop-up, and set them for the other.

The platform pop-up list is just a convenience that allows you to have one version of the project even though you're building for two platforms. It doesn't magically build an executable that will run on all the platforms you want. That is, if you're building an application on Windows, you'll get an application that runs on Windows, not on Mach. To get a version that runs on Mach, you'll need to transfer the project directory to a machine running OPENSTEP for Mach and build again.

Overriding Compiler and Linker Options

Most targets produce an optimized, debuggable executable and do not suppress compiler warnings. To change the compiler options used to produce the usual executable, override these macros in **Makefile.postamble**.

Postamble Macro	Description
OPTIMIZATION_CFLAG	Compiler optimization option, used by all but the debug target. The default is -O , which reduces code size and execution time.
LOCAL_DIR_INCLUDE_DIRECTIVE	Override if you don't want the current directory in the default search path. By default, this is defined as -I .
DEBUG_SYMBOLS_CFLAG	Compiler debug symbols options, used by all but the install target. The default is -g , which produces line number and symbol information.
WARNING_CFLAGS	Compiler warning message level, used by all targets. The default is -Wall , which suppresses none of the warning messages.
DEBUG_BUILD_CFLAGS	Compiler options used only by the debug target. The default is -DDEBUG , which defines the DEBUG preprocessor macro.
PROFILE_BUILD_CFLAGS	Compiler options used only by the profile target. The default is -pg , which produces information for gprof , and -DPROFILE , which defines the PROFILE preprocessor macro.

Setting Up Other Tools

Some tools are invoked by **make** if the project contains files with certain extensions. To set up these tools, set these macros in **Makefile.preamble**.

For more information about the tools listed here, see their man pages.

Preamble Macro	Description
PSWFLAGS	Options for the pswrap tool (invoked on .psw files).
YFLAGS	Options for the yacc tool (invoked on .y.c or .ym.m files).
LFLAGS	Options for the lex tool (invoked on .l.c or .lm.m files).
MSGFILES	Input files for the msgwrap tool. These should have the .msg extension.
DEFSFILES	Input files with a .defs extension for the mig tool.
MIGFILES	Input files with a .mig extension for the mig tool.
RPCFILES	Input files for the rpcgen tool (invoked on .x files).

Including More Files in the Build

There may be files that you don't want to add to the project but that should be included in the build. Use these macros in **Makefile.preamble** to have the build handle more files.

Preamble Macro	Description
OTHER_LIBS	Libraries to link with besides the libraries included in the project.
OTHER_OFILES	Object files to link into the executable besides those produced by the source files in the project.
OTHER_SOURCEFILES	Source files besides those included in the project.
INCLUDED_ARCHS	Architectures to which this project or subproject should be restricted to building for. Building for other architectures is skipped. Must be a subset of the architectures selected in the Build Options panel.
EXCLUDED_ARCHS	Similar to INCLUDED_ARCHS , but lists architectures that this project or subproject shouldn't build for instead of architectures it should build for. Don't use if using INCLUDED_ARCHS .

10

Basic Debugging

Starting a debugging session

Running the program in the debugger

Interrupting the program

Executing a single line of code

Stepping into a method or function

Setting breakpoints

Managing breakpoints

Executing several lines of code

Navigating using the stack

Examining the value of a variable or an object

Debugging object allocation and deallocation

Debugging a multithreaded program

Changing program execution while debugging

Changing code while debugging

Debugging multiple projects

Debugging frameworks

A fly, Sir, may sting a stately horse and make him wince; but one is but an insect, and the other is a horse still.

Samuel Johnson

Errors, like straws, upon the surface flow;
He who would search for pearls must dive below.

John Dryden

Starting a debugging session

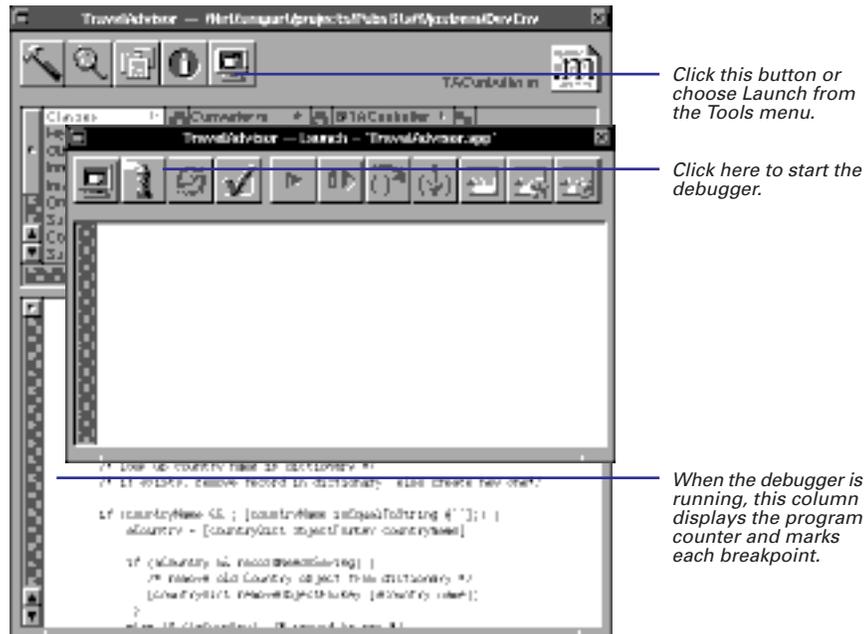
- 1 Click the launch button to bring up the Launch panel.
- 2 If necessary, click the checkmark button in the Launch panel to set the executable name and environment.

- 3 Click the debug button to start the debugger.

Or

- 3 Click the launch button to run your program without debugging.

You can use Project Builder's Launch panel to run your program in the debugger or, if you want, outside of the debugger. The Launch panel provides an interface to the Free Software Foundation's **gdb** debugger.



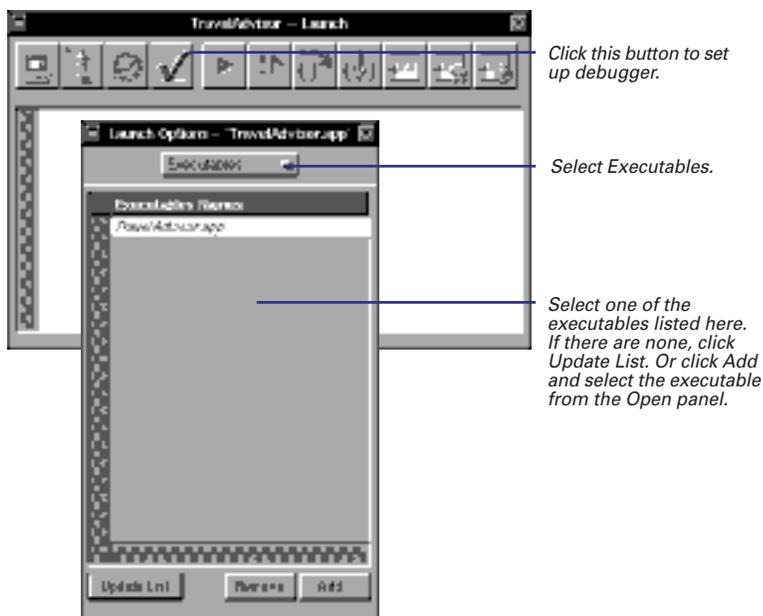
gdb is a command line tool, and it has a rich command-line interface. Project Builder provides a user interface for the most common debugging tasks. You perform other debugging tasks using the **gdb** command line interface.

After **gdb** starts up, you can click the run button to start up your program.

Project Builder automatically includes debugging symbols in your program if you use the default build target. See Chapter 9, "Building" for more information on building a program.

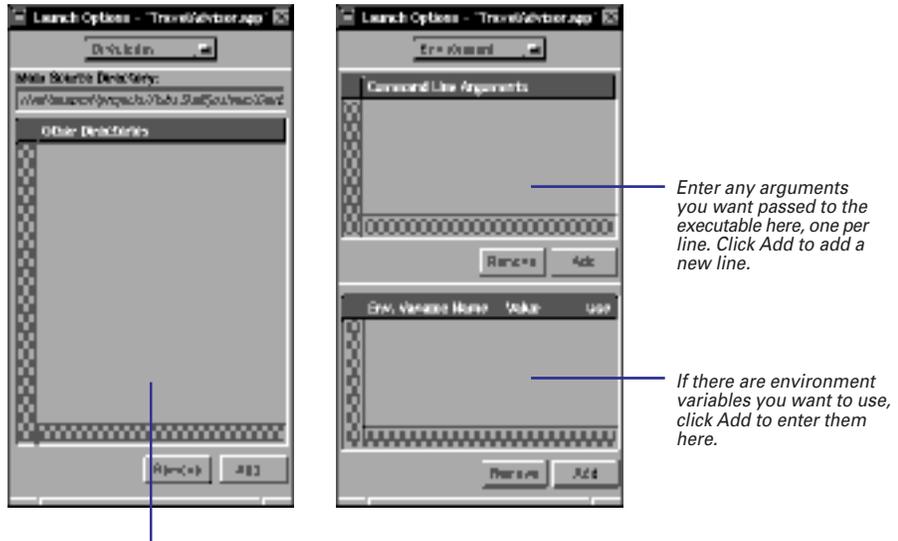
For a complete list of **gdb** commands, see the *OPENSTEP Development Tools Reference*.

If the Launch panel isn't displaying the name of the project executable in its titlebar, you need to set it in the Launch Options panel before you can run or debug the program. Needing to set the executable usually occurs when your project contains subprojects or builds a binary that doesn't run on its own (for example, it's a framework or a loadable bundle).



Setting Up the Program's Environment

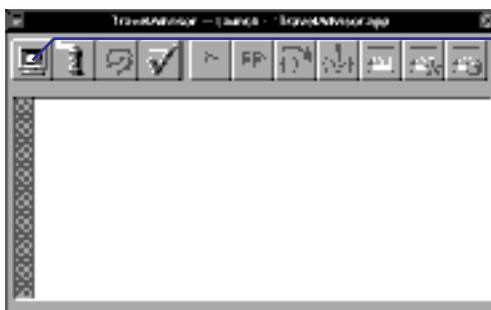
Besides allowing you to choose the executable, the Launch Options panel also lets you set environment variables, pass arguments to your program, and specify other directories that contain source code for this project.



You can change environment variables and command line arguments while your program is running, but they won't take effect until you restart your program.

Running the Program Outside of the Debugger

If you want to launch the program outside of the debugger, click the Launch button instead of the Debug button. The program is started up and operates independently, just as if you'd started it outside of Project Builder.



Click here to run the program.

Attaching to an Already Running Process

Sometimes, a problem occurs only when you launch an application outside the debugger. When you launch it inside **gdb**, the problem disappears. If this happens, launch the application using the launch button and use the **gdb** command **attach** to hook up to it:

```
(gdb) attach pid
```

pid is the process ID of the process you want to debug.

attach immediately stops the application. When you use **attach**, you can debug the process just like you normally would: by setting breakpoints, modifying storage, and so on.

When you're finished debugging, use **detach** (which takes no arguments) to detach the debugger from the process. The

process resumes executing. You'll kill the process if you try to quit **gdb** or if you try to start the program from the beginning. (**gdb** asks for confirmation before it allows you to do these things.)

If you're having trouble attaching to a process before the errant code is executed, send your program a stop signal as one of the first messages:

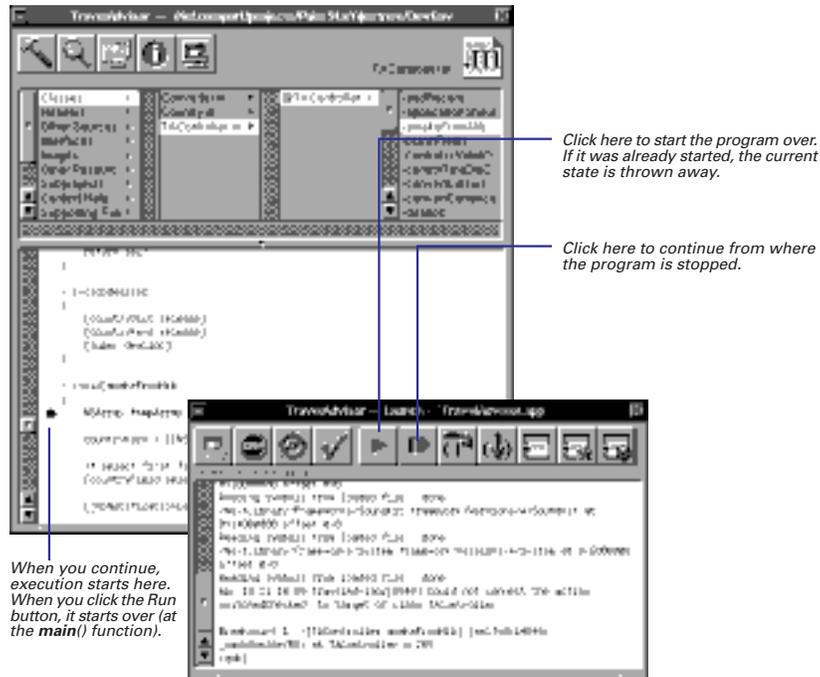
```
[NSThread sleepUntilDate:
 [NSDate distantFuture]];
```

This indefinitely suspends execution of the application. Once you attach in **gdb**, click the continue button to go on from there.

Running the program in the debugger

- ▶ Click the continue button to continue from where you left off.
Or
- ▶ Click the run button to start the program over.

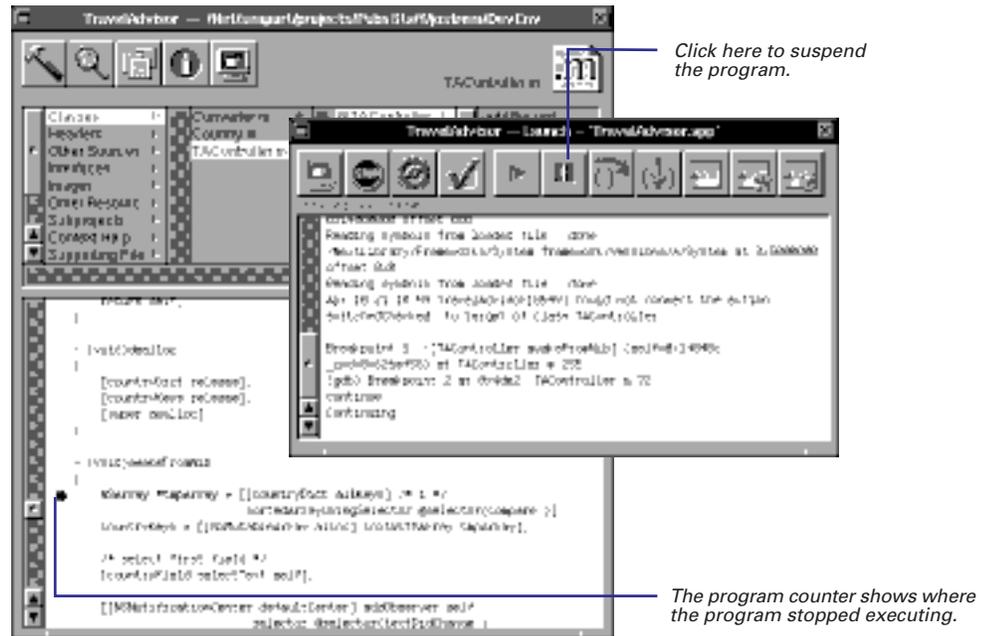
The run button starts you program from the beginning. Typically, you only use the run button to start the program for the first time. If you've interrupted the program or hit a breakpoint and you want to continue executing from that point, use the continue button.



Interrupting the program

- Click the suspend button.

It's often necessary to suspend the program that you're debugging so that you can examine its state or enter **gdb** commands. For example, to set a breakpoint or to see the value of a certain variable, you must interrupt the program first.

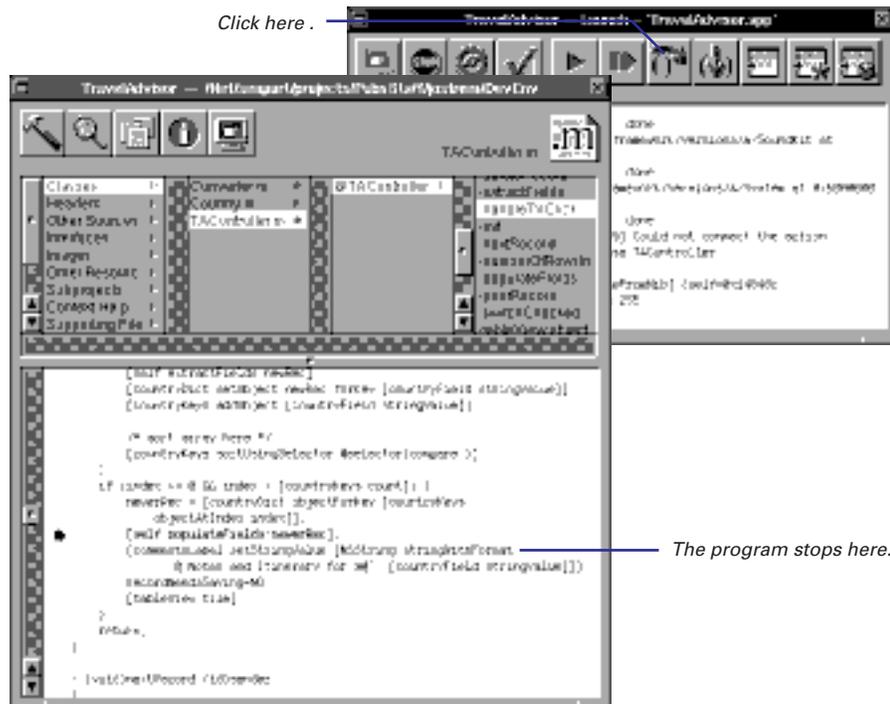


If you're debugging an application, it will display the spinning cursor when you suspend it. The application won't accept any input until you continue execution.

Executing a single line of code

- Click the next button.

If you know that a particular method has an error in it somewhere, you can execute that method a line at a time (called *stepping*) to see exactly where the error occurs.

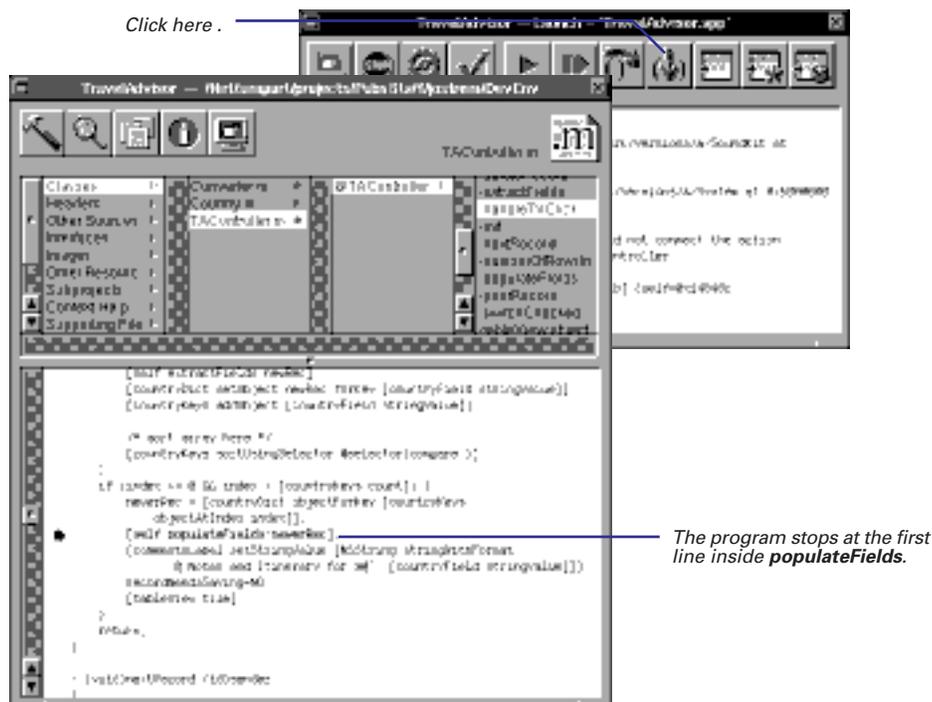


The next button always executes until control returns to the line of code directly below the current line. That is, if the current line contains a function call or invokes an Objective-C method, the entire function or method is executed, and the program doesn't stop until it returns (unless a breakpoint is hit).

Stepping into a method or function

- Click the step button.

If the program counter is pointing to a line containing a method invocation and you click the next button, that method executes before the program stops. If you want the program to stop inside of that method, click the step button instead.



The step button executes the program until control reaches a different line in the program. If the current line contains a method invocation, the program stops at the beginning of that method. If the method isn't part of your program (if it's in one of the OpenStep frameworks, for example), the entire method is executed. Execution doesn't stop until the next line of code you own.

Setting breakpoints

- 1 If the program is running, click the suspend button to stop it.
- 2 In the Project Builder main window, double-click in the gray area next to the line where you want the breakpoint.
- 3 Click the continue button to execute up to the breakpoint.

A breakpoint makes your program stop whenever a certain point in the program is reached. Every time the program encounters the line of code that has the breakpoint, it stops executing.



Double-click next to the line where you want the breakpoint.

If you're debugging an application, the cursor spins when it hits a breakpoint. When you enter this state, go to the Launch panel and examine the program's state (print values of variables, examine the stack, and so on).

Setting breakpoints on data

Sometimes you want to stop the program whenever the value of a variable changes, no matter which part of your code is doing the changing. To do this, use a watchpoint. To set a watchpoint:

```
(gdb) watch expr
```

where *expr* is any expression or variable.

gdb treats watchpoints and breakpoints the same. Anything you can do to a breakpoint, you can also do to a watchpoint (see “Cool Breakpoint Stuff” in this chapter). The

Breakpoints display of the Task Inspector provides information on both breakpoints and watchpoints.

When a watchpoint is set, your program runs much more slowly than if you had set a normal breakpoint, so use watchpoints sparingly. (One alternative is to set a conditional breakpoint, described in “Cool Breakpoint Stuff.”) However, watchpoints are sometimes the only way to catch an error when you don't know where the error occurs.

Cool Breakpoint Stuff

Using **gdb** commands, you can add more power to your breakpoints and make debugging a breeze. For complete information on **gdb** breakpoints, see the *OPENSTEP Development Tools Reference* manual. Here are some highlights.

Setting Breakpoints in Dynamically Loaded Code

gdb doesn't know about symbols in dynamically loaded code (such as code inside frameworks or loadable bundles) because it's not loaded until run time. This means you can't set a breakpoint in a framework until after you start the program that uses it. This is pretty frustrating when the framework is what you want to debug. However, you can set a future breakpoint when the framework isn't loaded yet. To do this, use the **future-break** command:

```
(gdb) future-break address
```

(*address* can be a method name, a function name, a file name and line number, and so on.) When you enter this command, **gdb** checks the loaded symbols for a symbol matching *address*. If one is found, it resolves the breakpoint. If not, it holds on to it. Then, whenever a dynamic shared library is loaded, **gdb** checks the breakpoint against the newly loaded symbols until it can resolve the symbol in the breakpoint. (If the symbol can never be resolved, the **future-break** just sits around doing nothing.)

When you quit the program, **gdb** unloads all of the breakpoints set in dynamic shared libraries. These breakpoints are converted into future breakpoints—when the library is loaded again, the breakpoints are resolved again.

Future breakpoints are just like normal breakpoints in every other respect; you can add commands to them, disable them, enable them, and so on. In the Breakpoints display of the Task Inspector, they are listed as “unloaded.”

Conditional Breakpoints

If you only want a breakpoint to stop when a certain condition is true, use the **condition** command:

```
(gdb) condition bnum expression
```

expression is any Boolean expression and it's associated with breakpoint number *bnum*. (The Breakpoints view of the Task Inspector tells you the breakpoint number.) From now on, this breakpoint will stop the program only if the value of *expression* is true. To remove a condition from a breakpoint, enter **condition** with no *expression*.

Ignoring Breakpoints

You can disable a breakpoint for a specific length of time with **gdb** command **ignore**:

```
(gdb) ignore bnum count
```

This command ignores the breakpoint the next *count* times it is reached. (0 means the program stops the next time it's reached.) If the breakpoint is a conditional breakpoint, the condition isn't checked unless the ignore count is 0.

Executing Commands at a Breakpoint

You can give any breakpoint a series of commands to execute when the program stops at it. For example, if you want to know what the value of the variable *x* is whenever breakpoint 5 is hit, enter the following. (You must type **end** when you're through to make the **gdb** prompt return.)

```
(gdb) commands 5
> print x
> end
```

This brings up a handy trick for ignoring breakpoints. Often, you don't know how many times you want to ignore a breakpoint (making the **ignore** command useless), but you know that you want to ignore it until a specific point in a program is reached. For example, say you want to stop at a method named **setCurrent**: but only if the message is sent by the **processParagraph** method. In this case, you can do the following:

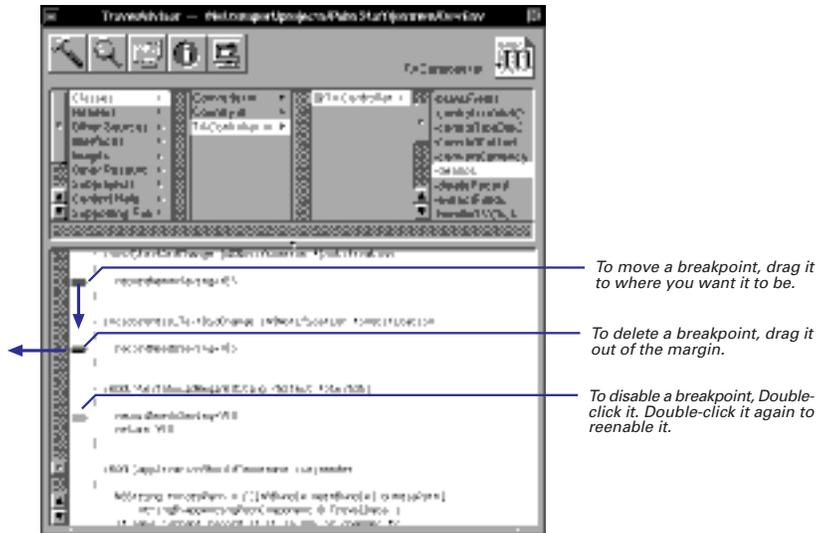
```
(gdb) break setCurrent:
Breakpoint #1 set
(gdb) break processParagraph
Breakpoint #2 set
(gdb) disable 1
(gdb) commands 2
> silent
> enable 1
> continue
> end
(gdb) continue
```

This example sets two breakpoints, one at the beginning of each method. Then, it disables the breakpoint at **setCurrent**. When the breakpoint at **processParagraph** is reached, it enables the breakpoint at **setCurrent**: and continues executing. (**silent** is just a convenience. It means that **gdb** won't print the usual stopped at breakpoint message.)

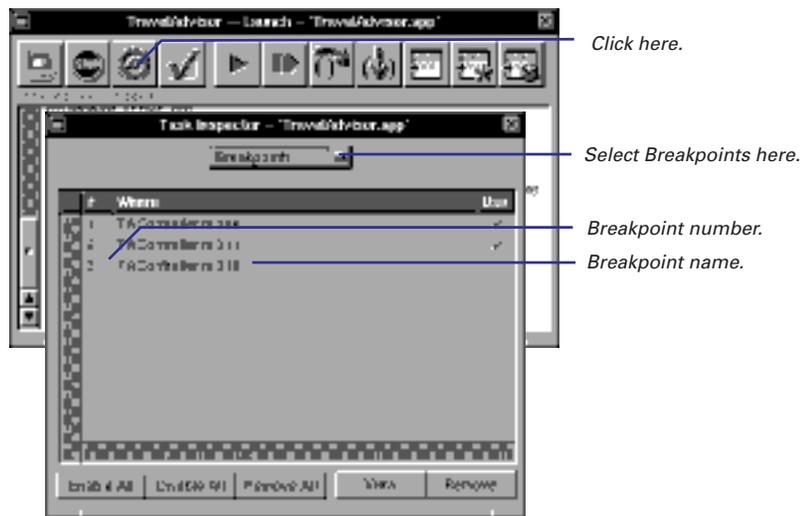
Managing breakpoints

- ▶ To move a breakpoint, drag it to where you want it to be.
- ▶ To delete a breakpoint, drag it off the Project Builder window.
- ▶ To disable a breakpoint, double-click it.
- ▶ To find out information about breakpoints, bring up the Breakpoints display of the Task Inspector.

When you no longer need to stop at a breakpoint anymore, you can delete the breakpoint; however, you might want to just disable it. When you delete a breakpoint, it is gone forever. When you disable a breakpoint, it still exists and is still displayed in the Project Builder main window, but the program does not stop at the breakpoint. You can enable the disabled breakpoint later.



Sometimes it's useful to know how many breakpoints you've set and where they are. The Breakpoints view of the Task Inspector provides this information. The first column of this display gives you the breakpoint number, which is used in many **gdb** commands.



Using the Breakpoints display, you can also enable and disable each breakpoint by clicking the Use column, or you can enable, disable, and remove all breakpoints. Use the View button to have Project Builder go to the line where the breakpoint is set.

See “Cool Breakpoint Stuff” in this chapter for some useful **gdb** commands involving breakpoints.

Executing several lines of code

- ▶ Drag the program counter to the line where you want execution to stop.

When you're stepping through code, you often hit a place where you'd like to execute several lines of code and stop again. For example, if you encounter a `for` loop that is executed several dozen times, you probably want to jump through the `for` loop and resume stepping after the loop ends. To do this, just drag the program counter to the first line of code past the `for` loop. The entire loop executes, and the program stops when it reaches the line of code you've dragged the counter to. You can then click the next or step button to resume stepping.



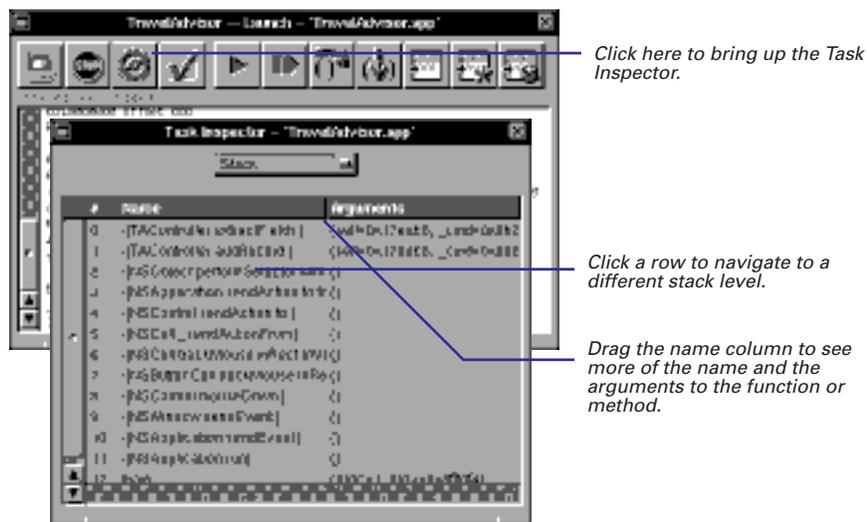
Dragging the program counter is more convenient than setting a breakpoint at the end of the loop. You don't have to delete the breakpoint when you're done.

Navigating using the stack

- 1 Click the inspector button on the Launch panel to bring up the Task Inspector.
- 2 Choose Stack from the pop-up list.
- 3 Click a stack frame to have the debugger jump to that stack frame.

Each time your program invokes a C function, a C++ member function, or an Objective-C method, the information about where in the program the call was made is saved in a block of data called a stack frame. The frame also contains the arguments of the call and the local variables of the function that was called.

The Task Inspector displays the stack on the right side of the window. Each row in the stack display represents one stack frame. The current stack frame is numbered 0, the frame that called it is 1, and so on.



At any given time, one of the stack frames is selected by **gdb**; many **gdb** commands refer implicitly to this selected frame. In particular, whenever you ask **gdb** for the value of a variable in the program, the value is found in the selected frame. You can select any frame by clicking it. You can then examine the values of variables pertaining to that stack frame. As you navigate to a different stack frame, the Project Builder main window shows you the currently executing line of code at that frame.

Tip: You can return from the current level by Shift-clicking the program counter.

For the Experts: More on Examining Variables

Making Sure Variables Stick Around

When you build the program using the default build target (for example, app for Application projects), an optimized, debuggable executable results. This executable is helpful if a bug surfaces only in the optimized version; however, debugging optimized code sometimes gives surprising results. Control flow may change and variables may disappear without a trace. You ask **gdb** to print such a variable and even though the source clearly shows it is in scope, **gdb** replies:

```
(gdb) print num
No symbol "num" in current context
```

To ensure that a variable be available in the debugger even after optimization, declare the variable **volatile**.

Value History

gdb maintains a value history for your session. This means that every expression you evaluate using the **print** command (or the Print, Print *, and PO buttons) is assigned a value number in the history, like this:

```
(gdb) print self
$7 = (struct NSApplication *) 0xbb5e4
```

You can refer to this value as **\$7** and use it in future expressions:

```
(gdb) print (char *) [$7 appName]
$8 = 0xb80cc "FunWithGDB"
```

Once a value is entered into the history, it doesn't change. The value is stored as **\$7**, not the expression that generated it. This means that **\$7** doesn't change to hold the new value of **self** when your program enters a different scope.

Also, at any time, **\$** refers to the last value in the history and **\$\$** to the next-to-last value.

The **output** command has the same semantics as the **print** command, but doesn't add the result to the value history. You can use this difference to avoid cluttering the value history with unimportant results. For more sophisticated printing needs, **gdb** provides a **printf** command similar to the C version that provides for formatted output. Like **output**, the results from **printf** are not entered into the value history.

Any name that begins with a **\$** can be used as the name of a **gdb** convenience variable. These variables are implicitly typed and created at first reference. Use **print** to get the value of a convenience variable and the **set** command to set or change the value. You can set the value to any valid C or Objective-C expression, including methods or functions:

```
(gdb) p $array = [NSArray array]
$24 = 793052
(gdb) p $num = 1230 % 4
$25 = 2
```

All registers have convenience variables associated with them. The **info registers** command dumps the contents of all registers so you can see the names associated with each register. The register convenience variables most often used are **\$fp**, which holds the frame pointer, **\$sp** for the stack pointer, and **\$pc** for the program counter.

Locating Your Variables

To find out how a variable is stored, use this command:

```
(gdb) info address self
Symbol "self" is a variable in register a2.
```

info address tells you if the variable is stored on the stack or in a register. This command is useful to determine if optimizations are causing problems, particularly on RISC machines.

Examining Raw Memory

Use the command **x** (for "examine") to examine memory without referencing the program's data types.

x is followed by a slash and an output format specification, followed by an expression for an address:

```
x/fmt addr
```

These *fmt* letters specify the size of unit to examine:

- b Examine individual bytes.
- h Examine halfwords (two bytes each).
- w Examine words (four bytes each).
- g Examine giant words (eight bytes).

These *fmt* letters specify how to print the contents:

- x Print as integers in unsigned hexadecimal.
- d Print as integers in signed decimal.
- u Print as integers in unsigned decimal.
- o Print as integers in unsigned octal.
- a Print as an address, both absolute and relative
- c Print as character constants (this implies size b).
- f Print as floating-point. This works only with sizes w and g.
- s Print a null-terminated string of characters.
- i Print a machine instruction in assembler syntax (or nearly).

Once you've entered **x** to see the value at an address, hit return to see the value at the next address.

Debugging object allocation and deallocation

- ▶ Use `enableFreedObjectCheck`: inside `gdb`.
Or
- ▶ Use the `oh` tool to see where and when objects are allocated and deallocated.
Or
- ▶ Use the `AnalyzeAllocation` tool to see where and when objects are allocated and deallocated.

Object allocation and deallocation are often trouble spots. Two common problems are using an object after it has been deallocated and releasing an object too many times. Here are some strategies and tools to debug object allocation and deallocation.

A typical autorelease error:

```
objc: FREED(id): message objectForKey: sent to freed object=0xfde44
```

Ignoring Autorelease Errors

You may want to debug the rest of your program first, saving the release problems until later. The **enableRelease**: convenience method defined in Foundation's `NSAutoreleasePool` class helps you ignore autorelease errors. `NSAutoreleasePool` defines the application's autorelease pool. When an object is autoreleased, it is added to the autorelease pool. At the top of the event loop, all objects in the pool are sent a **release** message, which decrements the reference count and potentially deallocates the object. `NSAutoreleasePool` allows you to control that pool.

If you receive messages from the debugger indicating that

you are sending messages to deallocated objects, enter this command:

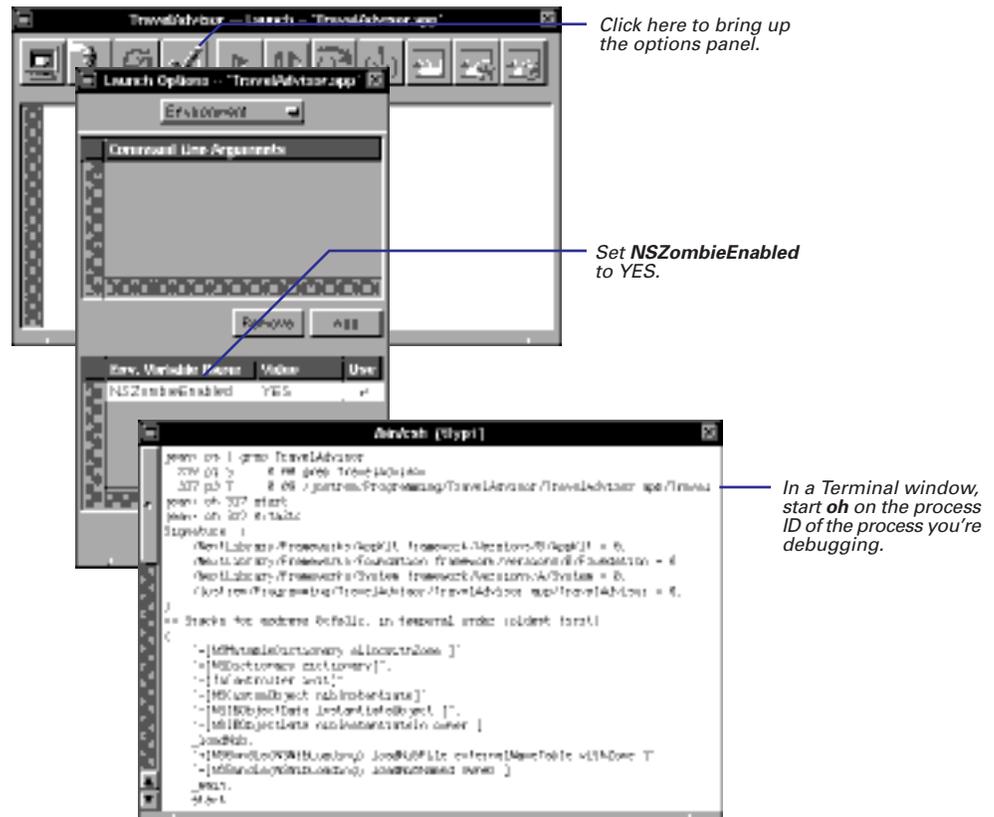
```
(gdb) call [NSAutoreleasePool
enableRelease:NO]
```

This message disables the deallocation of all objects in your program, ignoring autorelease errors.

Your program must be started when you send this message. It's often useful to break on `main()` and send this message after the first line or two of the program.

Using the oh Command

Another way to debug the **autorelease** and **release** errors is to use the **oh** command in conjunction with **gdb**. When you start the **oh** command, it starts recording allocation and deallocation events related to the process you specify. You set **NSZombieEnabled** so that the memory for deallocated objects is not reclaimed. (Released objects are just turned into “zombies.”) The advantage to setting this variable is that you can ensure that an object’s address is unique.



When you receive an autorelease error perform the command:

```
% oh pid address
```

where *address* is address of the object that is being release twice. **oh** will produce a report showing you the stack frame each time that object is allocated, copied, retained, or released, like the one shown on the next page.

```

== Stacks for address 0xfa31c, in temporal order (oldest first):
(
    "+[NSMutableDictionary allocWithZone:]",
    "+[NSDictionary dictionary]",
    "-[TAController init]",
    "-[NSCustomObject nibInstantiate]",
    "-[NSIBObjectData instantiateObject:]",
    "-[NSIBObjectData nibInstantiateIn:owner:]",
    _loadNib,
    "+[NSBundle(NSNibLoading) loadNibFile:...]",
    "+[NSBundle(NSNibLoading) loadNibNamed:owner:]",
    _main,
    start
)
(
    "+[NSDictionary dictionary]",
    "-[TAController init]",
    "-[NSCustomObject nibInstantiate]",
    "-[NSIBObjectData instantiateObject:]",
    "-[NSIBObjectData nibInstantiateIn:owner:]",
    _loadNib,
    "+[NSBundle(NSNibLoading) loadNibFile:...]"
    "+[NSBundle(NSNibLoading) loadNibNamed:owner:]",
    _main,
    start
)
(
    __NSAPDataReleaseToOffset,
    "-[NSAutoreleasePool release]",
    "+[NSBundle(NSNibLoading) loadNibFile:...]",
    "+[NSBundle(NSNibLoading) loadNibNamed:owner:]",
    _main,
    start
)
(
    "-[NSConcreteMutableDictionary release]",
    __NSAPDataReleaseToOffset,
    "-[NSAutoreleasePool release]",
    "+[NSBundle(NSNibLoading) loadNibFile:...]",
    "+[NSBundle(NSNibLoading) loadNibNamed:owner:]",
    _main,
    start
)
)

```

Keeping Memory Allocation Statistics

Another command, **AnalyzeAllocation**, lets you look at memory allocation after your program has finished executing. To use **AnalyzeAllocation**:

1. Set this environment variable:

```
% setenv NSKeepAllocationStatistics YES
```

The **NSKeepAllocationStatistics** variable tells your program to record information about memory allocation in a file named `/tmp/alloc_stats_name_pid`.

2. Run a specific task in your application. The allocation statistics file becomes very large very quickly, so it is important not to run too much of your program at once with **NSKeepAllocationStatistics** turned on.

3. Turn off the environment variable:

```
% unsetenv NSKeepAllocationStatistics
```

4. Perform this command in a Terminal window:

```
% AnalyzeAllocation -v /tmp/alloc_stats_name_pid
```

Common Autorelease Mistakes

Once you find the object with the autorelease error, look for the following:

- For every **autorelease** and **release** message in your application, make sure there is a corresponding **alloc**, **copy**, **mutableCopy**, or **retain** message sent to the same object. **autorelease** and **release** decrement an object's reference count. **alloc**, **copy**, **mutableCopy**, and **retain** increment the reference count. The number of increments and decrements for an object must be equal. Another way of thinking about this is: If you don't allocate, copy, or retain an object, you're not responsible for releasing it.
- When an NSArray, NSDictionary, or NSSet (known as the collection objects) is deallocated, the objects stored in the collection are released as well. If you need to access an object you stored in a collection after the collection is released, you must retain that object before you release the collection.
- Superviews retain subviews as you add them to the hierarchy and release subviews as you remove them from the hierarchy. If you swap views in and out of the hierarchy, you should retain the views that are not in the hierarchy.
- When you change a window's content view, the window releases the old content view and retains the new content view.
- Objects do not retain their delegates (to avoid retain cycles).
- **decodeValuesOfObjCTypes**: returns a retained object. **decodeObject** returns an autoreleased object. If you unarchive an instance variable with **decodeObject**, send it the **retain** or **copy** message.

Debugging a multithreaded program

- ▶ Use the `gdb` command `thread-list` to obtain information about all of the threads running in the program.
- ▶ Use the `thread-select` command to switch to a different thread.

A single program may have more than one thread of execution. A *thread* is an executable unit that has its own stack and is capable of independent I/O, but shares the address space of the other threads in a *task*.

`gdb` allows you to observe all threads while your program runs, but whenever `gdb` takes control, one thread in particular is always the focus of debugging. This thread is called the current thread. Debugging commands show program information from the perspective of the current thread. If you want to change to a different thread, use the `thread-select` command (passing it the thread number, which is displayed in the first column of the `thread-list` output).

```

gdb) help thread-select
Select and start a thread
gdb) help thread-list
List all of threads
gdb) thread-list
There are 2 threads
Thread State Suspend Port PC Name Function
0 S 0 0x0000 0x00401000 main
1 S 0 0x0000 0x00401000 gdb
thread
The current thread is thread 0
gdb) thread-select 1

```

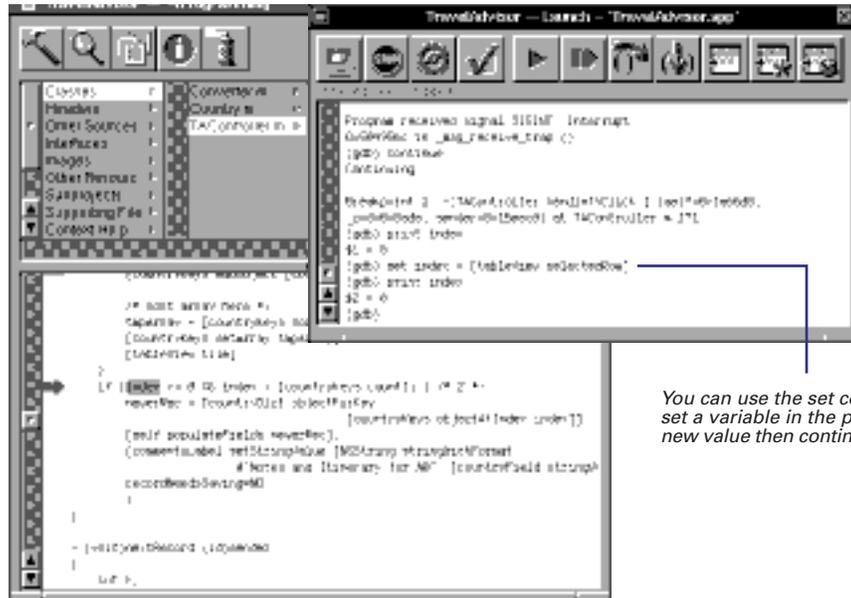
The `thread-list` command shows information about all of the threads in the program.

Use the `thread-select` command to jump to a different thread.

Changing program execution while debugging

- Use `gdb` commands to simulate a solution to a bug before building.

Once you find out what's wrong with your program, you might want to test that the solution you've come up with will work before you change the source code and rebuild. For example, what if you set a variable to a different value? Will that solve the problem?



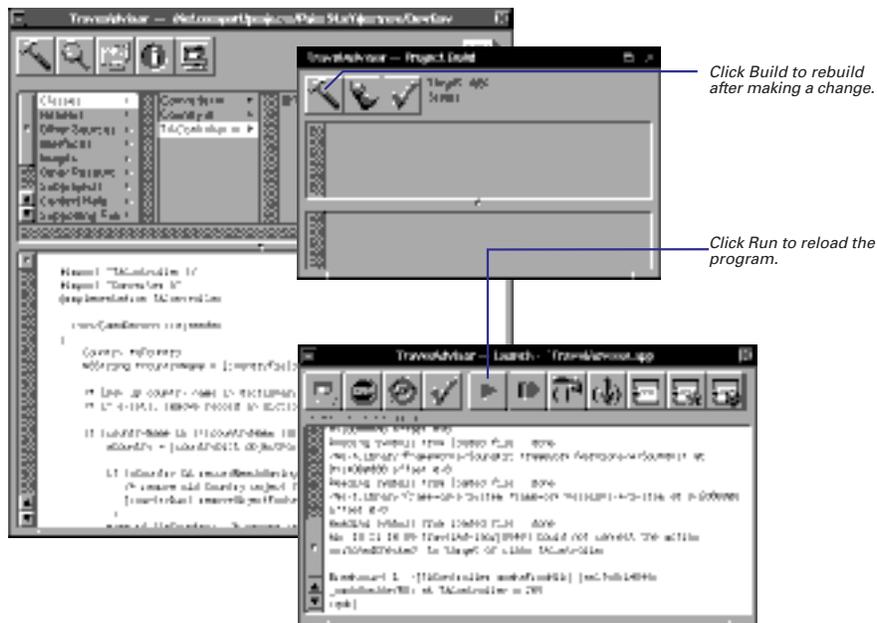
You can use the `set` command to set a variable in the program to a new value then continue executing.

Command	Description
<code>call function</code>	Executes the <i>function</i> . You can also use this for Objective-C messages.
<code>jump linenum</code>	Resume execution at line number <i>linenum</i> . Execution may stop immediately if there's a breakpoint there. The jump command doesn't change the current stack frame, or the stack pointer, or the contents of any memory location or any register other than the program counter. If <i>linenum</i> is in a different function from the one currently executing, the results may be wild if the two functions expect different patterns of arguments or of local variables. For this reason, the jump command requests confirmation if the specified line isn't in the function currently executing.
<code>jump *address</code>	Resume execution at the instruction at address <i>address</i> .
<code>set var = exp</code>	Perform an assignment.

Changing code while debugging

- 1 Use the `gdb` command `kill` to quit your program in the debugger.
- 2 Make changes to file in Project Builder.
- 3 Click the build button to bring up the Project Build panel.
- 4 Click the build button to build the program.
- 5 Go back to the Launch panel.
- 6 Click the debugger's run button.

After you've found a bug, you need to fix your code and rebuild the program. You don't need to quit the debugger. Just edit the file in Project Builder like you normally would, save it, and rebuild. When the build finished, stop and restart the program in `gdb`. When you click the run button, `gdb` checks for a more recent version of the executable and loads it if necessary. By not quitting `gdb`, you can preserve all of your breakpoints.

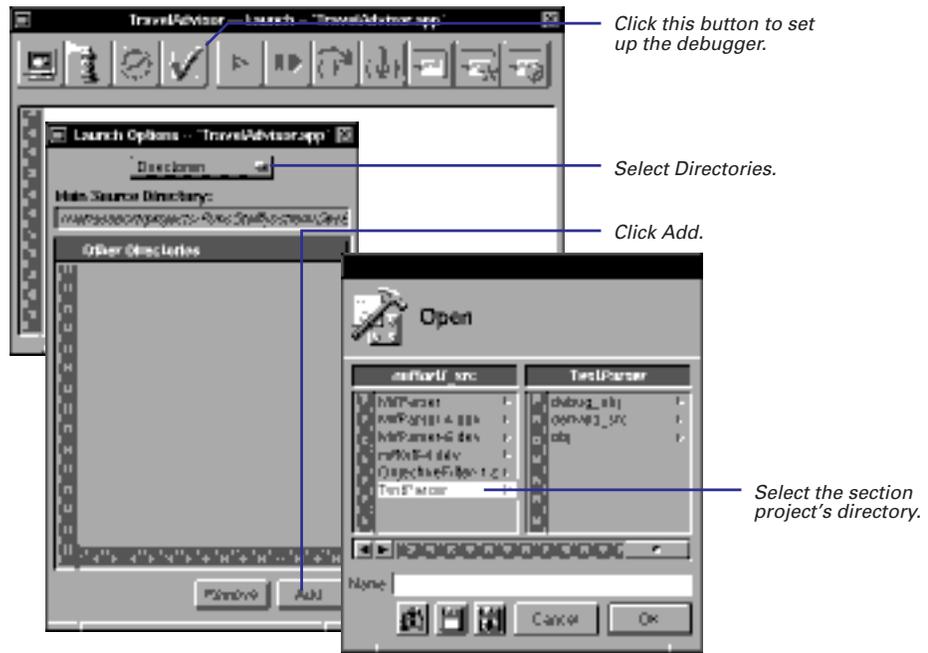


Debugging multiple projects

- 1 **Open the first project.**
- 2 **Click the checkmark button in the Launch panel.**
- 3 **In the Directories display of Launch Options panel, click the Add button.**
- 4 **Select the second project's directory in the Open panel.**
- 5 **Start the debugger.**

A debugger session applies to only one project file at a time. If you have more than one project open, the Launch panel displays the debugger session for the currently selected project. If you set a breakpoint in one project, it doesn't affect the other project's debugger session.

If you want to debug two projects that relate to each other (for example, an application and a framework), you need to make the debugging symbols of one project visible to the other project. You do this in the Launch Options panel.

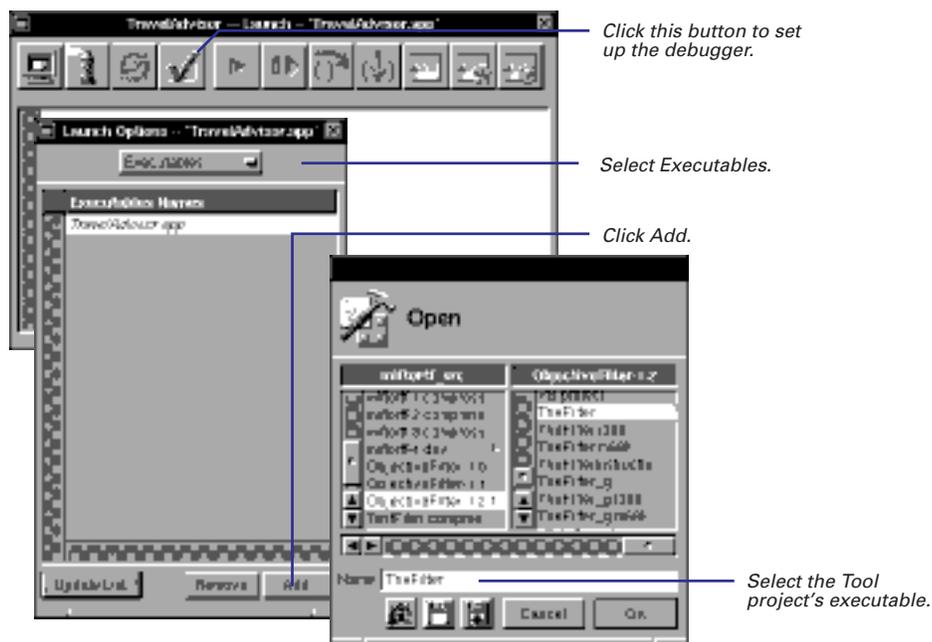


When you're debugging multiple projects and the debugger stops in code that's part of the unopened project, it displays the appropriate source code file under Non Project Files. If you want to go to other files in the second project and set breakpoints there, open them in the current project window.

Debugging frameworks

- 1 Create a tool that tests your framework.
- 2 In the framework project, bring up the Launch Options panel.
- 3 Select Executables.
- 4 Click the Add button.
- 5 Select the tool's executable in the Open panel that appears.
- 6 Start the debugger.

To debug a framework or library project, you usually create a tool project that uses all of the framework's features. However, you don't really want to debug the tool's code; you want to debug the framework's code. To debug the framework, you can select a tool's executable as the framework's debugger target. When you click the debugger's run button, the tool's executable is what is run. However, you can set breakpoints in the framework's code and step through it, just as if you were debugging an application.



11

Dynamic Loading

Loading nib files dynamically:
an info panel

Displaying an attention panel

Creating a window with
multiple displays

Creating dynamically loadable
bundles

...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.

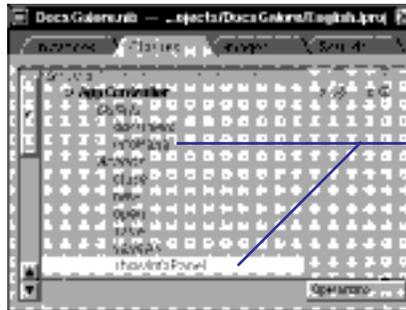
The last two commands (New Palette and New Inspector) are used when creating static palettes. If you're not working on a palette project, you use the New Empty command to create a nib file, unless you're specifically creating an info panel or an attention panel.

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.

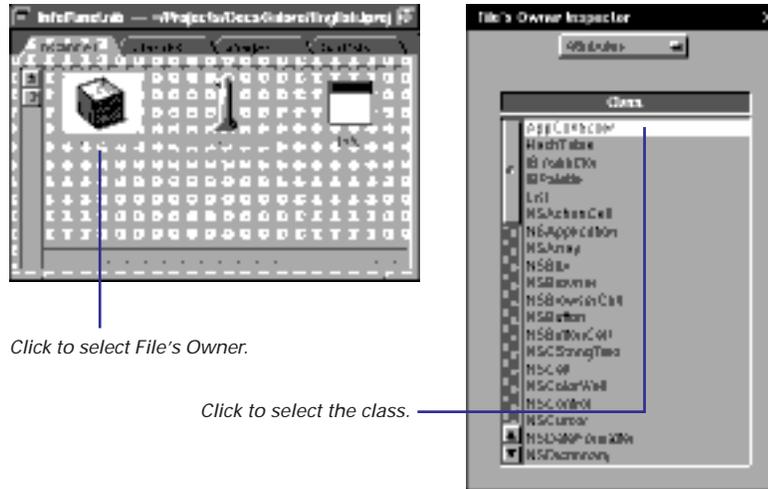
You cannot connect the Info Panel to the controller object in the main nib file because the panel is in the new auxiliary nib file. You must assign the controller class to File's Owner in the auxiliary nib file and then make the outlet connection between File's Owner and the panel. The first step in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

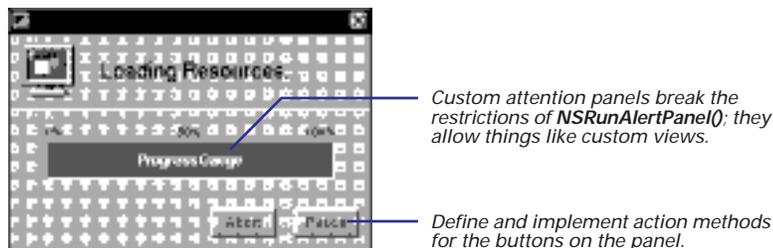
The Application Kit defines other functions related to `NSRunAlertPanel()`. For more information on these functions, see the “Functions” section of the *Application Kit Reference*.

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.



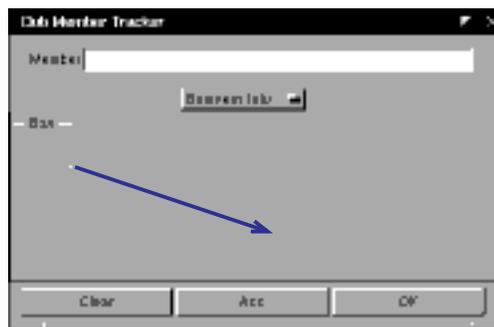
When the user selects an item from this list...



...the middle portion of the window changes.

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 **setContentViewController:** 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.

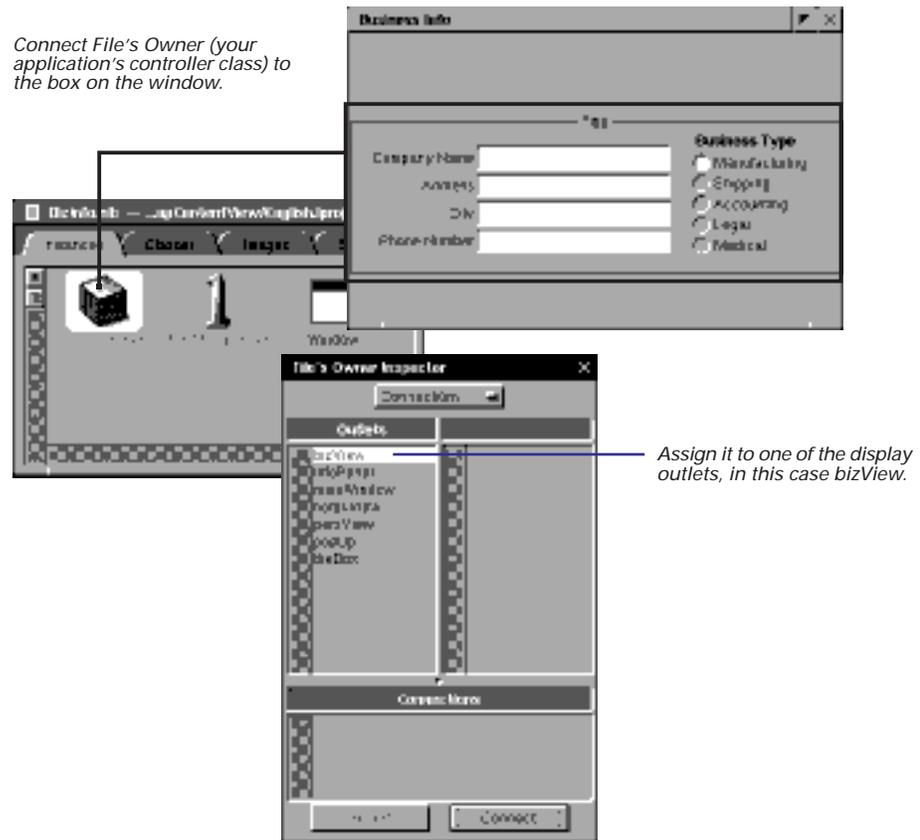


Place interface objects in a box in the auxiliary nib file's main window. When you're done, make the box invisible by choosing no border and no title in the Attributes display.

“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;
    }
}
```

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."

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.)

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.



In Project Builder, click here.



Choose Project Attributes.

Type the name of the principal class here.

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.

12

Creating Frameworks and Dynamic Shared Libraries

Setting up a framework
project

Making a header file private

Installing a precompiled
header

Providing backward
compatibility

Adding public API

Verifying compatibility
between two libraries

My library was dukedom large enough.

Shakespeare, *The Tempest*

I shall sleep, and move with the moving ships
Change as the winds change, veer with the tide.

Algernon Charles Swinburne

Since 'tis Nature's law to change, Constancy alone is
strange.

Hon Wilmot, Earl of Rochester, *A Dialogue between
Strephon and Daphne*

Setting up a framework project

- 1 Create a project with **Framework as the project type**.
- 2 Add header (.h) and implementation (.m) files to the project.
- 3 Specify which header files, if any, should be private.

A framework is a bundle containing a dynamic shared library. Both Framework and Library project types build dynamic shared libraries. The difference is that frameworks bundle the library file with its headers, documentation, and resources.



A project linked against a framework has easy access to headers.

You create a framework or library project from the New panel, just like any other project, but you must do some additional set-up using the Inspector and the makefiles.

The Framework vs. the Library

Because of their convenience, you'll want to create framework projects instead of library projects in most cases. However, if the project doesn't use resources and doesn't contain API that is public to your users (for example, if you distribute an application that uses a private library), you may choose to create a library project instead. If you need to create a static library (and you shouldn't need to), you must create a library project instead of a framework.

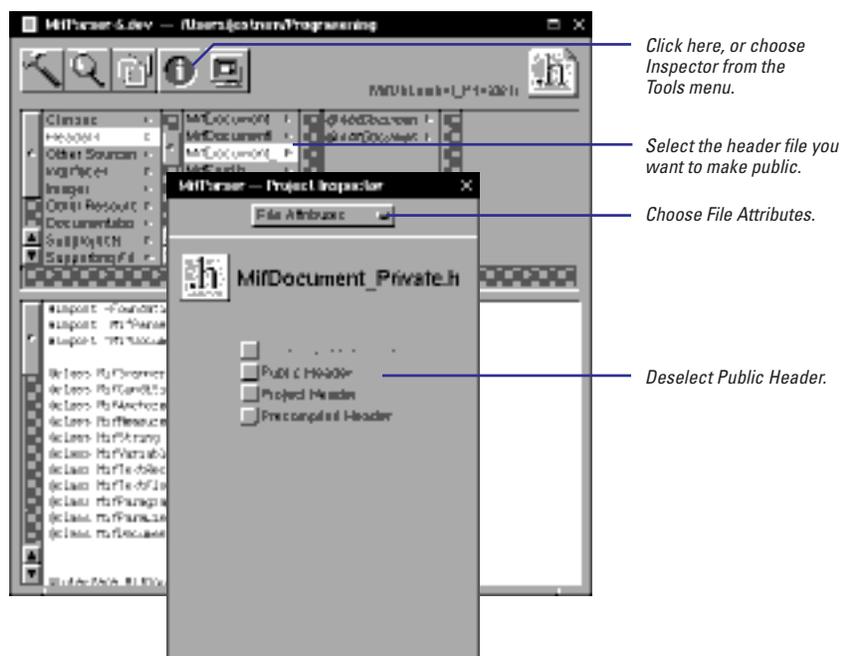
Creating a library project is very similar to creating a framework project. The tasks described in this chapter are things you do when you create either type of project. The main differences between creating a library project and creating a framework project are:

- The name of the binary file. For library projects. The name is **libProjectName.MajorVersion.dylib**. For framework projects it is just **ProjectName**.
- Publishing header files. For framework projects, all header files are public by default. In Library projects, header files are private by default. To install them so that the library's users may access them, you must use the File Attributes inspector to mark each header file as public, and you must specify where to install them using the **PUBLIC_HEADER_DIR** macro in the file **Makefile.preamble**.

Making a header file private

- 1 Select the header file in the project browser.
- 2 Click the Inspector button.
- 3 Choose File Attributes in the Inspector panel.
- 4 Deselect Public Header in the Inspector panel.

By default, all of a framework's header files are public. When the framework is installed, the headers are installed in the framework's **Headers** subdirectory and the framework's users can see those headers from their projects. If you have a header that you don't want your users to see, you must mark it as private.



A good reason to make a header file private is to make sure your users don't use the API. This frees you to change it later. See "Tips and Tricks to Changing the Major Version" in this chapter.

Setting the Search Path for Frameworks and Libraries

When you link a program with a framework (or library), the framework binary's full path is recorded in the program executable. By default, a program only looks in that one location for the binary. If it can't find it, the program won't launch.

To have a program look in more than one location, set the environment variable `DYLD_LIBRARY_PATH`. This variable works like the `PATH` environment variable. For example, if you enter the

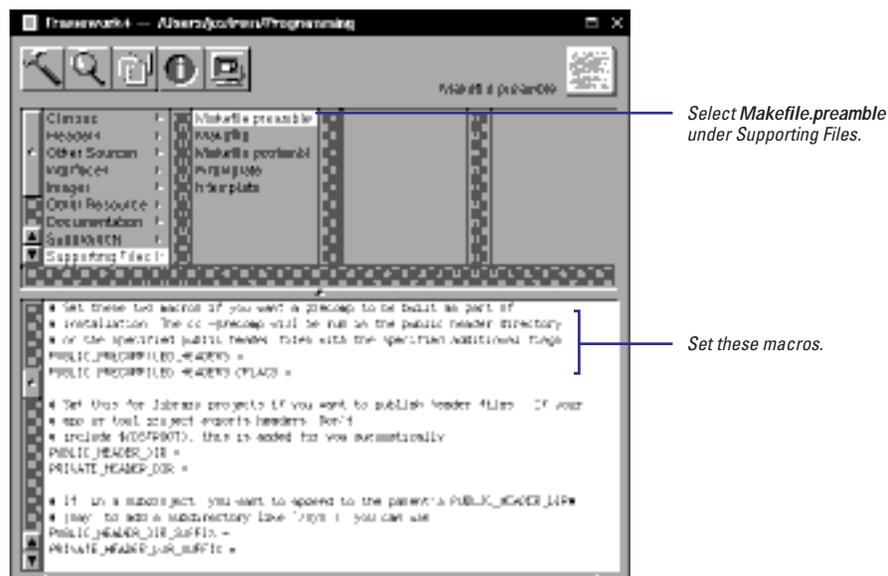
following commands in a Terminal window, the Foo application will look for the binary file **MyFramework** in two locations: the recorded location and in the directory `~/Library/MyFramework.framework`.

```
% setenv DYLD_LIBRARY_PATH \
~/Library/MyFramework.framework
% Foo.app/Foo
```

Installing a precompiled header

- 1 Select the `Makefile.preamble` file under **Supporting Files**.
- 2 Set the macros that affect the precompilation of a header after installation.

You might want to install a precompiled header file so that your users' projects compile faster. Installing a precompiled header is different from creating a precompiled header for a project because a header must be precompiled in its final location. When you create a precompiled header for a project, the header is compiled before the rest of the project. To install a precompiled header, you must first build and install the project in its destination, then precompile the header.



To precompile a header after it is installed, set these macros in **Makefile.preamble**:

Preamble Macro	Description
<code>PUBLIC_PRECOMPILED_HEADERS</code>	The names of the headers (<code>.h</code> extension) that should be precompiled after they are installed.
<code>PUBLIC_PRECOMPILED_CFLAGS</code>	The flags besides -precomp to pass to cc when precompiling.

Chapter 8 describes how to precompile a header for internal use by a project and the things to consider when you create the header file that is going to be precompiled.

Macros for the Makefile Hacker

The files **Makefile.preamble** and **Makefile.postamble** define several macros that affect frameworks and libraries. Using these macros, you can change the way a framework or library is built or installed. (See Chapter 9 for a description of the other macros in these files.)

By default, a framework project builds a bundle named *ProjectName.framework* with the subdirectories **Headers**, **Resources**, and **Versions**. Each major version is installed in a subdirectory under **Versions** along with its public headers, documentation, and resources in the appropriate subdirectories. Also under **Versions** is a subdirectory named **Current**, which contains links to the latest version. The subdirectories immediately under *ProjectName.framework* are really just symbolic links into **Current**.

A library project creates a binary file named **libProjectName.MajorVersion.dylib** and a symbolic link to this file named **libProjectName.dylib**. Both are installed in **/usr/lib**. No headers are installed by default.

Makefile.preamble Macros

SECTORDER_FLAGS Arguments to the linker's **-sectorder** option. See the **ld(1)** man page for more information.

OTHER_PUBLIC_HEADERS Header files that should be installed as public other than those marked as public in the File Attributes inspector.

OTHER_PRIVATE_HEADERS Header files that should be installed as private other than those included in the project.

PUBLIC_HEADER_DIR Location in which to install public headers. You must define this for library projects if you want header files to be installed when the library is installed. For frameworks, any header file marked as public is placed in the **Headers** subdirectory.

PUBLIC_PRECOMPILED_HEADERS Header files to be precompiled after installation. See "Installing a precompiled header" in this chapter.

PUBLIC_PRECOMPILED_HEADERS_CFLAGS See "Installing a precompiled header" in this chapter.

PRIVATE_HEADER_DIR Location in which to install private headers, which can be stripped away separately from your product build image. The default is not to install private headers.

PUBLIC_HEADER_DIR_SUFFIX Define this macro if a framework or library has a subproject whose public headers should be installed in a subdirectory of the parent's public header

directory. For example, if you define this macro as **/sys**, they are installed in **Headers/sys**.

PRIVATE_HEADER_DIR_SUFFIX The same as **PUBLIC_HEADER_DIR_SUFFIX**, but for private headers.

LIBRARY_STYLE If **STATIC**, builds a static archive library (**.a** extension) rather than a dynamic shared library.

BUILD_OFILES_LIST_ONLY If **YES**, links the object files in the project together but does not call **libtool** to create a dynamic shared library from the object files. This macro is useful if you want to use the modules in another, larger library project.

Makefile.postamble Macros

CURRENTLY_ACTIVE_VERSION If **YES**, a symbolic link to the framework's binary file is created in the directory **Versions/Current**. If **NO**, the link is not created. The default is **YES**. Set this to **NO** if you want to install a new version of a framework but you still want projects to link against the previously installed version. This macro does not affect library projects. Using this macro is the same as checking the current version box on the Project Attributes inspector.

DEPLOY_WITH_VERSION_NAME This is the same as changing the version name in the Project Attributes inspector. See "Providing backward compatibility" in this chapter.

CURRENT_PROJECT_VERSION The minor version number. See "CURRENT_PROJECT_VERSION: For That Extra Level of Checking" in this chapter.

COMPATIBILITY_PROJECT_VERSION The compatibility version number. See "Adding public API" in this chapter.

DYLIB_INSTALL_NAME The name of the binary file that is built. The default is **libProjectName.MajorVersion.dylib** for library projects, *ProjectName* for frameworks.

DYLIB_INSTALL_DIR Sets the path recorded in the library's binary file. **\$DYLIB_INSTALL_DIR/\$DYLIB_INSTALL_NAME** is passed as the argument to the **-install_name** option of **libtool**, which is used to set the name recorded in the library file to be something other than its path name. The default is not to use this option.

LIBRARY_STRIP_OPTS Options to pass to **strip** for statically linked libraries. You shouldn't have to create a static library, so you shouldn't have to use this macro.

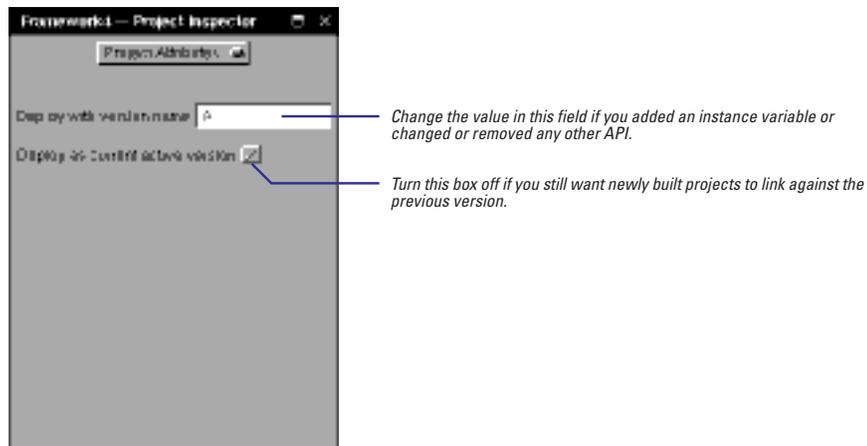
DYNAMIC_STRIP_OPTS Options to pass to **strip** for framework projects and dynamic shared library projects.

Providing backward compatibility

- 1 Click the **Inspector** button.
- 2 Choose **Project Attributes** from the Inspector panel.
- 3 Set the **Deploy with version name** field in the **Project Attributes** inspector if you have removed or changed API.
- 4 Build the project.

When you change a framework, you want to make sure not to break existing programs. If you do one of the following to your framework, you are in danger of breaking programs that link with it:

- Remove any public API.
- Change any API, such as a method or function declaration or a class name.
- Add instance variables to a class.
- Rearrange the order of instance variables in a class.
- Remove any of the architectures the framework is built for.



Whenever you make one of these changes, you should increment the framework's major version letter and provide both the new and old binary to your users. That way, programs linked against the older version of the framework will still run. New programs or modified programs will link with the newer version of the framework.

The **Install in:** field of the **Build Attributes** inspector provides the first half of the framework's full name. Variables such as **\$(HOME)** are expanded *before* the path name is recorded. For more information on the **Build Attributes** inspector, see Chapter 9.

A dynamic shared library's name contains the major version letter. This name is recorded in the executable when a program links with the library. Thus, any program that links with a dynamic shared library knows that library's major version. The program won't launch if it can't find a library with the correct name.

For example, if the program **MyProg** links with version **A** of the framework **Misc**, the path `/LocalLibrary/Frameworks/Misc.framework/Versions/A/Misc` is recorded in **MyProg**. Suppose you add an instance variable to a class in **Misc** and change version to **B**. This builds `Misc.framework/Versions/B/Misc` but leaves `Versions/A/Misc` intact. Because version **A** still exists, **MyProg** can still run. If you change **MyProg** and rebuild it, it links with version **B**.

Tips and Tricks to Changing the Major Version

If you don't change the framework's major version number when you need to, programs linked with it will fail in unpredictable ways. If you change the major version number and you don't need to, you're cluttering up the system with compatible frameworks. You can avoid errors in changing the major version number if you follow a few simple tricks.

Don't Do It

The first trick is to avoid having to change the version number in the first place. Some ways to do this are:

- Pad classes and structs with reserved fields. Whenever you add an instance variable to a public class, you must change the major version number because subclasses depend on a superclass's size. However, you can pad a class by defining an unused instance variable of type `id`. Then, if you need to add instance variables to the class, you can instead define a whole new class containing the storage you need and have your reserved instance variable point to it.
- Don't publish API unless you want your users to use it. You can freely change private API because you can be sure no programs are using it. Declare any API in danger of changing in a private header. See "Making a header file private" in this chapter.
- Don't delete things. If a method or function no longer has any useful work to perform, leave it in the API for compatibility purposes. Make sure it returns some reasonable value. (Even if you add additional arguments to a method, leave the old form around if at all possible.)

- Remember that if you *add* API rather than change or delete it, you don't have to change the major version number because the old API still exists. The exception to this rule is instance variables. (You do have to change the compatibility version number, however. See "Adding public API" in this chapter.)

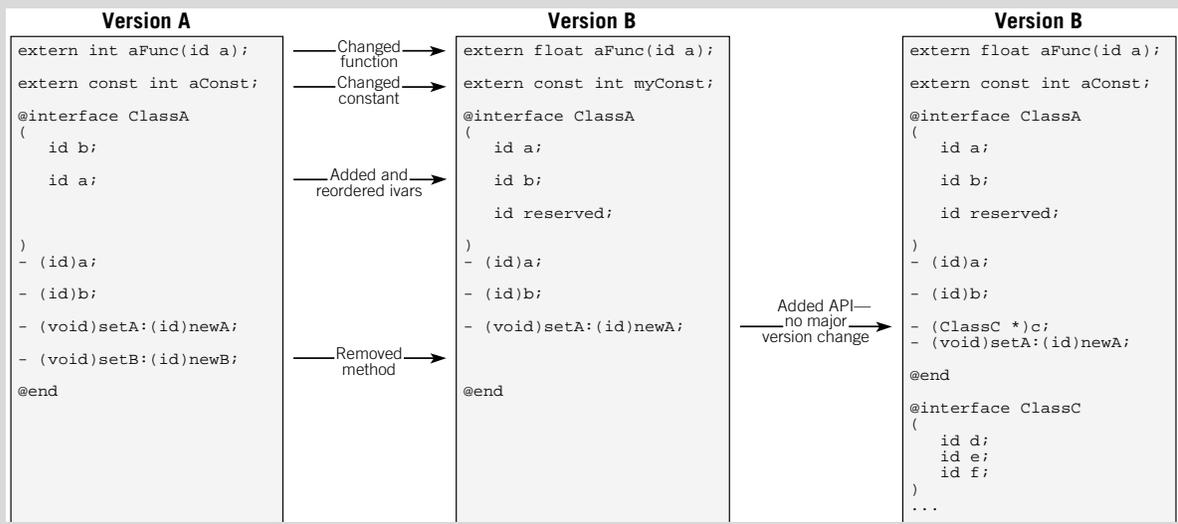
If You Do, Don't Clean It

make clean deletes the entire **.framework** bundle in the project directory, which means it deletes the old binaries in addition to the current binary. The subsequent build creates only the current version. You have no way of retrieving the earlier versions.

If you must perform a **make clean**, you'll need to create multiple copies of the project: one that builds the current version, and one for each of the previous versions. The projects that build the previous versions should set the **CURRENTLY_ACTIVE_VERSION** macro to `NO` so that the pointer to the current version is not changed when these older versions are installed. When you install, you'll need to install all versions.

Verify Whatever You Do

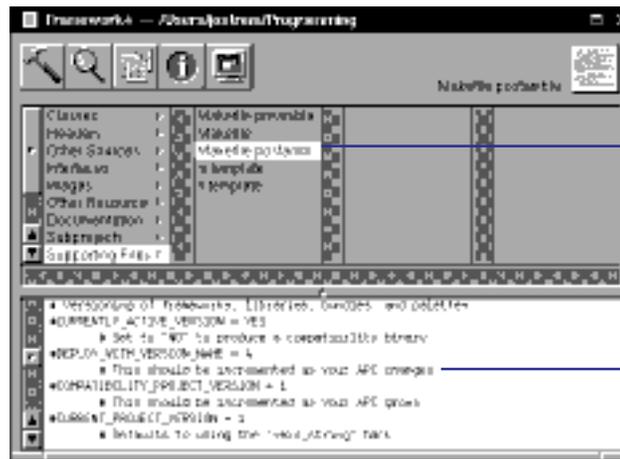
Use **cmpdylib** to make sure you did the right thing. If **cmpdylib** says the older library defines symbols that aren't defined in the newer library, you need to change the major version number. See "Verifying compatibility between two libraries" in this chapter.



Adding public API

- 1 **Change the compatibility project version number and the current project version number in `Makefile.postamble`.**
- 2 **Build the project.**

You shouldn't change the major version number when you add API (for example, if you add a class, add a method to an existing class, or add a function or constant). Adding API doesn't break existing programs. Existing programs are guaranteed to be using the older API and will still run because you've left the older API intact. However, new programs might use the new API, and therefore shouldn't try to run against older versions of the framework, which don't define that API.



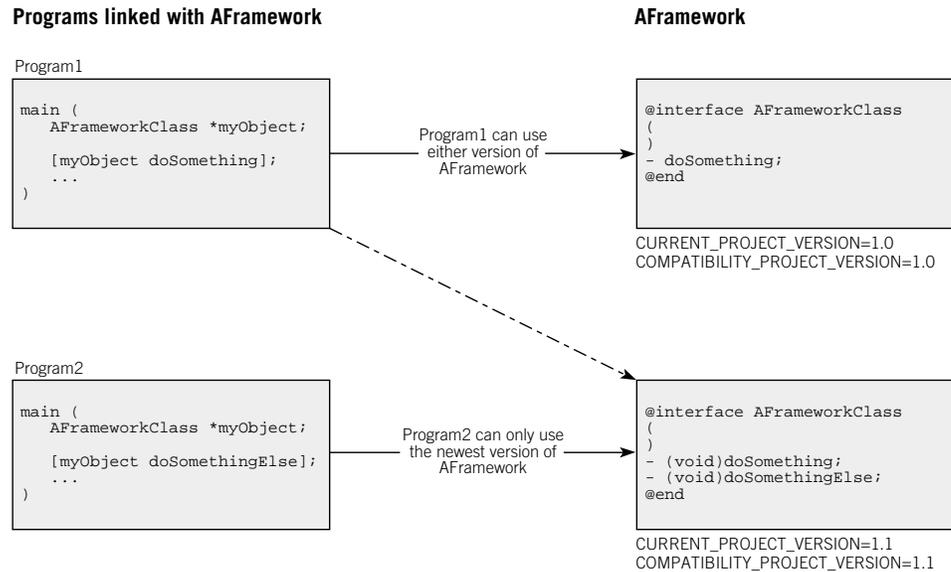
Select `Makefile.postamble` under Supporting Files.

Increment the value in `COMPATIBILITY_PROJECT_VERSION` and in `CURRENT_PROJECT_VERSION` when you add API, such as methods, classes, functions, or constants.

When you add API, increment the compatibility version number. The compatibility version number protects programs linked with newer versions of a library from running with older versions of the library. In order for a program to launch, the compatibility version number of the framework it runs with must be equal to or greater than the **CURRENT_PROJECT_VERSION** number of the framework it linked with.

Remember that adding instance variables to a class is an incompatible change, which means you should change the version name instead of the compatibility version number. See “Tips and Tricks to Changing the Major Version” for an explanation.

Increment the value in **CURRENT_PROJECT_VERSION** whenever you change the compatibility version number. See “CURRENT_PROJECT_VERSION: For That Extra Level of Checking” in this chapter.



Why shouldn't you just change the major version number when you add API? Because programs linked with the previous version of the framework still run with the new version. If you change the major version number, the previous version remains installed on your users' systems. By changing the compatibility version number instead, you can install just one version.

CURRENT_PROJECT_VERSION: For That Extra Level of Checking

In addition to the major version number, and the compatibility version number, a dynamic shared library has a third version number. This is the minor version number or current version number. You set the current version number in the macro **CURRENT_PROJECT_VERSION**, which is in **Makefile.postamble**.

At the very least, increment **CURRENT_PROJECT_VERSION** every time you increment **COMPATIBILITY_PROJECT_VERSION**. The **CURRENT_PROJECT_VERSION** stored in a program's executable is compared with the **COMPATIBILITY_PROJECT_VERSION** stored in the library's binary file. The version in the program must be greater than or equal to the version in the library for the program to launch.

The intent is that you increment **CURRENT_PROJECT_VERSION** every time you distribute the framework when you haven't changed or added API. For example, if you fix a bug in the way a

method works, you increment **CURRENT_PROJECT_VERSION**. Changes involving implementation only are almost always compatible. Programs linked against older versions of the framework can run against the new version and in fact are actually intended to run against the new version. Programs linked against the new version can still run against the old version (even though they will then encounter the bug that you have fixed).

In rare cases, someone may write a program that needs a fix from a certain version of the library. That program can use the function **NSVersionOfRuntimeLibrary()** to determine the current version of the library and take the appropriate action if the version isn't the one it needs: put up an alert panel, disable some feature of the program, or disable the entire program. Because of these rare cases where a program may need to check the version number, you should always increment **CURRENT_PROJECT_VERSION** when you distribute a new framework.

Verifying compatibility between two libraries

- 1 Start up the Terminal application.
- 2 Perform the `cmpdylib` command.

`cmpdylib` is a verification tool that you can use to make sure you've made the right choices about version numbers. The syntax is:

```
cmpdylib oldLibName newLibName
```

If *oldLibName* and *newLibName* are compatible, this command returns nothing. If they aren't compatible, it tells you why.

```
bin/csh (tty1)
jean> cmpdylib ParserKit1 ParserKit2
jean>
jean>
jean>
jean>
jean> cmpdylib ParserKit2 ParserKit3
compatibility versions are the same but new symbols defined in ParserKit3 not de
fined in ParserKit2:
.objc_class_name_MifFrame
jean>
jean>
jean>
jean>
jean> cmpdylib ParserKit3 ParserKit4
symbols defined in ParserKit3 not defined in ParserKit4:
.objc_class_name_MifParaLine
compatibility versions are the same but new symbols defined in ParserKit4 not de
fined in ParserKit3:
.objc_class_name_MifLineOfText
jean>
```

These should have different compatibility versions because the newer library has more symbols.

These should have different major versions because the older library has a symbol not defined in the newer library.

`cmpdylib` considers two libraries compatible if:

- They are built for the same architectures.
- *oldLibName* defines a subset of the symbols that *newLibName* does.
- *newLibName* defines symbols not in *oldLibName* and has a different compatibility version number.

The two libraries are incompatible if:

- They are built for different architectures.
- *oldLibName* defines symbols that aren't in *newLibName*.
- *newLibName* defines symbols not in *oldLibName* and has the same compatibility version number.

Currently, `cmpdylib` only checks C-level API and does not distinguish between public and private API. For example if you add a method, `cmpdylib` won't detect the change. Also if you change a private class, `cmpdylib` will report the change as an incompatibility.

