Currency Converter Tutorial Chapter 2

2

What You'll Learn

Creating a simple graphical user interface

Creating a custom subclass

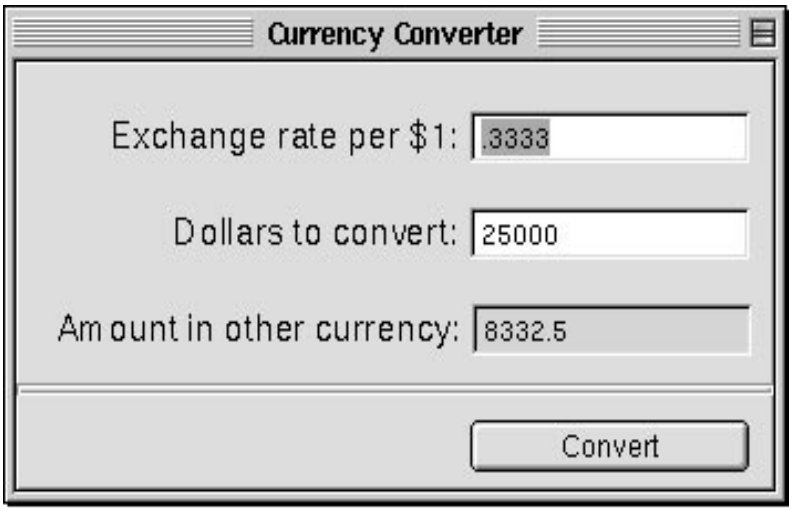
Connecting objects in the application

Sending a message to an object

Responding to a message

Building a project

Getting help



You can find the Currency Converter project in the **AppKit** subdirectory of **/System/Developer/Examples**.

Chapter 2
A Simple Application

The application that you are going to create in this tutorial is called Currency Converter. It is a simple application, yet it exemplifies much of what software development with OpenStep is about. As you'll discover, Currency Converter is amazingly easy to create, but it's equally amazing how many features you get "for free"— as with all OpenStep applications.

Currency Converter converts a dollar amount to an amount in another currency, given the rate of that currency relative to the dollar. You type a rate and an amount into text fields and then click a button to see the result. Instead of clicking the button, you can also press the Return key. You can double-click the converted amount, copy it (with the Edit menu's Copy command) and paste it in another application that takes text. You can tab between the first two fields. You can do many other things common to OpenStep applications.

By following the steps of this chapter, you will become familiar with the two most important OpenStep applications for program development: Interface Builder and Project Builder.

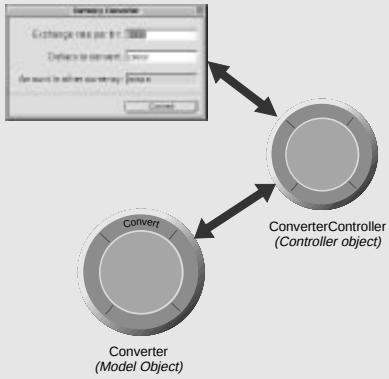
Currency Converter's Design, the Development Process, and a Design Paradigm

An object-oriented application should be based on a design that identifies the objects of the application and clearly defines their roles and responsibilities. You normally work on a design before you write a line of code. You don't need any fancy tools for designing many applications; a pencil and a pad of paper will do.

Currency Converter is an extremely simple application, but there's still a design behind it. This design is based upon the Model-View-Controller paradigm, a model behind many designs for object-oriented programs (see next page). This design paradigm aids in the development of maintainable, extensible, and understandable systems. But first, you might want to read "Why an Object Looks Like a Jelly Donut" on page 29 to understand the symbol used in the design diagram.

This design for Currency Converter is intended to illustrate a few points, and so may be overly designed for something so simple. It is quite possible to have the application's controller class, ConverterController, do the computation and do without the Converter class.

You can divide responsibility within Currency Converter among two custom objects and the user interface, taken as a collection of ready-made Application Kit objects. The Converter object is responsible for computing a currency amount and returning that value. Between the user interface and the Converter object is a *controller object*, ConverterController. ConverterController coordinates the activity between the Converter object and the UI objects.



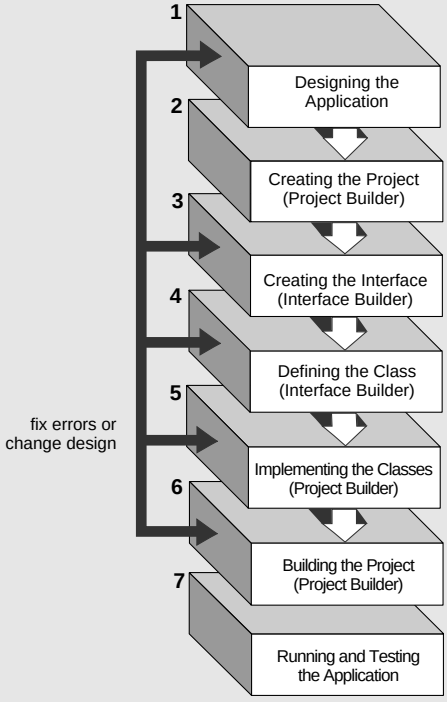
The ConverterController class assumes a central role. Like all controller objects, it communicates with the interface and with model objects, and it handles tasks specific to the application. ConverterController gets the values that users enter into fields, passes these values to the Converter object, gets the result back from Converter, and puts this result in a field in the interface.

The Converter class merely computes a value from two arguments passed into it and returns the result. As with any model object, it could also hold data as well as provide computational services. Thus, objects that represent customer records (for example) are akin to Converter. By insulating the Converter class from application-specific details, the design for Currency Converter makes it more reusable, as you'll see in the Travel Advisor tutorial.

Typical Development Workflow

This chapter introduces the typical flow of work involved in developing an OpenStep application

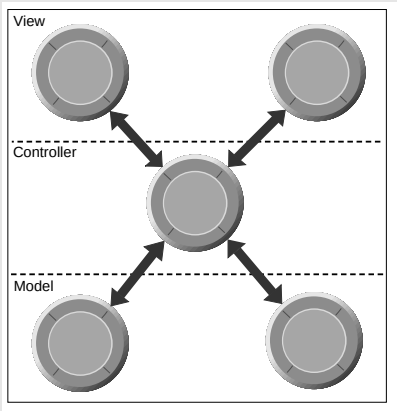
.



Note: Although this diagram shows the design phase at the beginning of the workflow process, application design can take place any time in the early stages of the project. It is often recommended as the first stage, however, and it is a good idea to review the design occasionally and modify it if necessary.

The Model-View-Controller Paradigm

A common and useful paradigm for object-oriented applications, particularly business applications, is Model-View-Controller (MVC). Derived from Smalltalk-80, MVC 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 directly displayed. They often are reusable, distributed, persistent, 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). You can also group View objects within a window in novel ways specific to an application. 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. (This last statement does not mean, however, that Controller objects *cannot* be reused; with a good design, they can.)

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 it's 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.

A Note on Terminology

The Application Kit and Enterprise Objects Framework reserve special meanings for "view object" and "model." A view object in the Application Kit denotes a user-interface object that inherits from NSView. In the Enterprise Objects Framework, a model establishes and maintains a correspondence between an enterprise object class and data stored in a relational database. This book uses "model object" only within the context of the Model-View-Controller paradigm.

Creating the Currency Converter Project

Every Rhapsody application starts out as a *project*. A project is a repository for all the elements that go into the application, such as source code files, makefiles, frameworks, libraries, the application's user interface, sounds, and images. You use the Project Builder application to create and manage projects.

1 Launch Project Builder.

Locate the Project Builder application (icon at right).

Double-click the icon to start the application.

Project Builder is located at **/System/Developer/Apps/ProjectBuilder.app**.

When Project Builder starts up, it displays the New Project panel. The New Project panel lets you specify a new project's name, location, and type.

1 Make a new project.

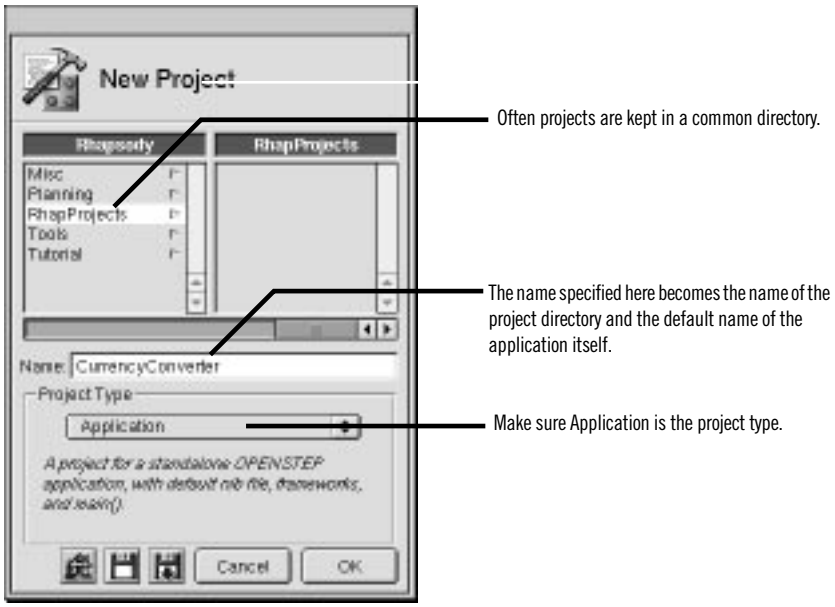
Choose New from the Project menu (Project ► New).

In the New Project panel, choose the Application project type from the pop-up list.

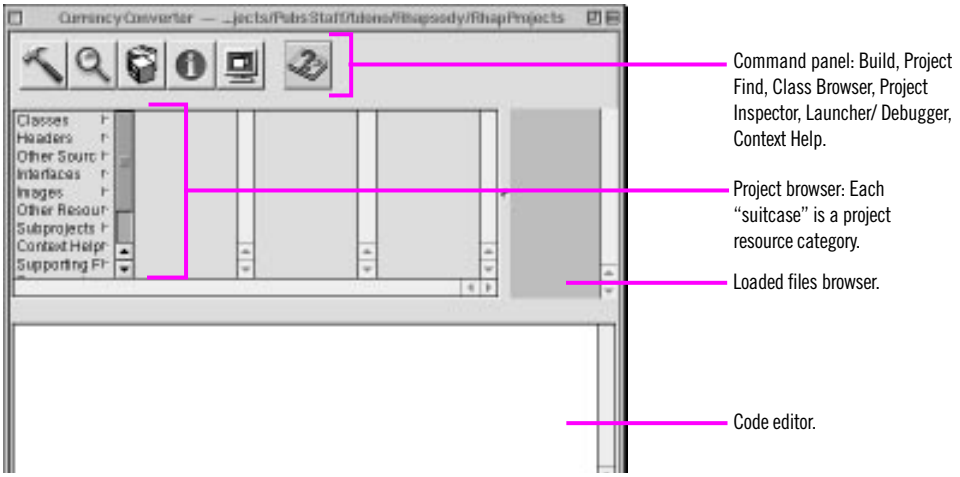
Using the file browser, go to the directory you want the project to be in.

Type "CurrencyConverter" in the Name field.

Click OK to create the project.



Project Builder creates a project directory named after the project—in this case CurrencyConverter—and populates this directory with an assortment of ready-made files and directories. It then displays its main window.



Go ahead and click an item in the left column of the project browser (a grouping of project resources sometimes called a "suitcase"); see what some of these suitcases contain already:

- **Other Sources:** This suitcase contains **CurrencyConverter_main.m**, the **main()** routine that loads the initial set of resources and runs the application. (You shouldn't have to modify this file.)
- **Interfaces:** This suitcase contains the nib files (extension".nib") which specify the application's user interface. More on nib files in the next step.
- **Supporting Files:** This suitcase contains the project's default makefiles and template source-code files. You can modify the preamble and postamble makefiles, but you must leave **Makefile** unchanged.
- **Frameworks:** This suitcase contains the frameworks (which are similar to libraries) which the application imports.

Project Indexing

When you create or open a project, after some seconds you may notice triangular "branch" buttons appearing after source code files in the browser. Project Builder has indexed these files.

During indexing Project Builder stores all symbols of the project (classes, methods, globals, etc.) in virtual memory. This allows Project Builder to access project-wide information quickly. Indexing is indispensable to such features as name completion and Project Find. (More on these features later.)

Usually indexing happens automatically when you create or open a project. You can turn off this option if you wish. Choose Preferences from the Tools menu and then choose the Indexing display. Turn off the "Index when project is opened" switch.

You can also index a project at any time by choosing Tools ▶ Indexer ▶ Index Subproject. If you want to do without indexing (maybe you have memory constraints), choose Tools ▶ Indexer ▶ Purge Indices.

Creating the Currency Converter Interface

When you create an application project, Project Builder puts the *main nib file* in the Interfaces suitcase. A nib file is primarily a description of a user interface (or part of a user interface). The main nib file contains the main menu and any windows and panels you want to appear when your application starts up; at start-up time, each application loads the main nib file.

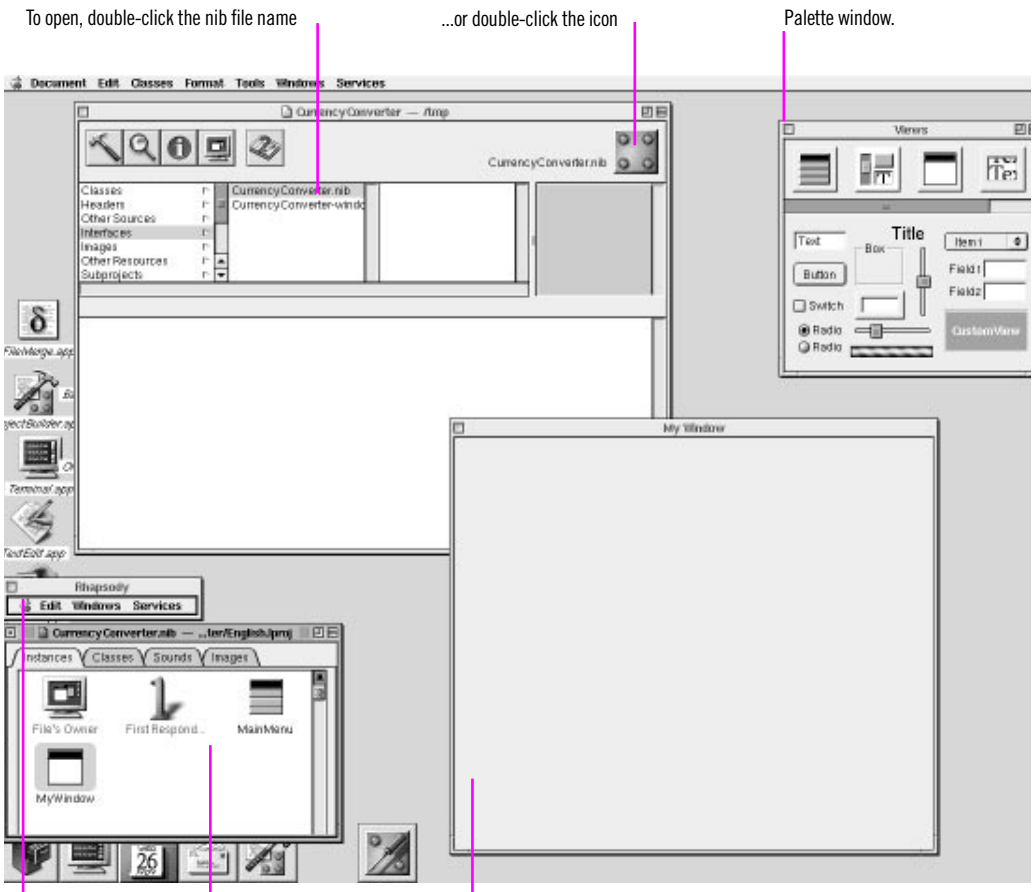
Customizing the Application's Window

At the beginning of a project, the main nib file is like a blank canvas, ready for you to craft the interface. Look in the Interfaces suitcase for nib files.

1 Open the main nib file.

Locate **CurrencyConverter.nib** in the project browser.

Double-click to open.



By default, a blank window entitled "My Window" will appear when the application is launched.

What's in a Nib File

Every application has at least one nib file. The main nib file contains the application 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.

Images Image files that you drag and drop over the nib file window or over an object that can accept them (such as a button or image view).

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 connect.

When You Load a Nib File

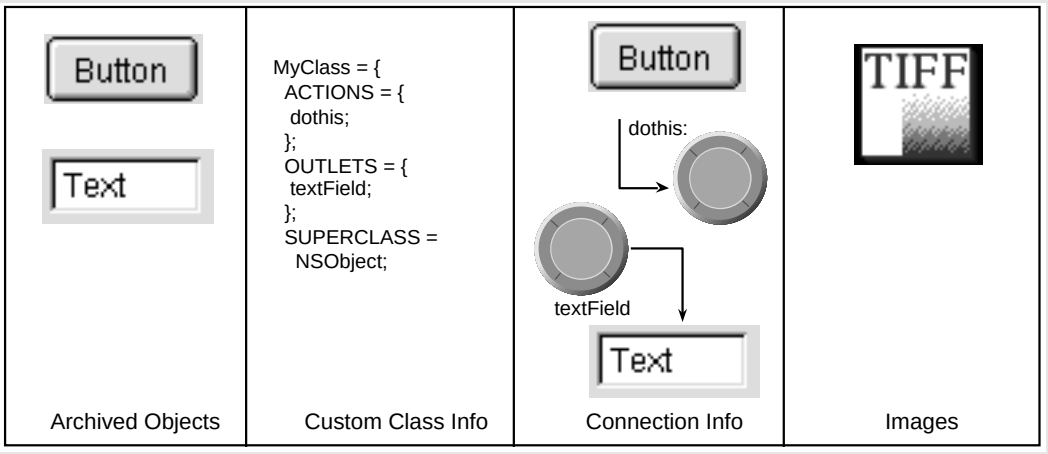
In your code, you can load a nib file by sending the NSBundle class **loadNibNamed:owner:** or **loadNibFile:externalNameTable:withZone:**. messages. When you do this, the run-time system does the following:

- It unarchives the objects from the object hierarchy, sending each object an **initWithCoder:** message after allocating memory for it.
- 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.
- It unarchives the connector objects and allows them to establish connections, including connections to File's Owner.
- It sends **awakeFromNib** to all objects that were derived from information in the nib file, signalling that the loading process is complete.

Connections and Accessor Methods

When OpenStep establishes connections during the course of loading a nib file, it sets the values of the source object's outlets. It first tries to set an outlet through the "set" accessor method if the source object implements it. For example, if the source object has an outlet named "contraption," the system first sees if that object responds to "setContraption:" and, if it does, it invokes the accessor method. If the source object doesn't implement the accessor method, the system sets the outlet directly.

Problems naturally ensue if a "set" accessor method does something other than directly set the outlet. One common example is an accessor method that sets the *string value* of an outlet referring to a text field (**setStringValue:**). After loading, the value of the outlet is **nil** because the "set" accessor method did not directly assign the value.



1 Resize the window.

Make the window smaller by dragging a corner of the window inward.

Most objects on an interface have attributes that you can set in the Inspector panel's Attributes display.

1 Set the window's title and attributes.

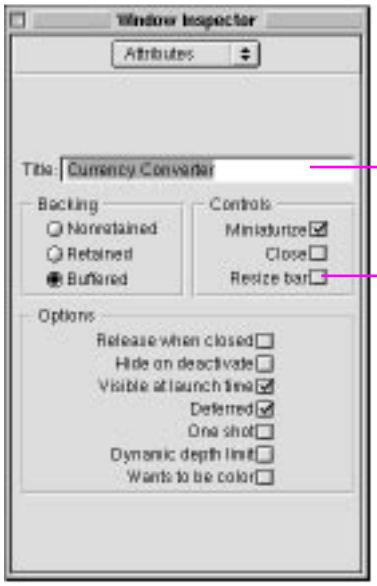
Click the window to select it.

Choose Tools ► Inspector.

Select the Attributes display from the pop-up list.

Enter the window title.

Turn off the resize bar.



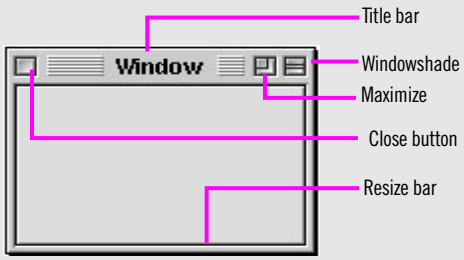
The title of the major window in an application is often the application name.

When this option is turned off, the windows's resize bar disappears.

Note: You can also bring up the Attributes display of the inspector by typing Control-1.

A Window in OpenStep

A window in OpenStep looks very similar to windows in other user environments such as Windows or Mac OS. It is a rectangular area on the screen in which an application displays controls, fields, text, and graphics. Windows can be moved around the screen and stacked on top of each other like pieces of paper. A typical OpenStep window has a title bar, a content area, and several control objects.



Many user-interface objects other than the *standard window* depicted above are windows. Menus, pop-up lists, and pull-down lists are primarily windows, as are all varieties of panels: attention panels, inspectors, and tool palettes, to name a few. In fact, *anything* drawn on the screen must appear in a window.

NSWindow and the Window Server

Two interacting systems create and manage OpenStep windows. On the one hand, a window is created by the Window Server. The Window Server is a process integrating the Window System and Display Postscript. The Window Server draws, resizes, hides, and moves windows using Postscript primitives. The Window Server also detects user events (such as mouse clicks) and forwards them to applications.

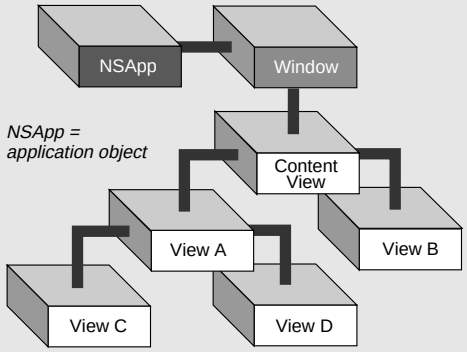
The window that the Window Server creates is paired with an object supplied by the Application Kit: an instance of the NSWindow class. Each physical window in an object-oriented program is managed by an instance of NSWindow (or subclass).

When you create an NSWindow object, the Window Server creates the physical window that the NSWindow object will manage. The Window Server references the window by its window number, the NSWindow by its own identifier.

Application, Window, View

In a running OpenStep application, NSWindow objects occupy a middle position between an instance of NSApplication and the views of the application. (A view is an object that can draw itself and detect user events.) The NSApplication object keeps a list of its windows and tracks the current status of each. Each window, on the other hand, manages a hierarchy of views in addition to its PostScript window.

At the "top" of this hierarchy is the *content view*, which fits just within the window's content rectangle. The content view encloses all other view (its *subviews*), which come below it in the hierarchy. The NSWindow distributes events to views in the hierarchy and regulates coordinate transformations among them.



Another rectangle, the *frame rectangle*, defines the outer boundary of the window and includes the title bar and the window's controls. The lower-left corner of the frame rectangle defines the window's location relative to the screen's coordinate system and establishes the base coordinate system for the views of the window. Views draw themselves in coordinate systems transformed from (and relative to) this base coordinate system.

See page 153 for more on the view hierarchy.

Key and Main Windows

Windows have numerous characteristics. They can be on-screen or off-screen. On-screen windows are "layered" on the screen in tiers managed by the Window Server. On-screen windows also can carry a status: *key* or *main*.

Key windows respond to key presses for an application and are the primary recipient of action messages from menus and panels. Usually a window is made key when the user clicks it. Key windows have black title bars. Each application can have only one key window.

An application has one main window, which can often have key status as well. The main window is the principal focus of user actions for an application. Often user actions in a modal key window (typically a panel such as the Font panel or an inspector) have a direct effect on the main window. In this case, the title bar of the main window (when it is not key) is a dark gray.

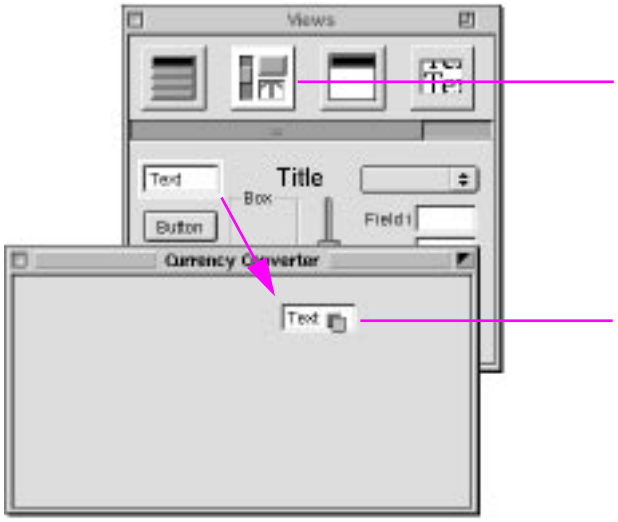
Fields and Buttons

Fields and buttons are the most common types of objects found on interfaces. Put these and other palette objects on the window using the "drag and drop" technique.

1 Put a text field on the interface and resize and initialize it.

Select the Views palette.

Drag a text field from the palette onto the window.



Click this icon to select the Views palette. This palette contains an assortment of commonly used Application Kit objects.

Drag a text field and drop it (that is, release the mouse button) over the "surface" of the window.

Move an object by dragging it around the surface of the window.

To initialize the text field, double-click "Text" and press Delete.

You must get rid of the word "Text" in this field; otherwise, that's what the field will show when the nib file is loaded.

The text field should be longer so it can hold more digits (you're dealing with millions here):

Lengthen the text field.

Drag a resize handle in the direction you want the object to grow.

Currency Converter needs two more text fields, both the same size as the first. You have two options: you can drag another object from the palette and make it the same size, or you can duplicate the first object.

1 Duplicate an object.

Select the text field.

Choose Edit ► Copy.

Choose Edit ► Paste.

The new text field appears slightly offset from the original field. Reposition it below the first text field.

Get the third field from the palette and make it the same size as the first field.

1 Make objects the same size.

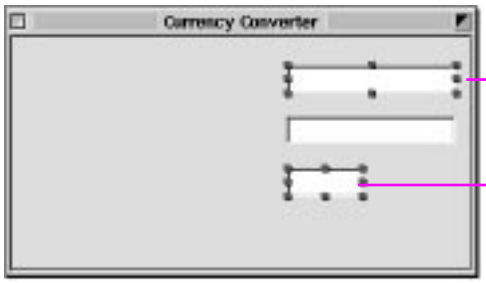
Drag a text field onto the window.

Delete "Text" from the text field.

Select the first text field.

Shift-click to select the new text field.

Choose Format ► Size ► Same Size



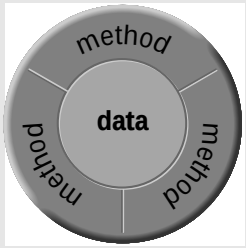
The first object you select should have the dimensions you want the other objects in the selection to take.

Shift-click multiple objects to include them in the selection.

You're not done yet with these text fields. The bottom text field displays the result of the computation. It should not be editable and therefore should, by convention, have a non-white background.

Why an Object Looks Like a Jelly Donut

Or a lifesaver. Or a slashed tire. Or segmented unity. This book depicts objects as this symbol:



Why this unlikely shape?

This symbol illustrates *data encapsulation*, the essential characteristic of objects. An object consists of both data and procedures for manipulating that data. Other objects or external code cannot access that data directly, but must send *messages* to the object requesting its data.

An object's procedures (called *methods*) respond to the message and may return data to the requesting object. As the symbol suggests, an object's methods do the encapsulating, in effect mediating access to the object's data. An object's methods are also its interface, articulating the ways in which the object communicates with the world outside it.

The donut symbol also helps to convey the *modularity* of objects. Because an object encapsulates a defined set of data and logic, you can easily assign it to particular duties within a program. Conceptually, it is like a functional unit—for instance, "Customer Record"—that you can move around on a design board; you can then plot communication paths to and from other objects based on their interfaces.

See the appendix "Object Oriented Programming," for a fuller description of data encapsulation, messages, methods, and other things pertaining to objects.

1 Change the attributes of a text field.

Select the third text field.

Choose Tools ► Colors.

Select the grayscale palette of the Color panel.

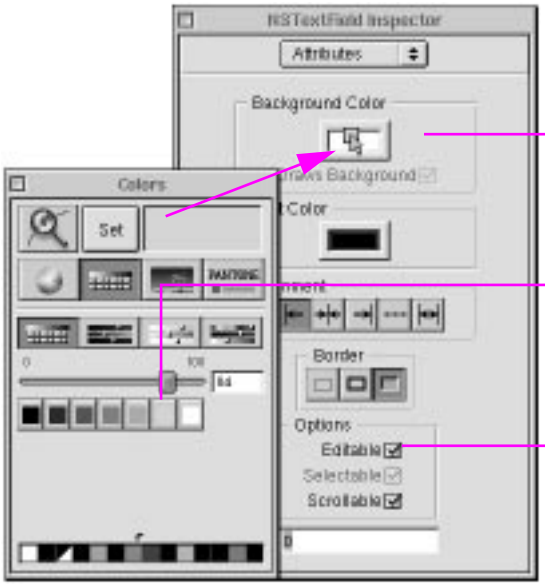
Select the color that is the same as the window background.

Choose Tools ► Inspector.

Select the Inspector panel's Attributes display.

Drag the gray color from the Color panel into the Background Color well.

Turn off the Editable and Scrollable options.



Drag the gray color into the well to apply the color to the selected object.

This color blends the text field into the window background.

With the Editable attribute turned off, users cannot alter the contents of the text field.

Keep Selectable as an option so users can copy and paste the contents to other applications.Keep Selectable as an option so users can copy and paste the contents to other applications.

The Views palette provides a "Title" object that you can easily adapt to be a text-field label. (The title object is actually a text field, set to have a gray background and no border, and to be non-editable and non-selectable.) Text in the title object is centered by default, but labels are often aligned from the right.

1 Assign labels to the fields.

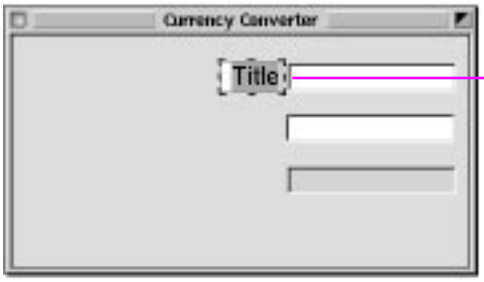
Drag a "Title" object onto the window.

Double-click to select the text.

Choose Inspector from the Tools menu.

Select the Attributes display.

Click the middle button under Alignment to align the text with the right edge of the text field.



The text is highlighted when it is selected.

The size of the text is rather large for a label, so change it. You set font family, typeface, and size with the standard OpenStep Font panel.

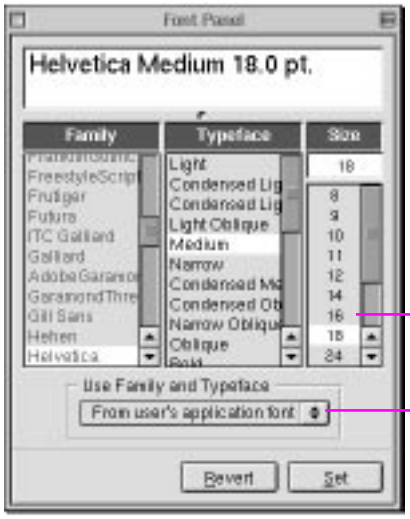
Make sure the object's text is selected.

Choose Format ► Font ► Font Panel.

Set the label text to 16 points.

Make two copies of the label.

Position all labels to the left of their text fields.

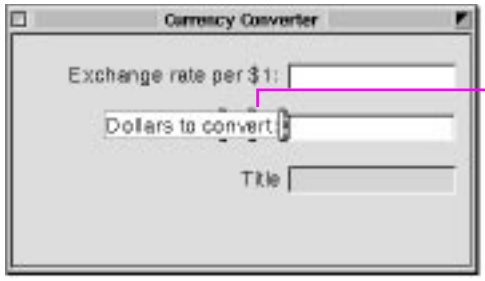


The font of the "Title" object is 18 points Helvetica. Click here and then click the Set button to set the font size to 16 points.

You should select the font that users request for applications in case the font you select is not available on the user's system.

When you cut and paste objects that contain text, like these labels, the object should be selected and not the text the object contains; if the text is selected, de-select it by clicking outside the text, then click the object again to select it.

Type the text of each label.



Double-click to select "Title," then type the text of the label in place of the selection.

1 Add a button to the interface and initialize it.

Drag the button object from the Views palette and put it on the lower-right corner of the window.

Make the button the same size as a text field.

Change the title of the button to "Convert".

You can resize buttons the same way you resize text fields or any other object on a window.

Double-click the title of the button to select the text.

Some Finishing Touches

Currency Converter's interface is almost complete. You've probably noticed that the final interface for Currency Converter (shown on the first page of this chapter) has a decorative line between the text fields and the button. This line is easy to make.

1 Create a horizontal decorative line.

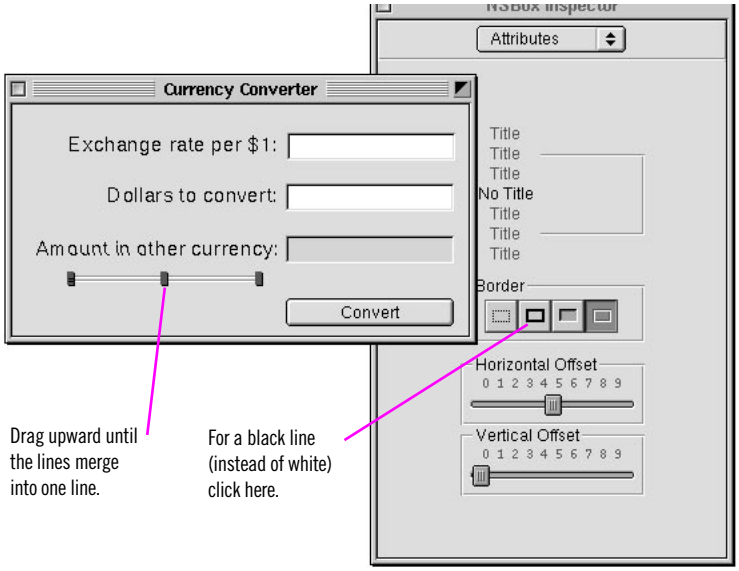
Drag a box object from the Views palette onto the interface.

Bring up the Attributes display for the box (Control-1), select No Title, and set the Vertical Offset to zero.

Drag the bottom-middle resize handle of the box upward until the horizontal lines meet.

Position the line above the button.

Drag the end points of the line until the line extends across the window.



Another finishing touch you might make is to align the text fields and labels in neat rows and columns. Interface Builder gives you several ways to align selected objects precisely on a window:

- Pressing arrow keys (with the grid off, the selected objects move one pixel)
- Using a reference object to put selected objects in rows and columns
- Specifying origin points in the Size display of the Inspector panel
- Using a grid (see side bar below)

For Currency Converter, use the columns-and-rows technique.

1 Align the text fields and labels in rows and columns.

Select the three text fields and choose Format ► Align ► Make Column.

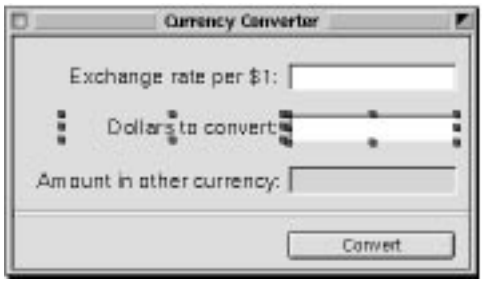
Select the first text field and its label and choose Format ► Align ► Make Row.

Repeat the last step for the second and third text fields and their labels.

COLUMNS

First select the object whose vertical position the other objects should adopt (the reference object).

Shift-click the other objects to include them in the selection. Making a column evens the spacing between objects in the selection.



ROWS

When you make a row, the selected objects rest on a common horizontal baseline.

The final step in composing the Currency Converter interface has more to do with behavior than appearance. You want the user to be able to tab from the first editable field to the second, and back again to the first.

How does this happen? Objects such as windows and views can acquire a temporary status called *first responder*. The first responder is the object on the window that is the current focus of keyboard events. All objects inheriting

Aligning on a Grid

You can align objects on a window by imposing a grid on the window. When you move objects in this grid, they "snap" to the nearest grid intersection like nails to a magnet. You set the edges of alignment and the spacing of the grid (in pixels) in the Alignment panel. Choose Format ►Align ► Alignment to display this panel.

Be sure the grid is turned on before you move objects (Format ►Align ► Turn Grid On).

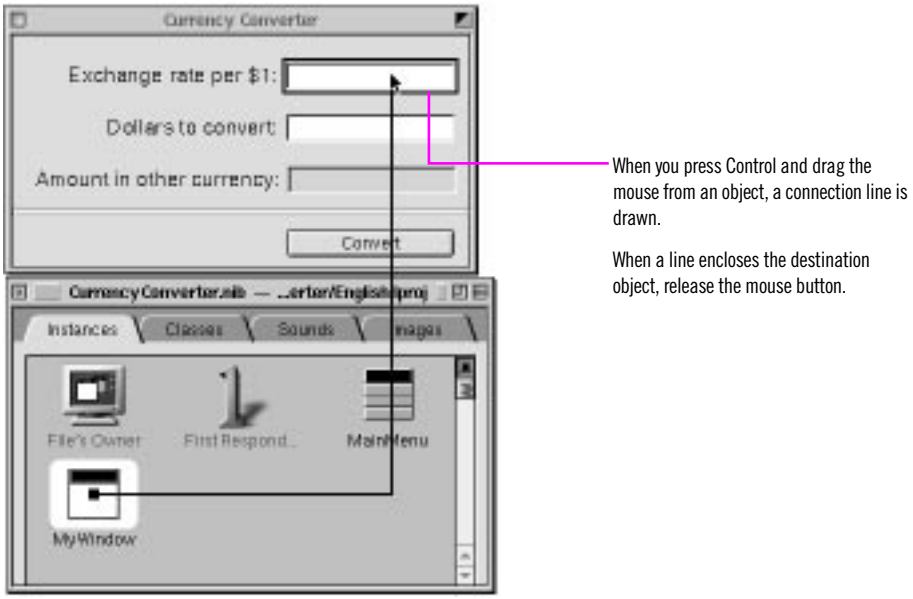
You can move selected user-interface objects in Interface Builder by pressing an arrow key. When the grid is turned on, the unit of movement is whatever the grid is set to (in pixels). When the grid is turned off, the unit of movement is one pixel.

from NSWindow have an outlet named **initialFirstResponder** for designating the first responder when the window is first opened.

1 Enable tabbing between text fields.

Select the window icon in the nib file window.

Control-drag a connection line from the icon to the first text field ("Exchange Rate per $1").

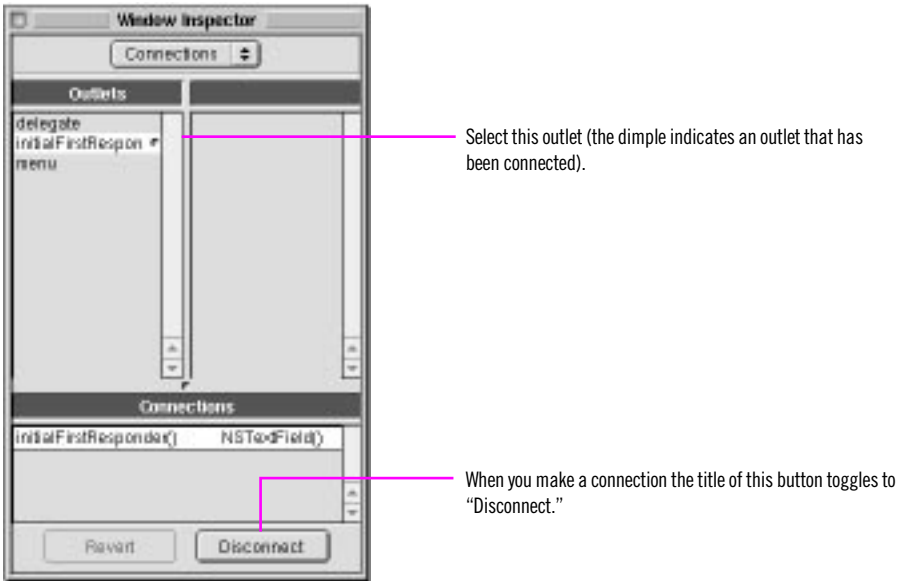


Modifier keys (such as Control) may vary by platform. You can customize some key bindings to suit your habits. See the on-line *Programming Topics* for more on custom key bindings.

When you make a visual connection such as this, Interface Builder brings up the Connections display of the Inspector panel:

In the Connections display of the Window inspector (which appears automatically), select **initialFirstResponder**.

Click Connect.



View objects on Interface Builder's palettes have an outlet named **nextKeyView** for designating the next object to become first responder. If the object is a text field, and the Send Action on Enter attribute is checked in Interface Builder, the field receives keyboard events when users press the Tab key. Since the default tabbing order follows the position in the view hierarchy, if you want a different order you must connect fields through the **nextKeyView** variable.

Select the first text field.

In the Attributes display of the inspector, check the Send Action on Enter switch.

Control-drag a connection line from it to the second text field.

In the Inspector panel (Connections display) select **nextKeyView** and click Connect.

Repeat the same procedure, going from the second text field to button.

Repeat again, this time going from the button to the first text field.

Don't connect the **nextKeyView** outlet of the "Amount in Other Currency" field; this field is not supposed to be editable.

The **initialFirstResponder** and **nextKeyView** variables are *outlets*. An outlet is the identifier of an object that another object stores as an instance variable. Outlets enable communication between objects. See page 40 for more information on outlets.

1 Test the interface.

Choose Document ► Save to save the interface to the nib file.

Choose Document ► Test Interface.

Try various operations in the interface (see suggestions on the following page).

When finished, choose Exit from the File menu.

The CurrencyConverter interface is now complete. Interface Builder lets you test an interface without having to write one line of code.

An OpenStep Application — What You Get "For Free"

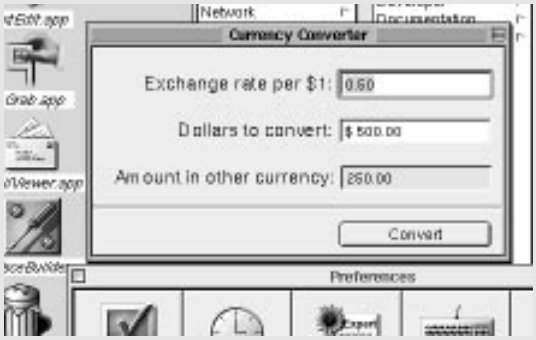
The simplest OpenStep application, even one without a line of code added to it, includes a wealth of features that you get "for free." You do not have to program these features yourself. You can see this when you test an interface in Interface Builder.

To enter test mode, choose Test Interface from the Document menu. Interface Builder simulates how your application (in this case, Currency Converter) would run, minus the behavior added by custom classes. Go ahead and try things out: move your windows, type in fields, click buttons.

Application and Window Behavior

In test mode Currency Converter behaves almost like any other application on the screen. Click elsewhere on the screen, and Currency Converter is deactivated, becoming totally or partially obscured by the windows of other applications.

.



Reactivate Currency Converter by clicking on its window or by double-clicking its icon (the default terminal icon) in the workspace. Then move the window around by its title bar. Here are some other tests you can make:

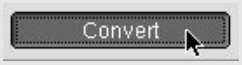
- Click the Edit menu. Its items appear and disappear when you release the mouse button, as with any application menu.
- Click the miniaturize button or choose the Hide command. Double-click the icon to get the application back.
- Click the close button, and the Currency Converter window disappears. (Choose Quit from the main menu and re-enter test mode to get the window back.)

If we had configured Currency Converter's window in Interface Builder to retain the resize bar, we could also resize it now. We could also have set the auto-resizing attributes of the window and its views so that the window's objects would resize proportionally to the resized window or would retain their initial size (see Interface Builder Help for details on auto-resizing).

Controls and Text

The buttons and text fields of Currency Converter come with many built-in behaviors. Click the Convert button. Notice how the button is highlighted momentarily.

.



If you had buttons of a different style, such as option buttons, they would also respond in characteristic ways to mouse clicks.

Now click in one of the text fields. See how the cursor blinks in place. Type some text and select it. Use the commands in the Edit menu to copy it and paste it in the other text field.

Do you recall the **nextKeyView** connections you made between Currency Converter's text fields? Insert the cursor in a text field, press the Tab key and watch the cursor jump from field to field.

When You Add Menu Commands

Interface Builder gives every new application a default main menu that includes the Info, Edit, Window, and Services menus. Some of these menus, such as Info, contain ready-made sets of commands. For example, with the Services menu (whose items are added by other applications at run time) you can communicate with other OpenStep applications, and with the Windows menu you can manage your application's windows.

Currency Converter needs only a few commands: the Quit and Hide commands and the Edit menu's Copy, Cut, and Paste commands. You can delete the unwanted commands if you wish. However, you could also add new ones and get "free" behavior. An application designed in Interface Builder can acquire extra functionality with the simple addition of a menu or menu command, *without* the need for compilation. For example:

- The Font submenu adds behavior for applying fonts to text in NSText objects, like the one in the scroll view object in the DataViews palette. Your application gets the Font panel and a font manager "for free."
- The Text submenu allows you to align text anywhere there is editable text, and to display a ruler in the NSText object for tabbing, indentation, and alignment.

Many objects that display text or images can print their contents as PostScript data. Later you'll learn how to add the Print menu command and have it invoke this capability.

See page 72 for an example of customizing OpenStep menus.

An OpenStep Application — The Possibilities

An OpenStep application can do an impressive range of things without a formidable programming effort on your part.

Document Management

Many applications create and manage repeatable, semi-autonomous objects called *documents*. Documents contain discrete sets of information and support the entry and maintenance of that information. A word-processing document is a typical example. The application coordinates with the user and communicates with its documents to create, open, save, close, and otherwise manage them.

The final tutorial in this book describes how to create an application based on a multi-document architecture.

File Management

An application can use the Open panel, which is created and managed by the Application Kit, to help the user locate files in the file system and open them. It can also use the Save panel to save information in files. OpenStep also provides classes for managing files in the file system (creating, comparing, copying, moving, and so forth) and for managing user defaults.

Communicating With Other Applications

OpenStep gives an application several ways to exchange information with other applications:

- **Pasteboard**: The pasteboard is a global facility for sharing information among applications. Applications can use the pasteboard to hold data that the user has cut or copied and may paste into another application.
- **Services**: Any application can access the services provided by another application, based on the type of selected data (such as text). An application can also provide services to other applications such as encryption, language translation, or record-fetching.
- **Drag-and-drop**: If your application implements the proper protocol, users can drag objects to and from the interfaces of other applications.

Custom Drawing and Animation

OpenStep lets you create your own custom views that draw their own content and respond to user actions. To assist you in this, OpenStep provides image-compositing and event-handling API as well as PostScript operators, operator functions, and client library functions.

Localization

OpenStep provides API and tool support for localizing the strings, images, sounds, and nib files that are part of an application

Editing Support

You can get several panels (and associated functionality) when you add certain menus to your application's menu bar in Interface Builder. These "add-ons" include the Font panel (and font management), the Color panel (and color management), and, although it's not a panel, the text ruler and the tabbing and indentation capabilities the Text menu brings with it.

Formatter classes enable your application to format numbers, dates, and other types of field values. Support for validating the contents of fields is also available.

Printing

With just a simple Interface Builder procedure, OpenStep automates simple printing of views that contain text or graphics. When a user clicks the control, an appropriate panel helps to configure the print process. The output is WYSIWYG.

Several Application Kit classes give you greater control over the printing of documents and forms, including features such as pagination and page orientation.

Help

You can create context-sensitive help for your application using Interface Builder, Project Builder, and an RTF text editor (such as TextEdit). If the user clicks an object on the application's interface while pressing a Help key, a small window containing concise information on the object is displayed. Your application can also incorporate Tool Tips—short descriptions that appear when the mouse pointer hovers over an object on the interface—and comprehensive Help in any format (for example, HTML).

Plug and Play

You can design some applications so that users can incorporate new modules later on. For example, a drawing program could have a tools palette: pencil, brush, eraser, and so on. You could create a new tool and have users install it. When the application is next started, this tool appears in the palette.

Defining the Classes of Currency Converter

Interface Builder is a versatile tool for application developers. It enables you not only to compose the application's graphical user interface, but it gives you a way to define much of the *programmatic* interface of the application's classes and to connect the objects eventually created from those classes.

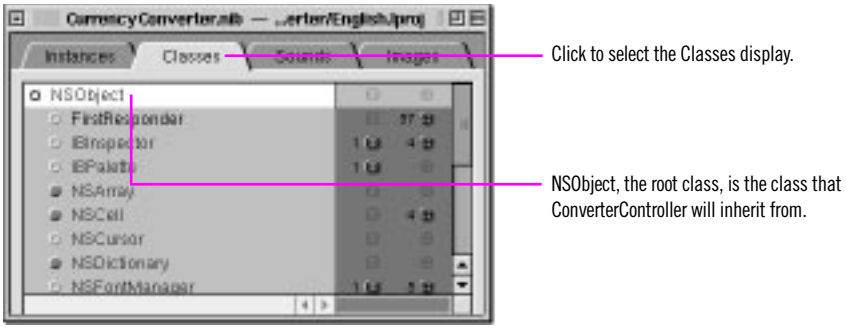
You must go to the Classes display of the nib file window to define a class. Once there, the first thing you must do is select the *superclass*, the class your new *subclass* will inherit from. Let's start with the ConverterController class.

1 Specify a subclass.

Go to the Classes display of the nib file window.

Select NSObject, the superclass of your custom classes.

Choose Classes ► Subclass.



After you choose the Subclass command, "MyNSObject" appears under "NSObject" highlighted.

Enter the name of the subclass: "ConverterController."

Press Return.

Now your class is established in the hierarchy of classes within the nib file. Next, specify the paths for messages travelling between the ConverterController object and other objects. In Interface Builder you specify these paths as *outlets* and *actions*.

Before You Go On

Here's some basic terminology:

Outlet An object held as an instance variable and typed as **id**. Objects in applications often hold outlets as part of their data so they can send messages to the objects referenced by the outlets. An outlet helps your program to track or manipulate something in the interface.

id The generic (or dynamic) type of objects (technically the address of an object).

Action Refers both to a message sent to an object when the user clicks a button or manipulates some other control object and to the method that is invoked.

Control object A user-interface object (a device) with which users can interact to affect events in the application. Control objects include buttons, text fields, forms, sliders, and browsers. All control objects inherit from NSControl.

See *Paths for Object Communication: Outlets, Targets, and Actions* on page 40. for a more detailed description of outlets and actions. See page 107 for more on control objects and their relation to cells and formatters.

Class Versus Object

To newcomers to the subject, explanations of object-oriented programming might seem to use the terms "object" and "class" interchangeably. Are an object and a class the same thing? And if not, how are they different? How are they related?

An object and a class are both programmatic units. They are closely related, but serve quite different purposes in a program.

First, classes provide a taxonomy of objects, a useful way of categorizing them. Just as you can say a particular tree is a pine tree, you can identify a particular object by its class. You can thereby know its purpose and what messages you can send it. In other words, a class describes the type of an object.

Second, you use classes to generate *instances*—or objects. Classes define the data structures and behavior of their instances, and at run time create and initialize these instances. In a sense, a class is like a factory, stamping out instances of itself when requested.

What especially differentiates a class from its instance is data. An instance has its own unique set of data but its class, strictly speaking, does not. The class defines the structure of the data its instances will have, but only instances can hold data.

A class, on the other hand, implements the behavior of **all** of its instances in a running program. The donut symbol used to represent objects is a bit misleading here, because it suggests that each object contains its own copy of code. This is fortunately not the case; instead of being duplicated, this code is shared among all current instances in the program.

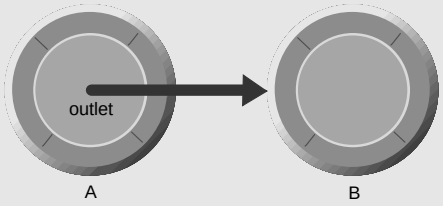
Implicit in the notion of a taxonomy is *inheritance*, a key property of classes. Classes exist in a hierarchical relationship to one another, with a *subclass* inheriting behavior and data structures from its *superclass*, which in turn inherits from its superclass.

See the appendix, "Object-Oriented Programming," for more on these and other aspects of classes.

Paths for Object Communication: Outlets, Targets, and Actions

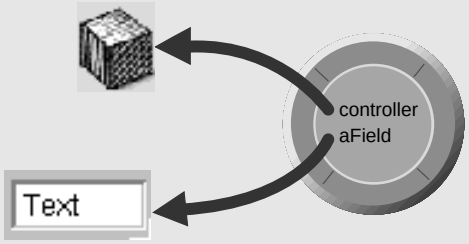
Outlets

An outlet is an instance variable that identifies an object.



You can communicate with other objects in an application by sending messages to outlets.

An outlet can reference any object in an application: user-interface objects such as text fields and buttons, windows and panels, instances of custom classes, and even the application object itself.



Outlets are declared as:

```
id  variableName;
```

You can use **id** as the type for any object; objects with **id** as their type are *dynamically typed*, meaning that the class of the object is determined at run time. You can statically type an object as a pointer to a class name, and you can declare these objects as instance variables. But statically typed objects are typically not outlets. What distinguishes outlets is their relationship to Interface Builder.

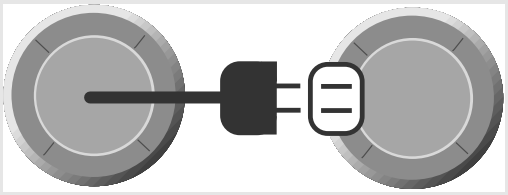
Interface Builder can "recognize" outlets in code by their declarations, and it can initialize outlets. You usually set an outlet's value in Interface Builder by drawing connection lines between objects. There are ways other than outlets to reference objects in an application, but outlets and Interface Builder's facility for initializing them are a great convenience.

When You Make a Connection in Interface Builder

As with any instance variable, outlets must be initialized at run time to some reasonable value—in this case, an object's identifier (**id** value). Because of Interface Builder, an application can initialize outlets when it loads a nib file.

When you make a connection in Interface Builder, a special connector object holds information on the source and destination objects of the connection. (The source object is the object with the outlet.) This connector object is then stored in the nib file. When a nib file is loaded, the application uses the connector object to set the source object's outlet to the **id** value of the destination object.

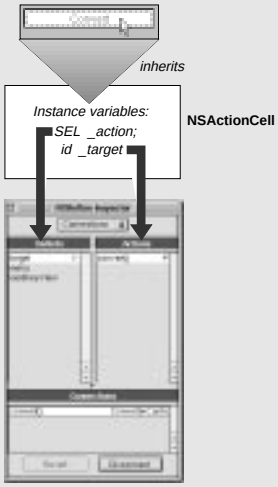
It might help to understand connections by imagining an electrical outlet (as used in the Classes display of the nib file window) embedded in the destination object. Also picture an electrical cord extending from the outlet in the source object. Before the connection is made the cord is unplugged and the value of the outlet is undefined; after the connection is made (the cord is plugged in), the **id** value of the destination object is assigned to the source object's outlet



Target/Action in Interface Builder—What's Going On

As you'll soon find out, you can view (and complete) target/action connections in Interface Builder's Connections inspector. This inspector is easy to use, but the relation of target and action in it might not be apparent. First, **target** *is an outlet of a cell object* that identifies the recipient of an action message. Well (you say) what's a cell object and what does it have to do with a button?

One or more cell objects are always associated with a control object (that is, an object inheriting from NSControl, such as a button). Control objects "drive" the invocation of action methods, but they get the target and action from a cell. NSActionCell defines the target and action outlets, and most kinds of cells in the Application Kit inherit these outlets



For example, when a user clicks the Convert button of Currency Converter, the button gets the required information from its cell and sends the message **convert:** to the target outlet, which is an instance of your custom class ConverterController.

In the Actions column of the Connections inspector are all action methods defined by the class of the target object and known by Interface Builder. Interface Builder identifies action methods because their declarations follow the syntax:

```
- (void)doThis:(id)sender;
```

It looks in particular for the argument **sender**.

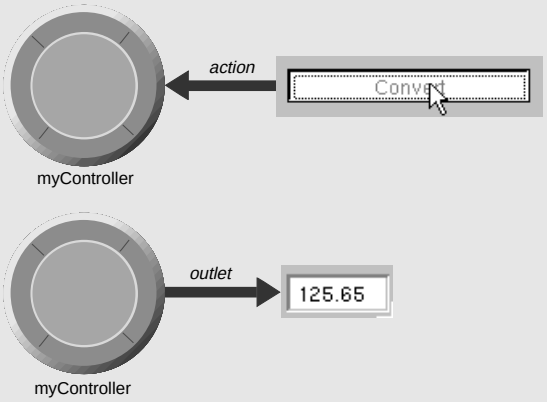
Which Direction to Connect?

Usually the outlets and actions that you connect belong to a custom subclass of NSObject. For these occasions, you need only follow a couple simple rules to know which way to draw a connection line in Interface Builder:

- To make an action connection, draw a line *to* the custom instance *from* a control object in the user interface, such as a button or a text field.
- To make an outlet connection, draw a line *from* the custom instance *to* another object in the application.

Another way to clarify connections is to consider who needs to find whom. With outlets, the custom object needs to find some other object, so the connection is from the custom object to the other object. With actions, the control object needs to find the custom object, so the connection is from the control object.

These are only rules of thumb for the common case, and do not apply in all circumstances. For instance, many OpenStep objects have a **delegate** outlet; to connect these, you draw a connection line from the OpenStep object to your custom object.



1 Define your class's outlets.

In the nib file window, click the electrical-outlet icon to the right of the class.

Choose Classes ► Add Outlet.

Type the name of the outlet in place of the selected "myOutlet." Name the first outlet **rateField**.

Press Return.

Repeat the last three steps to define two other outlets:

dollarField
totalField

ConverterController has one action method, **convert:**. When the user clicks the Convert button, a **convert:** message is sent to the target, ConverterController.

1 Define your class's actions.

In the Classes display of the nib file window, click the crosshairs icon.

Choose Classes ► Add Action.

Type the name of the action method, **convert:**.

Press Return.

Before You Go On

Exercise: ConverterController needs to access the text fields of the interface, so you've just provided outlets for that purpose. But ConverterController must also communicate with the Converter class (yet to be defined). To enable this communication, add an outlet named **converter** to ConverterController.

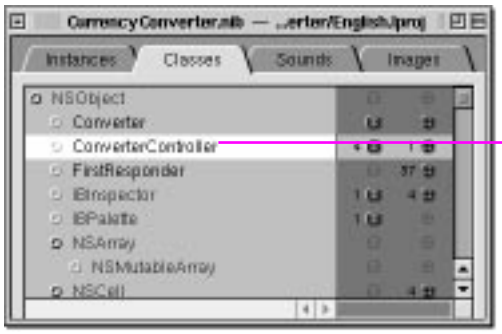
Connecting ConverterController to the Interface

As the final step of defining a class in Interface Builder, you create an instance of your class and connect its outlets and actions.

1 Generate an instance of the class.

In the Classes display, select the ConverterController class.

Choose the Classes ► Instantiate.



Note: The Instantiate command does not generate a true instance of ConverterController but creates a stand-in object used for establishing connections. When the nib file's contents are unarchived, Interface Builder will create true instances of these classes and use the proxy objects to establish the outlet and action connections.

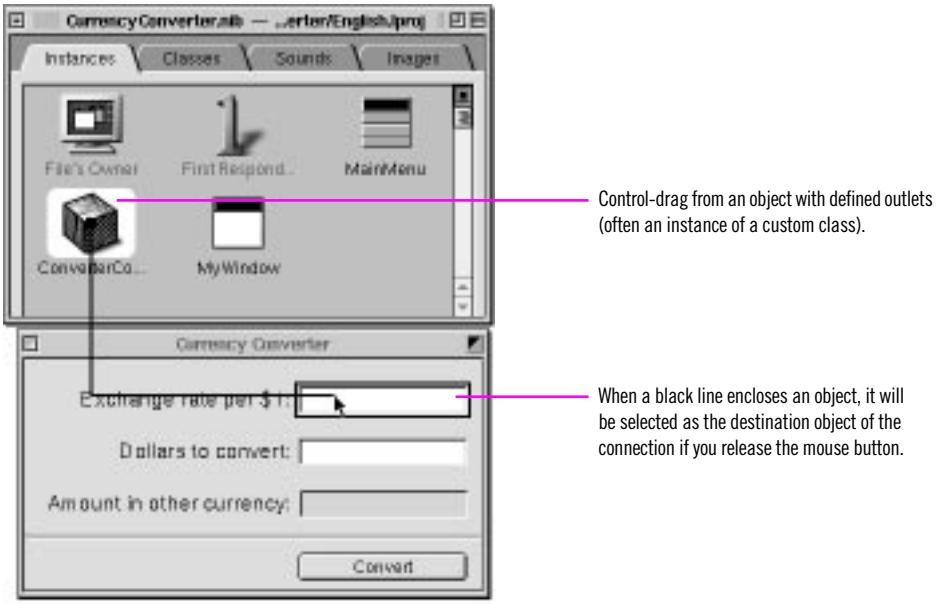
When you instantiate a class (that is, create an instance of it), Interface Builder switches to the Instances display and highlights the new instance, which is named after the class.

Now you can connect this ConverterController object to the user interface. By connecting it to specific objects in the interface, you initialize your outlets. ConverterController will use these outlets to get and set values in the interface.

1 **Connect the custom class to the interface via its outlets.**

In the Instances display of the nib file window, Control-drag a connection line from the ConverterController instance to the first text field.

When the field is outlined in black, release the mouse button.

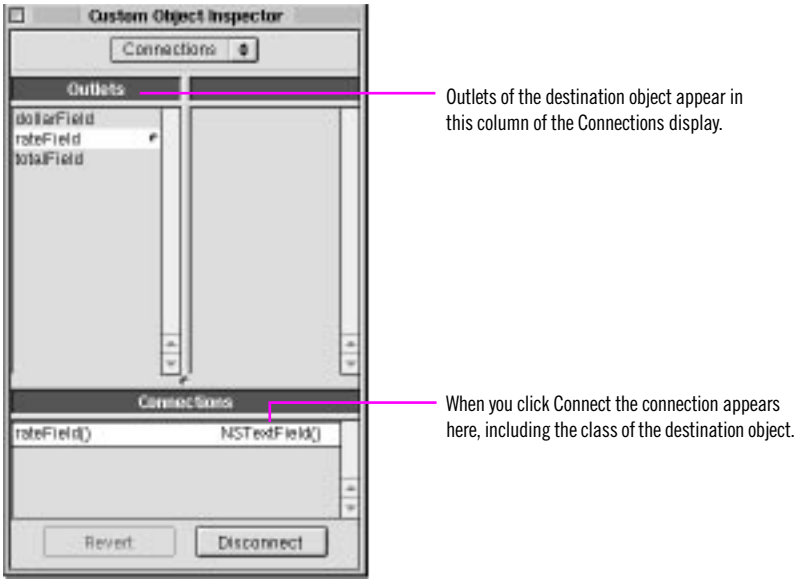


Interface Builder brings up the Connections display of the Inspector panel. This display shows the outlets you have defined for ConverterController.

In the Connections display, select the outlet that corresponds to the first field (**rateField**).

Click the Connect button.

Following the same steps, connect ConverterController's **dollarField** and **totalField** outlets to the appropriate text fields.



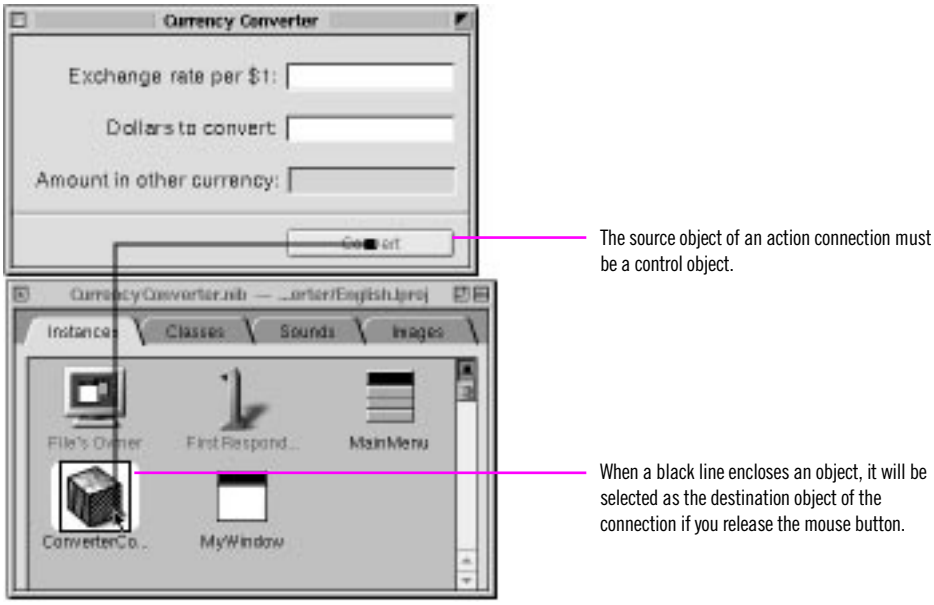
To receive action messages from the user interface—to be notified, for example, when users click a button—you must connect the control objects that emit those messages to CurrencyConverter. The procedure for connecting actions is similar to that for outlets, but with one major difference. When you connect an action, always start the connection line from a *control object* (such as a button, text field, or

form) that sends an action message; you usually end the connection at an instance of your custom class. That instance is the *target* outlet of the control object.

1 Connect the interface's controls to the custom object through the defined actions.

Control-drag a connection line from the Convert button to the ConverterController instance in the nib file window.

When the instance is outlined in black, release the mouse button.



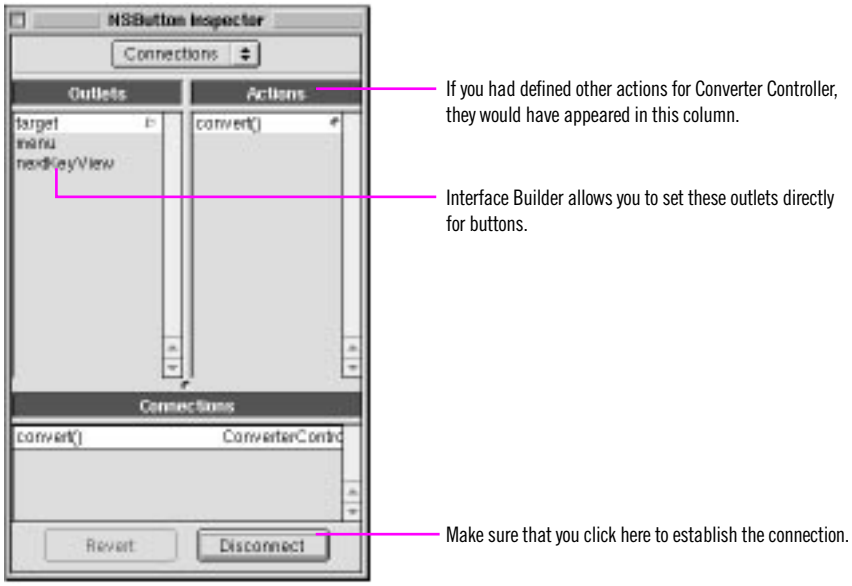
The Connections display of the Inspector panel shows the action methods you have specified for ConverterController.

In the Connections display, make sure **target** in the Outlets column is selected.

Select **convert:** in the Actions column.

Click the Connect button.

Save the CurrencyConverter nib file (Document ▶ Save).



You've finished defining the classes of Currency Converter—almost.

Before You Go On

Exercise: While connecting ConverterController's outlets, you probably noticed that one outlet remains unconnected: **converter**. This outlet identifies an instance of the Converter class in the Currency Converter application, but this instance doesn't exist yet.

Define the Converter class. This should be pretty easy because Converter, as you might recall, is a model class within the Model-View-Controller paradigm. Since instances of this type of class don't communicate directly with the interface, there is no need for outlets or actions. Here are the steps to be completed:

1. In the Classes display, make Converter a subclass of NSObject.

2. Instantiate the Converter class.

3. Make an outlet connection between ConverterController and Converter.

When you are finished, save **CurrencyConverter.nib**.

Optional Exercise

Text fields and action messages: Users can also activate the Convert button by pressing the Return key. In Currency Converter this key event occurs when the cursor is in a text field. Text fields are control objects just as buttons are; when the user presses the Return key and the cursor is in a text field, an action message is sent to a target object if the action is defined and the proper connection is made.

Connect the second text field (that is, the one with the "Dollars to Convert" label) to the **convert:** action method of ConverterController. You won't be disconnecting the prior action connection because multiple control objects in an interface can invoke the same action method.

Optionally, you can connect the second text field to the Convert button via the latter's **performClick:** action method. This method simulates a mouse click on the button and consequently invokes the action method of the button's target.

Implementing the Classes of Currency Converter

Interface Builder generates source code files from the (partial) class definitions you've made. These files are "skeletal," in the sense that they contain little more than essential Objective-C directives and the class-definition information. You'll usually need to supplement these files with your own code.

1 In Interface Builder, generate header and implementation files.

Go to the Classes display of the nib file window.

Select the ConverterController class.

Choose Classes ► Create Files.

Make sure your class is selected before you choose Create Files.

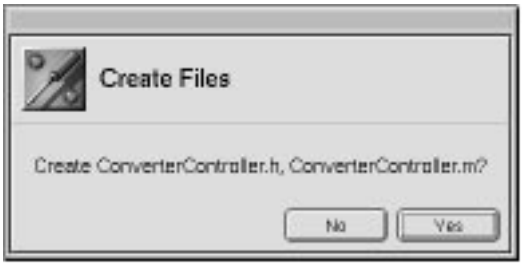
Interface Builder then displays two attention panels, one after the other:

Click Yes in response to a "create files" attention panel.

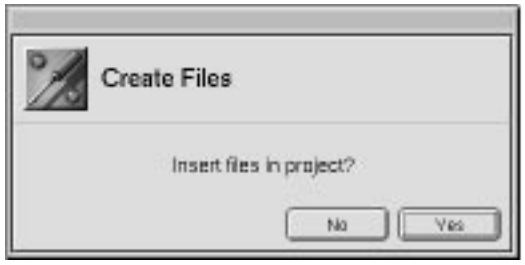
Click Yes in response to an "insert files in project" attention panel.

Repeat for the Converter class.

Save the nib file.



Click Yes to confirm that you want Interface Builder to generate the header and implementation files for your class.



Click Yes to confirm that you want the source code files added to the project. If, for example, you wanted to add the files to another project, you would click No.

Now we leave Interface Builder for this application. You'll complete the application using Project Builder.

Objective-C Quick Reference

The Objective-C language is a superset of ANSI C with special syntax and run-time extensions that make object-oriented programming possible. Objective-C syntax is uncomplicated, but powerful in its simplicity. You can mix standard C and even C++ code with Objective-C code.

The following summarizes some of the more basic aspects of the language. See *Object-Oriented Programming and the Objective-C Language* for complete details. Also, see "Object-Oriented Programming" in the appendix for explanations of terms that are italicized.

Declarations

- Dynamically type objects by declaring them as **id**:

```
id myObject;
```

  Since the class of dynamically typed objects is resolved at run time, you can refer to them in your code without knowing beforehand what class they belong to. Type outlets in this way as well as objects that are likely to be involved in *polymorphism* and *dynamic binding*.

- Statically type objects as a pointer to a class:

```
NSString *mystring;
```

  You statically type objects to obtain better compile-time type checking and to make code easier to understand.

- Declarations of *instance methods* begin with a minus sign (-); *class methods* begin with a plus sign (+):

```
- (NSString *)countryName;
+ (NSDate *)calendarDate;
```

- Put the type of value returned by a method in parentheses between the minus sign (or plus sign) and the beginning of the method name. (See above example.) Methods returning no explicit type are assumed to return **id**. Methods that return nothing should have a return type of **void**.

- Method argument types are in parentheses and go between the argument's *keyword* and the argument itself:

```
- (id)initWithName:(NSString *)name
    andType:(int)type;
```

  Be sure to terminate all declarations with a semicolon.

- By default, the scope of an instance variable is protected, making that variable directly accessible only to objects of the class that declares it or of a subclass of that class. To make an instance variable private (accessible only within the declaring class), insert the **@private** directive before the declaration.

Messages and Method Implementations

- Methods are procedures implemented by a class for its objects (or, in the case of class methods, to provide functionality not tied to a particular instance). Methods can be public or private; public methods are declared in the class's header file (see above). Messages are invocations of an object's method that identify the method by name.

- Message expressions consist of a variable identifying the receiving object followed by the name of the method you want to invoke; enclose the expression in brackets.

```
[anObject doSomethingWithArg:this];
```

  receiver *method to invoke (with possible arguments)*

  As in standard C, terminate statements with a semicolon.

- Messages often get values returned from the invoked method; you must have a variable of the proper type to receive this value on the left side of an assignment.

```
int result = [anObj calcTotal];
```

- You can nest message expressions inside other message expressions. This example gets the window of a form object and makes the returned NSWindow object the receiver of another message.

```
[[form window]
    makeKeyAndOrderFront:self];
```

- A method is structured like a function. After the full declaration of the method comes the body of the implementing code enclosed by braces.

- Use **nil** to specify a null object; this is analogous to a null pointer. Note that some OpenStep methods do not accept **nil** as an argument.

- A method can usefully refer to two implicit identifiers: **self** and **super**. Both identify the object receiving a message, but they affect differently how the method implementation is located: **self** starts the search in the receiver's class whereas **super** starts the search in the receiver's superclass. Thus,

```
[super init];
```

  causes the **init** method of the superclass to be invoked.

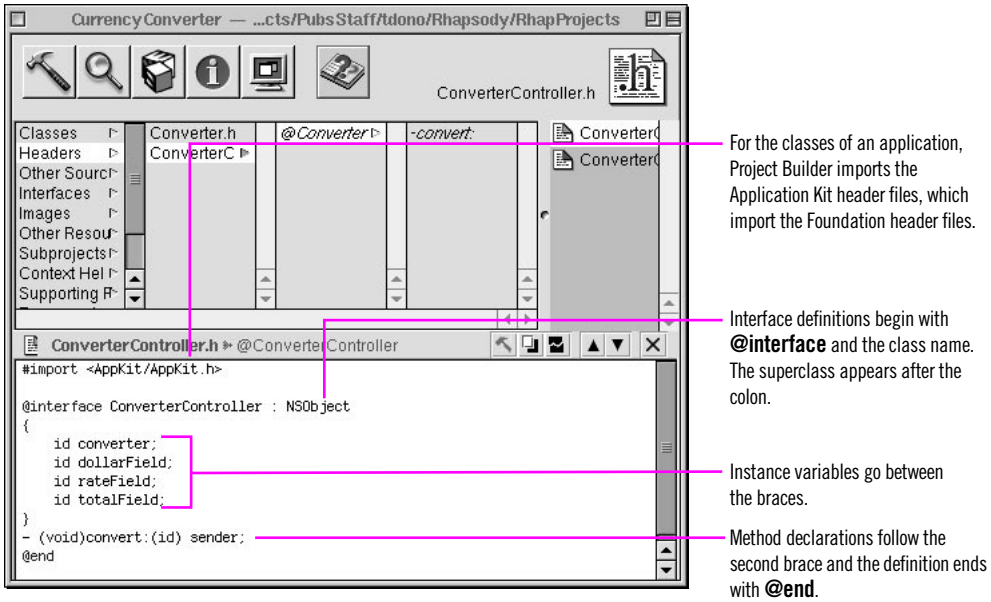
- In methods you can directly access the instance variables of your class's instances. However, *accessor methods* are recommended instead of direct access, except in cases where performance is of paramount importance. Chapter 4, "Travel Advisor Tutorial," describes accessor methods in greater detail.

1 Examine an interface (header) file in Project Builder.

Click Project Builder's main window to activate it.

Select Headers in the project browser.

Select **ConverterController.h**.



You can add instance variables or method declarations to a header file generated by Interface Builder. This is commonly done, but it isn't necessary in ConverterController's case. But we do need to add a method to the Converter class that the ConverterController object can invoke to get the result of the computation. Let's start by declaring the method in **Converter.h**.

1 Add a method declaration.

Select **Converter.h** in the project browser.

Insert a declaration for **convertAmount:byRate:**.

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

@interface Converter:NSObject
{
}
- (float)convertAmount:(float)amt byRate:(float)rate;

@end
```

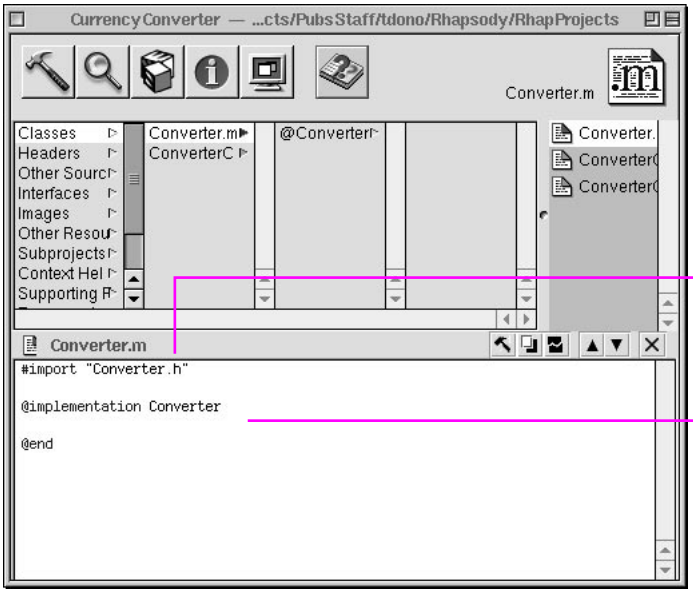
This declaration states that **convertAmount:byRate:** takes two arguments of type **float**, and returns a **float** value. When parts of a method name have colons, such as **convertAmount:** and **byRate:**, they are *keywords* which introduce arguments. (These are keywords in a sense different from keywords in the C language.) All declarations of instance methods begin with a dash (-), followed by a space.

Now you need to update both implementation files. First examine **Converter.m**.

1 Examine an implementation file.

Click Classes in the project browser.

Select **Converter.m**.



For this class, implement the method declared in **Converter.h**. Between **@implementation Converter** and **@end** add the following code:

1 Implement the classes.

Type the code at right between **@implementation** and **@end** in **Converter.m**.

```
- (float)convertAmount:(float)amt byRate:(float)rate
{
    return (amt * rate);
}
```

The method simply multiplies the two arguments and returns the result. Simple enough. Next update the "empty" implementation of the **convert:** method that Interface Builder generated.

Select **ConverterController.m** in the project browser.

Update the **convert:** method as shown by the example.

Import **Converter.h**.

```
- (void)convert:(id)sender
{
    float rate, amt, total=0.0;

    amt = [dollarField floatValue];                          A
    rate = [rateField floatValue];
    total = [converter convertAmount:amt byRate:rate];       B
    [totalField setFloatValue:total];                        C
    [rateField selectText:self];                             D
}
```

The **convert:** method does the following:

A Gets the floating-point values typed into the rate and dollar-amount fields.

B Invokes the **convertAmount:byRate:** method and gets the returned value.

C Uses **setFloatValue:** to write the returned value in the Amount in Other Currency text field (**totalField**).

D Sends **selectText:** to the rate field; this selects any text in the field or, if there is no text, inserts the cursor so the user can begin another calculation.

Be sure to import **Converter.h** (that is, include the directive **#import "Converter.h"**). ConverterController invokes a method defined in the Converter class, so it needs to be aware of the method's declaration.

Before You Go On

Each line of the **convert:** method shown above, excluding the declaration of **float**s, is a message. The "word" on the left side of a message expression identifies the object receiving the message (called the "receiver"). These objects are identified by the outlets you defined and connected. After the receiver comes the name of the method that the sending object (called the "sender") wants to invoke. Messages often result in values being returned; in the above example, the local variables **rate**, **amt**, and **total** hold these values.

Before you build the project, add a small bit of code to **ConverterController.m** that will make life a little easier for your users. When the application starts up, you want Currency Converter's window to be selected and the cursor to be in the Exchange Rate per $1 field. We can do this only after the nib file is unarchived, which establishes the connection to the text field **rateField**. To enable set-up operations like this, **awakeFromNib** is sent to all objects when unarchiving concludes. Implement this method to take appropriate action.

1 Implement the awakeFromNib method.

Type the code shown at right.

Save all code files.

```
- (void)awakeFromNib
{
    [rateField selectText:self];                          A
    [[rateField window] makeKeyAndOrderFront:self];       B
}
```

A You've seen the **selectText:** message before, in the **convert:** implementation; it selects the text in the text field that receives the message, inserting the cursor if there is no text.

B The **makeKeyAndOrderFront:** message does as it says: It makes the receiving window the key window and puts it before all other windows on the screen. This message also *nests* another message, **[rateField window]**. This message returns the window to which the text field belongs, and the **makeKeyAndOrderFront:** method is then sent to this returned object.

What Happens When You Build an Application

By clicking the Build button in Project Builder, you run the build tool. By default, the build tool is **make**, but it can be any build utility that you specify as a project default in Project Builder. The build tool coordinates the compilation and linking process that results in an executable file. It also performs other tasks needed to build an application.

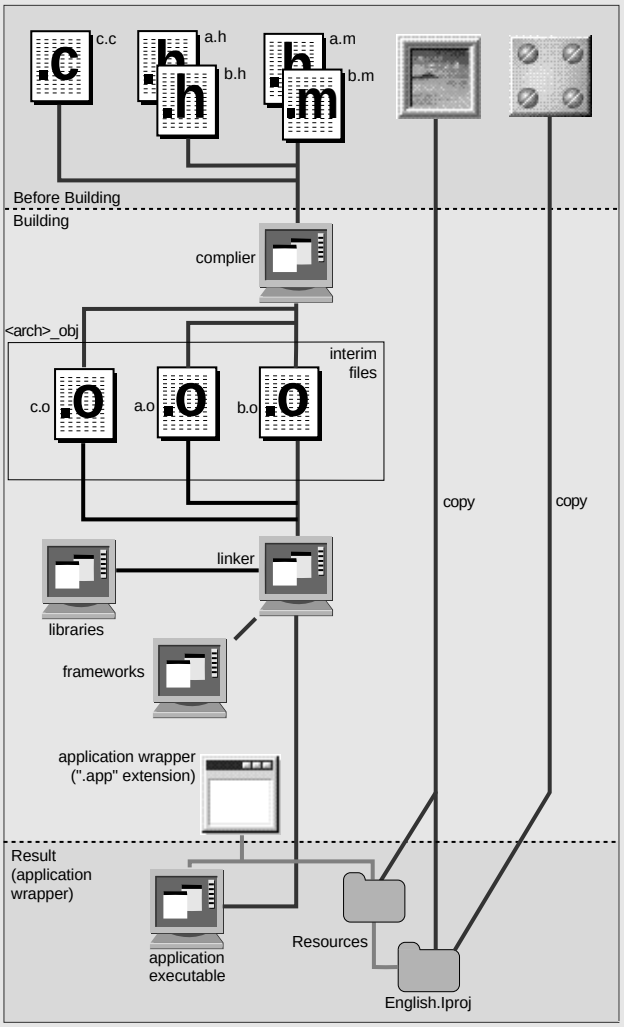
The build tool manages and updates files based on the dependencies and other information specified in the project's makefiles. Every application project has three makefiles: **Makefile**, **Makefile.preamble**, and **Makefile.postamble**. **Makefile** is maintained by Project Builder—don't edit it directly—but you can modify the other two to customize your build.

The build tool invokes the compiler, passing it the source code files of the project. Compilation of these files (Objective-C, C++, and standard C) produces machine-readable object files for the architecture or architectures specified for the build. The build utility puts these files in an architecture-specific subdirectory of **dynamic_obj**.

In the linking phase of the build, the build tool executes the linker, passing it the libraries and frameworks to link against the object files. Frameworks and libraries contain precompiled code that can be used by any application. Linking integrates the code in libraries, frameworks, and object files to produce the application executable file.

The build tool also copies nib files, sound, images, and other resources from the project to the appropriate localized or non-localized locations in the application wrapper.

An application wrapper on Windows is a directory with an extension of ".app". It contains the application executable and the resources needed by that executable.



Building the Currency Converter Project

The Build process in Project Builder compiles and links the application guided by the information stored in the project's makefiles. You must begin builds from the Project Build panel.

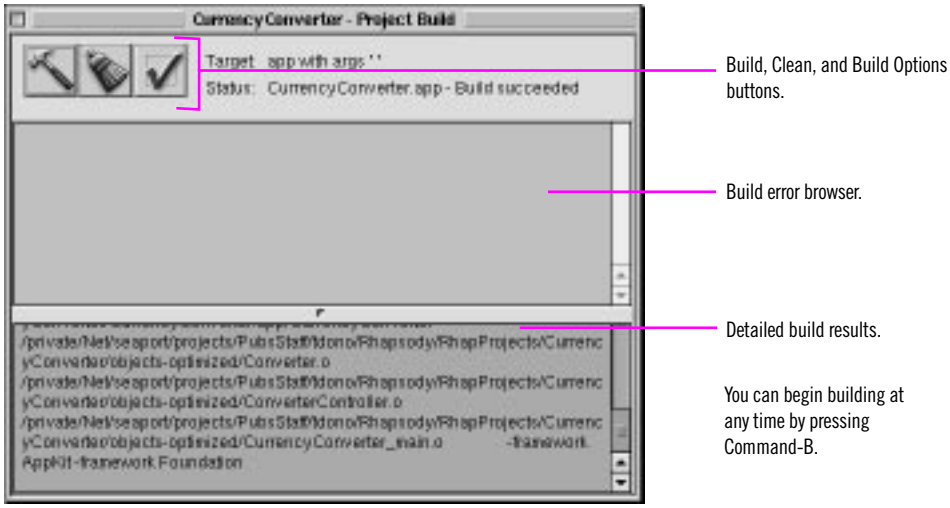
1 Build the project.

Save source code files and any changes to the project.

Click the Build button on the main window (icon shown at right).

Click the Build button on the Project Build panel (same icon).

When you click the Build button on the main window, the Project Build panel is displayed.



When you click the Build button on the Project Build panel, the build process begins; Project Builder logs the build's progress in the lower split view. When Project Builder finishes—and encounters no errors along the way—it displays "Build succeeded."

You don't have to maintain makefiles in Project Builder. It updates **Makefile** according to the variables specified through its user interface. You can customize the build process by modifying the **Makefile.preamble** and **Makefile.postamble** files. For more information on customizing these files, see the on-line Help for Project Builder and Interface Builder.

Of course, rare is the project that is flawless from the start. Project Builder is likely to catch some errors when you first build your project. To see the error-checking features of Project Builder, introduce a mistake into the code.

1 **Build the project after correcting errors.**

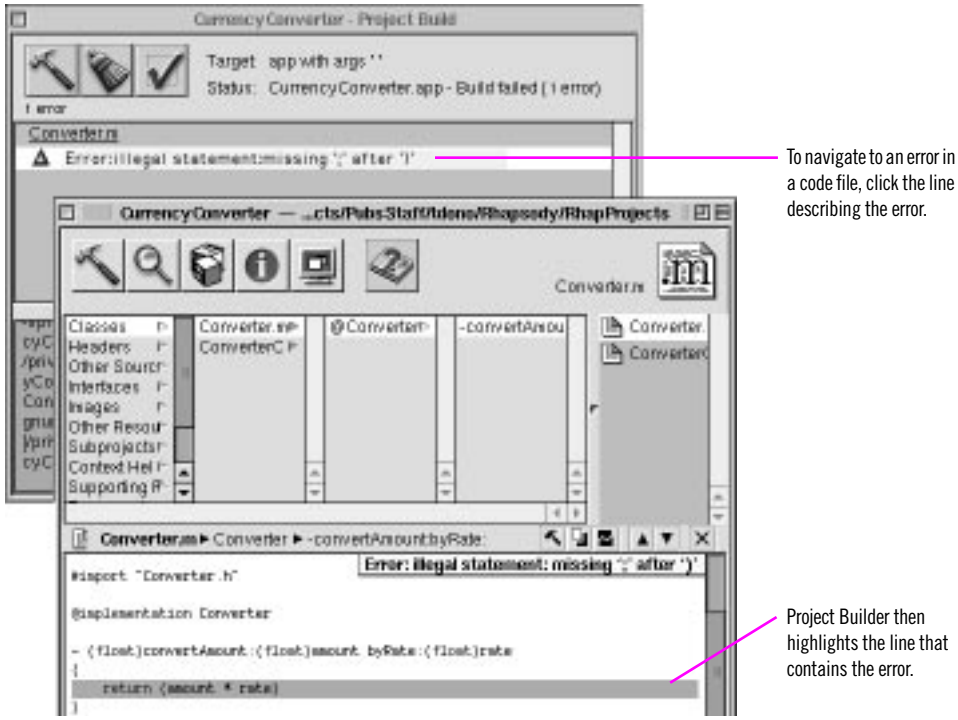
Delete a semicolon in the code, creating an error.

Click the Build button on the Project Build panel.

Click the error-notification line that appears in the build error browser (upper split view).

Fix the error in the code.

Re-build.



You can use Project Builder's graphical debugger or **gdb** to track bugs down. See "Using the Graphical Debugger" on page 110 for an overview of the graphical debugger.

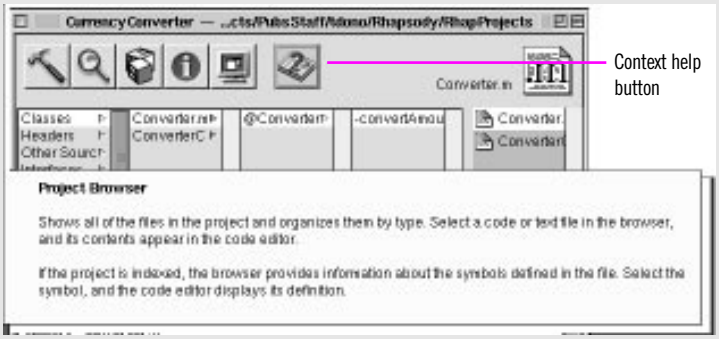
Where To Go For Help

Help on Development Tools

Project Builder and Interface Builder provide context-sensitive help on the details of their use. To activate context-sensitive help, Help-click a control, field, menu command, or other areas of the application. A small window appears that briefly describes the selected object. (The next click dismisses the window.)

These applications also provide Tool Tips, short descriptions of parts of the interface that briefly appear when the mouse pointer hovers over those areas. You can turn Tool Tips off.

Project Builder and Interface Builder also provide comprehensive task-based Help, accessible from the Apple menu.



Help on APIs

Project Builder gives you several ways to get information on OpenStep APIs when you're developing an application.

Project Find. The Project Find panel allows you to search for definitions of, and references to, classes, methods, functions, constants, and other symbols in your project. Since it is based on project indexing, searching is quick and thorough and leads directly to the relevant code. Help for Interface Builder or Project Builder contains full task-based instructions for using Project Find.

Reference Documentation Lookup. If the results of a search using Project Find include OpenStep symbols, you can easily get related reference documentation that describes that symbol. See "Finding Information Within Your Project" on page 96 for instructions on the use of this feature.

Frameworks. Under Frameworks in the project browser, you can browse the header files and documentation related to OpenStep frameworks within Project Builder. The Application Kit and Foundation frameworks always are included by default for application projects.

Rhapsody Technical Documentation

Most Rhapsody programming documentation is located in **/System/Documentation/Developer**.

Reference

- API reference documentation. Includes specifications of classes, protocols, functions, types, and constants. This documentation is located in the appropriate Rhapsody frameworks, except for information that is common to all frameworks (**Reference**).
- Development tools reference. Covers the compiler, the debugger, and other tools (**Reference/DevToolsRef**).

Tasks and Concepts

- *Discovering OpenStep: A Developer Tutorial* (this manual).
- "Object-Oriented Programming and the Objective-C Language": an on-line reference and users guide for Objective-C.
- "Topics in OpenStep Programming" contains concepts and programming procedures.

Run Currency Converter

Congratulations. You've just created your first OpenStep application. Find **CurrencyConverter.app** in the Workspace, launch it, and try it out. Enter some rates and dollar amounts and click Convert. Also, select the text in a field and choose the Services menu; this menu now lists the other applications that can do something with the selected text.

Of course, the more complex an application is, the more thoroughly you will need to test it. You might discover errors or shortcomings that necessitate a change in overall design, in the interface, in a custom class definition, or in the implementation of methods and functions.

Although it's a simple application, Currency Converter still introduced you to many of the concepts, tools, and skills you'll need to develop OpenStep applications. Let's review what you've learned:

- Composing a graphical user interface (GUI) with Interface Builder
- Testing the interface
- Designing an application using the Model-View-Controller paradigm
- Specifying a class's outlets and actions
- Connecting the class instance to the interface via its outlets and actions
- Class implementation basics
- Building an application and error resolution

Optional Exercise

Nesting Messages: You can nest message expressions; in other words, you can use the value returned by a message as the receiver of another message or as a message argument. It is thus possible to rewrite the first three messages of the ConverterController's **convert:** method as one statement:

```
total = [converter convertAmount:[dollarsField floatValue]
        byRate:[rateField floatValue]];
```

It is possible to go even further. Try to incorporate the fourth message (**[totalField setFloatValue:total]**) of the **convert:** method into the above statement.