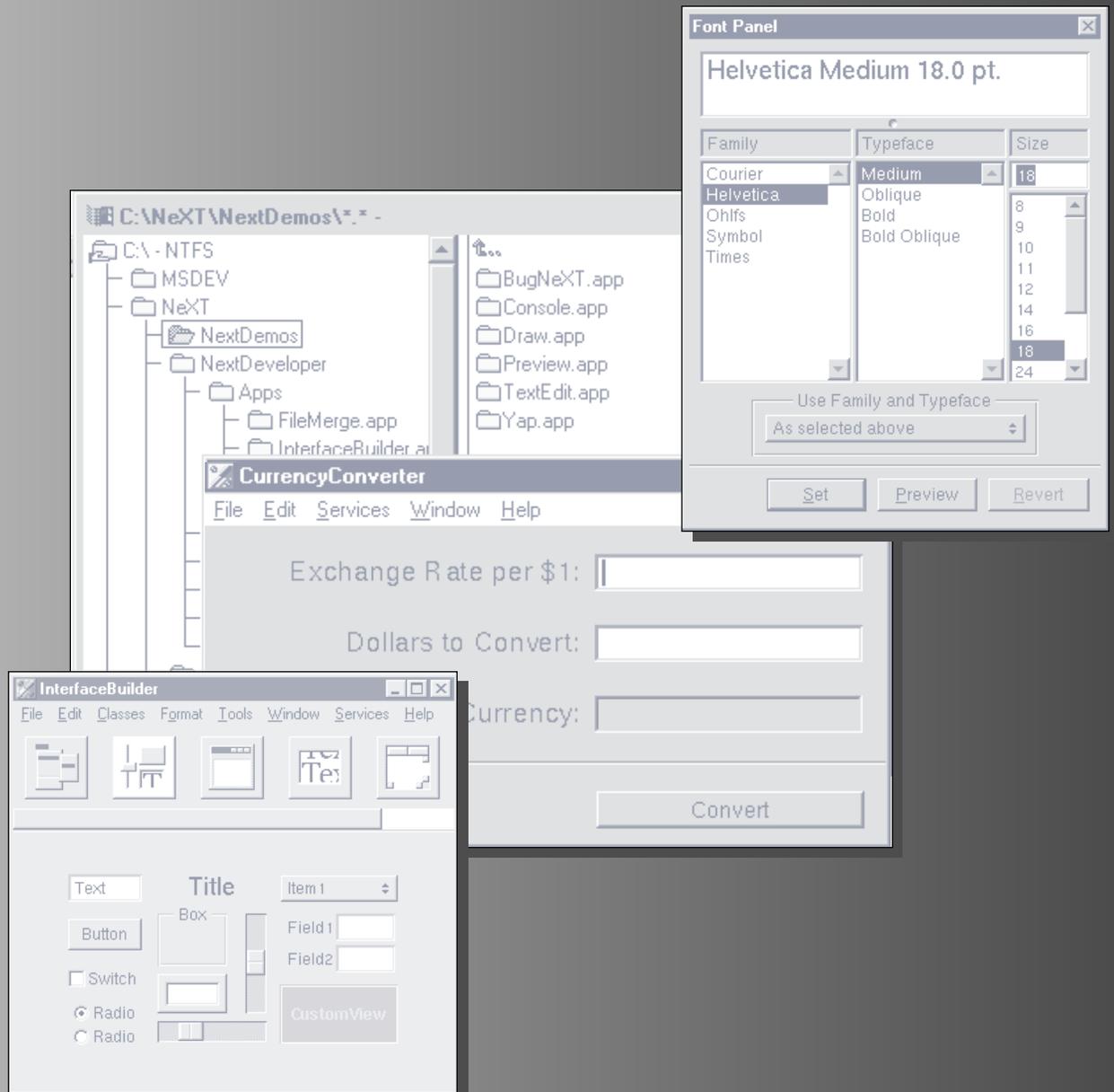


Currency Converter Tutorial Chapter 2



2

Chapter 2 Currency Converter Tutorial

Sections

Creating the Currency Converter Project

Creating the Currency Converter Interface

Designing the Currency Converter Application

Defining the Classes of Currency Converter

Implementing the Classes of Currency Converter

Building the Currency Converter Project

Run Currency Converter

Concepts

Project Indexing

A Window in OpenStep

An OpenStep Application — What You Get For Free

An OpenStep Application — the Possibilities

Why an Object is Like a Jelly Donut

The Model-View-Controller Paradigm

Class Versus Object

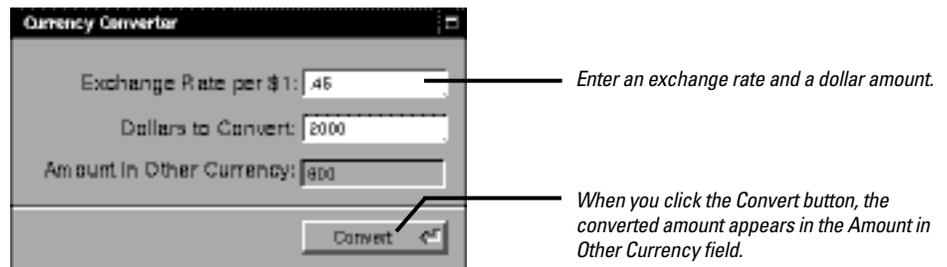
Paths for Object Communication: Outlets, Targets, and Actions

What Happens When You Build an Application

Where to Go For Help

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. Here's what it looks like:



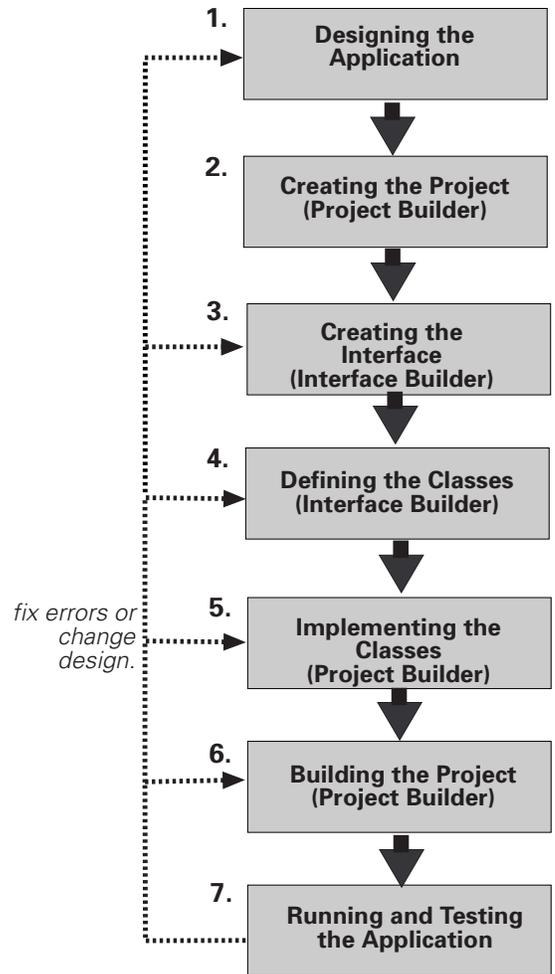
Instead of clicking the button, you can also press the Return key. You can double-click the converted amount, copy it (with the Edit menu's Copy command) and paste it in another application that takes text. You can tab between the first two fields. You can do many other things common to OpenStep applications.

In this tutorial you'll learn the basic things you must do to create a OpenStep application. You will discover how to:

- Create a project.
- Create an interface.
- Create a custom subclass.
- Connect an instance of the custom subclass to the interface.
- Design an application using a common object-oriented design paradigm.

You can find the CurrencyConverter project in the **AppKit** subdirectory of **/NextDeveloper/Examples**.

By following the steps of this chapter, you will become more familiar with the two most important OpenStep applications for program development: Interface Builder and Project Builder. You will also learn the typical work flow of OpenStep application development:



Note: Although this chapter discusses the design of the application midway through the tutorial, application design can take place anytime in the early stages of a project, and in fact is often recommended as the first stage.

Creating the Currency Converter Project

Every OpenStep 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.

In File Viewer navigate to the **/NextDeveloper/Apps** directory.

Select **ProjectBuilder.app** and double-click its icon (at right).



When Project Builder starts up, only its main menu appears on the screen. You must create or open a project to get Project Builder’s main window. The New Project panel allows you to specify a new project’s name and location.

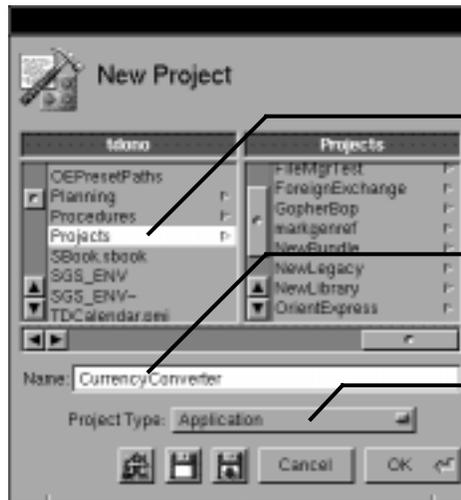
2 Make a new project.

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

In the New Project panel, select the project location.

Enter “CurrencyConverter” as the project name.

Click OK to create the project.



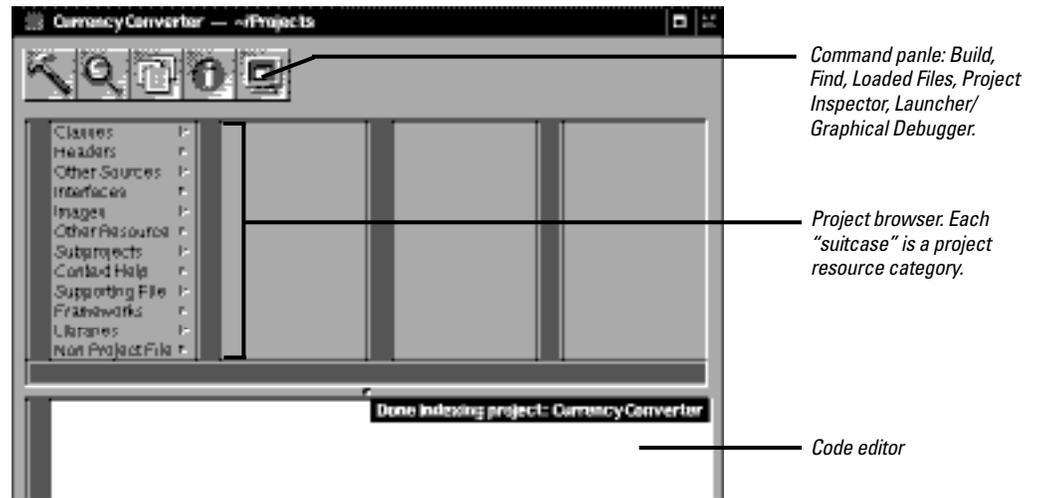
Often projects are kept in a common directory.

The name specified here becomes the name of the project directory and the default name of the application itself.

Make sure Application is the project type.

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.

Note: Here’s a variation on project creation: Create a project directory using File Viewer and then, in the New Project panel, navigate to that directory, type “PB.project” in the Name field, and click OK.



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:

A makefile specifies file dependency relations and compiler and linker instructions for building the project. See *OPENSTEP Development: Tips and Techniques* for common changes to **Makefile.preamble** and **Makefile.postamble**.

- **Other Sources:** This suitcase contains **CurrencyConverter_main.m**, the **main()** routine that loads the initial set of resources and runs the application. (Do not modify this file!)
- **Interfaces:** This suitcase contains **CurrencyConverter.nib**, the file that contains the application’s user interface. More on this file 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.

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 Info 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 Index Source Code from the Project menu. If you want to do without indexing (maybe you have memory constraints), choose Purge Indices from the Project menu.

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.

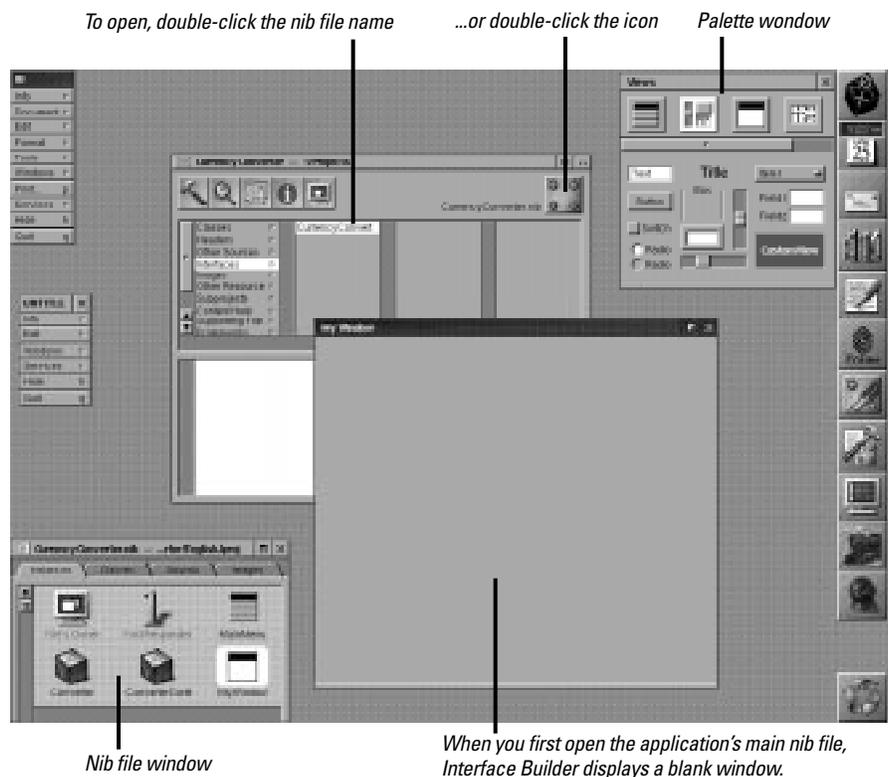
At the beginning of a project, the main nib file is like a blank canvas, ready for you to craft the interface. Look in the Interfaces suitcase for nib files.

1 Open the main nib file.

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

Double-click to open.

A nib file contains user-interface objects, definitions of custom classes, the connections between objects, and sounds and images that are used in the interface. Besides the main nib file, you can have nib files that you can load whenever you need them. These *auxillary* nib files, and the techniques related to using them, are described in the “To Do Tutorial,” page 118. See *OPENSTEP Development: Tools and Techniques* for an overview of nib files.



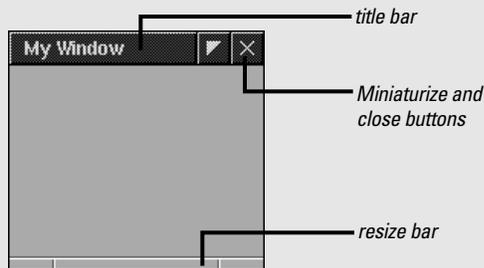
By default, the window entitled “My Window” will appear when the application is launched.

Note: The Interface Builder application is located in **/NextDeveloper/Apps**. The icon for the application is this:



A Window in OpenStep

A window in OpenStep looks very similar to windows in other user environments such as Windows or Macintosh. 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 NeXT 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).

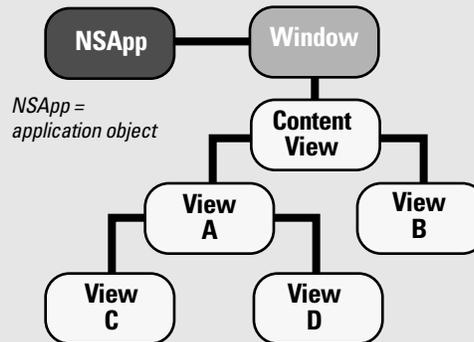
When you create an `NSWindow` object, the Window Server creates the physical window that the `NSWindow` object will manage. The Window Server references the window by its window number, the `NSWindow` by its own identifier.

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 149 for more on the view hierarchy.

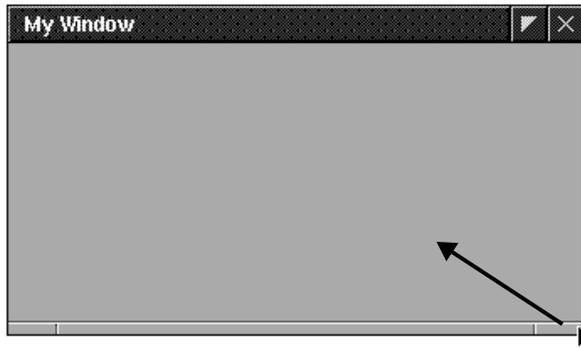
Key and Main Windows

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.

2 **Resize the window.**



Make the window smaller by dragging an edge of the window inward from a resize handle.

Between the two resize handles is the resize bar, which permits only vertical resizing.

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

3 **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 option.



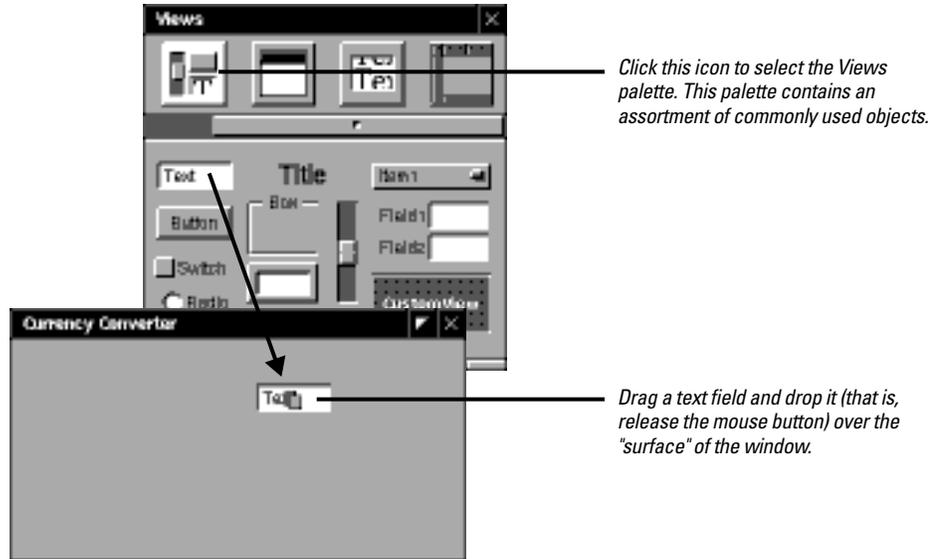
The title of the major window in an application is often the application name.

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

Put palette objects on the window using the “drag and drop” technique.

4 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.

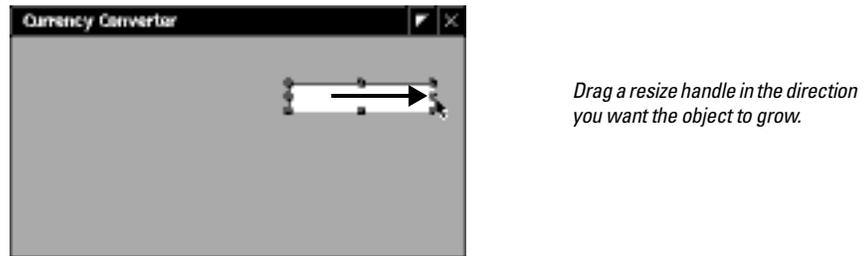


To initialize the text field, double-click "Text" and press Delete.

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):

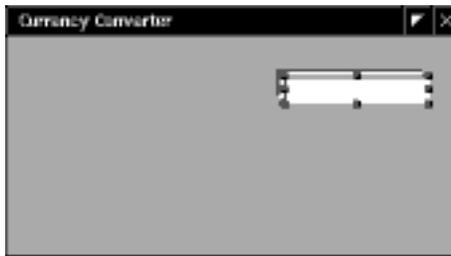
Lengthen the text field.



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

5 Duplicate an object.

- Select the text field.
- Choose Edit ► Copy.
- Choose Edit ► Paste.

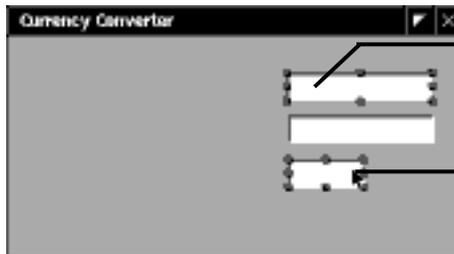


The new text field appears slightly offset from the original field. Reposition it under the first text field.

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

6 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



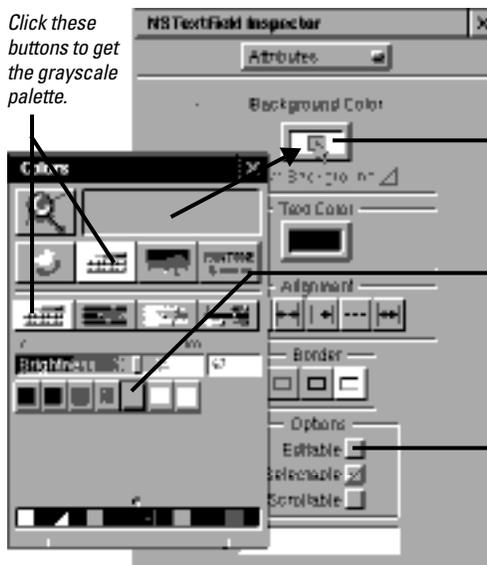
The first object you select should have the dimensions you want the other objects in the selection to take.

Shift-click multiple objects to include them in the same 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 gray background.

7 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 gray 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.



Click these buttons to get the grayscale palette.

Drag the gray color into this well to set the background color.

Click to get the color that 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 the user can select, copy, and paste the result 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

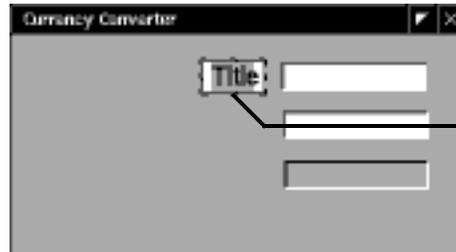
the title object is centered by default, but labels are usually aligned from the right.

8 Assign labels to the fields.

Drag a title object onto the window.

Double-click to select the text "Title".

Choose Format ► Text ► Align Right to align the text from the right.



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.

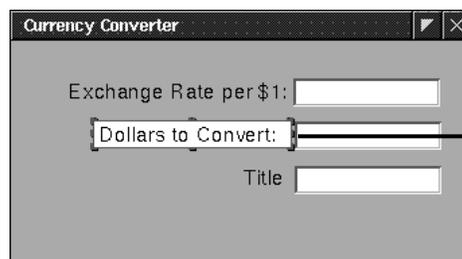
Position all labels to the left of their text fields.



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

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.



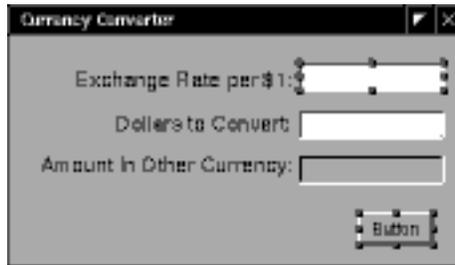
Double-click to select title, then type the text of the label in place of the selection.

9 **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.

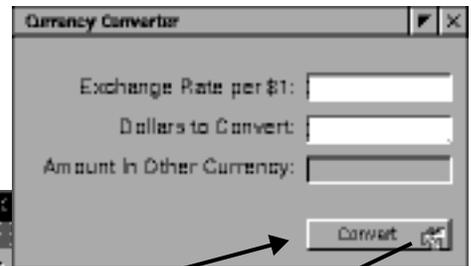
Double-click the title of the button to select it.

You can easily give the button the capacity for responding to carriage returns in addition to mouse clicks.

Select the Images display of the nib file window.

Drag the NSReturnSign image to the main window and drop it over the button.

Select a display of the nib file window by clicking a tab.



After you drop the image over the button, the image appears (by default) to the right of the button title.

If you check the attributes of the button in the Inspector panel, you'll notice two things have been added: **NSReturnSign** is now listed as the button's icon, and the Key field contains the escape sequence for a carriage return (**␣**).

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.

10 Create a horizontal decorative line.

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

Bring up the Attributes display for the box (Command-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.

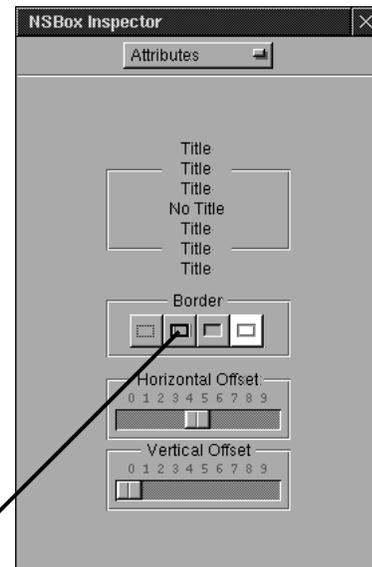
Position the line above the button.

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



Drag upward until lines merge into one line.

For a black line (instead of white) click here.



As you might have noticed, the Currency Converter has a main menu that holds, by default, the commands Info, Hide, and Quit, and the Edit, Services, and Windows menus. The menus contain ready-made sets of commands. The Edit menu includes commands for cutting, copying, and pasting text. The Windows menu lists the titles of open windows as well as common window commands. The Services menu allows your application to communicate with other applications, often with no work on the part of your application. For example, if your application handles text, you can use the Services menu to transfer information to other applications that accept text.

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.

Currency Converter’s interface is almost complete. One finishing touch might be to align the text fields and labels in neat rows and columns. Interface Builder gives you several ways to align selected objects on a window.

- Dragging objects with the mouse
- 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 preceding side bar)

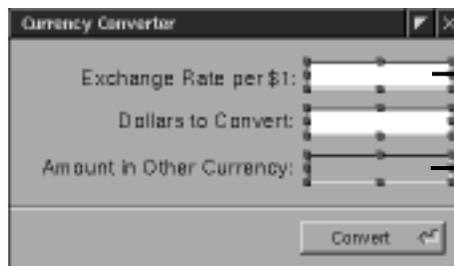
For Currency Converter, use the columns-and-rows technique.

11 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 **nextKeyView** variable is an *outlet*. 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.

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. Many objects on Interface Builder’s palettes have an instance variable named **nextKeyView**. This variable identifies the next object to receive keyboard events when the user presses the Tab key (or the previous object if Shift-Tab is pressed). If you want inter-field tabbing you must connect fields through the **nextKeyView** variable.

12 **Enable tabbing between text fields.**

Select the first text field.

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 to the first field.



When you press Control and drag the mouse from an object, a connection line is drawn.

When a line encloses the destination object, release the mouse button.

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



The nextKeyView outlet identifies the next object to respond to events after the Tab key is pressed.

Be sure to click the Connect button to confirm the connection (the button title then changes to Disconnect).

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

13 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 Quit from the main menu.

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

Note: You can also exit from test mode by double-clicking the Interface Builder icon, which changes to the following image to represent test mode:



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. Move the window around by its title bar.

Here’s some other tests you can make:

- Click the Edit submenu in Currency Converter’s main menu. It expands and contracts as in any application.
- Click the miniaturize button or choose the Hide command. Double-click the document icon to get the window back.
- Click the close box 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 *OPENSTEP Programming: Tools and Techniques* 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 radio 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 we made between Currency Converter’s text fields? While a cursor is in a text field, press the Tab key and watch the cursor jump from field to field.

When You Add Menu Commands

An application you design in Interface Builder can acquire extra functionality with the simple addition of a menu command or submenu. You’ve already seen what you get with the Services and Windows menu, both included by default. You can add other commands and submenus to the main menu for “free” functionality *without* compilation. For example:

- The Font submenu adds behavior for applying fonts to text in NSText objects, such as the one in the scroll view object in the DataViews palette. Your application gets the Font panel and a font-manager object “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.

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 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 and Account Management

An application can use the Open panel of the Application Kit to help the user locate files in the file system and open them. It can also make the Save panel available for saving information in files. NeXT's version of OpenStep also provides classes for managing files in the file system (creating, comparing, copying, moving, and so forth) and for managing system-account information and user defaults.

Communicating With Other Applications

OpenStep gives an application several ways of communicating information to and from 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 avail itself of the services provided by another application, based on the type of the 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.

Editing Support

You can get several panels (and associated functionality) when you add a submenu to your application's main menu in Interface Builder. These "add-ons" includes 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 it provides.

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 and Faxing

With just a simple Interface Builder procedure, OpenStep automates simple printing and faxing of views that contain text or graphics. When a user clicks the control, an appropriate panel helps to configure the print or fax 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 a help system for your application using Interface Builder, Project Builder, and an RTF text editor (such as Edit). The Application Kit includes a class for context-sensitive help. If the user clicks an object on the application's interface while pressing a Help key, a small window is displayed containing concise information on the object.

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.

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.

Designing the Currency Converter Application

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 "The Model-View-Controller Paradigm" on page 36). This design paradigm aids in the development of maintainable, extensible, and understandable systems. But first, you might want to read the sidebar below to understand the symbol used in the design diagram.

Note: This design for Currency Converter is intended to illustrate a few points, and so is perhaps 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.

Why an Object is Like a Jelly Donut

This book depicts objects as filled and segmented "donuts." 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.

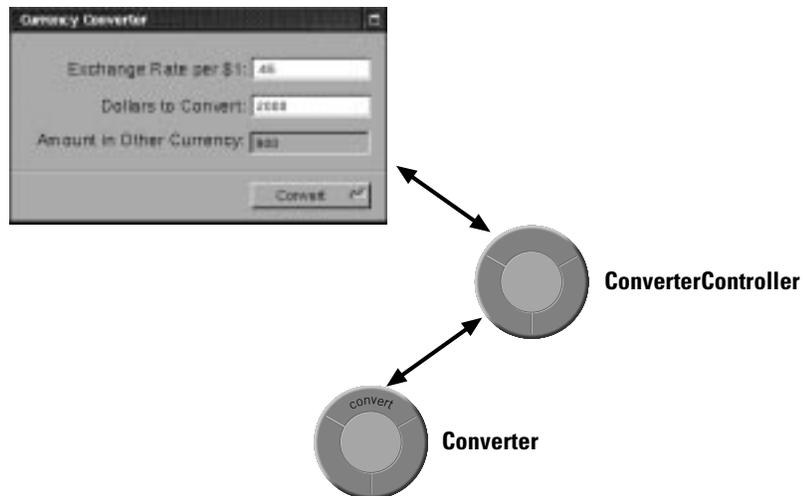
An object's procedures (called *methods*) respond to the message

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 properties of objects.

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.

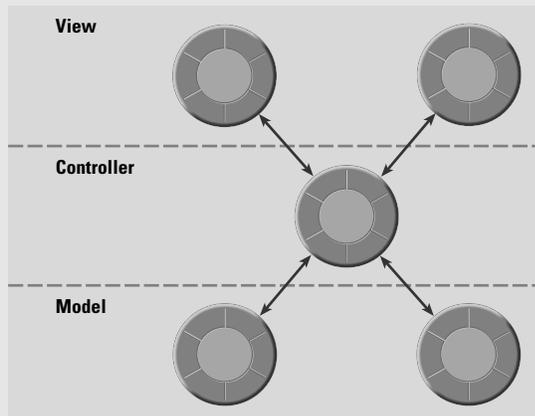


The ConverterController class assumes a central role. Like all controller objects, it communicates with the interface and with model objects, and it handles tasks specific to the application, such as managing the cursor. ConverterController gets the values users enter into fields, passes these values 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.

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 displayable. 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

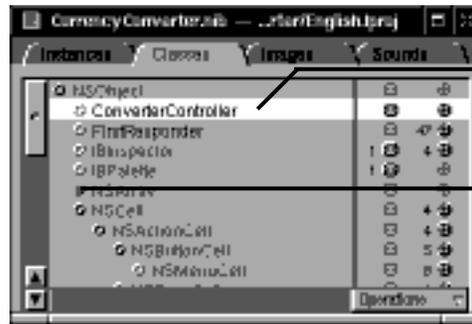
MVC, strictly observed, is not advisable in all circumstances. Sometimes it's best to combine roles. For instance, in a graphics-intensive application, such as an arcade game, you might have several View objects that merge the roles of View and Model. In some applications, especially simple ones, you 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.

Enter the name of the subclass:
"ConverterController."

Press Return.



After you name the class, it appears indented under its superclass in alphabetical order.

To see subclasses of a class, click a filled button (if the button is unfilled, there are no subclasses).

NSCell, for example, has several levels of subclasses; each level is indicated by indentation.

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:

See *Paths for Object Communication: Outlets, Targets, and Actions* on page 40, for a more detailed description of outlets and actions. See page 103 for more on control objects and their relation to cells and formatters.

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 lets you keep track of 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*.

2 Define your class's outlets.

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

Choose Add Outlet from the Operations pull-down menu

Type the name of the outlet over the highlighted "myOutlet." Name the first outlet **rateField**.

Press Return.

Repeat the last three steps to define two other outlets:

dollarField
totalField



Click here to begin specifying outlets.

"Outlets" appears indented underneath, highlighted (not shown).

Instead of choosing Add Outlets from the Operations menu, you can press Return when "Outlets" is highlighted to add an outlet.



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

3 Define your class's actions.

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

Choose the Add Action command from the Operations pull-down menu.

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

Press Return.



The crosshairs suggest the "target" in the target/action paradigm.

After you chose Add Action "myAction" appears indented under "Actions."

You only need to type **convert** here—Interface Builder adds the colon.

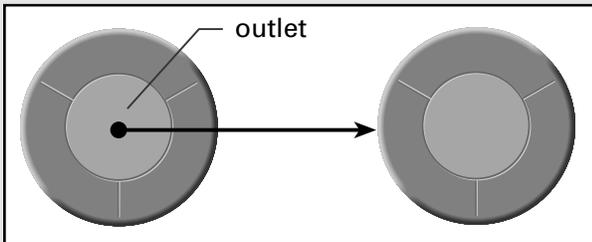
Before You Go On

Add an outlet: 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.

Paths for Object Communication: Outlets, Targets, and Actions

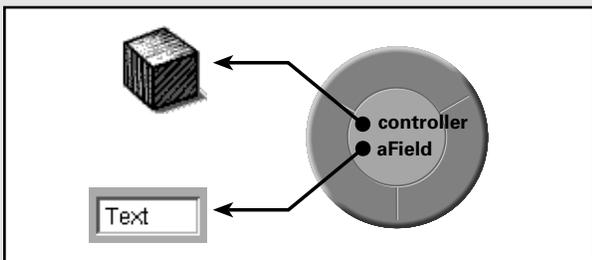
Outlets

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 anObject;
```

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; you can declare these objects as instance variables, but they are not outlets. What distinguishes outlets is their relationship to Interface Builder.

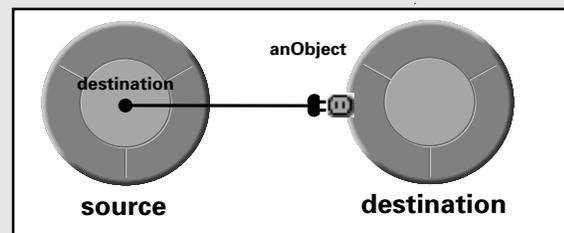
Interface Builder can “recognize” outlets in code by their declarations, and it can initialize outlets. You usually set an outlet’s value in Interface Builder by drawing connection lines between objects. There are ways other than outlets to reference objects in an application, but outlets and Interface 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 identifier of the destination object.

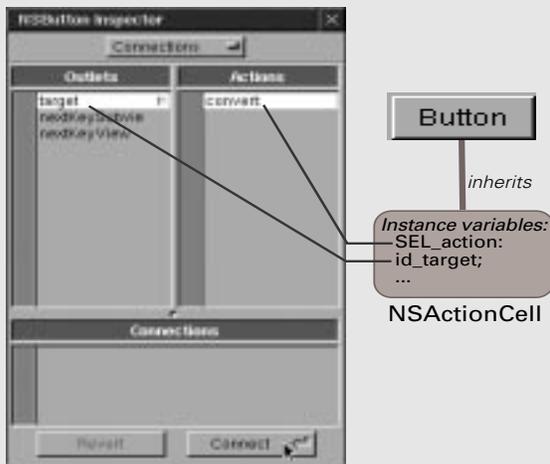
It might help to understand connections by imagining an electrical outlet (as used in the Classes display of the nib file window) embedded in the destination object. Also picture an electrical cord extending from the outlet in the source object. Before the connection is made the cord is unplugged and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

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**.

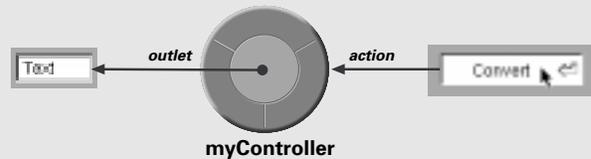
Which Direction to Connect?

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.



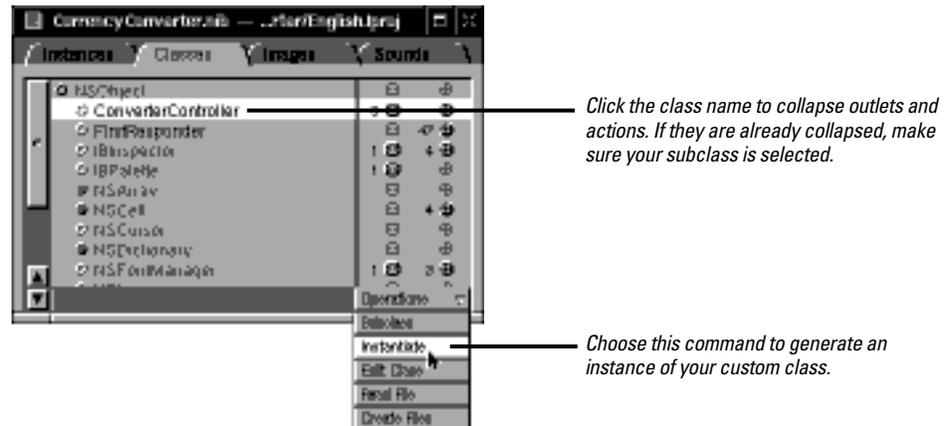
Connecting ConverterController to the Interface

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

4 Generate an instance of the class.

In the Classes display, select the ConverterController class.

Choose the Instantiate command from the Operations pull-down menu.



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.

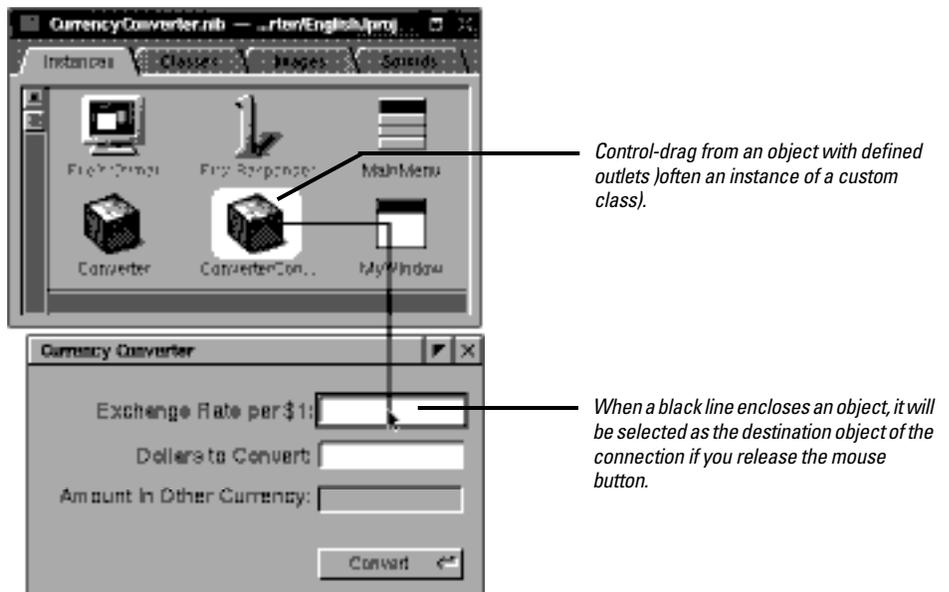


Now you can connect this ConverterController object to the user interface. By connecting it to specific objects in the interface, you initialize your outlets. ConverterController will use these outlets to get and set values in the interface.

5 **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.

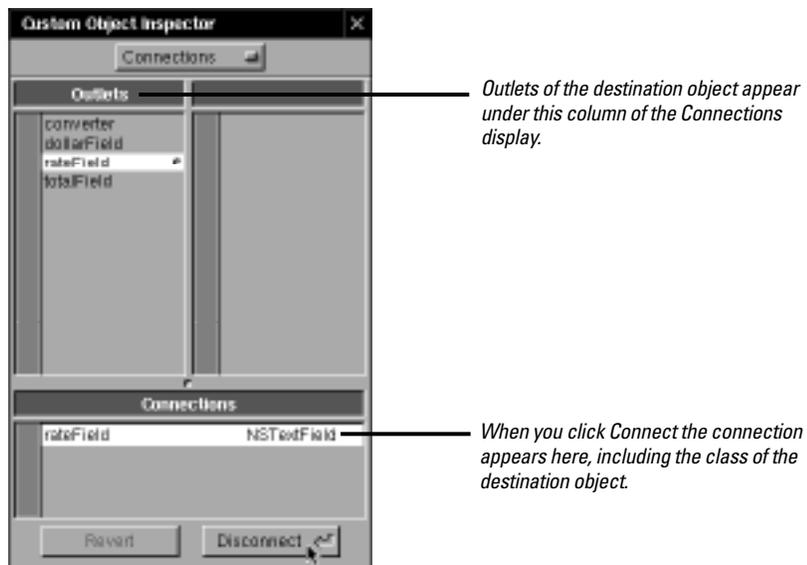


Interface Builder brings up the Connections display of the Inspector panel. This display shows the outlets you have defined for ConverterController.

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.



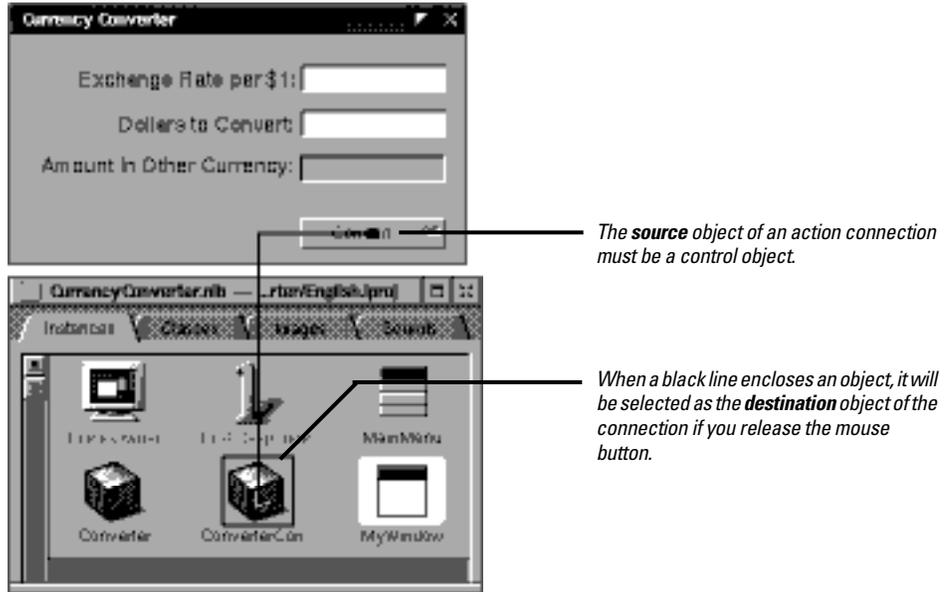
To receive action messages from the user interface—to be notified, for example, when users click a button—you must connect the control objects that emit those messages to CurrencyConverter. The procedure for connecting actions is similar to that for outlets, but with one major difference. When you connect 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.

6 **Connect the interface's controls to the custom class via its 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..



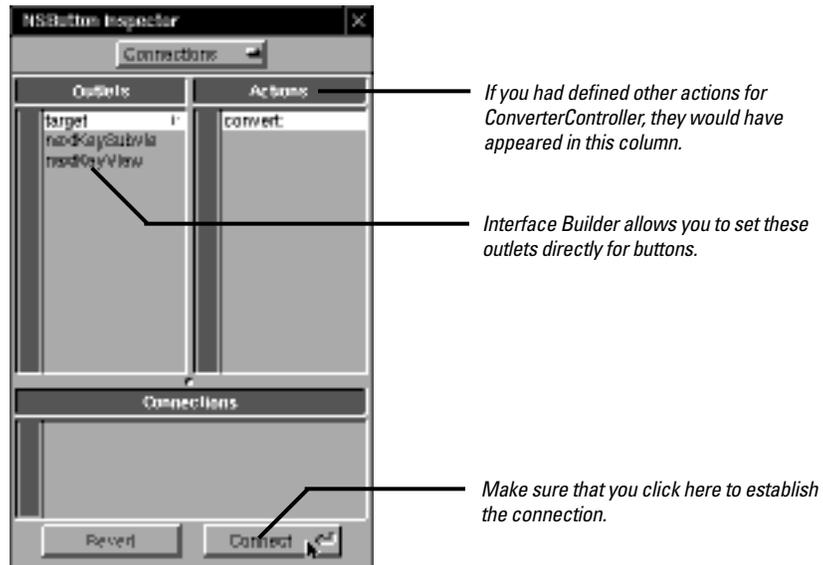
The Connections display of the Inspector panel shows the action methods you have specified for ConverterController.

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

Define the Converter Class: While connecting ConverterController’s outlets, you probably noticed that one outlet remains unconnected: **converter**. This outlet identifies the instance of the Converter class in the Currency Converter application, which 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: The **NSReturnsign** image that you embedded earlier in the Convert button indicates that users can activate this 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.

Implementing the Classes of Currency Converter

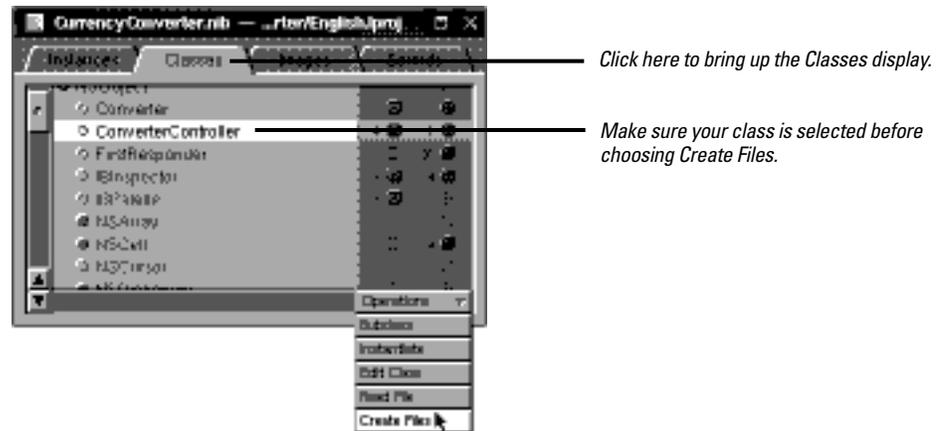
Interface Builder generates source code files from the (partial) class definitions you've made. These files are "skeletal," in the sense that they contain little more than essential Objective-C directives and the class-definition information. You'll usually need to supplement these files with your own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



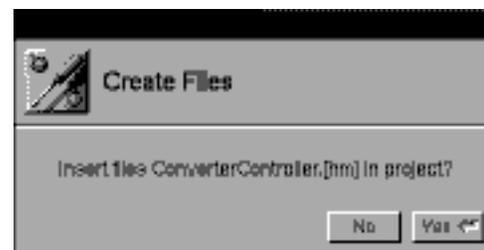
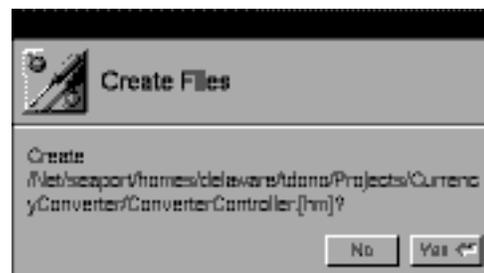
Interface Builder then displays two attention panels, one after the other:

When a Create Files panel is displayed, click Yes.

A second Create Files panel is displayed; click Yes again.

Repeat for the Converter class.

Save the nib file.



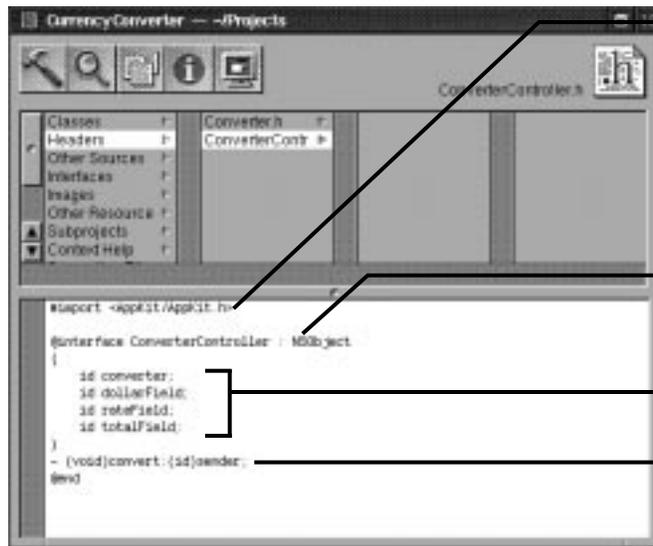
Now we leave Interface Builder for this application. You'll complete the application using Project Builder.

2 **Examine an interface (header) file in Project Builder.**

Hide Interface Builder and activate Project Builder.

Click Headers in the project browser.

Select **ConverterController.h**.



Project Builder imports the Application Kit header files, which import the Foundation header files.

Interface definition begin with **@interface** and the class name. The superclass appears after the colon.

Instance variables (here the outlets defined in Interface Builder) go between the braces. Method declarations follow the second brace. The declaration of the action method you specified in Interface Builder is inserted. The definition ends with **@end**.

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 with by declaring the method in **Converter.h**.

3 **Add a method declaration.**

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

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

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

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

@end
```

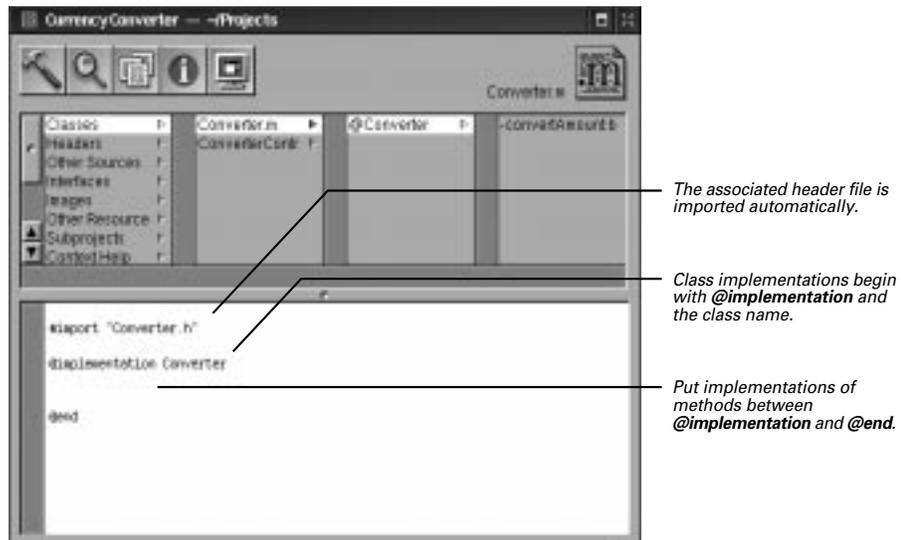
This declaration states that **convertAmount:byRate:** takes two arguments of type **float**, and returns a **float** value. When parts of a method name have colons, such as **convertAmount:** and **byRate:**, they are *keywords* which introduce arguments. (These are keywords in a sense different from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

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

4 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:

5 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;

    amt = [dollarField floatValue]; /* 1 */
    rate = [rateField floatValue];
    total = [converter convertAmount:amt byRate:rate]; /* 2 */
    [totalField setFloatValue:total]; /* 3 */
    [rateField selectText:self]; /* 4 */
}
```

The **convert** method does the following:

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

2. Invokes the **convertAmountByRate:** method and gets the returned value.
3. Uses **setFloatValue:** to write the returned value in the Amount in Other Currency text field (**totalField**).
4. Sends **selectText:** to the rate field; this puts the cursor in the rate field so the user begin another calculation.

Be sure to **#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 **floats**, 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.

6 Implement the **awakeFromNib** method.

Type the code shown at right.

```
- (void)awakeFromNib
{
    [rateField selectText:self]; /* 1 */
    [[rateField window] makeKeyAndOrderFront:self]; /* 2 */
}
```

1. 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.
2. 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]** returns the window to which the text field belongs, and the **makeKeyAndOrderFront:** method is then sent to this returned object.

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 (-) and, for *class methods*, 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**.
- Method argument types are in parentheses and go between the argument's *keyword* and the argument itself:

```
- 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
```

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 it the receiving object 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** objects as arguments.

- 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.

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.

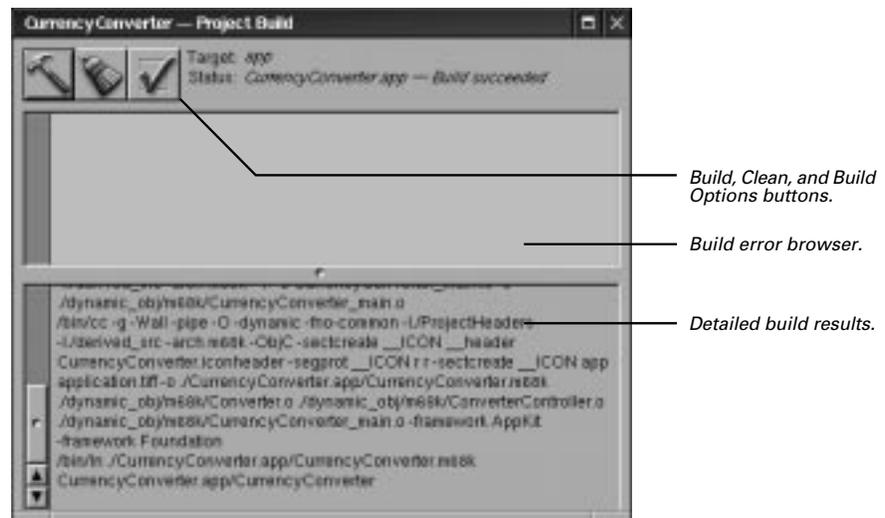
Save source code files and any changes to the project.

Click the Build button on the main window (icon 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.



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 *OPENSTEP Development: Tools and Techniques*

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.”

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.

What Happens When You Build an Application

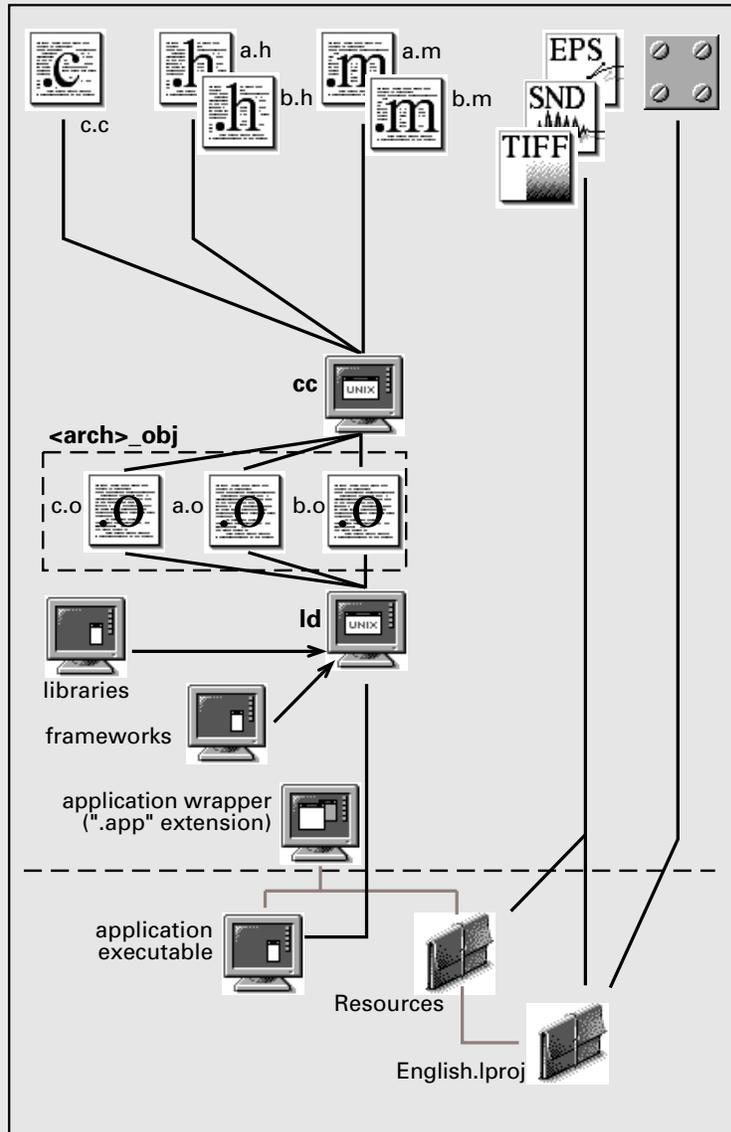
By clicking the Build button in Project Builder, you run the build tool. By default, the build tool is **gnumake**, but it can be any build utility that you specify as a project default in Project Builder. The build tool coordinates the compilation and linking process that results in an executable file. 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 tool **cc**, 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. It puts these files in an architecture-specific subdirectory of **dynamic_obj**.

In the linking phase of the build, the build tool executes the link editor **ld** (via **cc**), 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. If there are multiple architecture-specific object files, linking also combines these into a single "fat" executable.

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 is a file package with an extension of ".app". A file package is a directory that the Workspace Manager presents to users as a simple file; in other words, it hides the contents of the directory. The ".app" extension tells

the Workspace Manager that the application wrapper contains an executable that can be run ("launched") by double-clicking.

2 Build the project after correcting errors.

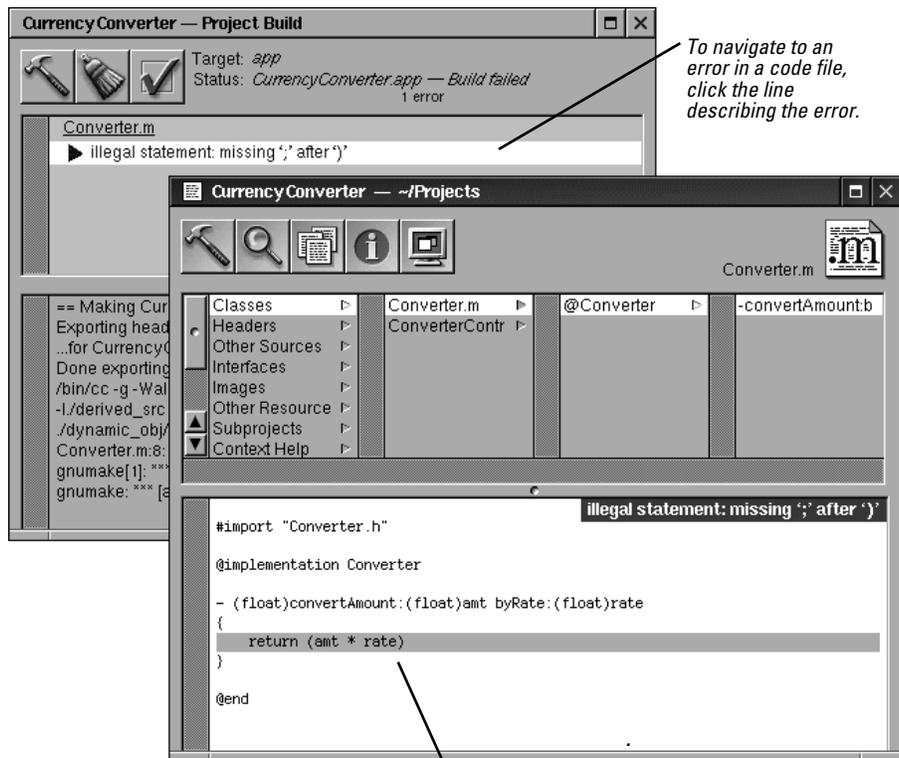
Delete a semicolon in the code, creating an error.

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.



To navigate to an error in a code file, click the line describing the error.

Project Builder highlights the line that contains the error

Where To Go For Help

Context-Sensitive Application Help

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 Help key varies by computer architecture. Consult user documentation for the Help key on your machine.



Digital Librarian

Digital Librarian is an application that quickly searches for a word (or other lexical unit) in an on-line manual (or other target) and lists the documents that contain the word. You click a listed item and the document is displayed at the point where the word occurs. The contents of documents are indexed, making searching very fast.

OpenStep includes **NextDeveloper.bshlf**, a Digital Librarian bookshelf for developers in **/NextLibrary/Bookshelves**. This bookshelf contains most of the targets you are likely to want, and includes (as the topmost target) instructions on creating your own bookshelf and customizing it to your needs. When you choose Help from Project Builder or Interface Builder, a Digital Librarian bookshelf is opened that contains the on-line version of *OPENSTEP Development: Tools and Techniques*.

You can find Digital Librarian as **Librarian.app** in **/NextApps**.





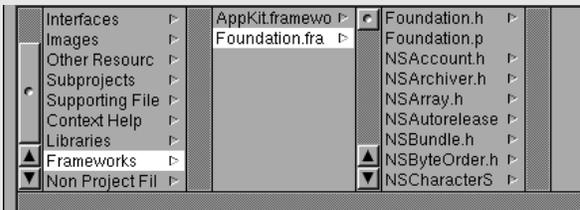
Project Builder

Project Builder gives you several ways to get the information you need when 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. See *OPENSTEP Development: Tools and Techniques* for a complete description of Project Find.

Reference Documentation Lookup: If the results of a search using Project Find includes OpenStep symbols, you can easily get related reference documentation that describes that symbol. See "Finding Information Within Your Project" on page 94 for instructions on the use of this feature.

Frameworks: Under Frameworks in the project browser, you can browse the header files related to OpenStep frameworks within Project Builder. The Application Kit and Foundation frameworks always are included by default for application projects. See chapter 5, "Where to Go From Here," for a fuller description



NeXT's Technical Documentation

Most OpenStep programming documentation is located on-line in NeXTLibrary/Documentation/NextDev. The document files are in RTF format, so you can open them in Project Builder, Edit, or in most word processors. NeXT includes the following manuals under the **/NextDev** directory:

Reference

- API Reference Documentation (specifications of classes, protocols, functions, types, and constants). This documentation is divided among, and located in, the frameworks NeXT provides, except for information that is common to all frameworks (/Reference).
- Development Tools Reference covering the compiler, the debugger, and other tools (Reference DevTools).
- *NeXT Assembler Manual*

Tasks and Concepts

- *Discovering OPENSTEP: A Developer Tutorial* (this manual)
- *Object-Oriented Programming and the Objective-C Language*
- *Topics in OPENSTEP Programming* (concepts and programming procedures)
- *OPENSTEP Development: Tools and Techniques* (a task-oriented approach to using the development tools)
- *OPENSTEP Conversion Guide* (step-by-step instructions for converting 3.x NEXTSTEP applications to run on OPENSTEP 4.0).

The **/NextDev** directory also includes release notes. It also contains documentation on the following products, if they're installed: Enterprise Objects Framework, Distributed Objects (DO), Portable Distributed Objects (PDO).

See chapter 5, "Where to Go From Here," for more information on NeXT's technical publications.

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.

You can use Project Builder's graphical debugger or **gdb** to track bugs down. See "Using the Graphical Debugger" on page 104 for an overview of the graphical debugger.

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.
