Inside Mac OS X

Developing Cocoa Objective-C Applications: A Tutorial

July 2002

Contents

C H A P T E R 1

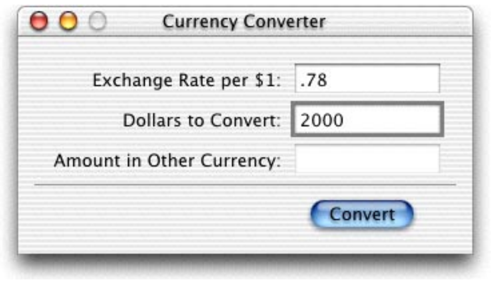
Introduction

This tutorial introduces the Cocoa application framework of Mac OS X, and teaches you how to use Apple's development tools and the Objective-C language to build robust, object-oriented applications. Cocoa provides the best way to build modern, multimedia-rich, objected-oriented applications for consumers and enterprise customers alike. This tutorial assumes familiarity with C programming, but does not assume any previous experience with Cocoa, Project Builder, or Interface Builder.

What You'll Learn in This Tutorial

In this tutorial you will build a simple application that converts a dollar amount to an amount in another currency, given the rate of that currency relative to the dollar. Here's what the finished application looks like:

Figure 1-1 The completed Currency Converter application



Currency Converter is a simple application, yet it exemplifies much of what software development with Cocoa is all about. As you'll discover, Currency Converter is amazingly easy to create, and it's equally amazing how many features you get "for free"—as with all Cocoa applications.

You can click in one of the text fields to enter a value, or use the tab key to move between them. The conversion is invoked by clicking the Convert button or by pressing the Return key. Because of the features inherited from the Cocoa application environment, you can select the converted amount, copy it (with the Edit menu's Copy command), and paste it in another application that takes text.

This tutorial will lead you through all of the basic steps for creating a Cocoa application. You'll learn how to:

- Create a Project Builder project.
- Create a graphical user interface using Interface Builder.
- Create a custom subclass of a Cocoa framework class.
- Connect an instance of your custom subclass to the interface.

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

1. Designing the application (your brain)
2. Creating the project (Project Builder)
3. Creating the interface (Interface Builder)
4. Defining the classes (Interface Builder)
5. Implementing the classes (Project Builder)
6. Building the project (Project Builder)
7. Running and testing the application

Along the way, you'll also learn how to design an application using a common object-oriented design paradigm. For additional background on object-oriented programming and Objective-C, see *Inside Mac OS X: The Objective-C Programming Language*.

The tutorial consists of the next two chapters, which you should complete in order:

1. "Designing the Currency Converter Application" (page 11)
2. "Building the Currency Converter Application" (page 17)

The final chapter, "Cocoa Resources" (page 69), describes where you can go to further improve your knowledge of Cocoa and Mac OS X programming in general.

C H A P T E R 2

Designing the Currency Converter Application

This part of the tutorial explains the most common object-oriented design paradigm used in Cocoa, Model-View-Controller, and guides you through the design of the Currency Converter application.

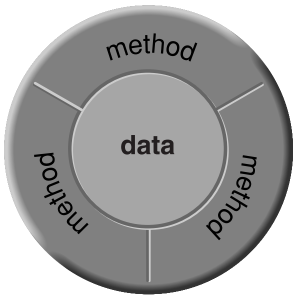
Designing the Currency Converter Application

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.

Object Notation

When designing an object-oriented application, it is often helpful to graphically depict the relationships between objects. This tutorial depicts objects graphically like Figure 2-1.

Figure 2-1 An object as a jelly donut



Though this representation might look a bit like a jelly donut, or a lifesaver, or a slashed tire, it 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.

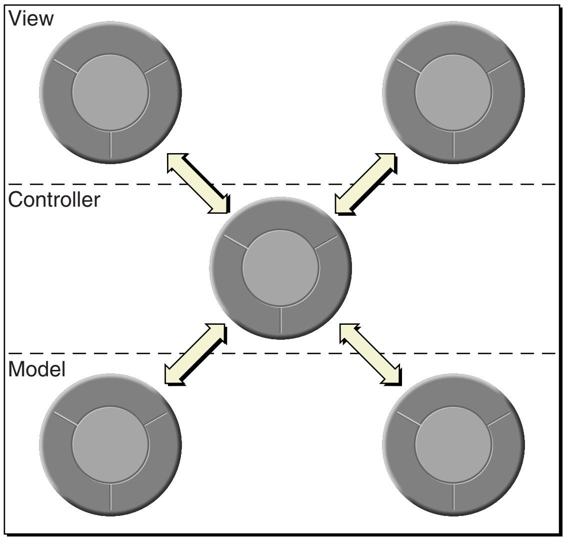
See *Inside Mac OS X: The Objective-C Language* for a fuller description of data encapsulation, messages, methods, and other things pertaining to object-oriented programming.

The Model-View-Controller (MVC) Paradigm

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, the model behind many designs for object-oriented programs. This design pattern aids in the development of maintainable, extensible, and understandable systems.

Model-View-Controller (MVC) was derived from Smalltalk-80. It proposes three types of objects in an application, separated by abstract boundaries and communicating with each other across those boundaries.

Figure 2-2 Object relationships in the Model-View-Controller paradigm



Model Objects

This type of object represents special knowledge and expertise. Model objects hold 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 about 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.

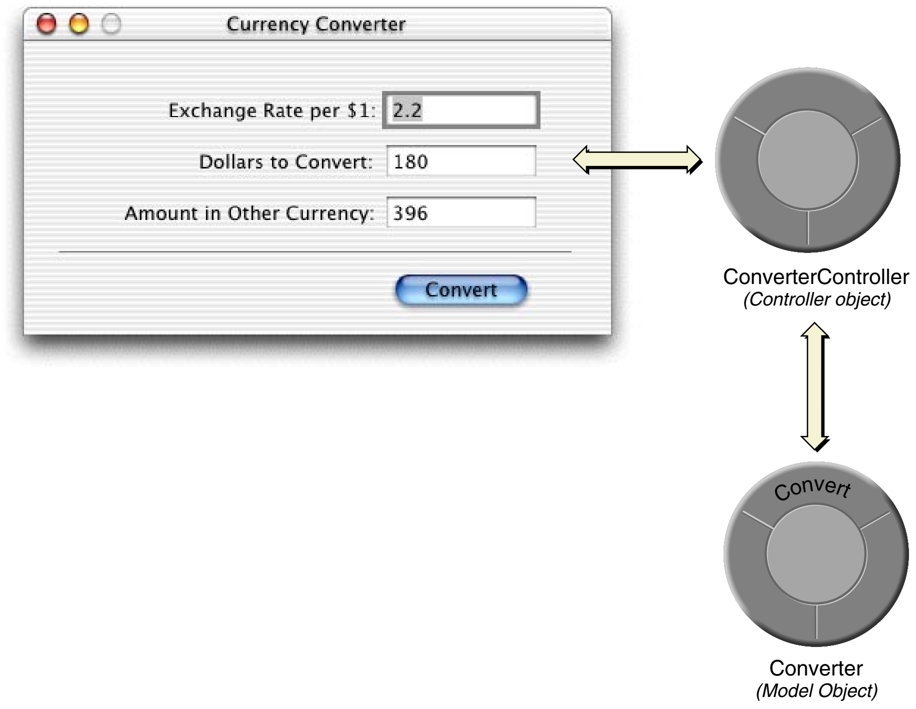
Hybrid Models

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.

MVC in Currency Converter's Design

Currency Converter consists of two custom objects (Model and Controller) and a user interface (View), implemented using a collection of ready-made Application Kit objects. A 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.

Figure 2-3 Model-View-Controller relationships in Currency Converter



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.

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, perform the computation and do without the Converter class.

Building the Currency Converter Application

This part of the tutorial guides you through building the Currency Converter application and in the process teaches you the steps essential to building a Cocoa application using Objective-C.

The tasks in this chapter include:

Creating the Currency Converter Project

Every Cocoa 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, frameworks, libraries, the application's user interface, sounds, and images. You use the Project Builder application to create and manage your project.

There are three basic steps to create a Cocoa project:

"Open Project Builder" (page 18)

"Choose the New Project Command" (page 18)

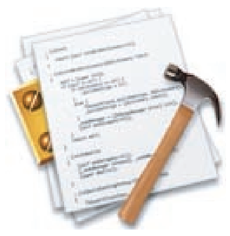
"Select Project Type" (page 18)

Open Project Builder

To open Project Builder:

1. Find Project Builder in `/Developer/Applications/`.
2. Double-click the icon, shown in Figure 3-1.

Figure 3-1 The Project Builder application icon



The first time you start Project Builder, you'll be asked a few setup questions. The default values should work for the majority of users.

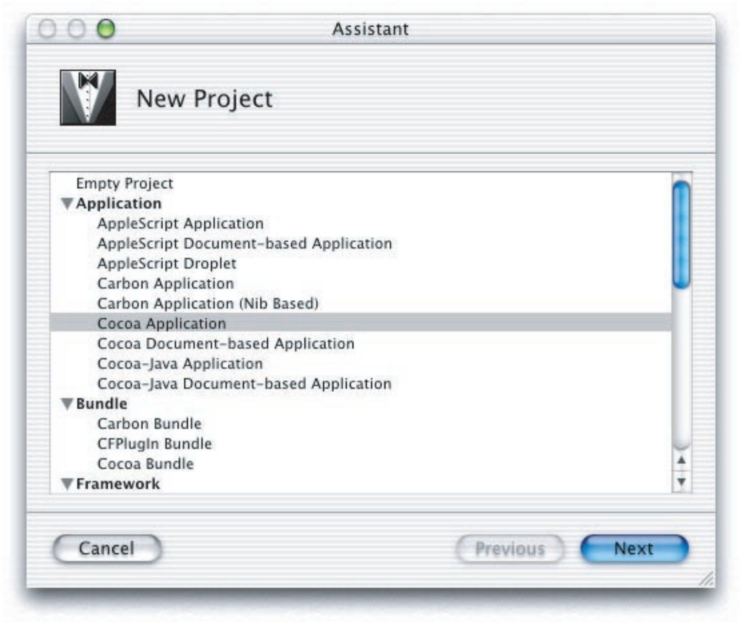
Choose the New Project Command

When Project Builder is launched, only its menus appear. To create a project, choose New Project from the File menu. Project Builder will then display the New Project window.

Select Project Type

Project Builder can build many different types of applications, including everything from Carbon and Cocoa applications to Mac OS X kernel extensions and Mac OS X frameworks. For this tutorial, select Cocoa Application and click Next, as shown in Figure 3-2 (page 19).

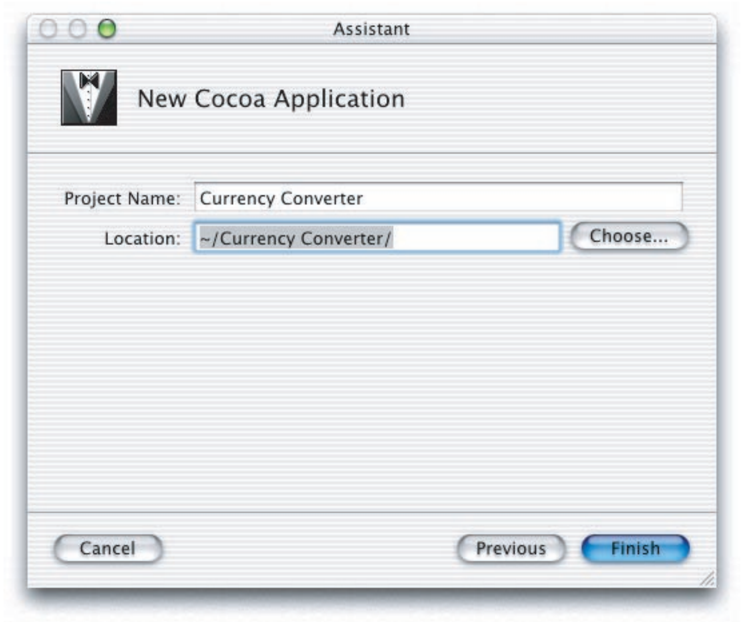
Figure 3-2 Project Builder's New Project Assistant



Empty Project
▼Application
AppleScript Application
AppleScript Document-based Application
AppleScript Droplet
Carbon Application
Carbon Application (Nib Based)
Cocoa Application
Cocoa Document-based Application
Cocoa-Java Application
Cocoa-Java Document-based Application
▼Bundle
Carbon Bundle
CFPlugIn Bundle
Cocoa Bundle
▼Framework

1. Using the file-system browser, navigate to the directory where you want your project to be.
2. Type the name of the project in the Name field. For the current project, type the name "Currency Converter."
3. Click Finish.

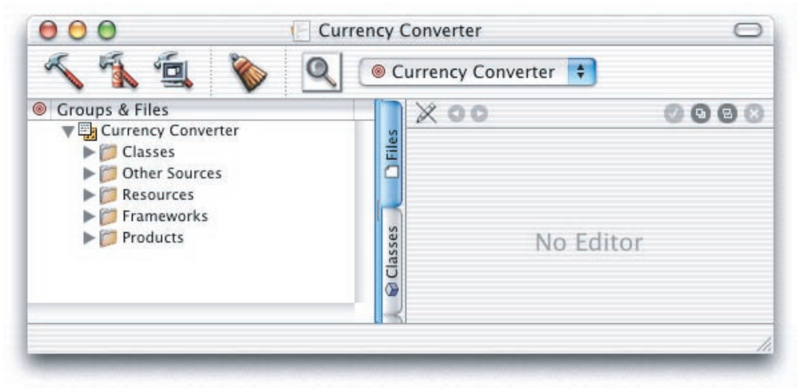
Figure 3-3 Selecting a name and location in Project Builder's New Project Assistant



Project Name: Currency Converter
Location: ~/Currency Converter/

When you click Finish, Project Builder creates and displays a project window, as shown in Figure 3-4.

Figure 3-4 The new Currency Converter project in Project Builder.



Notice that Project Builder uses hierarchical groups to organize a project. These groups are very flexible in that they do not necessarily reflect either the on-disk layout of the project or the build system handling of the files. They are purely for organizing your project. The default groups created for you by the templates can be used as is or rearranged however you like.

Go ahead and click an item in the left column of the project browser and see what some of the default groups contain:

Classes This group is empty at first, but it will be used to hold the implementation and header files for your project.

Other Sources This group contains `main.m`, the `main()` routine that loads the initial set of resources and runs the application. (You shouldn't have to modify this file.)

Resources This group contains the nib files (extension `.nib`) and other resources which specify the application's user interface. More on nib files in the next step.

Frameworks This group contains the frameworks (which are similar to libraries) which the application imports.

Products This group contains the results of project builds and is automatically populated with references to the products created by each target in the project.

Curious folks might want to look in the project directory to see what kind of files it now contains. Among the project files are:

`English.lproj`
: A directory containing resources localized to your preferred language. In this directory are nib files automatically created for the project. You may find other localized resources, such as `Dutch.lproj`. For more information on nib files, see the developer documentation for Interface Builder.

`main.m`
: A file, generated for each project, that contains the entry-point code for the application.

`Currency Converter.pbproj`
: This file contains information that defines the project. It too should not be modified. You can open your project by double-clicking this file in the Finder.

Creating the Currency Converter Interface

This section of the tutorial guides you through the twelve code-free steps involved in creating a functioning graphical interface for Currency Converter, and explains interesting and important aspects of Cocoa programming along the way.

The steps are as follows:

"Open the Main Nib File" (page 23)

"Resize the Window" (page 25)

"Set the Window's Title and Attributes" (page 27)

"Place a Text Field, Resize and Initialize It" (page 28)

"Duplicate an Object" (page 28)

"Change the Attributes of a Text Field" (page 29)

"Assign Labels to the Fields" (page 29)

Nib Files

Every application with a graphical user interface has at least one nib file. The main nib file is loaded automatically when an application launches and contains the application menu and generally as least one window along with various other objects. An application can have other nib files as well. Each nib file contains:

Archived Objects Also known in object-oriented terminology as "flattened" or "serialized" objects—meaning that the object has been encoded in such a way that it can be saved to disk (or transmitted over a network connection) and later restored in memory. Archived objects contain information such as their size, location, and position in the object hierarchy. 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 (typically the one that loaded the nib file from disk).

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 Cocoa 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.

Open the Main Nib File

1. Locate `MainMenu.nib` in the Resources group in Project Builder.

2. Double-click to open it. This will open Interface Builder and bring up the nib file.

A default menu bar and window titled “Window” will appear when the nib file is open.

Windows in Cocoa

A window in Cocoa looks very similar to windows in other user environments such as Windows or Mac OS 9. It is a rectangular area on the screen in which an application displays things such as 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 Cocoa window has a title bar, a content area, and several control objects.

NSWindow and the Window Server

Many user-interface objects other than the standard window are windows. Menus, pop-up lists, and pull-down lists are primarily windows, as are all varieties of utility windows and dialogs: attention dialogs, Info windows, and tool palettes, to name a few. In fact, anything drawn on the screen must appear in a window. End-users, however, may not recognize or refer to them as “windows”.

Two interacting systems create and manage Cocoa windows. On the one hand, a window is created by the Window Server. The Window Server is a process which uses the internal window management portion of Quartz (the low-level drawing system) to draw, resize, hide, and move windows using Quartz graphics 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 a Cocoa 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 Cocoa 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 NSWindow object, on the other hand, manages a hierarchy of views in addition to its 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 views (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. Cocoa uses the lower-left corner of the frame rectangle as the origin for the base coordinate system, unlike Carbon and Classic applications, which use the upper-left corner. Views draw themselves in coordinate systems transformed from (and relative to) this base coordinate system.

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.

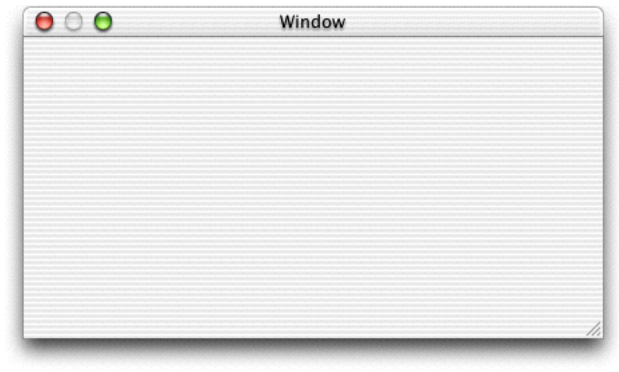
Key windows respond to key presses for an application and are the primary recipient of messages from menus and panels. Usually a window is made key when the user clicks it. Each application can have only one key window.

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 window or an Info window) have a direct effect on the main window.

Resize the Window

1. Make the window smaller by dragging an edge of the bottom-right corner of the window inward.

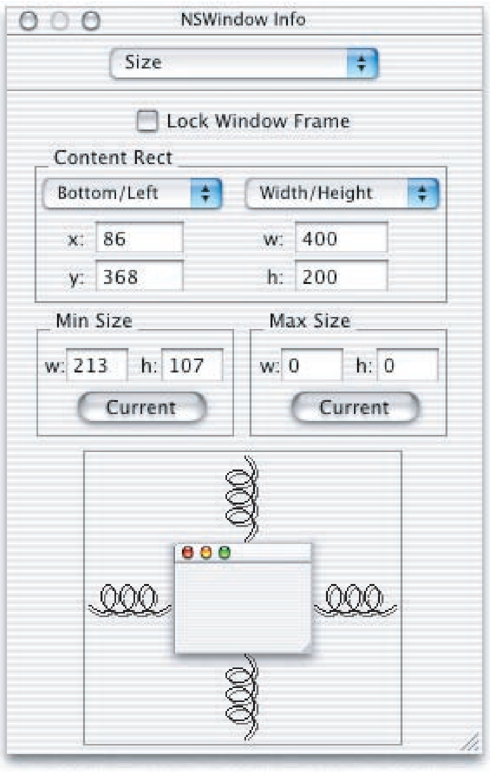
Figure 3-5 A resized window in Interface Builder



You can resize the window more precisely by using the Size menu of the Window Info Window.

1. Choose Show Info from the Tools menu.
2. Select Size from the pop-up menu.
3. In the “Content Rect” area, select “Width/Height” from the right-hand pop-up menu. In the text fields under the Width/Height menu, type 400 in the width (w) field and 200 in the height (h) field, as shown in Figure 3-6 (page 27).

Figure 3-6 Resizing a window with Interface Builder's Info palette



x: 86
y: 368
w: 400
h: 200

Min Size: w: 213 h: 107

Max Size: w: 0 h: 0

Set the Window's Title and Attributes

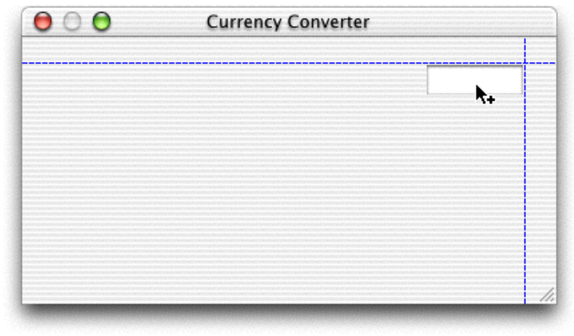
While the Info Window is open, set other attributes for the window.

1. Select Attributes from the Info window's pop-up menu and change the window's title to "Currency Converter".
2. Verify that the "Visible at launch time" option is selected.
3. De-select the "Resize" check box in the "Controls" area.

Place a Text Field, Resize and Initialize It

1. Drag a text field object onto the Currency Converter window. Notice that Interface Builder helps you place objects according to the Aqua Interface Guidelines by displaying "pop-up" guides when an object is dragged close to the proper distance from neighboring objects or the edge of the window.
2. Resize the text field by grabbing a handle and dragging in the direction you want it to grow. In this case, drag the left handle to the left to enlarge the text field.

Figure 3-7 Adding a text field in Interface Builder



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.

Duplicate an Object

1. Select the text field, if it is not already selected.
2. Choose Duplicate (Command-D) from the Edit menu. The new text field appears slightly offset from the original field.
3. Reposition the new text field under the first text field. Notice that guides appear and assist you by "snapping" the second text field into place.

4. To make the third text field, type Command-D again. Notice that IB remembered the offset from the previous Duplicate command and automatically applied it to the newly created text field.

Change the Attributes of a Text Field

The bottom text field will display the results of the computation and should therefore have different attributes than the other text fields. It should not be editable or selectable.

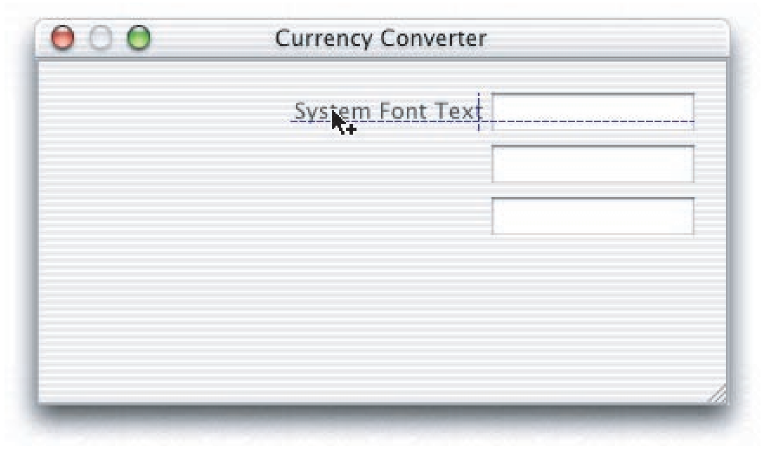
1. Select the third text field.
2. Bring up the Info Window and choose Attributes from the pop-up menu.
3. Turn off the Editable attribute in the Options section of the Info window so that users will not be able to alter the contents of the field. Keep the Selectable attribute so that users can copy and paste the contents to other applications.

Assign Labels to the Fields

Text fields without labels would be confusing, so add labels by using the ready-made label object from the Views palette.

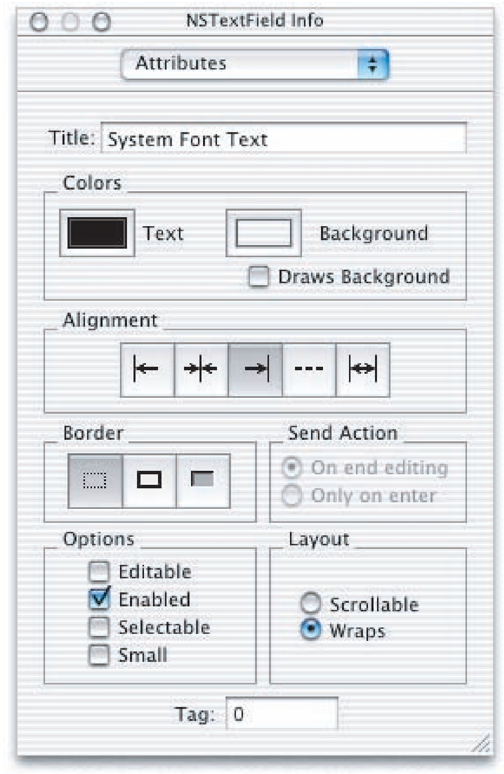
1. Drag a System Font Text object onto the window from the Views palette.

Figure 3-8 Adding a text label in Interface Builder



2. Make the text right aligned; with the System Font Text object selected, click on the third button from the left in the Alignment area of the Info window.

Figure 3-9 Right-aligning a text label in Interface Builder



Title: System Font Text

[ ] Draws Background

Send Action: ( ) On end editing ( ) Only on enter

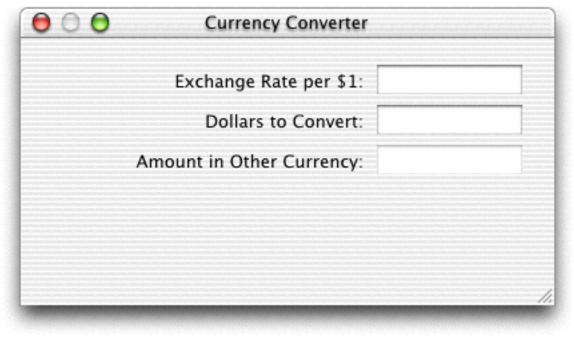
Options:
[ ] Editable
[✓ Enabled]
[ ] Selectable
[ ] Small

Layout: ( ) Scrollable [✓ Wraps]

Tag: 0

3. Duplicate the text label twice, enter the text for each, and align them as shown in Figure 3-10.

Figure 3-10 Aligned text fields and labels in Interface Builder



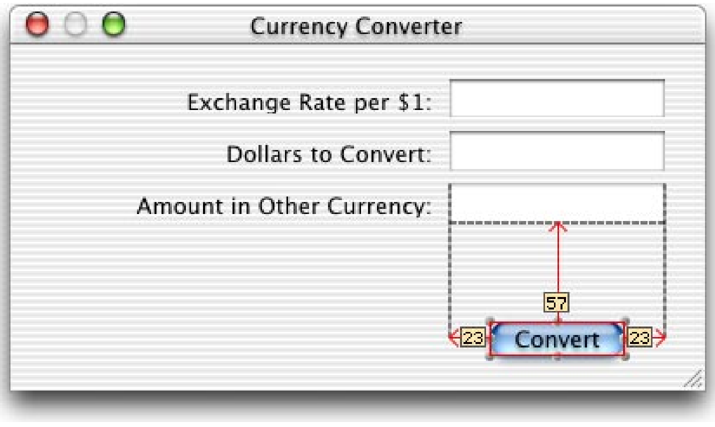
Add a Button to the Interface and Initialize It

The currency conversion can be invoked either by clicking a button or pressing Return.

1. Drag the button object from the Views palette and put it in the lower-right portion of the window.
2. Double click the title of the button to select its text label and change the title to “Convert”.
3. Choose Attributes in the NSButton Info Window, and then choose Return from the pop-up menu labeled “Equiv:”. This will give the button the capacity to respond to the Return key as well as to mouse clicks.
4. Align the button under the text fields.

   First, drag the button downward until the Aqua guide appears. With the button still selected, hold down the Option key. If you move the cursor around Interface Builder will show you the distance from the button to the object that you’ve indicated with the cursor. With the Option key still down and the cursor over the bottom text field, use the arrow keys to nudge the button to the exact center of the text fields.

Figure 3-11 Measuring distances in Interface Builder

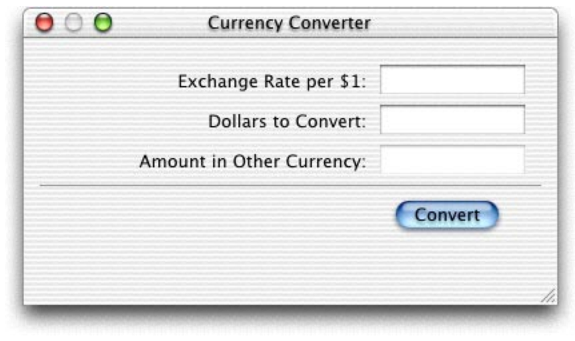


Add a Horizontal Decorative Line

You've probably noticed that the final interface for Currency Converter has a decorative line between the text fields and the button.

1. Drag a horizontal separator object from the Views palette onto the interface. It's located just beneath the radio button matrix object in the lower left hand corner of the Views palette.
2. Drag the endpoints of the line until the line extends across the window.

Figure 3-12 Adding elements to the Currency Converter window in Interface Builder



3. Move the Convert button back up until the Aqua guide appears.

Aqua Layout and Object Alignment

In order to make an attractive user interface, you must be able to visually align interface objects in rows and columns. “Eyeballing” the alignments can be very difficult, and typing in x/y coordinates by hand is tedious and time consuming. Aligning Aqua interface widgets is made even more difficult because the objects have shadows and UI guideline metrics don't typically take the shadows into account. Interface Builder uses visual guides and layout rectangles to help you with object alignment.

In Cocoa, all drawing is done within the bounds of an object’s frame. Because interface objects have shadows, they don't visually align correctly if you align the edges of the frames (as is done with Mac OS 9). For example, the Aqua UI guidelines say that a push button should be 20 pixels tall, but you actually need a frame of 32 pixels for both the button and its shadow. The layout rectangle is what you must align.

You can view the layout rectangles of objects in IB using “Show Layout Rectangles” in the Layout menu (Command-L). Also, the IB size pane has a pop-up to toggle between the frame and layout rectangle so you can set values by hand when appropriate.

Interface Builder gives you many ways to align objects in a window:

- Dragging objects with the mouse in conjunction with Aqua guides
- 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
- Using the built-in alignment functions
- Specifying origin points in the Size display of the Info window
- Using horizontal and/or vertical guides

Look in the Alignment and Guides sub-menus of the Layout menu for various alignment commands and tools. You can also use the alignment tool (choose Alignment in the Tools menu) which provides a floating window with buttons that perform various types of alignment.

Finalize the Window Layout

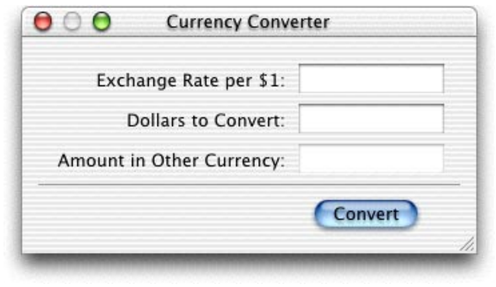
Currency Converter's interface is almost complete. The finishing touch is to resize the window so that all of the object are centered and properly aligned to each edge; currently the objects are only aligned to the top and right edge.

For Currency Converter, you will continue using the automated Aqua guides along with a few Layout commands.

1. Select the third text label "Amount in Other Currency", then extend the selection (Shift-click) to include the other two.
2. Resize all the labels to their minimum width by choosing "Size to Fit" in the layout menu.
3. Select "Same Size" from the Layout menu to make the selected text labels the same size.
4. Drag the labels towards the left edge of the window, and release them when the Aqua guide appears.
5. Select all three text fields and drag them to the left, again using the guides to help you find the proper position.
6. Shorten the horizontal separator and move the button into position again under the text fields.
7. Resize the window using the guides to give you the proper distance from the text fields on the right and the Convert button on the bottom.

At this point the application's window should look like Figure 3-13.

Figure 3-13 Currency Converter's final user interface in Interface Builder

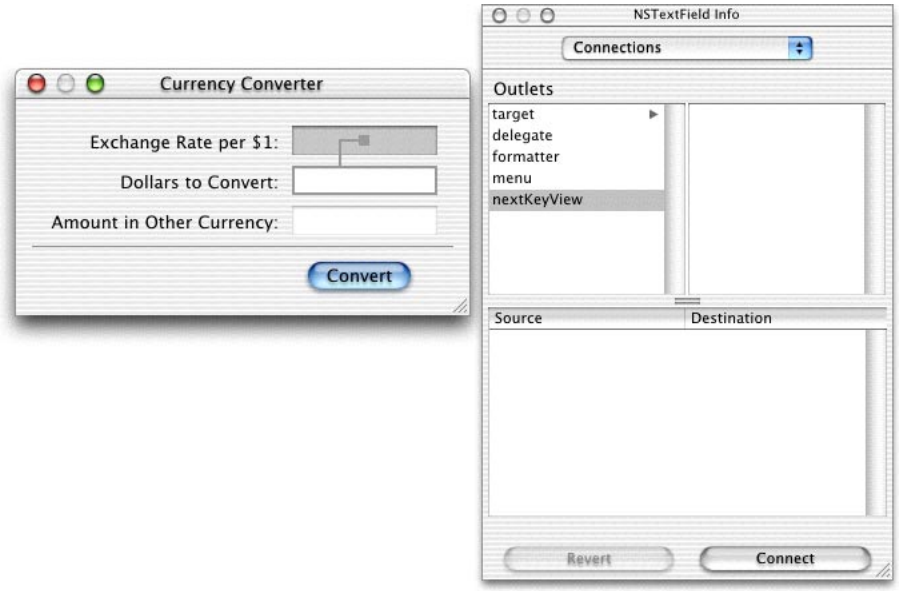


Enable Tabbing Between Text Fields

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

1. Select the first text field.
2. Control-drag a connection line from it to the second text field.

Figure 3-14 Connecting `nextKeyView` outlets in Interface Builder



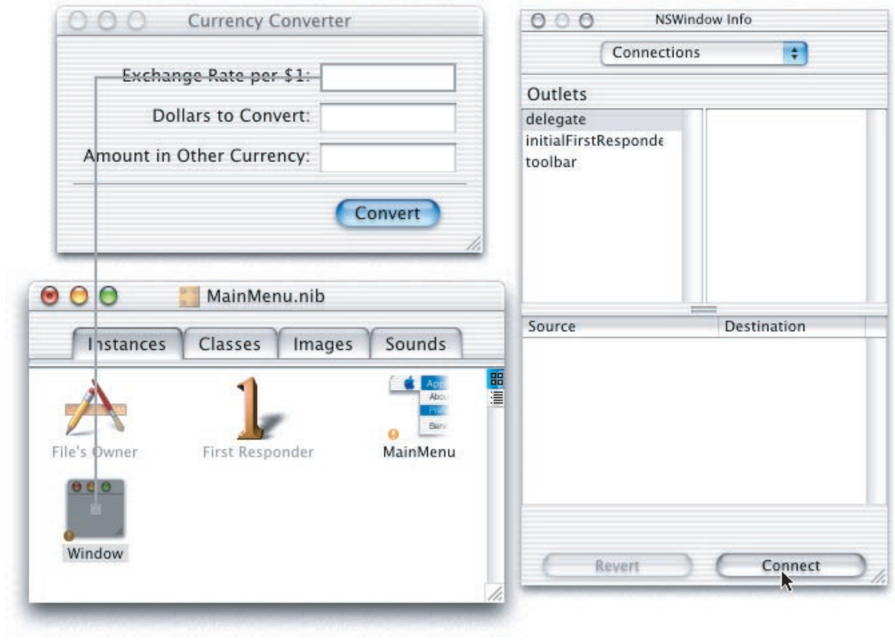
3. In the Info window, select `nextKeyView` and click Connect. The `nextKeyView` outlet identifies the next object to respond to events after the Tab key is pressed.
4. Repeat the same procedure, going from the second field to the first.

Set the initialFirstResponder for the Window

In the previous section, you set up the key view loop using Interface Builder, establishing connections between the `nextKeyView` outlets of the two text fields. Now you must set the window's `initialFirstResponder` outlet to the text field that you want selected when the window is first placed on screen. If you do not set this outlet, the window sets a key loop (not necessarily the same as the one you would have specified!) and picks a default initial first responder for you.

1. Control-drag a connection line from the Window instance to the first text field.

Figure 3-15 Setting the `initialFirstResponder` outlet in Interface Builder



2. In the Info window, select `initialFirstResponder` and click Connect.

Test the Interface

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

1. Choose File > Save All to save your work
2. Choose File > Test Interface.
3. Try various operations in the interface, such as tabbing, and cutting and pasting between text fields.
4. When finished, choose Quit NewApplication from the Interface Builder application menu to exit test mode.

Notice that the screen position of the Currency Converter window in Interface Builder is used as the initial position for the window when the application is launched. Place the window near the top left corner of the screen so it will be in a convenient (and traditional) initial location.

For Free with Cocoa

The simplest Cocoa 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.

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. Then move the window around by its title bar. Here are some other tests you can make:

1. Click the Edit menu. Its items appear and disappear when you release the mouse button, as with any application menu.
2. Click the miniaturize button. Click the window's icon in the Dock get the application back.
3. 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 hadn't configured Currency Converter's window in Interface Builder to remove the resize box, we could 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. Notice that the Convert button "throbs" as is the default for Aqua buttons associated with the Return key. Click the Convert button. Notice how the button is highlighted momentarily.

If you had buttons of a different style 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.

Menu Commands

Interface Builder gives every new application a default main menu that includes the Application, File, Edit, Window, and Help menus. Some of these menus, such as Edit, contain ready-made sets of commands. For example, with the Services sub-menu (whose items are added by other applications at run time) you can communicate with other Cocoa applications, and with the Window 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 text view objects, like the one in the scroll view object in the Data Views palette. Your application gets the Font window and a font manager "for free."
- The Text submenu allows you to align text anywhere text is editable, 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 PDF data.

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.

File Management

An application can use the Open dialog, 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 dialog to save information in files. Cocoa 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

Cocoa 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

Cocoa lets you create your own custom views that draw their own content and respond to user actions. To assist you in this, Cocoa provides objects and functions for drawing such as the NSBezierPath class.

Localization

Cocoa 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 window (and font management), the color picker (and color management), and, 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, Cocoa automates simple printing of views that contain text or graphics. When a user clicks the control, an appropriate dialog 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 very easily create context-sensitive help—knows as "tool tips"—for your application using the Info window in Interface Builder. After you've entered the tool tip text, the user can then hold the cursor over an object on the application's interface and a small window will appear containing concise information on the object.

Plug-in Architecture

You can design your application 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.

There are three steps to define the classes of Currency Converter:

"Specify a Subclass" (page 44)

"Define the Outlets of the Class" (page 48)

"Define the Actions of the Class" (page 50)

Classes and Objects

To newcomers, 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 software 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.

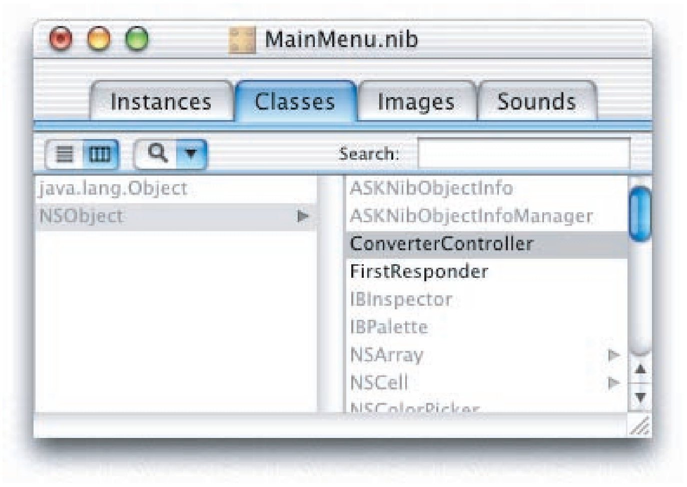
See the appendix, "Object-Oriented Programming," for more on these and other aspects of classes.

Specify a Subclass

You must go to the Classes display of the nib file window to define a class. Let's start with the ConverterController class.

1. In Interface Builder, select the Classes display of the `MainMenu.nib` window.
2. In the leftmost column of the pane, click on NSObject and hit Return to create an NSObject subclass called "MyObject." Type "ConverterController" to rename it.

Figure 3-16 Subclassing NSObject



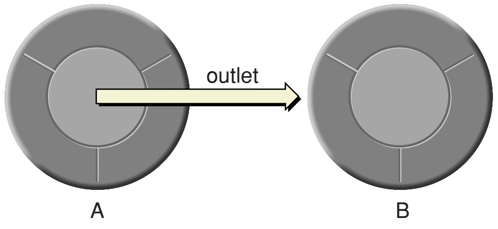
Paths for Object Communication: Outlets, Targets, and Actions

In Interface Builder, you specify the paths for messages travelling between the ConverterController object and other objects as **outlets** and **actions**.

Outlets

An outlet is an instance variable that identifies an object.

Figure 3-17 An outlet pointing from one jelly donut to another



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 dialogs, instances of custom classes, and even the application object itself. What distinguishes outlets is their relationship to Interface Builder.

Outlets are declared as:

```
IBOutlet 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.

When you don't need a dynamically typed object, you can—and should—statically type it as a pointer to an object:

```
IBOutlet NSButton *myButton;
```

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.

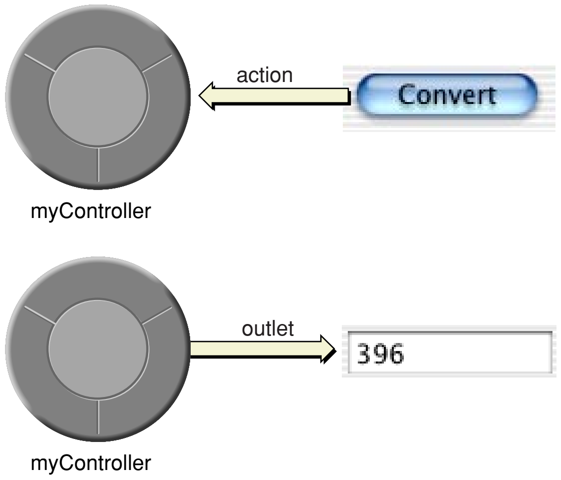
It might help to understand connections by imagining an electrical outlet embedded in the destination object. Also picture an electrical cord extending from the outlet in the source object. Before the connection is made the cord is unplugged and the value of the outlet is undefined; after the connection 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

As you'll soon find out, you can view (and complete) target/action connections in Interface Builder's Connections Info window. This interface is easy to use, but the relation of target and action in it might not be apparent. First, a 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.

Figure 3-18 Relationships in the target-action paradigm



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 Info window 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 simple rule to know which way to draw a connection line in Interface Builder. Draw the connection in the direction that messages will flow:

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

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

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.

Define the Outlets of the Class

1. Type the name of the method, `convert`, and hit Return. IB adds the “:” for you.

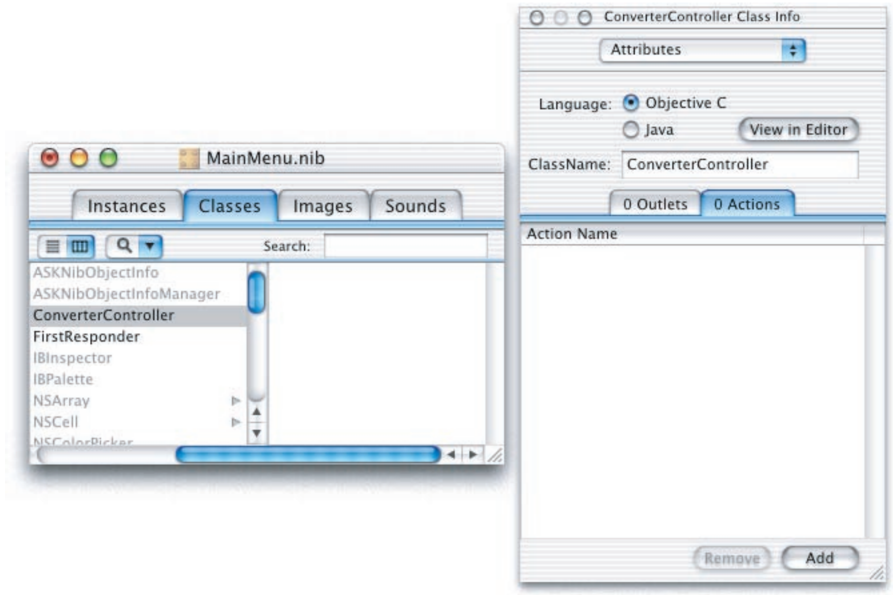
1. Select ConverterController in the Classes window.
2. Choose Add Outlet from the Classes menu, or:

a. Make sure that the Info window is visible and the Attributes pane is selected.

b. Make sure that the tab labeled "0 Outlets" is active.

c. Click the Add button.

Figure 3-19 Outlets and actions in the Interface Builder Info window



3. Name this outlet `rateField` and press Return.
4. Since the `rateField` outlet is still selected, all you have to do to create more outlets is press Return. Do this once to create the `dollarField` outlet, and again for the `totalField` outlet.

Notice the Type column in the table of outlets. By default, the type of outlets is set to `id`. It works to leave it as `id`, since Objective-C is a dynamically typed language. However, it's a good idea to get into the habit of setting the types for outlets, since statically typed instance variables receive much better compile-time error checking. Change the type of the three outlets to NSTextField by selecting it from the combo box lists currently set to `id`.

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.

Define the Actions of the Class

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. 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.

1. Choose Add Action from the Classes menu, or:
    a. Make sure that the Info window is visible and the Attributes pane is selected.
    b. Make sure that the tab labeled "0 Actions" is active.
    c. Click the Add button.
2. Type the name of the method, `convert`, and hit Return. IB adds the ":" for you.

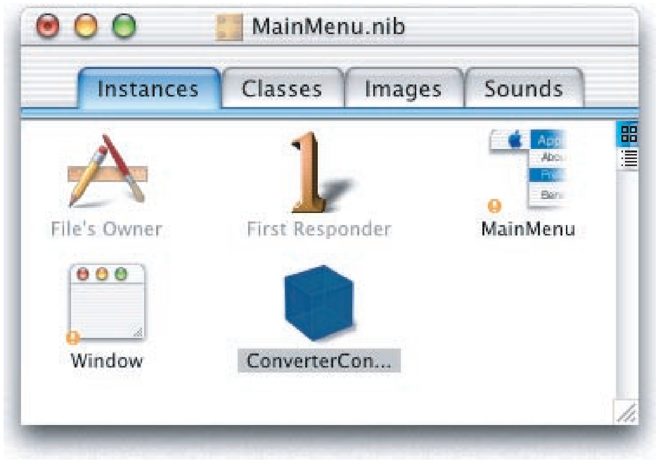
Connecting ConverterController to the Interface

Generate an Instance of the Class

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. Select ConverterController in the Classes window if it is not already selected.
2. Choose Instantiate from the Classes menu. The instance will appear in the Instances view as shown highlighted in Figure 3-20.

Figure 3-20 A newly-instantiated ConverterController object in Interface Builder

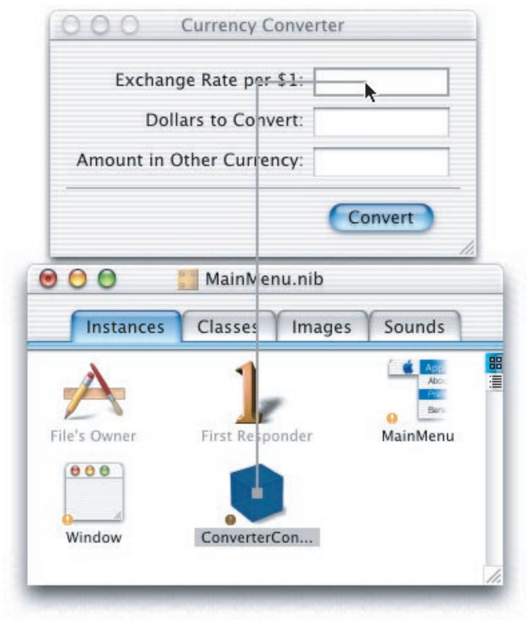


Connect the Custom Class to the Interface

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

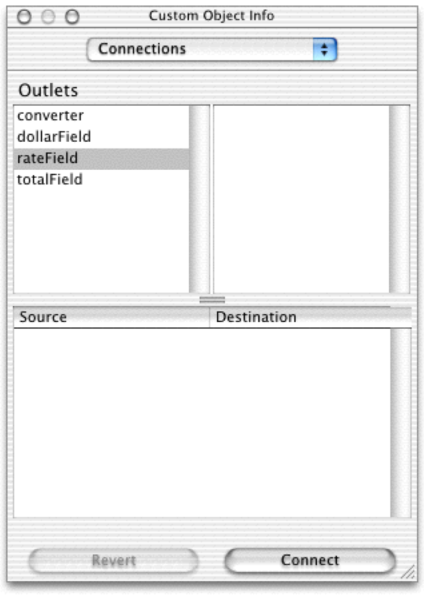
1. In the Instances display of the nib file window, Control-drag a connection line from the ConverterController instance to the first text field, as shown in Figure 3-21 (page 52). When the text field is outlined, release the mouse button.

Figure 3-21 Connecting the `rateField` outlet in Interface Builder



2. Interface Builder brings up the Connections display of the Info window. Select the outlet that corresponds to the first field (`rateField`).

Figure 3-22 Choosing the `rateField` outlet in the Info window



3. Click the Connect button.
4. Following the same steps, connect ConverterController's `dollarField` and `totalField` outlets to the appropriate text fields.

Connect the Interface Controls to the Class's Actions

1. Control-drag a connection from the Convert button to the ConverterController instance in the nib file window. When the instance is outlined, release the mouse button.
2. In the Connections display, make sure `target` in the Outlets column is selected.
3. Select `convert` in the Actions column.
4. Click the Connect button.
5. Save the nib file.

Define the Converter Class

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. Hint: Control-drag from the ConverterController instance to the Converter instance.
4. Save `MainMenu.nib`.

Implementing the Classes of Currency Converter

The final step in building the Currency Converter application is to implement the classes you defined in the previous steps.

Generate the Source Files

1. Go to the Classes display of the nib file window.
2. Select the ConverterController class.
3. Choose Create Files in the Classes menu.
4. Verify that the check boxes in the Create column next to the `.h` and `.m` files are selected.
5. Verify that the check box next to "Insert into Currency Converter" is selected.
6. Click the Choose button.

7. Repeat for the Converter class.
8. Save the nib file.

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 with Objective-C code.

The following summarizes some of the more basic aspects of the language. See *Inside Mac OS X: The Objective-C Programming Language* for additional details.

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];
```

As in standard C, terminate statements with a semicolon.

Messages often result in values being 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 Cocoa 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' instances. However, accessor methods are recommended instead of direct access, except in cases where performance is of paramount importance.

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 and objects in this way if they 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 (`-`), a space after the minus sign is optional.

```
- (NSString *)countryName;
```

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 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.

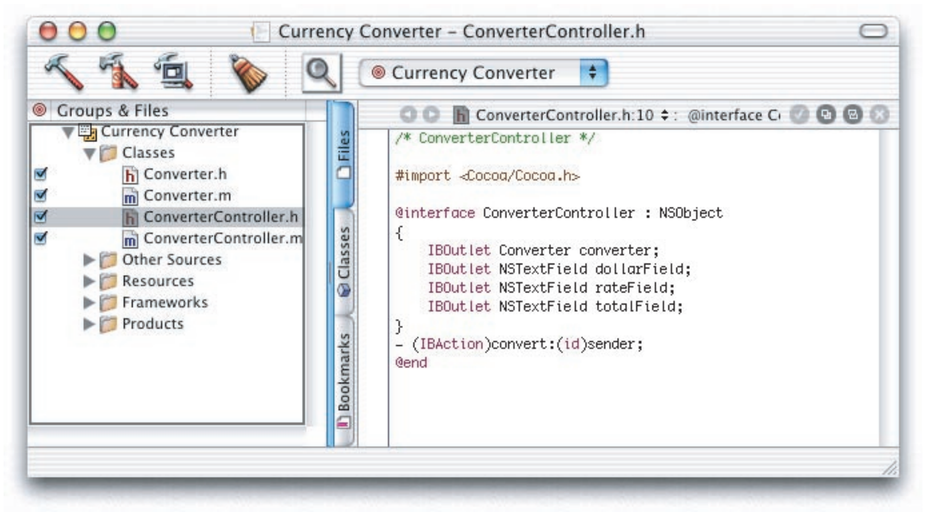
By default, the scope of an instance variable is protected, making that variable directly accessible only to objects of the class that declares it or of a subclass of that class. To make an instance variable private (accessible only within the declaring class), insert the @private directive before the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

Examine an Interface (header) File in Project Builder

When Interface Builder adds the header and source files to the Currency Converter project, the source files appear in whatever group was selected when the files were added from Interface Builder. Since these files are class implementations, add them to the Classes group.

1. Click Project Builder's main window to activate it.
2. Select all four files in the project browser and drag them into the Classes group.

Figure 3-23 Source files in Currency Converter's Classes group



```
/* ConverterController */

#import <Cocoa/Cocoa.h>

@interface ConverterController : NSObject
{
    IBOutlet Converter converter;
    IBOutlet NSTextField dollarField;
    IBOutlet NSTextField rateField;
    IBOutlet NSTextField totalField;
}
- (IBAction)convert:(id)sender;
@end
```

Add a method declaration

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. Select `Converter.h` in the project browser.
2. Insert a declaration for `convertAmount:atRate:`.

```
#import <Cocoa/Cocoa.h>
@interface Converter:NSObject
{
}
- (float)convertAmount:(float)amt atRate:(float)rate;

@end
```

This declaration states that `convertAmount:atRate:` takes two arguments of type `float`, and returns a `float` value. When parts of a method name have colons, such as `convertAmount:` and `atRate:`, they are keywords which introduce arguments. (These are keywords in a sense different from keywords in the "C" language.)

Now you need to update both implementation files.

Implement Currency Converter's Classes.

For the Converter class, implement the method you just declared in `Converter.h`. Method implementations go between `@implementation <class name>` and `@end` so this is where you will add the code for Converter.

1. Select `Converter.m` from the Classes group in Project Builder's main window.
2. Insert the code in for `convertAmount:`

```
#import "Converter.h"
@implementation Converter
- (float)convertAmount:(float)amt atRate:(float)rate
{
    return (amt * rate);
}
@end
```

The method simply multiplies the two arguments and returns the result. Simple enough.

1. Next, update the "empty" implementation of the `convert:` method in `ConverterController.m` that Interface Builder generated for you.

```
- (IBAction)convert:(id)sender
{
    float rate, amt, total;

    amt = [dollarField floatValue];
    rate = [rateField floatValue];

    total = [converter convertAmount:amt atRate:rate];

    [totalField setFloatValue:total];
    [rateField selectText:self];
    }
```

2. Make sure that `ConverterController.m` imports `Converter.h` by adding the following line at the top of the source file.

```
    #import "Converter.h."
```

The `convert:` method does the following:

- Gets the floating-point values typed into the rate and dollar-amount fields.
- Invokes the `convertAmount:atRate:` method and gets the returned value.
- Uses `setFloatValue:` to write the returned value in the Amount in Other Currency text field (`totalField`).
- 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.

Each line of the `convert:` method, 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.

Implement the awakeFromNib method

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. Add the following code to `ConverterController.m`.

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

The `makeKeyAndOrderFront:` message does as it says: It makes the receiving window the key window (the window that receives input from the keyboard) 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.

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.

Turbo Coding With Project Builder

When you write code with Project Builder you have a set of "workbench" tools at your disposal, among them:

Project Find

Project Find (available from the Find pane in Project Builder) allows you to search both your project's code and the system headers for identifiers. Project Find uses a project index that stores all of a project's identifiers (classes, methods, globals, etc.) on disk.

For C-based languages, Project Builder automatically gathers indexing information while the source files are being compiled, so it's necessary to build the project to create the index before you can use Project Find.

Integrated Documentation Viewing

Project Builder supports viewing HTML documentation directly in the application. You can access documentation and release notes about Project Builder, other developer tools, Carbon, Cocoa, AppleScript Studio, and even access UNIX man pages.

Additionally, you can jump directly from fully or partially completed identifiers in your code to developer documentation and header files. To retrieve the HTML documentation for an identifier, Option-double-click it, and to retrieve its declaration in a header file, Command-double-click it.

Indentation

In Preferences you can set the characters at which indentation automatically occurs, the number of spaces per indentation, and other global indentation characteristics. The Edit menu includes the Indentation submenu, which allows you to indent lines or blocks of code on a case-by-case basis.

Delimiter Checking

Double-click a brace (left or right, it doesn't matter) to locate the matching brace; the code between the braces is highlighted. In a similar fashion, double-click a square bracket in a message expression to locate the matching bracket and double-click a parenthesis character to highlight the code enclosed by the parentheses. If there is no matching delimiter, Project Builder emits a warning beep.

Emacs Bindings

You can use the most common Emacs commands in Project Builder's code editor. (Emacs is a popular editor for writing code.) For example, there are the commands page-forward (Control-v), word-forward (Meta-f), delete-word (Meta-d), kill-forward (Control-k), and yank from kill ring (Control-y).

Some Emacs commands may conflict with some of the standard Macintosh key bindings. You can modify the key bindings the code editor uses to substitute other "command" keys—such as the Option key or Shift-Control— for Emacs' Control or Meta keys. For information on key bindings, see the concept "About Key Bindings" in "Programming Topic: Text Input Management," in the Cocoa developer documentation.

Build, Debug, and Run Currency Converter

This section explains how to build, debug, and run the application.

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 jam, 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 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.

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 package. An application package is a directory that contains the application executable and the resources needed by that executable. This directory appears as a single file in the Finder that can be double-clicked to launch the application.

Build the Project

You begin builds by clicking the Build button.

1. Save source code files and any changes to the project.
2. Click the Build button in the Project Builder window, as shown in Figure 3-24.

Figure 3-24 Project Builder's Build button

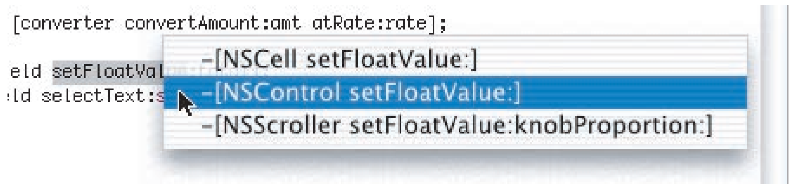
When you click the Build button, the build process begins. When Project Builder finishes—and encounters no errors along the way—it displays "Build succeeded" in the lower left corner of the window.

Look Up Documentation

When Project Builder builds a project, it also generates an index of the source files, included frameworks, and associated documentation. You can jump directly to documentation and header files within Project Builder. Try it out:

1. Return to `ConverterController.m`.
2. Option-double-click the word `setFloatValue` in the code. (Hold down the Option key and double-click the word.) A pop-up menu appears with method names, such as `-[NSControl setFloatValue:]`.

Figure 3-25 The documentation pop-up menu



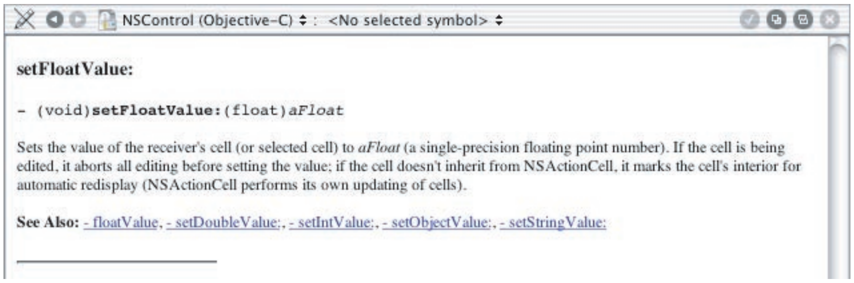
```
[converter convertAmount:amt atRate:rate];

eld setFloatVal
ld selectText:s
```

-[NSCell setFloatValue:]
-[NSControl setFloatValue:]
-[NSScroller setFloatValue:knobProportion:]

3. Select the method associated with NSControl. Project Builder opens an HTML-formatted copy of the documentation right in the editor pane, and jumps to a description of the method you requested.

Figure 3-26 Documentation in Project Builder


NSControl (Objective-C) : <No selected symbol>

setFloatValue:

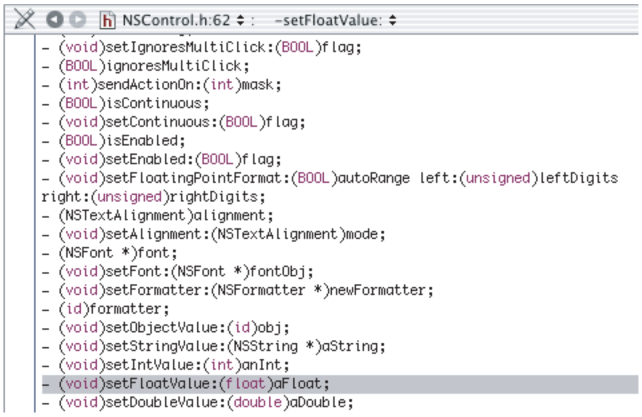
`- (void)setFloatValue:(float)aFloat`

Sets the value of the receiver's cell (or selected cell) to *aFloat* (a single-precision floating point number). If the cell is being edited, it aborts all editing before setting the value; if the cell doesn't inherit from NSActionCell, it marks the cell's interior for automatic redisplay (NSActionCell performs its own updating of cells).

See Also: - floatValue, - setDoubleValue:, - setIntValue:, - setObjectValue:, - setStringValue:

4. Click the Back button (the left arrow at the top of the editor pane) to return to your code.
5. Command-double-click the same word. A similar pop-up menu appears.
6. Again select the `-[NSControl setFloatValue:]` method. This time, Project Builder jumps to the method declaration in the associated header file.

Figure 3-27 Header lookup in Project Builder


NSControl.h:62 : -setFloatValue:

```
- (void)setIgnoresMultiClick:(BOOL)flag;
- (BOOL)ignoresMultiClick;
- (int)sendActionOn:(int)mask;
- (BOOL)isContinuous;
- (void)setContinuous:(BOOL)flag;
- (BOOL)isEnabled;
- (void)setEnabled:(BOOL)flag;
- (void)setFloatingPointFormat:(BOOL)autoRange left:(unsigned)leftDigits
right:(unsigned)rightDigits;
- (NSTextAlignment)alignment;
- (void)setAlignment:(NSTextAlignment)mode;
- (NSFont *)font;
- (void)setFont:(NSFont *)fontObj;
- (void)setFormatter:(NSFormatter *)newFormatter;
- (id)formatter;
- (void)setObjectValue:(id)obj;
- (void)setStringValue:(NSString *)aString;
- (void)setIntValue:(int)anInt;
- (void)setFloatValue:(float)aFloat;
- (void)setDoubleValue:(double)aDouble;
```

7. Click the Back button.

If you cannot get the pop-up menus to appear, the project is probably not indexed. To index the project, build it.

Debug the Project

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.

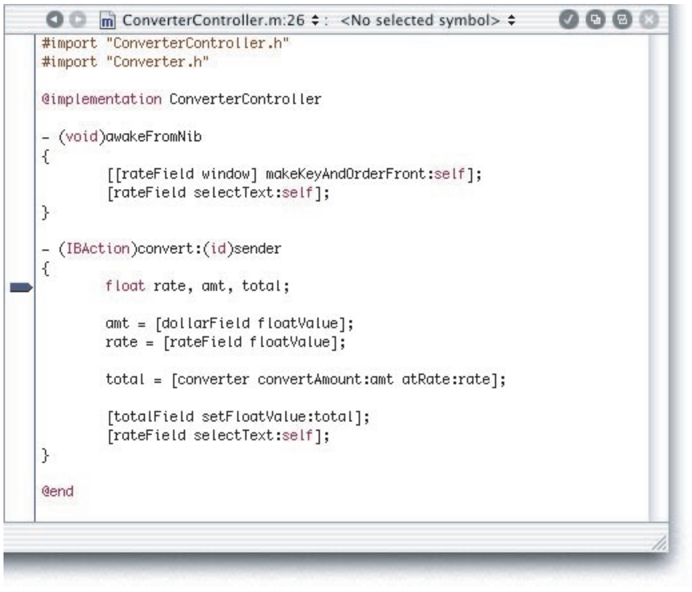
1. Delete a semicolon in the code, creating an error.
2. Click the Build button on the Project Build panel.
3. Click the error-notification line that appears in the build error browser.
4. Fix the error in the code.
5. Re-build the project.

Using the Graphical Debugger

To smooth the task of debugging, Project Builder puts a graphical user interface over the GNU debugger, GDB. To help familiarize you with the debugger (not every step is likely to go so smoothly as the currency conversion!) use it now to step through the currency conversion code.

1. Click `ConverterController.m` in Project Builder's Groups & Files view.
2. Locate the `convert:` method.
3. Set a breakpoint by clicking in the column to the left of the code listing.

Figure 3-28 Setting a breakpoint in Project Builder



```
#import "ConverterController.h"
#import "Converter.h"

@implementation ConverterController

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

- (IBAction)convert:(id)sender
{
        float rate, amt, total;

        amt = [dollarField floatValue];
        rate = [rateField floatValue];

        total = [converter convertAmount:amt atRate:rate];

        [totalField setFloatValue:total];
        [rateField selectText:self];
}

@end
```

4. Run the debugger by clicking the Debug button, or type Command-R.
5. Test the conversion by entering values and clicking the Convert button.
6. When you click Convert, the debugger will activate and allow you to step through the code line by line and examine the results of each step.

To perform complex debugging tasks, you can use the GDB console. Click the Console tab at the top right-hand corner of the screen to expose the Console. When you click in the Console window and press Return, the `(gdb)` prompt appears.

There are many GDB commands that you can type at this prompt, which are not represented in the user interface. For on-line information on these commands, enter "help" at the prompt.

Run Currency Converter

Enter some rates and dollar amounts and click Convert. Also, select the text in a field and choose Services sub-menu in the Application 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 consolidates all of the concepts and techniques introduced in the previous chapter. By now you have a much better grasp of the skills you’ll need to develop Cocoa 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

C H A P T E R 4

Cocoa Resources

This chapter describes a number of resources that will help you on the road to learning Cocoa: developer documentation, websites, email lists, and books.

Cocoa Developer Documentation

Apple's Cocoa Developer Documentation includes a reference for the complete Cocoa API. It also includes Programming Topics, which contain conceptual material and specific tasks in a variety of programming domains. Also see the "Getting Started" section for material targeted at those new to Cocoa programming.

You can access the Cocoa Developer Documentation in a number of ways:

- Choose Cocoa Help from Project Builder's Help menu, or from Mac Help's Developer Help Center.
- Navigate to `/Developer/Documentation/Cocoa` on your hard disk.
- Visit http://developer.apple.com/techpubs/macosx/Cocoa/CocoaTopics.html with your web browser for the latest version of the documentation.
- Use Project Builder's Find pane to search through developer documentation and header files, as well as identifiers in your source files. The Find pane is available in the main window in Single Window mode, or by choosing Show Batch Find from the Find menu.
- In a Project Builder editor pane, option-double-click a identifier to retrieve documentation, or command-double-click it to retrieve it associated header file.

You can also browse frameworks in your project directly from Project Builder. For Cocoa developers, browse Foundation.framework > Headers and AppKit.framework > Headers inside Frameworks > Other Frameworks, located in your target's group in the Groups & Files pane.

Developer Tools Help

Apple provides help documentation for the majority of the developer tools in Mac OS X, including Project Builder, Interface Builder, command-line tools, and others.

For help with Project Builder, choose Project Builder Help from Project Builder's Help menu.

For help with Interface Builder, choose Interface Builder Help from Interface Builder's Help menu.

For all developer tools, there are a number of places you can go for help:

- Choose Developer Tools Help from Project Builder's Help menu, or from Mac Help's Developer Help Center.
- Navigate to `/Developer/Documentation/DeveloperTools` on your hard disk.
- Visit http://developer.apple.com/techpubs/macosx/DeveloperTools/devtools.html for the latest version of the documentation.

Additionally, Project Builder and Interface Builder provide help tags, also known as tool tips, to describe parts of the interface when the mouse arrow hovers over those parts.

Mailing Lists

Apple supports many public mailing lists to let developers and user communicate about various topics. The cocoa-dev mailing list is a great resource for Cocoa developers, from beginners to experts. The studentdev list is a place for students to discuss all facets of development with Apple products, including Cocoa.

You can sign up for mailing lists and browse list archives at the following address:

http://lists.apple.com

Books

O'Reilly & Associates has published two books on Cocoa development, one authored by Apple:

- *Learning Cocoa*, by Apple Computer, Inc.
- *Building Cocoa Applications: A Step-by-Step Guide*, by Simson Garfinkel and Michael K. Mahoney

Several other books on Cocoa development have been written, and more will be published in the future, so visit the programming section of your favorite bookstore.

Other Resources

- The Apple Developer Connection (ADC) provides many useful technical and business services for Cocoa developers. See http://developer.apple.com for developer documentation, sample code, information about developer programs, and business-related support.

- There are several third-party websites devoted to Cocoa development. You can search Google to find many Cocoa resources on the Web:

  http://www.google.com/search?hl=en&lr=&q=cocoa+programming