Cocoa Application Tutorial

2006-11-07

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures and Listings

Introduction to Cocoa Application Tutorial

This document introduces the Cocoa application environment using the Objective-C language and teaches you how to use the Xcode Tools development suite to build robust, object-oriented applications. Cocoa provides the best way to build modern, multimedia-rich, object-oriented applications for consumers and enterprise customers alike. This document assumes you are familiar with C programming but does not assume you have previous experience with Cocoa or Xcode Tools.

The Xcode Tools package is part of the Mac OS X installation media. You must install this package on your computer before following the instructions in this document. Once installed, you can get further information in *About Xcode Tools* in `/Developer`.

Important: This document is targeted for Mac OS X v10.4 and later, and Xcode Tools 2.2 and later. To find out which version of Xcode Tools is installed on your computer, launch Xcode (`/Developer/Applications`) and choose About Xcode from the Xcode application menu.

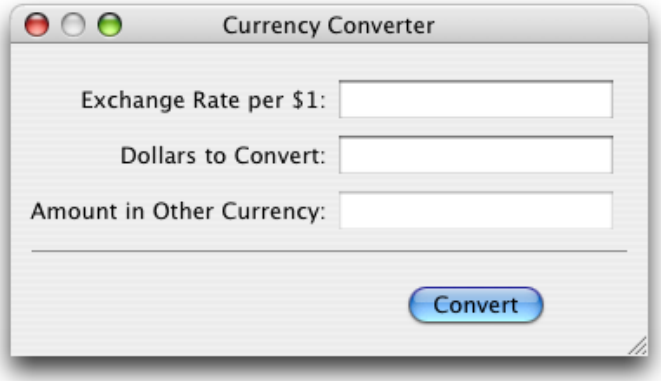
This document is intended for programmers interested in developing Cocoa applications or people curious about Cocoa.

This document shows how to build Currency Converter, an application that converts a dollar amount to an amount in another currency, given the rate of that currency relative to the dollar. You may try this tutorial using a development environment different from the one specified earlier. However, you may need to adjust to user interface and implementation discrepancies.

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

The main window of the finished application is shown in Figure I-1

Figure I-1 The Currency Converter main window



This document leads you through the basic steps for creating a Cocoa application. It shows how to:

- Create an Xcode 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 user interface
- Build an application and correct problems

By following the instructions provided in this document, you familiarize yourself with the two most important applications used for developing Mac OS X applications: Interface Builder and Xcode. You also learn the typical workflow of Cocoa application development:

1. Designing the application (your brain)
2. Creating the project (Xcode)
3. Creating the user interface (Interface Builder)
4. Defining the classes that implement the application's functionality (Interface Builder)
5. Implementing the application's functionality (Xcode)
6. Building the application (Xcode)
7. Running and testing the application (Xcode)

Note: To help you troubleshoot problems as you follow the tutorials, this document includes the finalized Currency Converter project as a companion archive (`ObjCTutorial_companion.zip`). The archive also contains files needed to follow some of the instructions in this document.

Organization of This Document

This document consists of the following chapters:

- "The Essence of Cocoa" (page 13) introduces basic concepts whose understanding is required when developing Cocoa applications.
- "Creating the Currency Converter Project and User Interface" (page 17) guides you through the development of the Currency Converter user interface.
- "Implementing Currency Converter" (page 47) shows how to define the custom behavior of the application.
- "Building Currency Converter" (page 53) explains how to build the application.
- "Configuring Currency Converter" (page 57) describes the basic identifying properties that application bundles require, including version information and icon, and explains how to configure them in an application.
- "Expanding on the Basics" (page 71) explains some of the behavior Cocoa applications get by default.

This document also contains the appendix "Objective-C Quick Reference" (page 77)—which summarizes some of the basic aspects of the Objective-C language—and a revision history.

See Also

These documents provide detailed information on Cocoa development:

- *Getting Started with Cocoa* provides a road map for learning Cocoa.
- *Cocoa Fundamentals Guide* describes the Cocoa application environment.
- *The Objective-C Programming Language* introduces Objective-C and describes the Objective-C runtime system, which leads to much of Cocoa's dynamic behavior and extensibility.
- *Apple Human Interface Guidelines* explains how to lay out user interface elements to provide a pleasant user experience.

The Essence of Cocoa

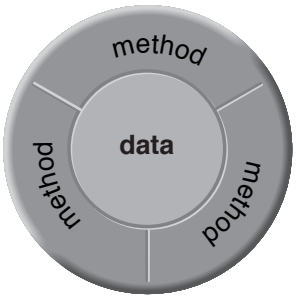
If you've done only procedural programming before, you may feel a bit intimidated reading about Cocoa and the concept of object-oriented programming. Even if you have done object-oriented design before, you may be wondering how that model fits into the world of Cocoa. This chapter covers the most common object-oriented design paradigm used in Cocoa and shows how that paradigm is applied to the Currency Converter application.

An object-oriented (OO) 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. The following sections describe important OO design concepts.

Object Notation

When designing an object-oriented application, it is often helpful to graphically depict the relationships between objects. This document depicts objects graphically as shown in Figure 1-1

Figure 1-1 An object



This representation illustrates data encapsulation, the essential characteristic of objects. An object consists of both data and methods for manipulating that data. Other objects or external code cannot access the object's data directly, but they request data from the object by sending messages to it. Read that sentence again, as it speaks from the very heart of OO development. *Other objects or external code cannot access the object's data directly, but they request data from the object by sending messages to it.* Your job is to make those objects talk to one another and share information through their methods.

An object's methods respond to messages and may return data to the object requesting it. An object's methods do the encapsulating, in effect regulating 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.

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.

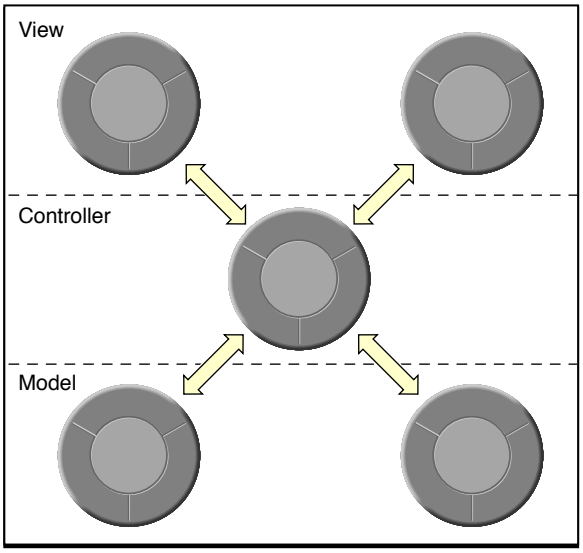
See *The Objective-C Programming Language* in Cocoa Objective-C Language Documentation for a fuller description of data encapsulation, messages, methods, and other things pertaining to object-oriented programming.

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, as illustrated in Figure 1-2

Figure 1-2 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—name, address, and phone number, for example—and has access to methods that can access and distribute that information. 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. Application Kit, one of the frameworks that compose Cocoa, 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 Objects

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. A Controller object, for example, could mediate the transfer of a street address (from our Customer model object) to a visible text field on a window (our View object). It also performs all the application-specific chores, such as loading nib files and acting as the 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 even make your View and Model objects available to other developers from a palette in Interface Builder.

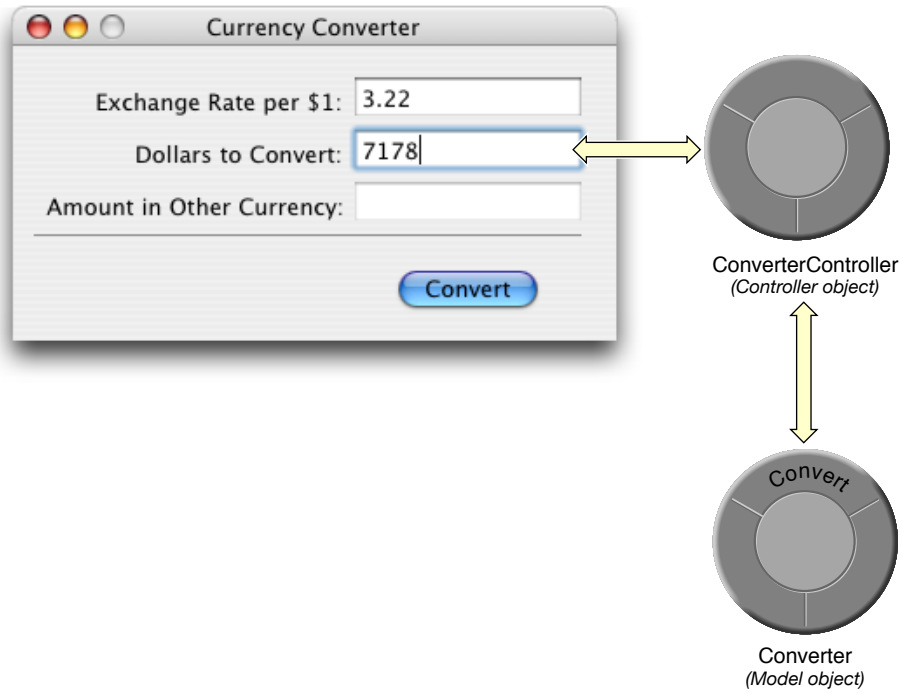
Hybrid Models

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 classes. 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`, which coordinates the activity between the `Converter` object and the user-interface elements. Figure 1-3 shows how `ConverterController` manages the user interface and the `Converter` model class.

Figure 1-3 Model-View-Controller relationships in Currency Converter



The `ConverterController` class assumes a central role. Like all controller objects, it communicates with interface elements and 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 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. By adhering to the MVC paradigm in this tutorial, however, you learn the fundamentals of good Cocoa design which assist you greatly as you begin to work on more advanced projects.

Creating the Currency Converter Project and User Interface

This chapter guides you through the development of the user interface of the Currency Converter application and teaches you the essential steps required for building a Cocoa application.

Important: Before continuing, make sure your development environment meets the requirements specified in "Introduction to Cocoa Application Tutorial Using Objective-C." (page 9)

Creating the Currency Converter Project

Every Cocoa application starts life 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 Xcode application to create and manage your project.

The following sections cover the steps necessary to create the Currency Converter project.

Open Xcode

To open Xcode:

1. In the Finder, go to `/Developer/Applications`.
2. Double-click the icon, shown in Figure 2-1

Figure 2-1 The Xcode application icon

The first time you start Xcode, it takes you through a quick set-up process. The default settings should work for the majority of users.

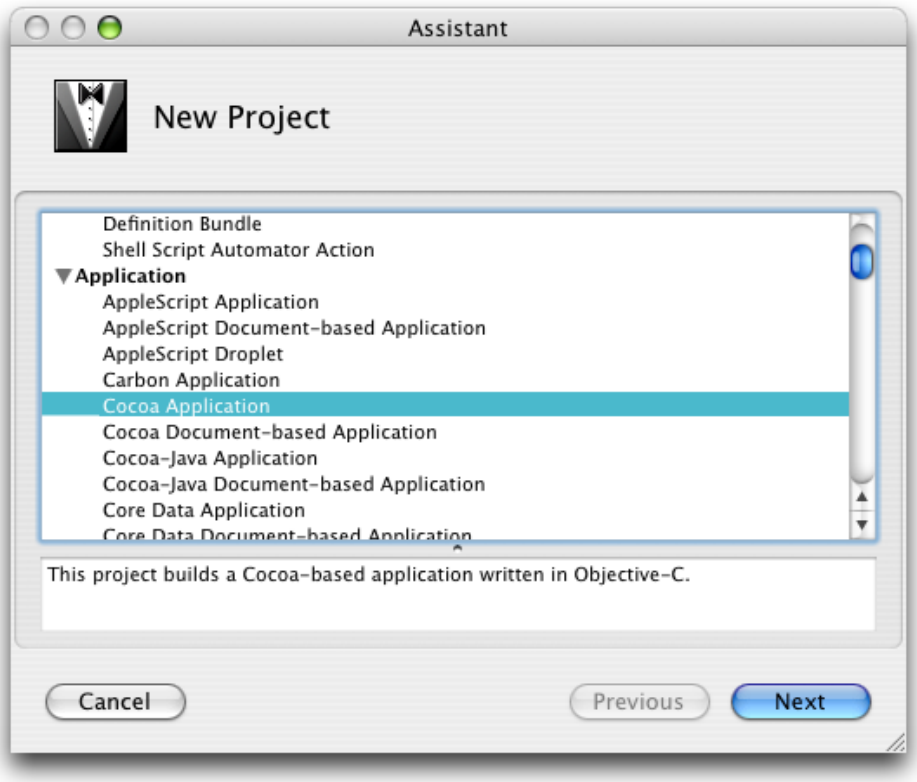
Choose the New Project Command

When Xcode is launched, only its menu bar appears. To create a project, choose New Project from the File menu. The New Project Assistant appears.

Choose a Project Type

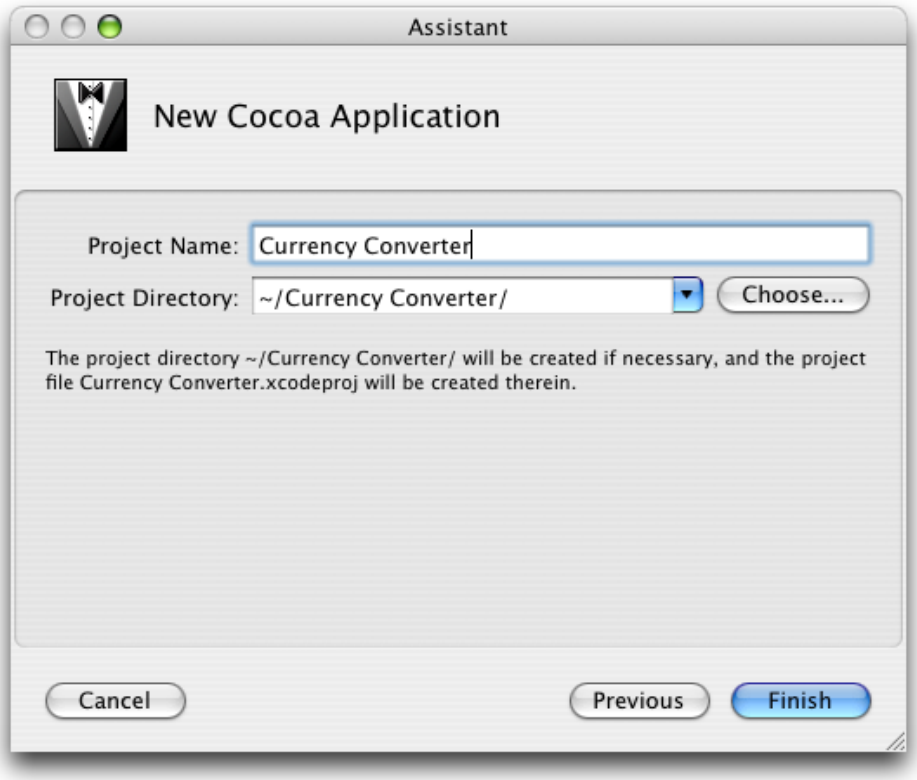
Xcode can build many different types of applications, including everything from Carbon and Cocoa applications to Mac OS X kernel extensions and Mac OS X frameworks. For this tutorial, select Cocoa Application and click Next, as shown in Figure 2-2

Figure 2-2 Xcode's New Project Assistant



1. Type `Currency Converter` in the Project Name field, as shown in Figure 2-3

Figure 2-3 Entering a project's name and choosing its location in Xcode New Project Assistant

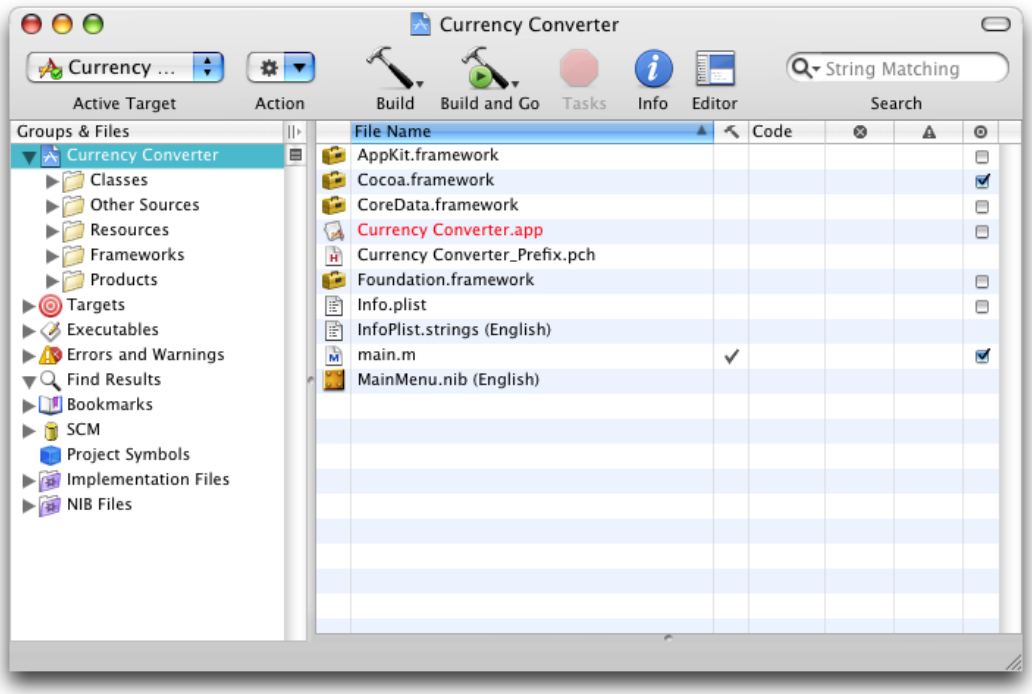


2. Click Choose to navigate to the directory where you want your project to be stored. The drop-down menu next to the Project Directory field eventually fills up with your frequently used directories. Use this to save time in the future.

3. Click Finish.

When you click Finish, Xcode creates the project's files and displays the project window, shown in Figure 2-4

Figure 2-4 The new Currency Converter project in Xcode.



The Groups & Files list is comprised of all the source files, images, and other resources that make up a project. These files are grouped in the project group, the first item in the Groups & Files list; this group is named after the project. The project's files are grouped into subgroups, such as Classes, Other Sources, Resources, and so on, as shown in Figure 2-4 These groups are very flexible in that they do not necessarily reflect either the on-disk layout of the project or the way the build system handles it. They are purely for organizing your project. The groups created by Xcode should be suitable for most developers, but you can rearrange them however you like.

These are the groups Xcode sets up for Cocoa applications:

Classes. This group is empty at first. However, you can place in it the classes required by your application.

Other Sources. This group contains `main.m`, which defines the `main` function that runs the application. (You shouldn't have to modify this file.) It also contains `Currency Converter_Prefix.pch`. This "prefix header" helps Xcode to reduce compilation time. This file is not important for this tutorial.

Resources. This group contains the nib files and other resources that specify the application's user interface. "What Is a Nib File?" (page 21) describes nib files.

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

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.

Below the project group are other groups, including **smart groups**. Smart groups—identified by the purple folders on the left side of the list—allow you to sort the project's files using custom rules in a way similar to using smart playlists in iTunes.

These are some of the other groups in the Groups & Files list:

Targets. Lists the end results of your builds. This group usually contains one target, such as an application or a framework, but it can consist of multiple items.

Executables. Contains the executable products your project creates.

Errors and Warnings. Displays the errors and warnings encountered in your project when you perform a build.

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

`CurrencyConverter.xcodeproj`
: This package contains information that defines the project. You should not modify it directly. You can open your project by double-clicking this package in the Finder.

`main.m`
: An Objective-C file, generated for each project, that contains the `main` function of the application.

`English.lproj`
: A directory containing resources localized to the English language. In this directory are nib files automatically created for the project. You may find other localized resource directories, such as `Dutch.lproj`.

Creating the Currency Converter Interface

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

What Is a Nib File?

Every Cocoa application with a graphical user interface has at least one nib file. The main nib file is loaded automatically when an application launches. It contains the menu bar and generally at 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 to another computer) 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 place into 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 your custom class information.
- **Connection Information.** Information about how objects within the class hierarchies are interconnected. Connector objects special to Interface Builder store this information. When you save a document, its connector objects are archived in the nib file along with the objects they connect.

Open the Main Nib File

You use Interface Builder to define an application's user interface. To open the Currency Converter's main nib file in Interface Builder:

1. Locate `MainMenu.nib` in the Resources subgroup of your project.
2. Double-click the nib file. This opens the nib file in Interface Builder.

A default menu bar and a window titled "Window" appear when the nib file is opened.

Windows in Cocoa

A window in Cocoa looks very similar to windows in other user environments such as Windows. 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, drawers, utility windows, and tool palettes, to name a few. In fact, anything drawn on the screen must appear in a window. Users, however, may not recognize or refer to them as "windows."

Two interacting systems create and manage Cocoa windows. A window is created by the Window Server. The Window Server is a process that uses the internal window management portion of Quartz (the low-level drawing system) to draw, resize, hide, and move windows using Quartz graphics routines. 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 framework (AppKit). The object supplied is an instance of the `NSWindow` class. Each physical window in a Cocoa program is managed by an instance of `NSWindow` or a subclass of it. For information on AppKit, see What Is Cocoa? in *Cocoa Fundamentals Guide*.

When you create an `NSWindow` object, the Window Server creates the physical window that the `NSWindow` object manages. The `NSWindow` class offers a number of instance methods through which you customize the operation of its onscreen window.

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` object 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 bottom-left corner of the frame rectangle as the origin for the base coordinate system, unlike Carbon and Classic applications, which use the top-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 onscreen or offscreen. Onscreen windows are "layered" on the screen in tiers managed by the Window Server. Onscreen 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

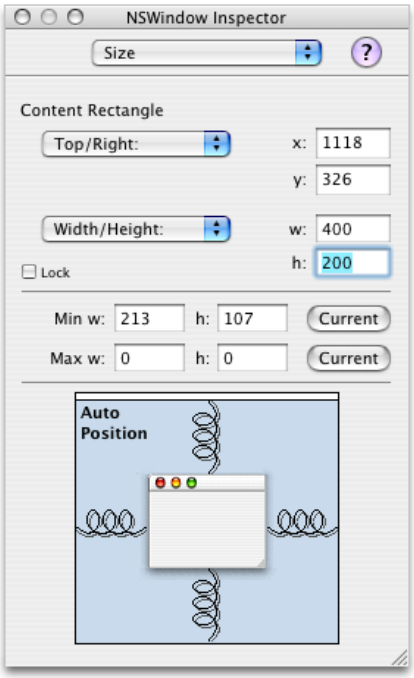
Make the window smaller by dragging the bottom-right corner of the window inward, as shown in Figure 2-5

Figure 2-5 Resizing a window manually

You can resize the window more precisely by using the Size menu of the `NSWindow` inspector.

1. Choose Show Inspector from the Tools menu.
2. Choose Size from the pop-up menu.
3. In the Content Rectangle group, choose Width/Height from the second pop-up menu.
4. Type `400` in the width (“w”) field and `200` in the height (“h”) field, as shown in Figure 2-6

Figure 2-6 Resizing a window with the `NSWindow` inspector



Set the Window's Title and Attributes

Set other attributes for the window in the `NSWindow` inspector:

1. Choose Attributes from the inspector pop-up menu and change the window's title to "Currency Converter". Press Return to lock in the change.
2. Verify that the "Visible at launch time" option is selected.
3. Deselect the Zoom option in the "Title bar controls" group.

Set the Application Name in the Menu

Interface Builder places the term "NewApplication" in place of the application name in the menu bar and throughout an application's menu hierarchy by default. You must change this text to the application name in all menu items that include the application name, such as the application menu and the Help menu.

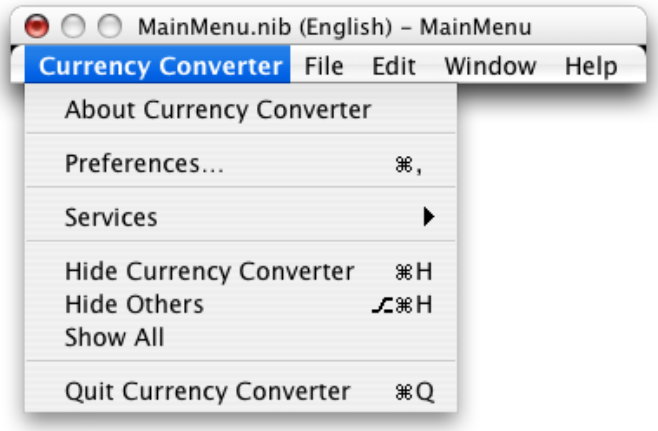
1. Rename the application menu:
    a. In the MainMenu window, double-click NewApplication, and press the Space bar.
    b. In the `NSMenuItem` inspector, enter `Currency Converter` in the Title text field and press Return.

    Important: At runtime, the title of the application menu is determined by the value of the application-name property (the value of the `CFBundleName` information property list key), not the title you specify in the nib file. See "Essential Application Identification Properties" (page 57) for details.

2. Modify items in the application menu:
    a. In the MainMenu window, click Currency Converter, double-click About NewApplication, and replace `NewApplication` with `Currency Converter`.
    b. Change Currency Converter > `Hide NewApplication` to `Hide Currency Converter`.

c. Change Currency Converter > `Quit NewApplication` to `Quit CurrencyConverter`. The Currency Converter application menu should now look like Figure 2-7

Figure 2-7 The Currency Converter application menu

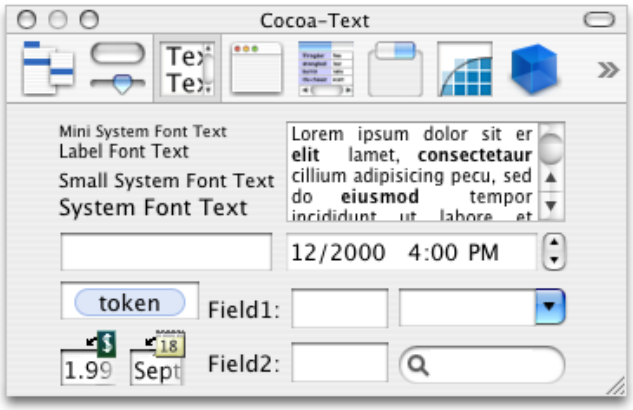


3. Click Help and replace `NewApplication Help` with `Currency Converter Help`.

Configure a Text Field

The Interface Builder palette window contains several user-interface elements that you can drag into a window or menu to create an application's user interface. Open the Interface Builder palette window—shown in Figure 2-8—by choosing Tools > Palettes > Show Palettes.

Figure 2-8 Cocoa text controls in the Interface Builder palette window

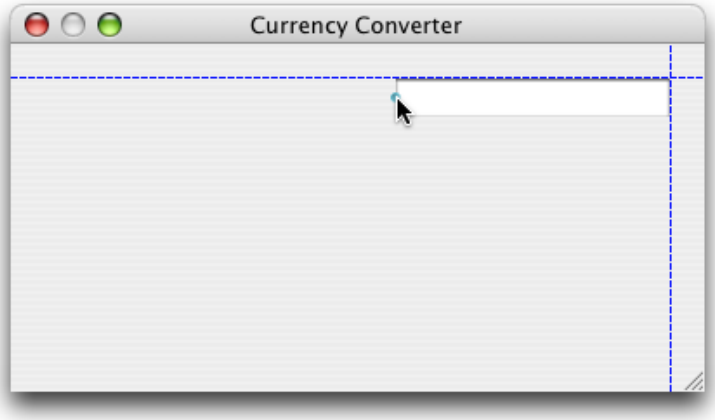


Place a text field in the Currency Converter window:

1. Click the text toolbar item in the palette window (shown as the third toolbar item in Figure 2-8) and drag a text field object to the top-right corner of the Currency Converter window. Notice that Interface Builder helps you place objects according to the Apple human interface guidelines by displaying layout guides when an object is dragged close to the proper distance from neighboring objects or the edge of the window.

2. Increase the text field's size so that it's about a third wider. 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, as shown in Figure 2-9

Figure 2-9 Resizing a text field



Currency Converter needs two more text fields, both the same size as the first. There are two options: You can drag another text field from the palette to the window and make it the same size as the first one; or you can duplicate the text field already in the window. The "Duplicate an Object" (page 27) section demonstrates the latter.

Duplicate an Object

To duplicate the text field in the Currency Converter window:

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. Position the new text field under the first text field. Notice that the layout guides appear and Interface Builder snaps the text field into place.

4. To make the third text field, press Command-D. Notice that Interface Builder remembers the offset from the previous Duplicate command and automatically applies it to the newly created text field.

As a shortcut, you can also Option-drag the original text field to duplicate it.

Change the Attributes of a Text Field

The bottom text field displays the results of the currency conversion computation and should therefore have different attributes than the other text fields: It must not be editable by the user.

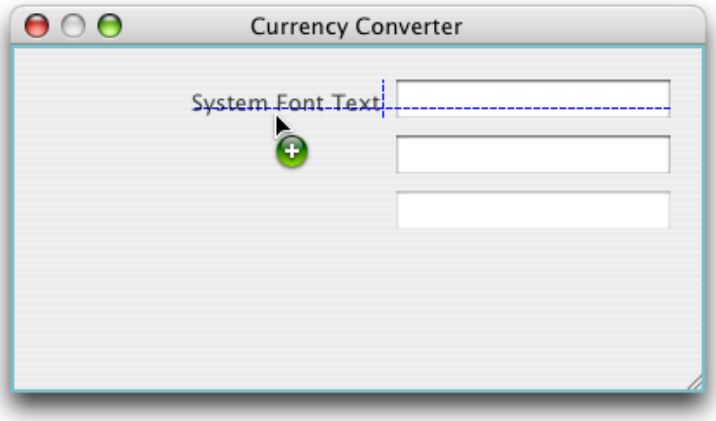
To make the text field that shows the result noneditable by the user:

1. Select the third text field.
2. In the `NSTextField` inspector, choose Attributes from the pop-up menu.
3. Deselect the Editable option so that users are not allowed to alter the contents of the text field. Make sure the Selectable option is selected so that users can copy the contents of the text field 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 Text palette.

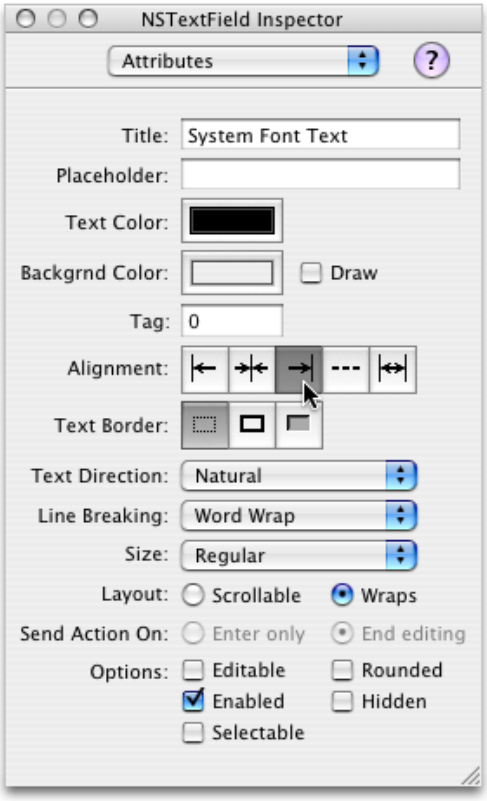
1. Drag a System Font Text element onto the window from the Cocoa Text palette.



2. Make the text label right aligned.

With the System Font Text element selected, click the third button from the left in the Alignment area in the `NSTextField` inspector, as shown in Figure 2-10

Figure 2-10 Right-aligning a text label in Interface Builder

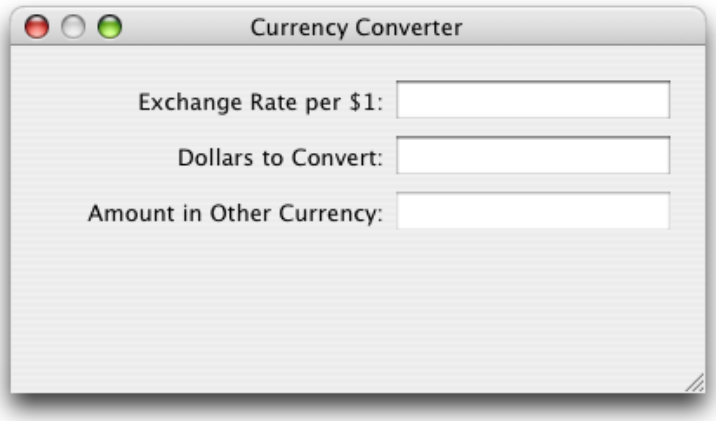


3. Enter `Exchange Rate per $1:` in the Title text field.

4. Duplicate the text label twice. Set the title of the second text label to “Dollars to Convert:” and the title for the third text label to “Amount in Other Currency:”.

5. Expand the text fields to the left so that their entire titles are visible, as shown in Figure 2-11

Figure 2-11 Text fields and labels in the Currency Converter window



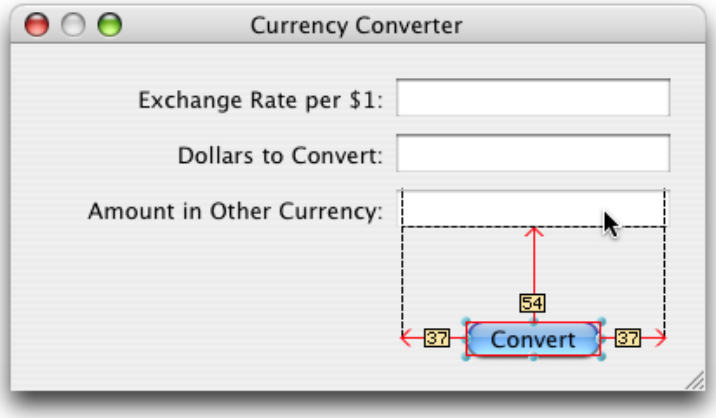
Configure a Button

The currency conversion should be invoked either by clicking a button or pressing Return. To add the Convert button to the Currency Converter window:

1. Drag the Button element from the Cocoa Controls palette to the bottom-right corner of the window.
2. Double-click the button and change its title to "Convert".
3. Choose Attributes from the `NSButton` inspector pop-up menu and then choose Return from the Key Equiv pop-up menu. This makes the button respond to the Return key as well as clicks.
4. Align the button under the text fields:
    a. Drag the button downward until the layout guide appears and then release it.
    b. With the button still selected, hold down the Option key. If you move the pointer around, Interface Builder shows you the distance from the button to the element over which the pointer is hovering.

c. With the Option key still down and the pointer over the Amount in Other Currency text field, use the arrow keys to nudge the button so that its center is aligned with the center of the text field, as shown in Figure 2-12

Figure 2-12 Measuring distances in Interface Builder



Add a Horizontal Decorative Line

You probably noticed that the final interface for Currency Converter has a decorative line between the text fields and the button. To add the line to the Currency Converter window:

1. Drag a horizontal line element from the Cocoa Controls palette to the Currency Converter window.
2. Drag the endpoints of the line until the line extends across the window, as shown in Figure 2-13

Figure 2-13 Adding a horizontal line to the Currency Converter window

3. Move the Convert button up until the layout guide appears below the Amount in Other Currency text field, and shorten the window until the horizontal layout guide appears below the Convert button.

Interface 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 user interface elements is made even more difficult because the elements have shadows and user interface guideline metrics do not 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 do not visually align correctly if you align the edges of the frames. For example, *Apple Human Interface Guidelines* says 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 Interface Builder using the Show Layout Rectangles command in the Layout menu.

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

- Dragging objects with the mouse in conjunction with the layout guides
- Pressing the 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 pane in the inspector

The Alignment and Guides submenus in the Layout menu provide various alignment commands and tools, including the Alignment window, which contains controls you can use to perform common alignment operations.

Finalize the Window Layout

Currency Converter's interface is almost complete. The finishing touch is to resize the window so that all the user-interface elements are centered and properly aligned to each edge. Currently, the objects are aligned only to the top and right edges.

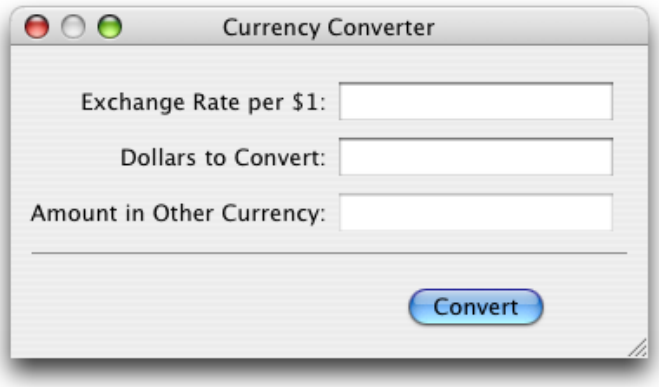
Perform these steps to finalize the Currency Converter window:

1. Select the Amount in Other Currency text label and extend the selection (Shift-click) to include the other two.
2. Resize all the labels to their minimum width by choosing Size to Fit from the Layout menu.
3. Deselect all the labels and reselect them, starting from the Amount in Other Currency label.
4. Choose Same Size from the Layout menu to make the selected text labels the same size.

5. Choose Layout > Alignment > Align Left Edges.
6. Drag the labels towards the left edge of the window, and release them when the layout guide appears.
7. Select the three text fields and drag them to the left, again using the guides to help you find the proper position.
8. Shorten the horizontal separator and move the button into position again under the text fields.
9. Make the window shorter and narrower until the layout guides appear to the right of the text fields.

At this point the application's window should look like Figure 2-14

Figure 2-14 Currency Converter's final user interface in Interface Builder



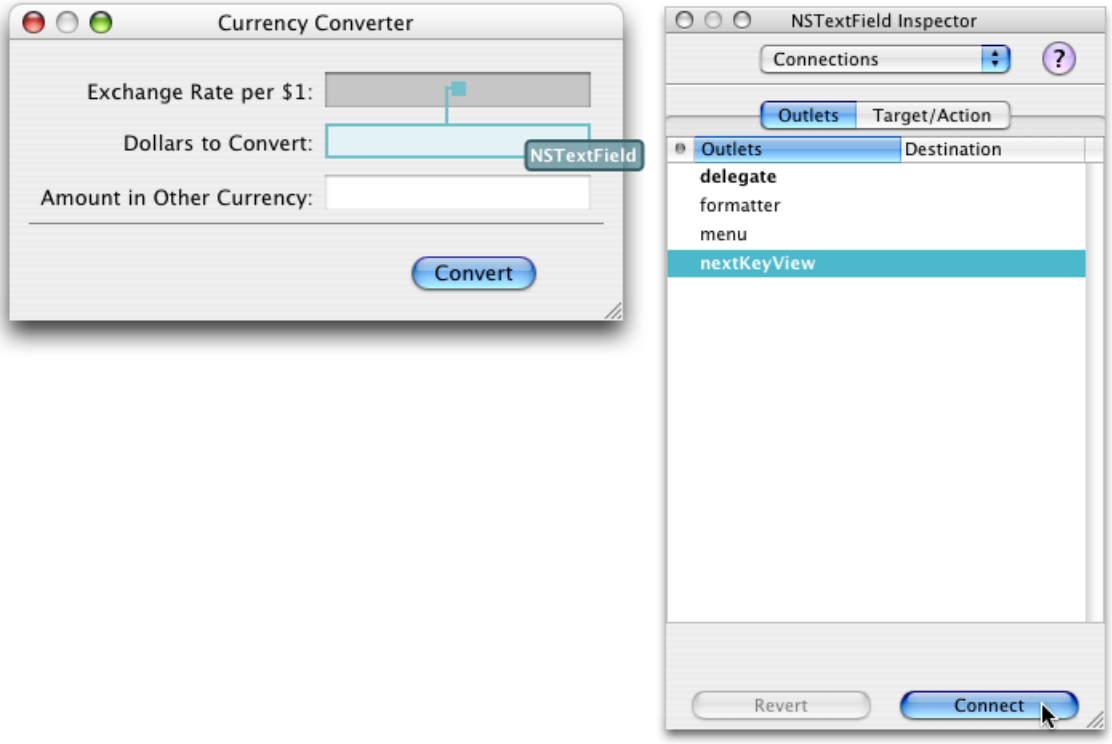
Enable Tabbing Between Text Fields

The final step in composing the Currency Converter user 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 outlet named `nextKeyView`. This variable identifies the next object to receive keyboard events when the user presses the Tab key (or the previous object when Shift-Tab is pressed). A Cocoa application by default makes its "best guess" about how to handle text field tabbing, but this guess often produces unexpected results. If you want correct interfield tabbing, you must connect fields through the `nextKeyView` outlet:

1. Select the Exchange Rate text field.

2. Control-drag a connection from the Exchange Rate text field to the Dollars to Convert text field, as shown in Figure 2-15 (To Control-drag, press Control then drag the connection line.)

Figure 2-15 Connecting `nextKeyView` outlets in Interface Builder



3. In the inspector for the Dollars to Convert text field click Outlets, 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 Dollars to Convert text field to the Exchange Rate text field.

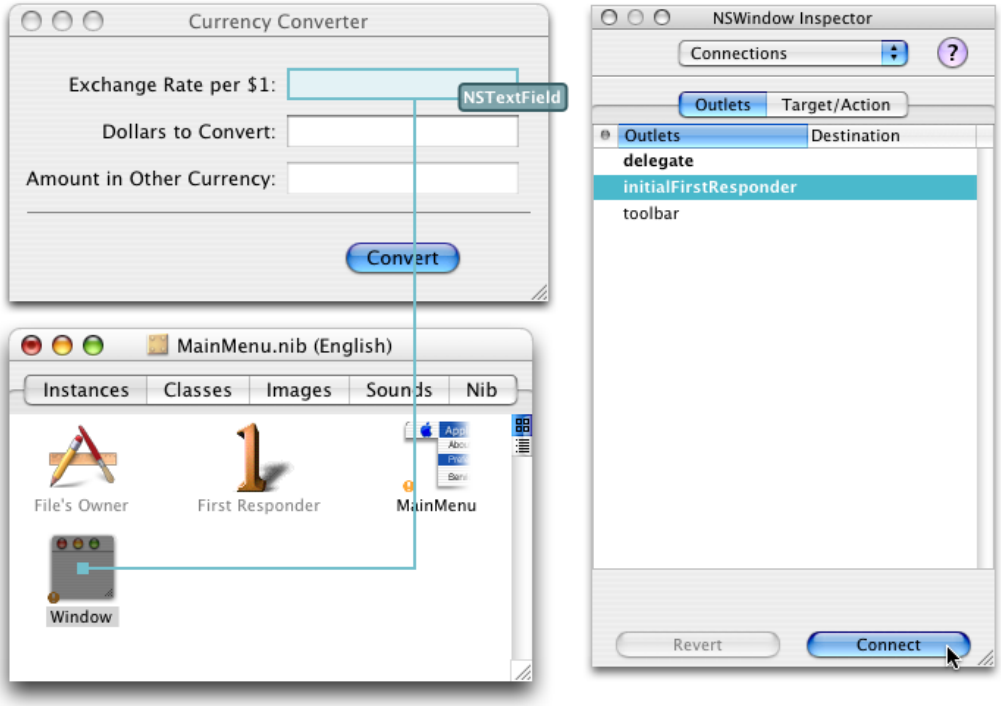
Set the First Responder for the Currency Converter Window

In "Enable Tabbing Between Text Fields," (page 33) 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 displayed onscreen. If you do not set this outlet, the window sets a key loop and picks a default initial first responder for you (not necessarily the same as the one you would have specified).

To set the `initialFirstResponder` outlet for the Currency Converter window:

1. Control-drag a connection from the Window instance in the `MainMenu.nib` window to the Exchange Rate text field, as shown in Figure 2-16

Figure 2-16 Setting the `initialFirstResponder` outlet in Interface Builder



2. In the inspector for the Exchange Rate text field, select `initialFirstResponder` and click Connect.

The Currency Converter user interface is now complete.

Test the Interface

Interface Builder lets you test an application's user interface without having to write code. To test the Currency Converter user interface:

1. Choose File > Save to save your work.
2. Choose File > Test Interface.
3. Try various user operations, such as tabbing, and cutting and pasting between text fields.
4. When finished, choose Quit Currency Converter 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 that it's in a convenient (and traditional) initial location.

Defining the ConverterController Class

Interface Builder is a versatile tool for application developers. It enables you to 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.

The following sections show how to define the `ConverterController` class and connect it to Currency Converter's user interface.

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 that 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 of them—or objects. Classes define the data structures and behavior of their instances, and at runtime create and initialize these instances. In a sense, a class is like a factory, stamping out instances of itself (objects of its class) 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 have, but only instances can hold data. The class also implements the behavior of all its instances in a running program.

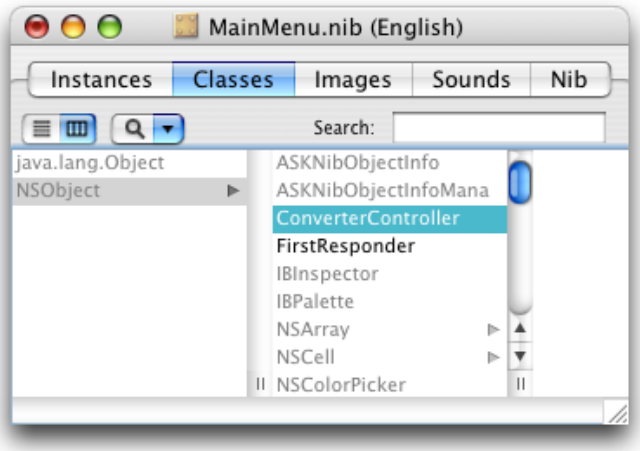
Implicit in the notion of a taxonomy is inheritance, a key property of classes. Classes exist in a hierarchical relationship to one another, with a subclass inheriting behavior and data structures from its superclass, which in turn inherits from its superclass.

Specify the ConverterController Class

You must go to the Classes pane of the nib file window to define a class. To design the `ConverterController` class:

1. In the `MainMenu.nib` window, click Classes.
2. In the leftmost column of the browser, select `NSObject` and press Return to create a `NSObject` subclass called `MyObject`.
3. Type `ConverterController` to rename `MyObject`, and press Return. Figure 2-17 shows the result of this operation.

Figure 2-17 Subclassing `NSObject`



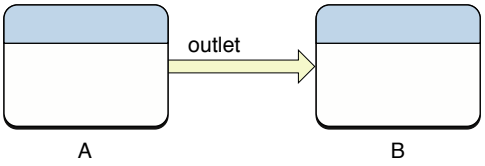
Paths for Object Communication: Outlets, Targets, and Actions

In Interface Builder, you specify the paths for messages traveling between the `ConverterController` object and other objects as outlets and actions. The following sections explain how the objects that implement the Currency Converter user interface communicate with each other in the running application.

Outlets

An **outlet** is an instance variable that identifies an object. Figure 2-18 illustrates how an outlet in one object points to another object.

Figure 2-18 An outlet pointing from one object to another



Objects 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;
```

Note: `IBOutlet` is a null-defined macro, which the C preprocessor removes at compile time. Interface Builder uses it to identify outlet declarations.

Objective-C gives you incredible freedom. 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 runtime. The dynamically typed object's class can be changed as needed, *even during runtime*, which should invoke a sense of both excitement and extreme caution in even the most grizzled OO veteran. This can be a tremendous feature and allows for very efficient use of memory, but casting a type to an object that cannot respond to the messages for that type can introduce puzzling and difficult-to-debug problems into your application.

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

```
IBOutlet NSButton* myButton;
```

You usually set an outlet's target 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.

At application load time, the instance variables that represent outlets are initialized to point to the corresponding target. For example, the `rateField` of the `ConverterController` instance would be initialized with a reference to the Exchange Rate text field object (see "Connect the ConverterController Class to the Text Fields" (page 42) for details). When an outlet is not connected, the value of the corresponding instance variable is `null`.

It might help to understand connections by imagining an electrical outlet plugged into the destination object. Also picture an electrical cord extending from the outlet in the source object. Before the connection is made, the cord is not plugged in, and the value of the outlet is `null`; after the connection is made (the cord is plugged in), a reference to the destination object is assigned to the source object's outlet.

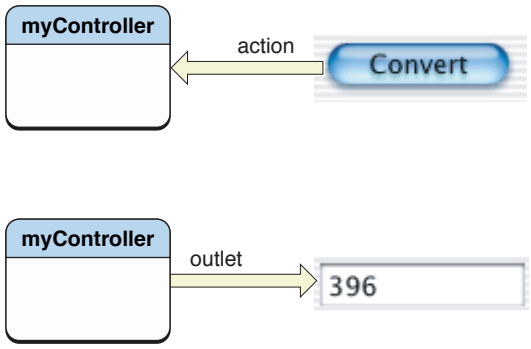
Target/Action in Interface Builder

You can view (and complete) target/action connections in the Connections pane in the Interface Builder inspector. This pane 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 may 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 AppKit inherit these outlets.

For example, when a user clicks the Convert button in the Currency Converter window, the button gets the required information from its cell and invokes the `convert` method on the target outlet object, which is an instance of the custom class `ConverterController`. Figure 2-19 shows the interactions between the ConverterController class, the Convert button, and the Amount in Other Currency field.

Figure 2-19 Relationships in the target-action paradigm



In the Actions column in the Connections pane in the 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 names follow the syntax:

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

Here, it looks 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 to follow a simple rule to know in which way to specify a connection in Interface Builder. Create the connection from the object that sends the message to the object that receives the message:

- To make an action connection, create the connection from an element in the user interface, such as a button or a text field, to the custom instance you want to send the message to.
- To make an outlet connection, create the connection from the custom instance to another object (another instance or user-interface element) in the application.

These are only rules of thumb for common cases 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 to the custom object.

Define the User Interface and Model Outlets of the ConverterController Class

The `ConverterController` object needs to communicate with the user-interface elements in the Currency Converter window. It must also communicate with an instance of the `Converter` class, defined in "Defining the Converter Class." (page 44) The `Converter` class implements the conversion computation.

To add the outlets required by the `ConverterController` class:

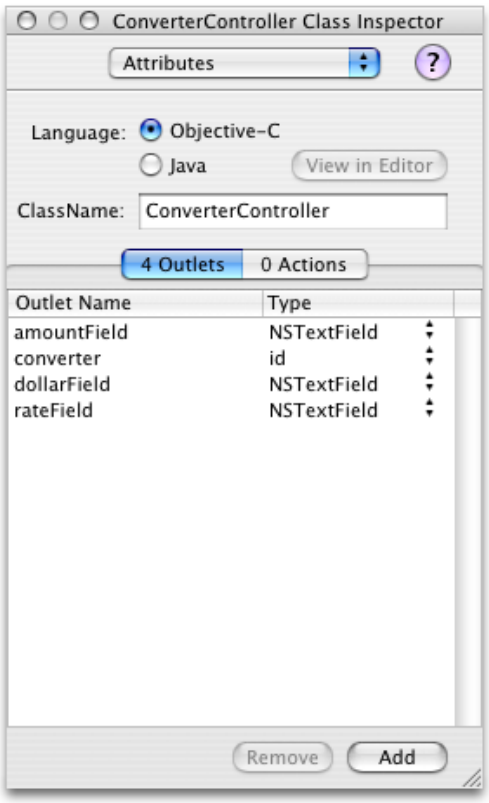
1. Select `ConverterController` in the Classes pane in the `MainMenu.nib` window.
2. Choose Add Outlet to `ConverterController` from the Classes menu, or:
    a. Choose Attributes from the inspector pop-up menu.
    b. Click 0 Outlets.
    c. Click Add.
3. Name this outlet `rateField` and press Return.
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 `amountField` outlet.
5. Add another outlet named `converter`. This is the outlet the `ConverterController` instance uses to communicate with the `Converter` instance. (The `Converter` class is defined later in this chapter.)

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` because 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 `amountField`, `dollarField`, and `rateField` outlets to `NSTextField` by choosing it from the pop-up menus currently set to `id`.

Note: The `converter` outlet cannot be typed at this time because the `Converter` class has not been defined.

The result of these operations is shown in Figure 2-20

Figure 2-20 Outlets and actions in the `ConverterController` class inspector



Define the Actions of the ConverterController Class

The `ConverterController` class needs one action method, `convert`. When the user clicks the Convert button, the `convert:` message is sent to the target object, an instance of the `ConverterController` class.

To add the `convert` method to the `ConverterController` class:

1. Select `ConverterController` in the Classes pane in the `MainMenu.nib` window.
2. Choose Add Action to `ConverterController` from the Classes menu, or:
    a. Choose Attributes from the inspector pop-up menu.
    b. Click 0 Actions.
    c. Click Add.
3. Type `convert:` in the Action Name list and press Return.

Interconnecting the ConverterController Class and the User Interface

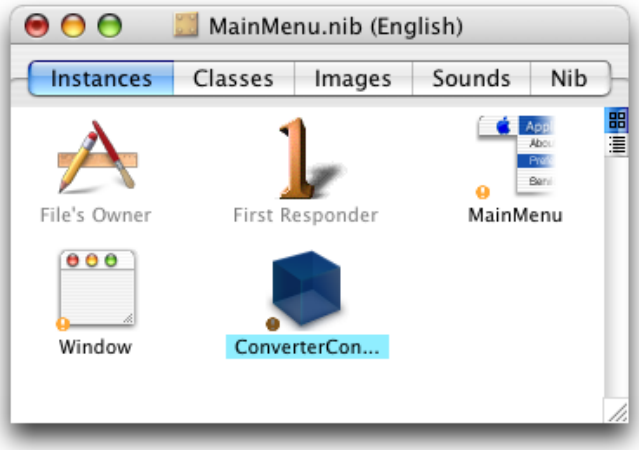
The following sections show how to connect the Currency Converter user interface and the ConverterController class to each other.

Create an Instance of the ConverterController Class

As the final step of defining a class in Interface Builder, you create an instance of the `ConverterController` class and connect its outlets and actions. To carry out this task, perform these steps:

1. Select `ConverterController` in the Classes pane in the `MainMenu.nib` window.
2. Choose Instantiate `ConverterController` from the Classes menu. The instance appears in the Instances pane as shown in Figure 2-21 Notice that the instance has a yellow badge with an exclamation point next to it. This means the instance contains unconnected outlets.

Figure 2-21 A newly instantiated `ConverterController` instance



Connect the ConverterController Instance to the Text Fields

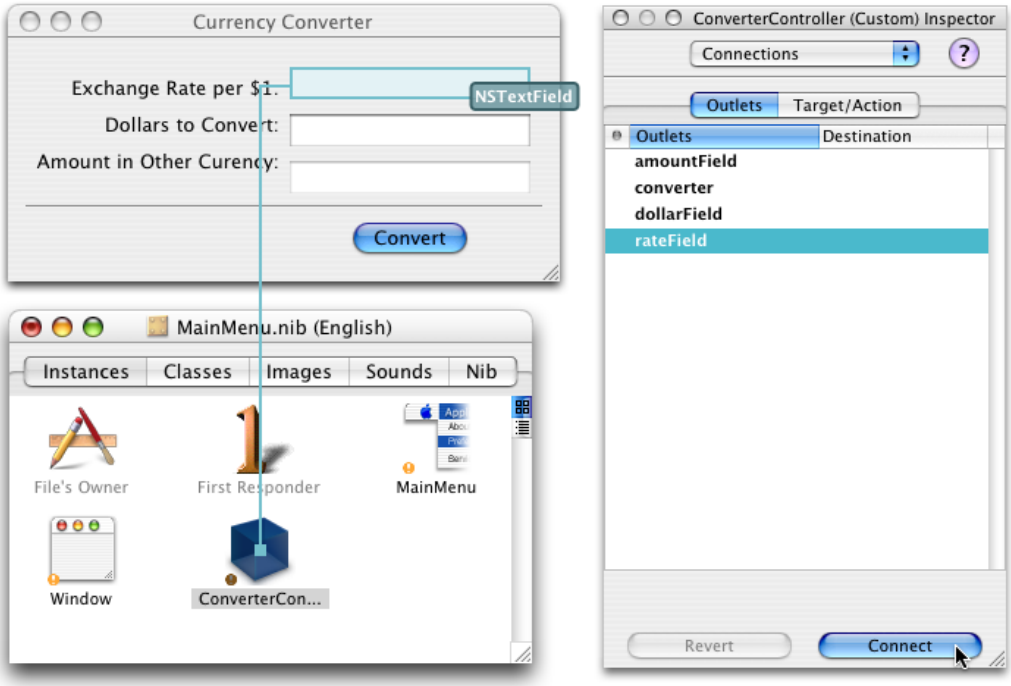
By connecting the `ConverterController` instance to specific objects in the interface, you initialize its outlets. The `ConverterController` class uses these outlets to get and set values in the user interface. Follow these steps to connect the instance to the user interface:

1. In the Instances pane in the nib file window, Control-drag a connection from the `ConverterController` instance to the Exchange Rate text field.

2. Interface Builder displays the Connections pane in the inspector. Select the outlet that corresponds to the first field, `rateField`.

3. Click Connect, as shown in Figure 2-22

Figure 2-22 Connecting ConverterController to the `rateField` outlet



4. Following the same steps, connect the `ConverterController` class's `dollarField` and `amountField` outlets to the appropriate text fields.

Connect the Convert Button to the ConverterController convert Action Method

Follow these steps to connect the user interface elements in the Currency Converter window to the methods of the `ConverterController` class:

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

2. In the Connections pane in the inspector, make sure the Target/Action pane is displayed.

3. Select `convert:` in the Actions in `ConverterController` list and click Connect.

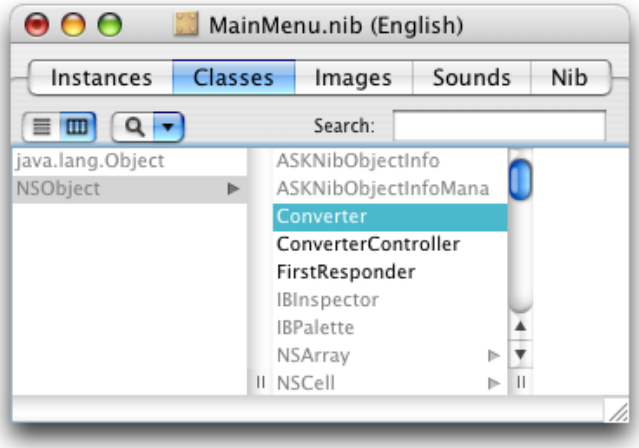
4. Save the nib file.

Defining the Converter Class

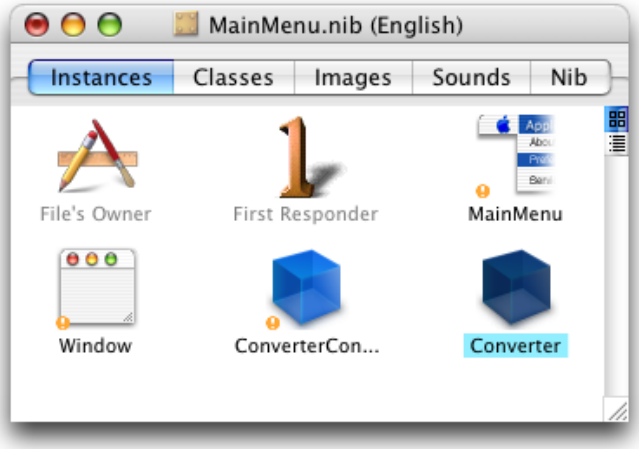
While connecting the `ConverterController` class's outlets, you probably noticed that one outlet remains unconnected: `converter`. This outlet identifies an instance of the `Converter` class in the Currency Converter application, but this instance doesn't exist yet.

The `Converter` class implements a **model object**. Model objects contain special knowledge and expertise. They hold data and define the logic that manipulates that data. For example, a customer object, common in business applications, is a model object. Since instances of this type of class don't communicate directly with the user interface, there is no need for outlets or actions. Here are the steps to be completed:

1. In the Classes pane in the nib file window, create a subclass of `NSObject` named `Converter`. See "Specify the ConverterController Class" (page 36) for details.

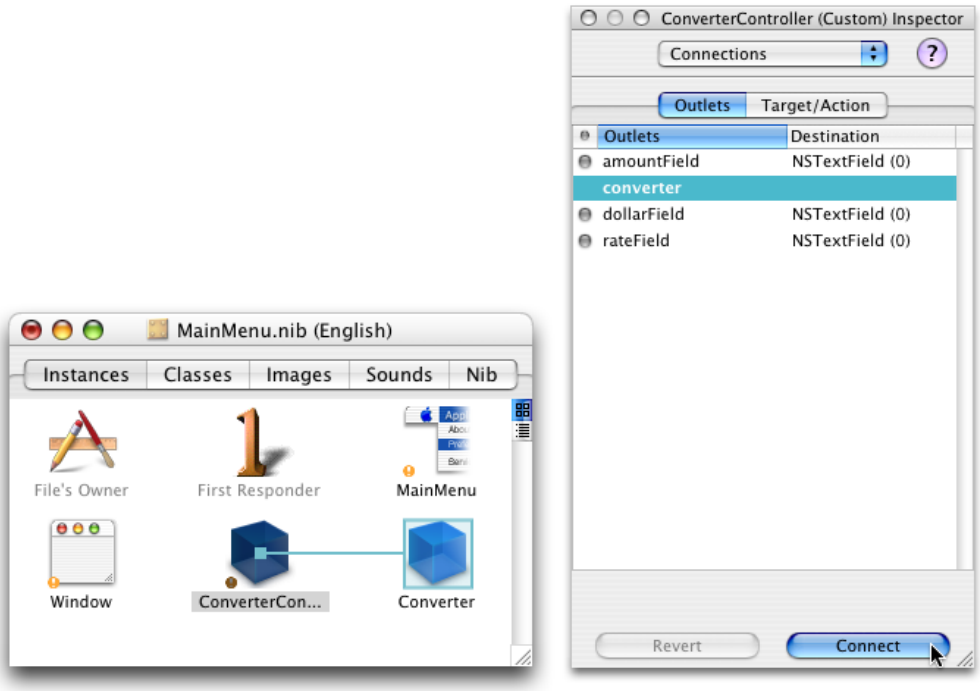


2. Instantiate the `Converter` class. See "Create an Instance of the ConverterController Class" (page 42) for details.



3. Connect the `converter` outlet of the `ConverterController` instance to the `Converter` instance, as shown in Figure 2-23

Figure 2-23 Connecting the `ConverterController` instance to the `Converter` instance



4. Change the type of the `converter` outlet in the `ConverterController` class from `id` to `Converter`. As explained in "Define the User Interface and Model Outlets of the ConverterController Class," (page 39) you should do this to improve error checking during compilation.

    a. In the Classes in the nib file window, select the `ConverterController` class.

    b. In the inspector for the `ConverterController` class, choose Attributes from the pop-up menu.

    c. In the Outlets pane, select the `converter` outlet and choose `Converter` from the corresponding Type pop-up menu.

5. Save the nib file.

Implementing Currency Converter

In "Creating the Currency Converter Project and User Interface" (page 17) you created the skeleton and the user interface for your application. This chapter guides you through defining the custom behavior of the application and in the process teaches you the final steps essential to implementing a Cocoa application.

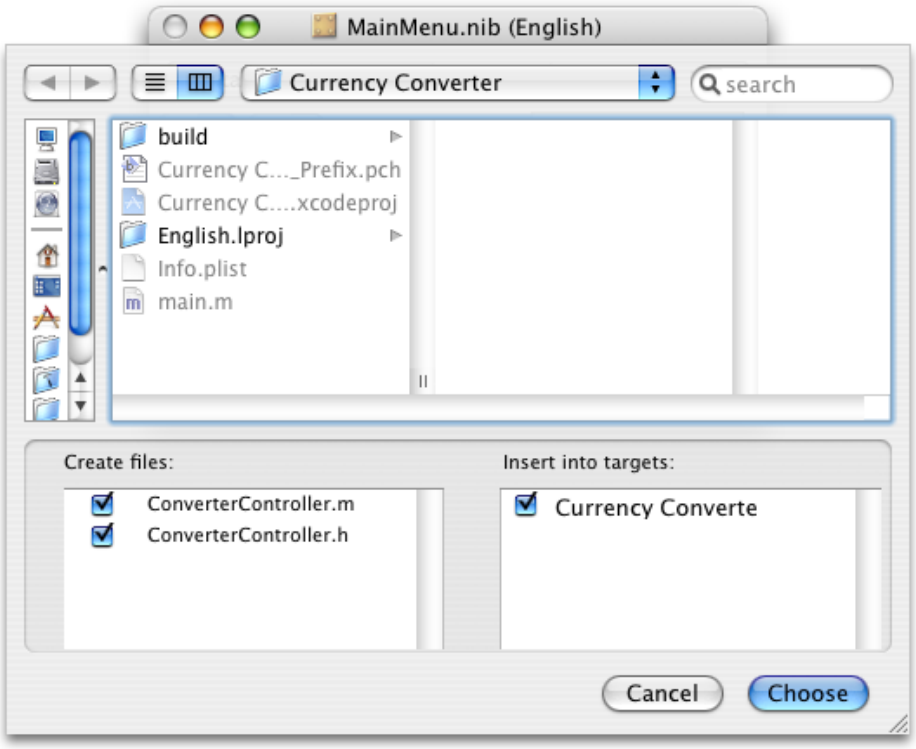
Note: If you're new to Objective-C, you may benefit from the information in "Objective-C Quick Reference" (page 77) before going through this chapter.

Generate the Source Files

To generate the source files for the Currency Converter application:

1. In the Classes pane in the nib file window, select the `ConverterController` class.
2. Choose Classes > Create Files for `ConverterController`.

3. Make sure `ConverterController.h` and `ConverterController.m` in "Create files" and Currency Converter in "Insert into targets" are selected.



Note: If "Insert into targets" is empty, close the nib file and reopen it from the Currency Converter project.

4. Click Choose.
5. Repeat steps 1 through 4 for the `Converter` class.
6. Save the nib file.

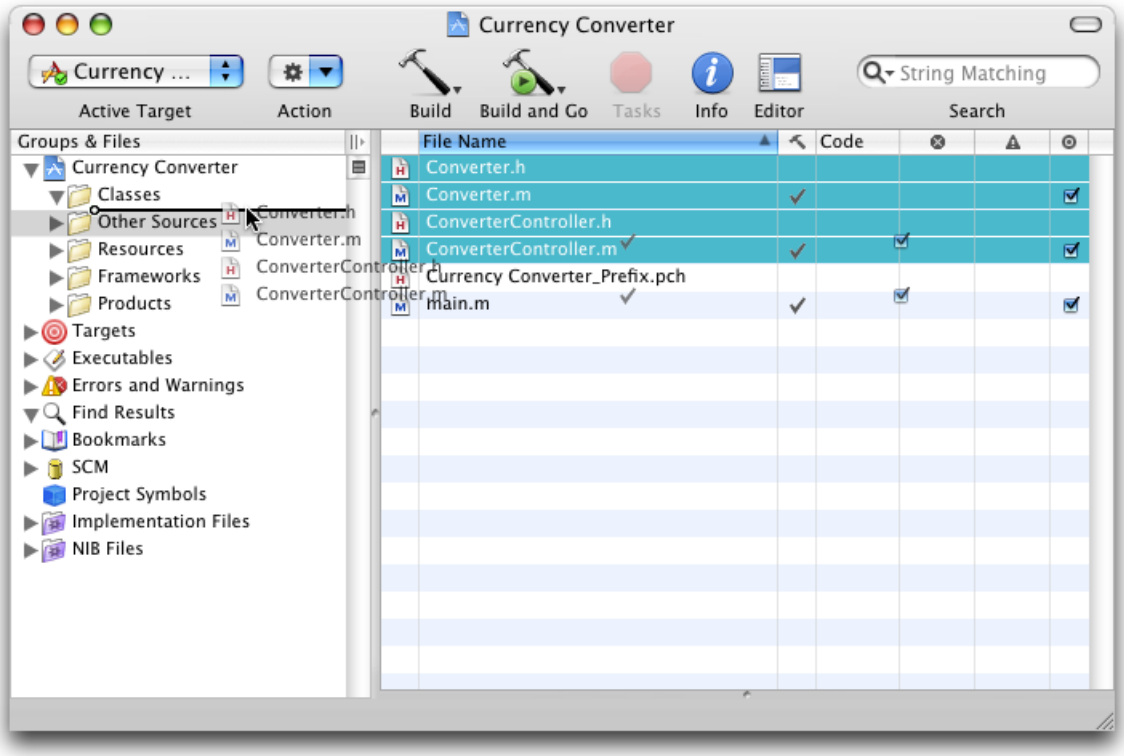
Quit Interface Builder.

Place the Implementation Files in the Appropriate Group

When Interface Builder adds the header and implementation files to the Currency Converter project, it puts them in the Other Sources group. While this is perfectly acceptable—the build system doesn't care where the files are—they are classes of your project and should be organized accordingly. Move these files to the Classes group:

1. In the Xcode project window, select the header and implementation files for the `Converter` and `ConverterController` classes in the Other Sources group.

2. Drag the files to the Classes group.



Finalize ConverterController.h

When Interface Builder generates the header file for a class you designed in it, it doesn't add the `#import` statements that may be necessary to incorporate types declared in other Interface Builder–generated header files. You have to add any missing `#import` statements before your project can build correctly. In this case, `ConverterController.h` needs to import `Converter.h` because it declares the type of the `converter` outlet as `Converter`.

Insert the highlighted line in Listing 3-1 into `ConverterController.h`.

Listing 3-1 The `ConverterController.h` header file

```
#import <Cocoa/Cocoa.h>
#import "Converter.h"

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

```
@end
```

Implement Currency Converter's Classes

The `Converter` class needs a method that computes the currency conversion. This is the method the `ConverterController` object uses to perform the conversion.

Define the `convertCurrency:atRate:` method:

1. In the Classes group, double-click `Converter.h` to open this file in an editor window.
2. Insert the highlighted line in Listing 3-2 into `Converter.h`.

Listing 3-2 Declaration of the `convertCurrency:atRate:` method in `Converter.h`

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

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

3. In the Classes group, double-click `Converter.m` to open this file in an editor window.
4. Insert the highlighted lines in Listing 3-3 into `Converter.m`.

Listing 3-3 Definition of the `convertCurrency:atRate:` method in `Converter.m`

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

The `convertCurrency:atRate:` method multiplies its two arguments and returns the result.

When clicked, the Convert button sends the `convert:` message to the `ConverterController` object. Complete the definition of the `convert:` method in the `ConverterController` class, so that it sends the `convertCurrency:atRate:` message to the `Converter` instance to execute the conversion:

1. In the Classes group, double-click `ConverterController.m` to open this file in an editor window.
2. Insert the highlighted lines in Listing 3-4 into `ConverterController.m`.

Listing 3-4 Definition of the `convert:` method in `ConverterController.m`

```
#import "ConverterController.h"
@implementation ConverterController
- (IBAction)convert:(id)sender {
    float rate, currency, amount;
    currency = [dollarField floatValue];
    rate =     [rateField floatValue];

    amount =   [converter convertCurrency:currency atRate:rate];

    [amountField setFloatValue:amount];
    [rateField selectText:self];
}
@end
```

The `convert:` method does the following:

- Gets the floating-point values entered into the Exchange Rate (`rateField`) and Dollars to Convert (`dollarField`) text fields.
- Sends the `convertCurrency:atRate:` message with the dollar and rate values to the object pointed to by the `converter` outlet and gets the returned value.
- Uses `setFloatValue:` to write the returned value to the Amount in Other Currency text field (`amountField`).
- Sends the `selectText:` message to the rate field. As a result, any text in the field is selected; if there is no text, the insertion point is placed in the text field so the user can begin another calculation.

Each code line in the `convert:` method, excluding the declaration of floating-point variables, 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 the receiver to execute. Messages often result in values being returned; in the above example, the local variables `rate`, `currency`, and `amount` hold these values.

You've now completed the implementation of Currency Converter. Are you surprised how little code you had to write, given your application now has a fully functional currency-converting system and a beautiful user interface? "Building Currency Converter" (page 53) shows how to build and run the application.

Building Currency Converter

This chapter guides you through building the Currency Converter application and, in the process, teaches you the steps essential to building a Cocoa application.

Overview of the Build Process

You start the build process by clicking the Build toolbar item in the Xcode project window. During this process, Xcode coordinates the compilation and linking tasks that result in an executable file. It also performs other tasks needed to build an application.

While building a project, Xcode invokes the compiler, passing it the source code files of the project. The compilation of these files produces object files for the architectures specified for the build.

In the linking phase of the build, Xcode executes the static 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.

Xcode also copies nib files, sound files, image files, and other resources from the project directory to the appropriate locations in the **application bundle**. An application bundle 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; it can be double-clicked to launch the application.

Build the Currency Converter Application

To build the Currency Converter application:

1. Choose Save All from the Xcode File menu to save the changes made to the project's source files.
2. Click the Build toolbar item in the project window.

The status bar at the bottom of the project window indicates the status of the build. When Xcode finishes—and encounters no errors along the way—it displays "Build succeeded" in the status bar. If there are errors, however, you need to correct them and start the build process again. See "Correct Build Errors" (page 54) for details.

Note: If you get a build error because `converter` is not declared in `ConverterController.m`, go back to "Define the User Interface and Model Outlets of the ConverterController Class" (page 39) and re-generate the header file for the `ConverterController` class, as explained in "Generate the Source Files." (page 47)

Look Up Documentation

Xcode gives you access to ADC Reference Library content. You can jump directly to documentation and header files while you work on a project. Try it out:

1. Open `ConverterController.m` in an editor window.
2. Option–double-click the word `setFloatValue` in the code. (Hold down the Option key and double-click the word.) The Developer Documentation window appears with a list of relevant method names in its detail view. This Reference Library access system provides a fast way to get to reference material. Read more in "Expanding on the Basics." (page 71)
3. Close the Developer Documentation window.
4. Command–double-click the same word. A pop-up menu with a list of method names appears.
5. Choose `[NSCell setFloatValue]`. This time, Xcode displays the `NSCell.h` header file in an editor window and highlights the declaration of the `setFloatValue` method.
6. Close the header file.

Run Currency Converter

Your hard work is about to pay off. Because you haven't edited any code since the last time you built the project, the application is ready to run. Choose Build > Build and Go.

After the Currency Converter application launches, enter a rate and a dollar amount and click Convert. Now, select the text in a text field and choose the Services submenu from the Application menu. The Services menu lists other applications that can operate on the selected text.

Quit Currency Converter by choosing Quit Currency Converter from the application menu.

Correct Build Errors

Of course, rare is the project that is flawless from the start. For most applications you write, Xcode is likely to catch some errors when you first build them. Thankfully, Xcode offers tools to help you catch those bugs and move on.

To get an idea of the error-checking features of Xcode, introduce a mistake into the code:

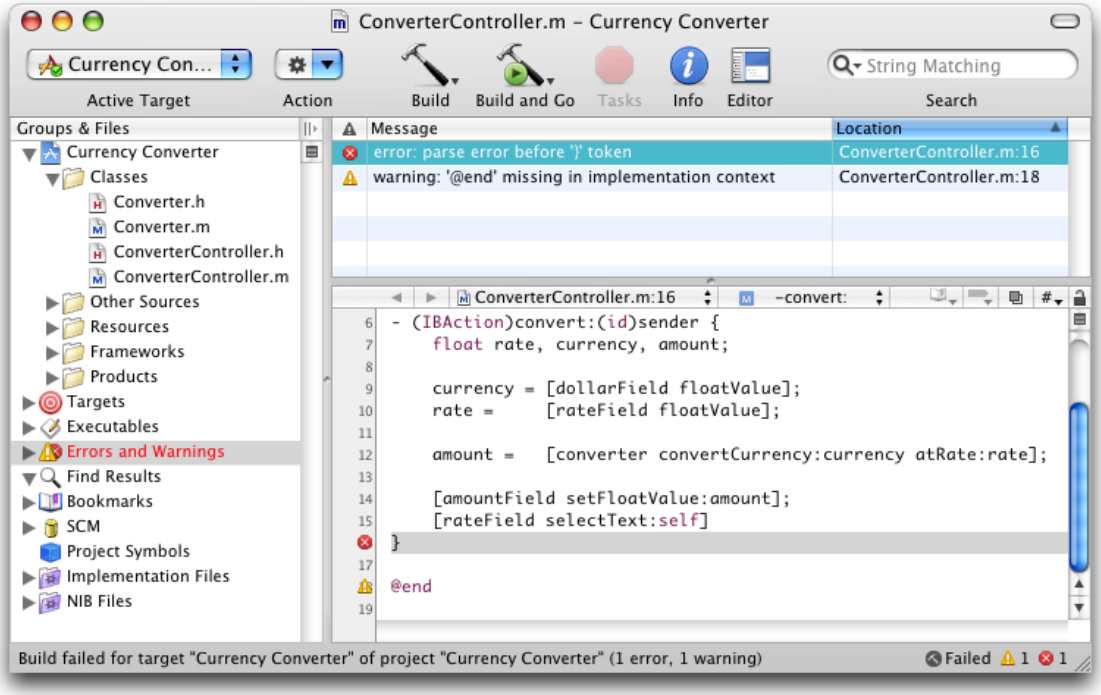
1. Open `ConverterController.m`.
2. Delete the semicolon from the code line that sends the `selectText:` message.
3. Click the Build toolbar item.

Uh-oh! Something is amiss. You can now see that the left column of your code contains one error indicator.

While the error indicator helps you understand the location of the error, you may want to examine the nature of the problem. In the Groups & Files list, disclose the Errors and Warnings group if it's not already disclosed. Xcode lists the files that contain build errors. In this case, the `ConverterController.m` file is the only file with a problem.

Select the file in the Errors and Warnings group or open it in an editor window to display the error. Xcode displays information about the error in the detail view, as shown in Figure 4-1

Figure 4-1 Identifying build errors



Fix the error in the code and build the application again. The Errors and Warnings group clears and the status bar indicates that the build is successful.

Great Job!

Although Currency Converter is a simple application, creating it illustrates many of the concepts and techniques of Cocoa programming. Now you have a much better grasp of the skills you need to develop Cocoa applications. Let's review what you learned:

- Composing a graphical user interface (GUI) in Interface Builder
- Testing a user interface in Interface Builder
- Specifying a class's outlets and actions in Interface Builder
- Connecting controller-instances to the user interface by using outlets and actions in Interface Builder
- Implementing a model class in Xcode
- Building applications and correcting build errors in Xcode

Configuring Currency Converter

Mac OS X applications contain information to help single them out from each other, to the benefit of both developers and users. This information includes the application's primary and secondary version numbers, and the icon the Finder and the Dock use to represent it. The file that stores these details is known as the *information property list* file (named `Info.plist`). This property list file is stored in the `Contents` directory of the application bundle.

Note: A **bundle** is a directory that groups files in a structured hierarchy. To make it easy for users to manipulate bundles, bundles can be represented as files instead of folders in the Finder; these bundles are known as **packages**. An **application bundle** stores the executable files and resources that make up an application. Although it's more correct to refer to application bundles as *application packages* because they're always shown to users as single files in the Finder, this chapter adopts the term application bundle instead of application package. For more information on bundles and packages, see *Bundle Programming Guide*.

This chapter describes the essential identification properties required of Mac OS X applications. It also walks you through the process of configuring these properties in Currency Converter.

Essential Application Identification Properties

There are several essential properties that identify applications to users and to Mac OS X: application identifier, build version number, release version number, copyright notice, application name, and application-icon filename.

- The **application-identifier property** specifies a string Mac OS X uses to identify an application. This property does not identify a specific application bundle in the filesystem or a particular version of the application. In normal conditions, users don't see application identifiers.

  The application-identifier property is specified with the `CFBundleIdentifier` key in the `Info.plist` file.

  Application identifiers are uniform type identifiers (UTIs) or reverse Domain Name System (DNS) names; that is, the top-level domain comes first, then the subdomains, separated by periods (`.`). There are two parts to an application identifier: the *prefix* and the *base*. The **application-identifier prefix** identifies the company or organization responsible for the application and is made up of two or more domains. The first prefix domain, or top-level domain, is normally `com` or `org`. The

second domain is the name of the company or organization. Subsequent domains can be used as necessary to provide a narrower scope. Prefix domains use lowercase letters by convention. For example, Apple applications use application identifiers that start with `com.apple`.

The **application-identifier base** comprises a single domain, which refers to the application proper. This domain should use word capitalization, for example, `AddressBook`. See *Uniform Type Identifiers Overview* for more information about uniform type identifiers.

Mac OS X uses application identifiers to precisely refer to application bundles irrespective of their location on the filesystem. For example, some Mac OS X features, such as parental controls, use only application identifiers to refer to applications on a user's computer. The Parental Controls preferences pane contains a list of application filenames, in which administrators select the applications for which a user is to have managed access, as shown in Figure 5-1 Without application identifiers, administrators would have to navigate to the location of each managed application, a relatively tedious task.

Figure 5-1 Benefits of using application identifiers

- The **build-version-number property** identifies an iteration of the application.

  The build-version-number property is specified with the `CFBundleVersion` key in the `Info.plist` file.

  The build version number is made up of a string of period-separated integers. Each integer must be equal to or greater than zero. For example, `55`, `1.2`, and `1.2.0.55`, are all valid build version numbers.

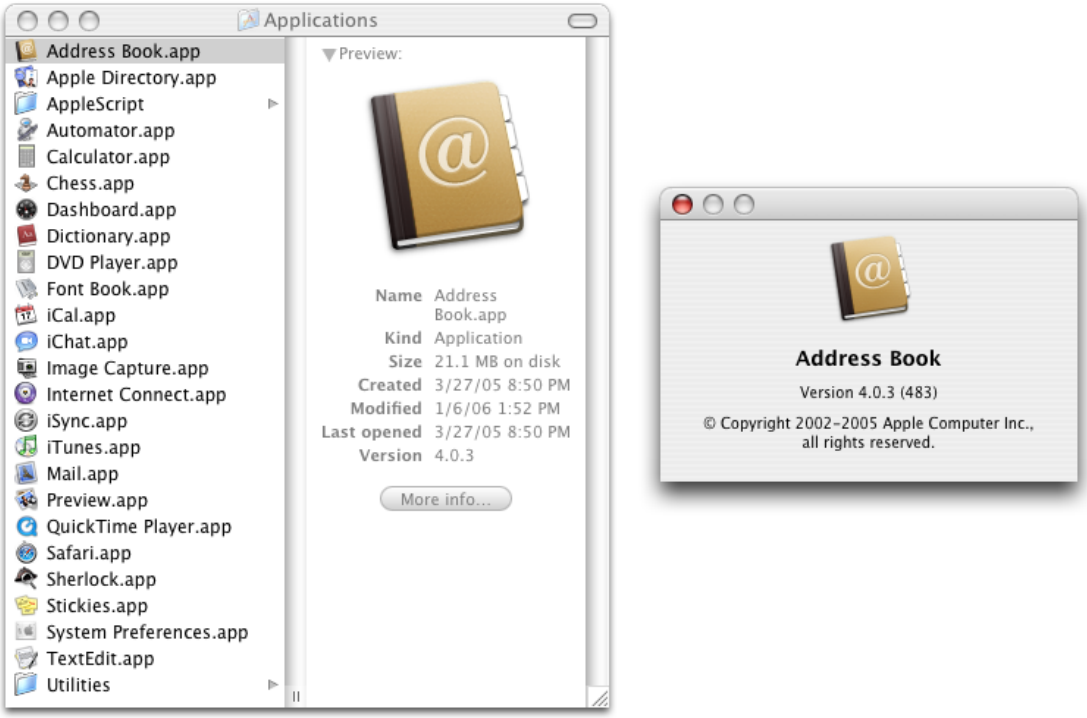
  Mac OS X uses the build version number to, for example, decide which version of an application to launch to open a file when there are multiple versions in the filesystem (as determined by their application identifiers). In such cases, Mac OS X launches the application bundle with the highest build version number. To ensure the accurate operation of this mechanism, you must adhere to

one version-numbering style as you release new versions of your application. That is, if you publish your application for the first time using a multi-integer build version number, subsequent publicly available versions must use the same number of integers in their build version numbers.

Note: The application's build version number does not appear in Finder windows.

- The **release-version-number property** specifies the version information the Finder displays for the application. When you specify both a build version number and a release version number, the About window displays the release version number, followed by the build version number in parenthesis, as shown in Figure 5-2

Figure 5-2 Build and release version numbers in Finder preview panes and About windows



The release-version-number property is specified with the `CFBundleShortVersionString` key in the `Info.plist` file.

The release version number identifies a released iteration of the application. Similar to the build version number, the release version number is made up of a string of period-separated integers. However, you should specify no more than three integers for the release version number. By convention, the first integer represents major revisions to the application, such as revisions that implement new features or major changes. The second integer denotes revisions that implement less prominent features. The third integer represents maintenance releases.

- The **copyright-text property** specifies the copyright notice for the application, for example, `© 2006, My Company`. This notice is shown to users in the About window of the application.

  The copyright-notice property is specified with the `NSHumanReadableCopyright` key in the `Info.plist` file.

- The **application-name property** specifies the title of the application menu in the menu bar when the application opens and the name of the application in its About window.

  The application-name property is specified with the `CFBundleName` key in the `Info.plist` file.

  Note: When you create an application project in Xcode, the name you enter as the project name is used as the application name.

- The **application-icon-filename property** specifies the icon the Finder, the Dock, and the application's About window display for the application.

  The application-icon-filename property is specified with the `CFBundleIconFile` key in the `Info.plist` file.

  An icon file contains one or more images that depict an application's icon at various resolutions. These separate versions of an icon allow the Finder and the Dock to render icons as sharp as possible at different sizes. You create icon files using the Icon Composer application.

For further details on these and other application properties, see *Runtime Configuration Guidelines*.

The following sections show how to specify these properties for Currency Converter.

Specifying Currency Converter's Identifier, Version, and Copyright Information

This section shows how to specify Currency Converter's identifier, release version number, and copyright text.

Important: To complete this task, you need to open this document's companion archive, `ObjCTutorial_companion.zip`.

Apple_Note: The following instructions are based on Xcode Tools 2.2.1 on Mac OS X v10.4.4.

Currency Converter's name property is set to the project name you entered in "Creating the Currency Converter Project," (page 17) `Currency Converter`. Therefore, you don't need to change the value of this property.

To set the the application-identifier, build-version-number, release-version-number, and copyright-text properties, follow these steps:

1. In the Currency Converter project window, select the Resources group in the Groups & Files list.
2. Remove the `InfoPlist.strings` file from the project (this file is used for internationalization, a subject outside the scope of this document):
   a. In the detail view, select the `InfoPlist.strings` file.
   b. Choose Edit > Delete.

c. In the dialog that appears, click Delete References.

3. In the detail view, double-click the `Info.plist` file. The file opens in an editor window.
4. Set the application identifier to `com.mycompany.CurrencyConverter`:
   a. Locate the `CFBundleIdentifier` key in the file.
   b. Set the `string` element under the `CFBundleIdentifier` key to `com.mycompany.CurrencyConverter`.
5. Set the build version number to `100`:
   a. Locate the `CFBundleVersion` key in the file.
   b. Set the corresponding `string` element to `100`.
6. Set the release version number `1.0.0`:

   When the `CFBundleShortVersionString` key is not defined in the `Info.plist` file, you must add it:

   a. Place the cursor after the last key/value pair in the `Info.plist` file, just before the `</dict>` tag.
   b. Press Return.
   c. Enter the following lines:

   ```
   <key>CFBundleShortVersionString</key>
   <string>1.0.0</string>
   ```

7. Set the copyright text to `© 2006, My Company`:

   Add the following key/value pair to the `Info.plist` file (press Option-G to produce the © character):

   ```
   <key>NSHumanReadableCopyright</key>
   <string>© 2006, My Company</string>
   ```

8. Save the `Info.plist` file.

Listing 5-1 shows how Currency Converter's property list file might look after the changes. The highlighted lines have been either modified or added.

Listing 5-1 Specifying domain, version, and copyright information in the Currency Converter `Info.plist` file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
```

```
    <key>CFBundleDevelopmentRegion</key>
    <string>English</string>
    <key>CFBundleExecutable</key>
    <string>${EXECUTABLE_NAME}</string>
    <key>CFBundleIconFile</key>
    <string></string>
    <key>CFBundleIdentifier</key>
    <string>com.mycompany.CurrencyConverter</string>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundleName</key>
    <string>${PRODUCT_NAME}</string>
    <key>CFBundlePackageType</key>
    <string>APPL</string>
    <key>CFBundleSignature</key>
    <string>????</string>
    <key>CFBundleVersion</key>
    <string>100</string>
    <key>NSMainNibFile</key>
    <string>MainMenu</string>
    <key>NSPrincipalClass</key>
    <string>NSApplication</string>
    <key>CFBundleShortVersionString</key>
    <string>1.0.0</string>
    <key>NSHumanReadableCopyright</key>
    <string>© 2006, My Company</string>
</dict>
</plist>
```

Clean the project, and build and run the application.

Note: If the application builds correctly but fails to run with an error similar to the one in Figure 5-3 review the changes you made to the `Info.plist` file. Make sure that the file is well formed. Then clean the project and build the application again.

Figure 5-3 Build error caused by a non–well-formed `Info.plist` file

Quit Currency Converter.

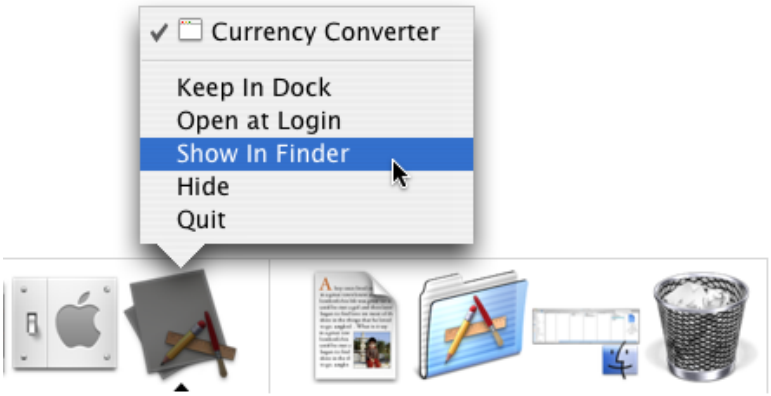
To see how application identifiers can be beneficial, execute the following command in a Terminal window:

```
> open -b com.mycompany.CurrencyConverter
```

The `open` command locates and launches Currency Converter based on its application identifier. This command can also use the filenames of application bundles to locate and launch applications (the `.app` suffix is optional).

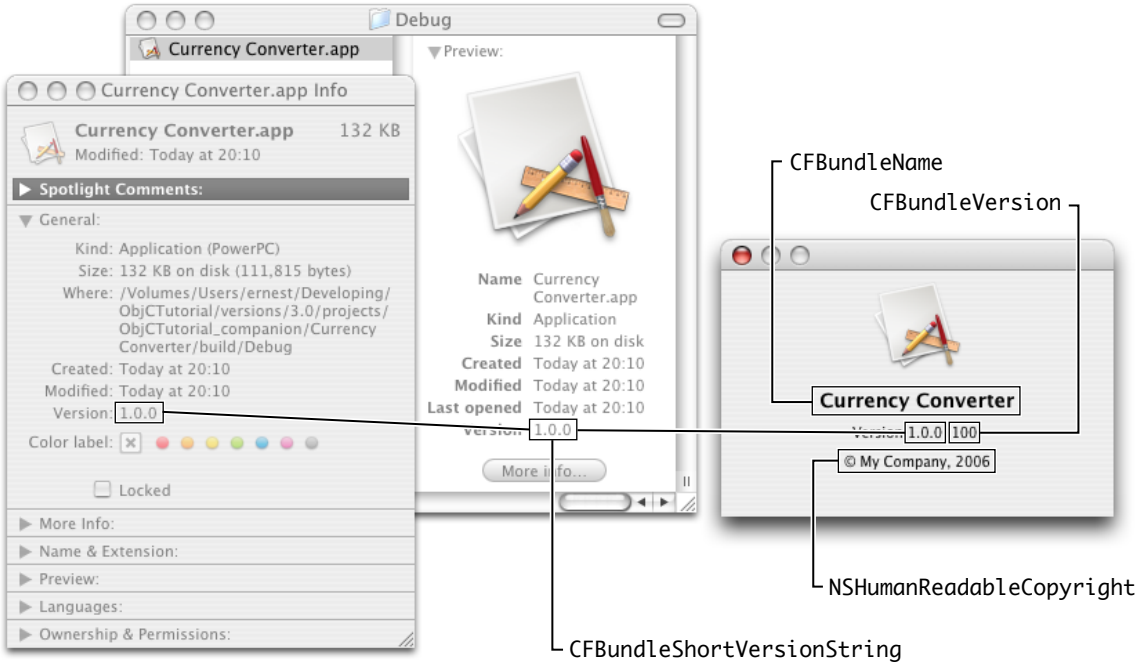
In the Dock, Control-click Currency Converter and choose Show In Finder from the shortcut menu, as shown in Figure 5-4

Figure 5-4 Locating the application bundle from the Dock



The Finder opens a window, (shown in Figure 5-5) displaying the Currency Converter application bundle. Notice that the release version number (`CFBundleShortVersionString`) appears in the preview column (in column view) and in the Info window for the application bundle. The About Currency Converter window shows the application name (`CFBundleName`), build version number (`CFBundleVersion`) in parenthesis, release version number, and copyright text (`NSHumanReadableCopyright`).

Figure 5-5 Application properties as seen by the user



Quit Currency Converter. Now the only essential application identification information left unspecified for Currency Converter is its icon.

Creating the Currency Converter Icon File

When you create a Cocoa application without specifying an icon for it, the Finder and the Dock assign it the generic application icon, as shown in Figure 5-5 (page 64) To make your applications more appealing to their users and to differentiate them from other applications, you should give your applications distinctive icons. As a result, your applications stand out from other applications in Finder windows and in the Dock. This section describes the process of creating an icon file using Icon Composer and configuring Currency Converter to use the icon file.

Follow these steps to create the icon file for Currency Converter:

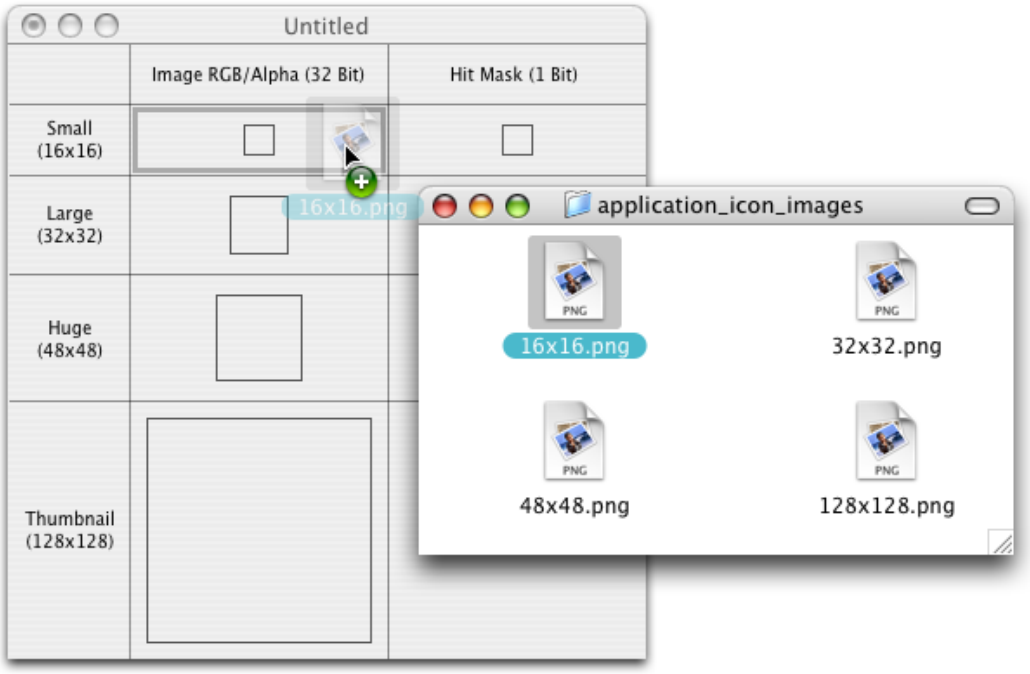
1. Launch Icon Composer, located in `/Developer/Applications/Utilities`. Icon Composer displays an empty icon file editor window.

2. In the Finder, navigate to the `ObjCTutorial_companion/application_icon_images` directory. This directory contains four image files that depict the Currency Converter application icon.

3. Add the image files to the icon file:

a. Drag `16x16.png` from the Finder window to the Small Image image well in the icon file editor, as shown in Figure 5-6

Figure 5-6 Dragging `16x16.png` to the icon file editor



In the dialog that appears, click Extract Mask.

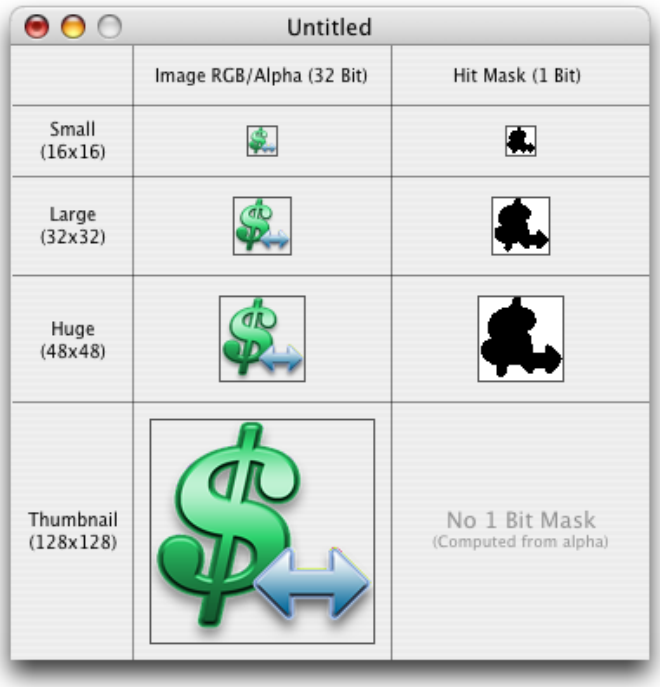
b. Drag `32x32.png` to the Large Image well and extract the hit mask.

c. Drag `48x48.png` to the Huge Image well and extract the hit mask.

d. Drag `128x128.png` to the Thumbnail Image well. Icon Composer doesn't compute a hit mask for Thumbnail images.

The icon file editor should look like Figure 5-7

Figure 5-7 Icon file editor with icon images at several resolutions



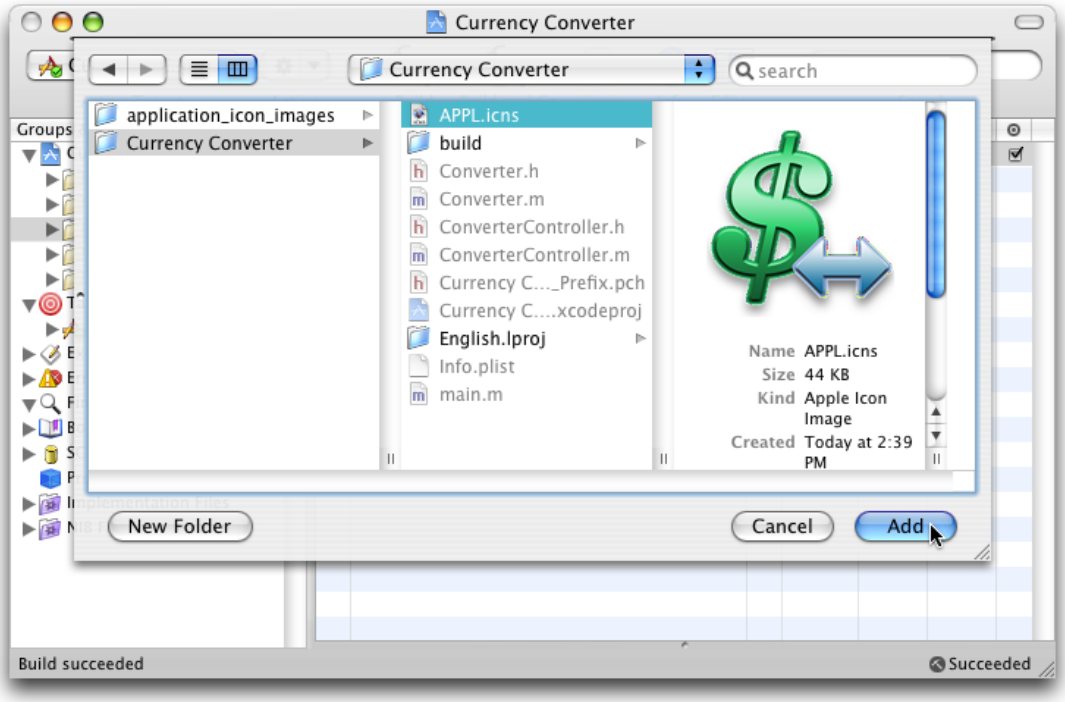
4. Save the icon file:
    a. Choose File > Save As.
    b. In the Save dialog, navigate to the Currency Converter project directory.
    c. In the Save As text field, enter `APPL.icns`.
    d. Click Save.
5. Quit Icon Composer.

Although the Currency Converter project directory contains the `APPL.icns` file, you still need to add it to the project:

1. In the Currency Converter project window, select the Resources group in the Groups & Files list.
2. Choose Project > Add to Project.

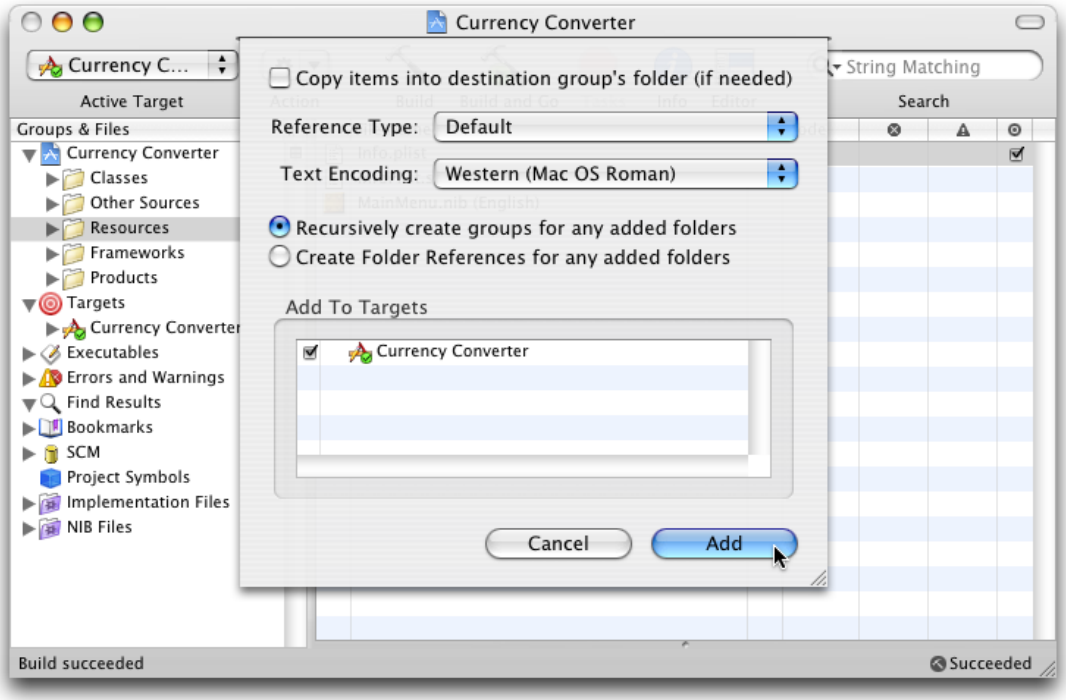
3. In the dialog that appears, select the `APPL.icns` file in the Currency Converter project directory, and click Add, as shown in Figure 5-8

Figure 5-8 Selecting the icon file to add to the Currency Converter project



4. In the dialog that appears next, shown in Figure 5-9 click Add.

Figure 5-9 Specifying project file-addition options



Finally, set the application-icon-filename property in the Currency Converter `Info.plist` file:

1. In the Currency Converter project, locate the `CFBundleIconFile` key in the `Info.plist` file.
2. Set the corresponding `string` element to `APPL.icns`.
3. Save the `Info.plist` file.

The `Info.plist` file should look like Listing 5-2 The highlighted line points out the last modification.

Listing 5-2 Specifying a custom application icon in the Currency Converter `Info.plist` file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>CFBundleDevelopmentRegion</key>
    <string>English</string>
    <key>CFBundleExecutable</key>
    <string>${EXECUTABLE_NAME}</string>
    <key>CFBundleIconFile</key>
    <string>APPL.icns</string>
    <key>CFBundleIdentifier</key>
    <string>com.mycompany.CurrencyConverter</string>
```

```
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundleName</key>
    <string>${PRODUCT_NAME}</string>
    <key>CFBundlePackageType</key>
    <string>APPL</string>
    <key>CFBundleSignature</key>
    <string>????</string>
    <key>CFBundleVersion</key>
    <string>100</string>
    <key>NSMainNibFile</key>
    <string>MainMenu</string>
    <key>NSPrincipalClass</key>
    <string>NSApplication</string>
    <key>CFBundleShortVersionString</key>
    <string>1.0.0</string>
    <key>NSHumanReadableCopyright</key>
    <string>© My Company, 2006</string>
</dict>
</plist>
```

Clean the project. Build and run the application. Currency Converter now has a distinguishing icon, shown in Figure 5-10

Figure 5-10 Currency Converter sporting an elegant icon

Configuring your applications appropriately is essential for providing a good experience to your customers. This practice also lets you to take advantage of Mac OS X services, such as **managed installations**. Managed installations rely on application identifiers and version numbers to determine, for example, whether a user is trying to install an earlier version of an application over a newer one. As a result, the user is protected from unknowingly performing a potentially disruptive operation.

You benefit from managed installations because, when you create a delivery solution for your product, you don't have to worry about the details of installing the product for the first time, upgrading to a newer version, or downgrading to an earlier version. Mac OS X handles the details for you.

Expanding on the Basics

This chapter describes other integrated components of Cocoa. Do you recall how little code was required to build Currency Converter into a working application? You may be surprised how many classes and features come packaged with Cocoa to minimize the time you spend coding.

For Free With Cocoa

The simplest Cocoa application, even one without a line of code added to it, includes a wealth of features 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 Interface Builder's 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.

If you closed your application, run it again. Once the Currency Converter window is open, move it around by its title bar. Here are some other tests you can do:

1. Open the Edit menu. Its items appear and then 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 to get the application back.
3. Click the close button; the Currency Converter window disappears.

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 autoresizing attributes of the window and its views so that the window's elements would resize proportionally to the resized window or would retain their initial size (see Interface Builder Help for details on autoresizing).

Controls and Text

The buttons and text fields of Currency Converter come with many built-in behaviors. Notice that the Convert button pulsates as is the default for buttons associated with the Return key. Click the Convert button. Notice how the button is highlighted for a moment.

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 insertion point 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 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 submenu (whose items are added by other applications at runtime) you can communicate with other Mac OS X applications. You can manage your application's windows with the Window menu.

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 text view object in the Text palette. Your application gets the Font window and a font manager "for free." Text elements in your application can use this functionality right out of the box. See *Font Panel* for more information.
- 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.
- Thanks to the PDF graphics core of Mac OS X, many objects that display text or images can print their contents as PDF documents.

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. You could also save your Currency Converters as documents, with a little extra code.

See Document-Based Applications in Cocoa Design Guidelines Documentation for more information.

File Management

An application can use the Open dialog, which is created and managed by the Application Kit framework, 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:

- **Pasteboards**. Pasteboards are a global facility for sharing information among applications. Applications can use the pasteboards to hold data that the user has cut or copied and may paste into another application. Each pasteboard can have multiple pasteboards accepting multiple data types.
- **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.

Internationalization

Cocoa provides API and tool support for internationalizing the strings, images, sounds, and nib files that are part of an application. Internationalization allows you to easily localize your application to multiple languages and locales without significant overhead.

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), 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 executes the Print command, an appropriate dialog helps to configure the print process. The output is WYSIWYG (what you see is what you get).

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—known as "help tags"—for your application using the Interface Builder inspector. After you've entered the help tag text for the user-interface elements in your application, a small window containing concise information on the element appears when the user places the pointer over these elements.

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.

Turbo Coding With Xcode

When you write code with Xcode you have a set of "workbench" tools at your disposal. Some of these tools are described in the following sections:

Project Find

Project Find (available from the Find window in Xcode) 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, and so forth) on disk.

For C-based languages, Xcode automatically gathers indexing information while the source files are being compiled; therefore, it is not necessary to build the project to create the index before you can use Project Find.

Code Sense and Code Completion

Code Sense indexes your project files to provide quick access to the symbols in your code and the frameworks linked by your project. Code Completion uses this indexing to automatically suggest matching symbols as you type. These features can be turned on in the Code Sense preferences pane in the Xcode Preferences window.

Since Code Sense and Code Completion use Xcode's speedy indexing system, the suggestions they provide appear instantaneously as you type. If you see an ellipsis (...) following your cursor, Xcode could not find an exact match.

Integrated Documentation Viewing

Xcode supports viewing HTML-based ADC Reference Library content directly in the application. You can access reference material about the Xcode application, 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 reference information and header files. To retrieve the reference information for an identifier, Option–double-click it; to retrieve its declaration in a header file, Command–double-click it.

The search bar in the Developer Documentation window also offers you a quick and easy way to find an identifier in any of Cocoa's programming interfaces.

Indentation

In the Indentation preferences pane in the Xcode Preferences window you can set the characters at which indentation automatically occurs, the number of spaces per indentation, and other global indentation characteristics. The Format menu also offers commands to indent code blocks individually.

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, Xcode emits a beep.

Emacs Bindings

You can use the most common Emacs commands in Xcode'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 in the Key Bindings preferences pane in Xcode Preferences to substitute other "command" keys—such as the Option key or Shift-Control—for Emacs's Control or Meta keys. For information on key bindings, see About Key Bindings in *Text Input Management*.

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 sections summarize some of the basic aspects of the language. See *The Objective-C Programming Language* for 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. 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 differ in 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 paramount.

Declarations

Dynamically type objects by declaring them as `id`:

```
id myObject;
```

Since the class of dynamically typed objects is resolved at runtime, 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 for class methods) and the beginning of the method name. Methods that return nothing must 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.

Document Revision History

This table describes the changes to *Cocoa Application Tutorial*.

Date	Notes
2006-11-07	Changed title from "Cocoa Application Tutorial Using Objective-C."
2006-05-23	Added chapter on setting essential application properties, including the application identifier, the application icon filename, and version information.
	Added "Configuring Currency Converter" (page 57) to explain how to configure application properties.
	Improved instructions to customize the Currency Converter default menu hierarchy in "Set the Application Name in the Menu." (page 25)
	Made minor editorial changes.
2006-01-10	Added the finalized Currency Converter project. Specified Xcode Tools version requirements. Made other small changes.
	Added finalized Currency Converter project as this document's companion archive.
	Added "Finalize ConverterController.h" (page 49) to instruct that the `#import "Converter.h"` code line has to be added to `ConverterController.h`.
	Updated the introduction and "Creating the Currency Converter Project and User Interface" (page 17) to specify the development environment required to successfully complete the tasks described in this document.
	Updated "Paths for Object Communication: Outlets, Targets, and Actions" (page 37) to indicate how Interface Builder (in Xcode Tools 2.2 and later) defines outlets in the header files it generates.
	Updated "Define the User Interface and Model Outlets of the ConverterController Class" (page 39) to explain why the `converter` outlet cannot be typed.

Date	Notes
	Corrected Listing 3-2 (page 50) by moving the method declaration after the right curly brace.
	Corrected Listing 3-3 (page 50) by including a tag number in the code line with the right brace.
	Corrected Listing 3-4 (page 51) by removing the `#import "Converter.h"` code line.
2005-10-04	Updated for Mac OS X v10.4 and Xcode Tools 2.2. Changed title from "Developing Cocoa Objective-C Applications: A Tutorial."
2003-08-07	Updated for new Developer Tools and Mac OS X version 10.3.
	Screenshots updated for Xcode.
	Chapter reorganization to flatten the document structure.
2003-05-03	Revision history was added to existing document. It will be used to record changes to the content of the document.