Cocoa Application Tutorial

General

2009-08-03

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

Contents

Figures and Listings

Chapter 7 **Configuring Your Application 55**

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.

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

This document uses Xcode 3.2 running on Mac OS X version 10.6.

Organization of This Document

This document consists of the following chapters:

- "The Essence of Cocoa" (page 11) introduces basic concepts whose understanding is required when developing Cocoa applications.
- "Creating a Project in Xcode" (page 15) guides you through creating a project using Xcode.
- "Defining the Model" (page 19) guides you through defining the underlying functionality of an application.
- "Defining the View: Building the User Interface" (page 23) guides you through the development of a basic user interface using Interface Builder.
- "Bridging the Model and View: The Controller" (page 39) shows how to create a controller object to mediate communication between the model and view.
- "Building and Running Your Application" (page 51) explains how to build and test the application.
- "Configuring Your Application" (page 55) explains how to configure the basic identifying properties that application bundles require, including the version information and application icon.
- "Expanding on the Basics" (page 67) explains some of the behavior Cocoa applications get by default.

Goals for Learning

Throughout this tutorial you will learn:

- What Cocoa is

- What the application development process looks like in an object-oriented environment
- How to make a Cocoa application
- Where to go from here by adapting your knowledge

Prerequisites

In order to maximize the instructional potential of this document, you should know C but not necessarily object-oriented programming or application design.

To help you troubleshoot problems as you follow the tutorial, 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.

The Xcode development environment is part of the Mac OS X installation media or available for download from developer.apple.com. You must install Xcode on your computer before following the instructions in this document.

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 is the basis of 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

This chapter covers the most common object-oriented design pattern used in Cocoa and shows how that paradigm is applied to application development. A design pattern is a template for a design that solves a general, recurring problem in a particular context. If you have done object-oriented design before, you may be wondering how that model fits into the world of Cocoa. This tutorial will help you understand what Cocoa is as an object-oriented framework. If you've done only procedural programming before, don't worry. This tutorial also teaches the basics of the object-oriented programming. You will learn the MVC design pattern, which is a very common practice used by application developers.

What Is Cocoa?

Cocoa is an object-oriented library of tools that contains many of the objects and methods needed to develop great applications for Mac OS X. By providing these tools to programmers, it takes the tedium out of writing applications. It also makes user interface development simple. If you've ever tried to design a user interface by writing the actual code for it, you understand how difficult it is. In this tutorial, you'll learn how easy it is to create a beautiful user interface by merely dragging objects onto a window.

If you would like more detail on what you can do with Cocoa and how it fits into Mac OS X, see *Cocoa Fundamentals Guide*.

Classes and Objects

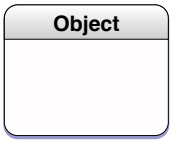
An object consists of both data and methods for manipulating that data. An object is an instance of a class, which means that there is memory allocated for that specific instance of the class. An essential characteristic of objects is that they encapsulate data. Other objects or external code cannot 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 object-oriented 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 invokes methods corresponding to messages that are passed to it and may return data to whoever sent the message. 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 outside world.

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.

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

Figure 1-1 An object

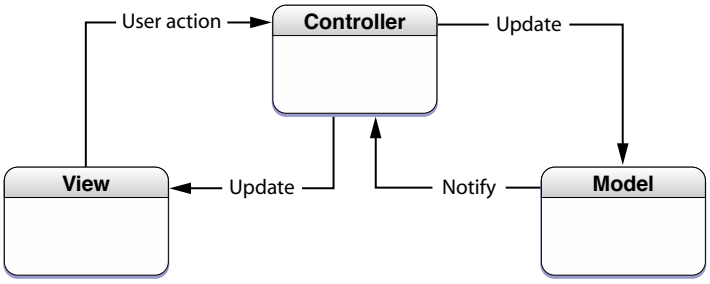


See *The Objective-C Programming Language* for a fuller description of data encapsulation, messages, methods, and other things pertaining to object-oriented programming.

The MVC Design Pattern

Model-View-Controller (MVC) is a design pattern that 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. This is the design pattern behind many designs for object-oriented programs. This design pattern aids in the development of maintainable, extensible, and understandable systems.

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, suppose you have a customer object, a common object in business applications, that holds all of the salient facts about a customer, including the customer's name, date of birth, and phone number. That object would be a model object because it represents the data your application manipulates, and has access to methods that can access and distribute that information. A more specialized model might be one in a meteorological system called Front; objects of this type would contain the data and intelligence to represent weather fronts. Model objects are not directly accessed by the user of the application. They are often reusable, distributed, persistent, and portable to a variety of platforms.

View Objects

A view object represents something visible on the user interface (a window or a button, for example). A view object is "ignorant" of the source of the data it displays. The Application Kit, one of the frameworks that Cocoa is composed of, usually provides all the basic 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 tend to be very reusable and so provide consistency between applications.

Controller Objects

Acting as mediators between model objects and view objects in an application are controller objects. Controller objects communicate data back and forth between the model and view objects. A controller object, for example, could mediate the transfer of a street address (from the customer model object) to a visible text field on a window (the view object). It also performs all the chores specific to the application it is part of. 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 always the best solution. 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.

The Currency Converter Application

The MVC design pattern, albeit a very simplified one, can be applied directly to the Currency Converter application. Currency Converter is a simple application that converts US dollars to another currency based on an exchange rate entered by the user. Though it's a very simple application, it can still be used to convey the fundamental elements of Cocoa application development using the MVC design pattern.

The underlying functionality, which is the model, converts US dollars to another currency based on an exchange rate. For this, there is a single function, `convertCurrency`, which multiplies the amount in US dollars by the exchange rate to get the amount in the other currency.

All graphical applications have a user interface that the user interacts with. This is the view. Here, the user inputs the exchange rate in one text field, the amount of dollars to convert in the next text field, and clicks the convert button, or presses Return, to perform the calculation. The final result appears in the last text box, which is the amount of the other currency.

The view needs to send the data the user entered to the model somehow. This is done by creating a controller, which gathers the exchange rate and amount in US dollars from the view, sends the values to the model, and writes the result into the view.

Creating a Project in Xcode

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. In this chapter, you will learn to create a project, and see what all the various parts of a project are for. In the process, you will learn how to navigate the Xcode user interface, and you will learn about all the different folders available for you.

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 setup process. The default settings should work for the majority of users.

Make a New Project

This section will guide you through the process of creating a project. Though it is focused on creating a project suitable for Currency Converter, you will learn about other options available to you when creating a new project.

Choose the New Project Command

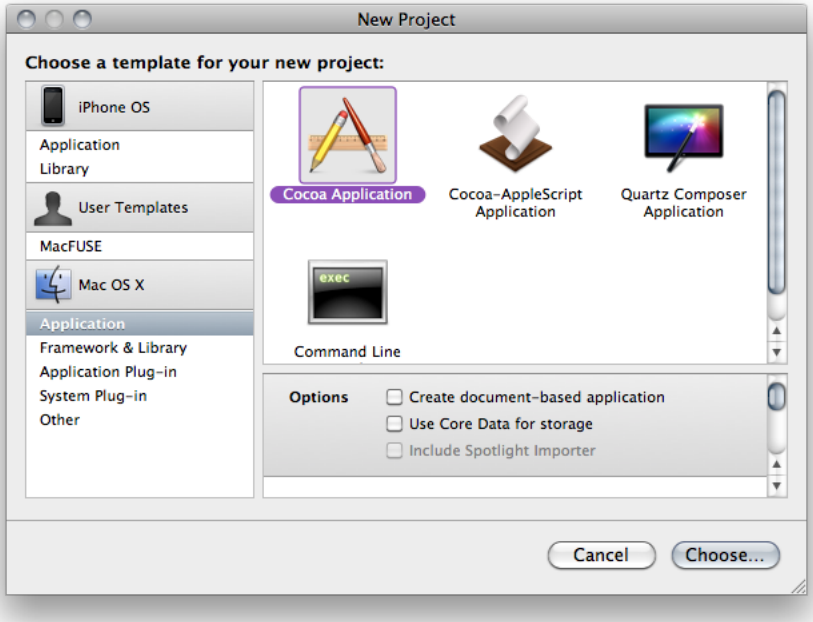
To create a project, choose File > New Project. The New Project Assistant appears.

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, use the Cocoa Application template.

1. Select Cocoa Application and click Choose…, as shown in Figure 2-2

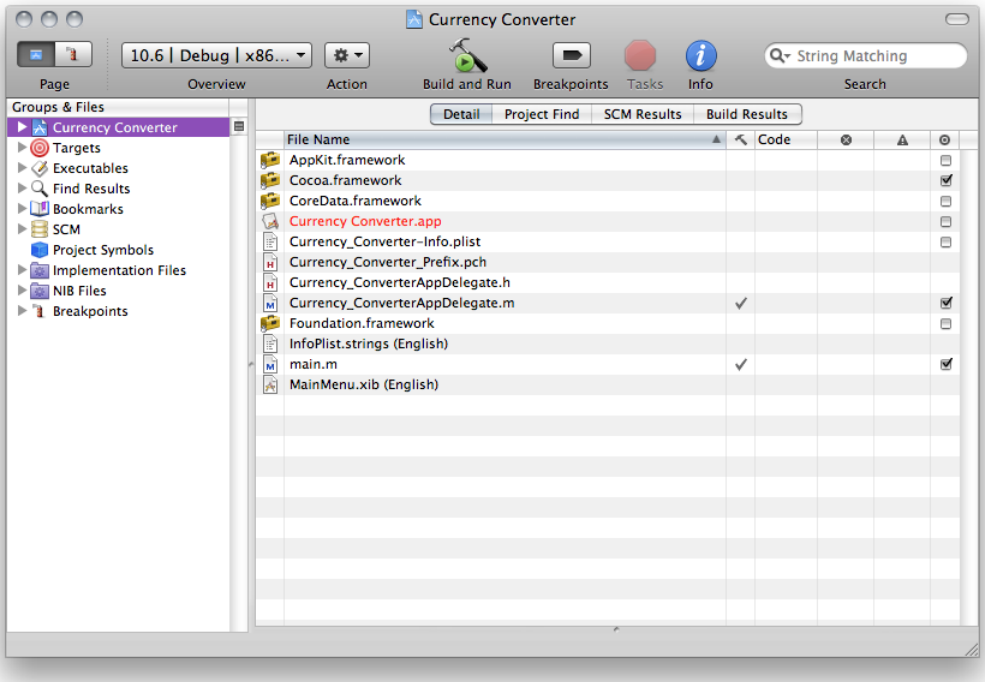
Figure 2-2 The New Project Assistant in Xcode



2. Type the project's name, Currency Converter, in the save sheet, and choose where you want to save the project. Then click save.

Xcode creates the project's files and displays the project window, shown in Figure 2-3.

Figure 2-3 The new Currency Converter project in Xcode



The Xcode Project Interface

The Groups & Files list allows you to easily navigate through your project's files and traits. All the source files, images, and other resources that make up a project are grouped in the project group, the first item in the Groups & Files list; this group is named after the project (here, Currency Converter). The project's files are grouped into subgroups, such as Classes, Other Sources, Resources, and so on, as shown in Figure 2-3. Groups are very flexible because they do not necessarily reflect either the on-disk layout of files in the project or the way the build system handles those files. Groups are purely for organizing your project. The groups created by Xcode should be suitable for most developers, but you can customize them however you like.

These are the groups Xcode sets up for Cocoa applications:

- **Classes.** This group holds all the classes required by your application.
- **Other Sources.** This group contains the `main.m` file, 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 reduce compilation time. You don't need to worry about this file.
- **Resources.** This group contains the nib files and other resources that specify the application's user interface.
- **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 access 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.** This group 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.** This group contains the executable products your project creates.
- **Errors and Warnings.** This group displays the errors and warnings encountered in your project when you perform a build.

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

`Currency Converter.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.

`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`.

What's Next?

In this chapter, you made an Xcode project that will contain all the resource files for a Cocoa application. In the next chapter, you will learn how to create the model, the basic functionality behind an application.

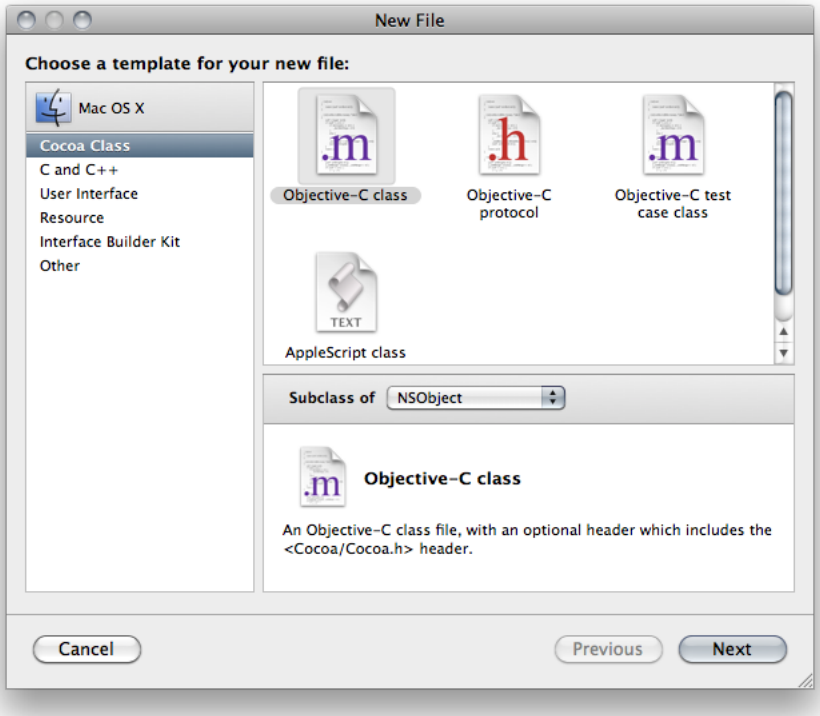
Defining the Model

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. In Currency Converter, the model class you're going to create is the converter class. In the MVC design pattern, instances of a model class do not communicate directly with the user interface.

In this chapter, you will create the model for your application. First, you will learn how to create a new class in Xcode. Then, you will define the interface for your model before implementing it. Finally, you will implement the entire functionality of the class. In the process, you will learn some of the syntax of Objective-C, including how to declare variables and methods. You will also be introduced to the concept of declared properties—a feature in Objective-C 2.0 that makes writing accessor methods incredibly quick and simple.

Specify the Model Class

1. Select the Classes group in the Groups & Files list.
2. Choose File > New File.
3. Select Objective-C class in the Cocoa Class group. Leave it as a subclass of NSObject, and click Next.



4. Name your new class `Converter` and click Finish.

Xcode automatically puts the interface (`Converter.h`) and implementation (`Converter.m`) files for your new class in the Classes group and opens the interface file so you can get right to work.

Declare the Model Interface

The model for the currency converter is a simple class that contains two variables: a dollar amount in US dollars, and the exchange rate. The model's job is to multiply these two numbers and return the result. This means the model needs:

- One class: Converter
- Two variables of type `float`: `sourceCurrencyAmount` and `rate`
- One method: `(float)convertCurrency`

Declare Instance Variables

The two variables are defined between the braces of the converter class. Define the variables:

1. If `Converter.h` is not already open for editing, double-click `Converter.h` in the Classes group of the Groups & Files menu. This opens the file in an editor window.
2. Insert the third line in Listing 3-1 into `Converter.h` in the location shown.

Listing 3-1 Declaration of the member variables in `Converter.h`

```
#import <Cocoa/Cocoa.h>
@interface Converter : NSObject {
    float sourceCurrencyAmount, rate;
}
```

Declared Properties and Accessor Methods

Objects encapsulate data, as explained in "Classes and Objects" (page 11). The scope of variables in an object is limited to that particular object—that is, the instance of that class. One instance of a customer class, for example, can only see its own data, not the data of any other customer object. But say a specific customer needs to compare itself with another customer. In order for this to be possible, classes supply accessor method to read and write to the data an object encapsulates. This gives the classes discretion over what data they share. For example, if a class only wants to use certain variables for internal use, it can simply not have accessor methods for those variables. Classes can also provide external entities with read-only access to its data. In that case, the class provides methods that get values for that data, but no methods that set values.

Objective-C 2.0 provides a feature called **declared properties**. A property declaration is, effectively, a shorthand for declaring the setter and getter for an attribute of an instance of a class. In addition to the declaration itself, there are directives to instruct the compiler to synthesize the accessors and to inform the compiler that you will provide the methods yourself at runtime. There are two parts to a property, its declaration and its implementation.

You declare a property as follows:

```
@property(attributes) Type variableNameList;
```

where *attributes* is a comma-separated list of keywords such as `readwrite` and `copy`, for example:

```
@property(copy) NSString *name;
```

Your converter class wants to provide access to its data to external objects, so you need to declare accessor methods.

1. Place your insertion point after the closing brace of the instance method declarations in `Converter.h`, just before the `@end` directive.
2. Add the following property declaration for your two instance variables.

   ```
   @property(readwrite) float sourceCurrencyAmount, rate;
   ```

At compile time, the compiler treats this as if you had declared the following methods:

```
- (float)sourceCurrencyAmount;
- (void)setSourceCurrencyAmount:(float)newSourceCurrencyAmount;
- (float)rate;
- (void)setRate:(float)newRate;
```

The property declaration simply declares what the accessor methods will look like. In the next section, you will provide an implementation for the accessor methods.

Declare the Model Method: convertCurrency

The one method the model requires is a simple function to multiply the two values encapsulated by the converter class.

Add the following line as the next line of code in `Converter.h`:

```
- (float)convertCurrency;
```

Although this method takes no arguments, it nonetheless provides the result of multiplying the source currency amount with the conversion rate. This is possible because instance methods have access to the instance variables of their object. At runtime, the instance method treats all instance variables as if they were local variables. It is common practice to use accessor methods even when the variable is in scope. This practice makes it clear that the code is accessing an instance variable, rather than a local variable.

Implementing the Model

Now it's time to define the behavior of the functions you declared in `Converter.h`.

1. In the Classes group of the Groups & Files menu, double-click `Converter.m` to open the file for editing.
2. Create the getters and setters for the two member variables—`sourceCurrencyAmount` and `rate`.

   Remember you used `@property` to create the prototypes for the getter and setter methods in `Converter.h`. Similarly, you can use the `@synthesize` directive in `@implementation` blocks to trigger the compiler to generate accessor methods for you.

   Add the following line into `Converter.m` after the `@implementation Converter` line:

   ```
   @synthesize sourceCurrencyAmount, rate;
   ```

This line defines the body of the getter and setter methods for the variables `sourceCurrencyAmount` and `rate` based on the properties you set in the `Converter.h` file.

For more information on properties and the various options available, see Declared Properties.

Define the convertCurrency Method

Insert the definition of the convertCurrency method into `Converter.m`, as shown in Listing 3-2.

Listing 3-2 Definition of the `convertCurrency` method in `Converter.m`

```
#import "Converter.h"

@implementation Converter
@synthesize sourceCurrencyAmount, rate;

- (float)convertCurrency {
    return self.sourceCurrencyAmount * self.rate;
}

@end
```

The `convertCurrency` method multiplies the values of the converter class's two instance variables and returns the result. Notice the use of the notation `self.sourceCurrencyAmount` to access the `sourceCurrencyAmount` instance variable. `self` is an Objective-C keyword that stands for "the object executing the method". Declared properties can be accessed using the familiar dot notation, just like accessing fields in a struct.

What's Next?

You just defined and implemented the basic functionality of your application by creating the model. In the next chapter, you will create the view—the user interface for the application.

Defining the View: Building the User Interface

Every application has a user interface. It can be a tedious and infuriating task to design user interfaces without a good program to help you. Interface Builder, an application supplied with Xcode, makes designing user interfaces simple, and so turns a tedious task into something easy.

This chapter teaches you how Interface Builder integrates with Cocoa. It guides you through building the user interface for your application. You learn where to locate the tools you need and how to use the features in Interface Builder to create the perfect user interface for your application.

User Interface Elements and Interface Builder

Interface Builder is an application that helps you design the user interface and connect the different parts of your application. It contains a drag-and-drop interface for UI design, and an inspector that allows you to view and change the properties of the various objects in your design, as well as the connections between them.

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 as "flattened" or "serialized" objects—meaning that the object has been encoded in such a way that it can be saved to disk 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, at runtime, becomes the actual object that owns the nib file (typically the one that loaded the nib file from disk).
- **Images.** Image files that appear somewhere in your application's interface, 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 because it doesn't have access to their 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. When you save a document, information about the connections between the objects in the nib is archived along with the objects themselves.

Windows and Menus in Cocoa

Windows have numerous characteristics. They can be onscreen or offscreen. Onscreen windows are "layered" on the screen in tiers managed by the window server. A typical Cocoa window has a title bar, a content area, and several control objects.

Key and Main Windows

Onscreen windows can carry a status: key or main. Key windows respond to keypresses 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 key window (typically a panel such as the Font window or an Info window) have a direct effect on the main window.

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. 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 the Application Kit, 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 applications, which use the top-left corner. Views draw themselves in coordinate systems transformed from (and relative to) this base coordinate system.

Creating the Currency Converter Window

Currency Converter has a main window that the user will interact with. This section guides you through the process of designing Currency Converter's main window.

Create the Window

You use Interface Builder to define an application's user interface. Open your application's main nib file in Interface Builder:

1. Locate `MainMenu.xib` in the Resources subgroup of your project.
2. Double-click the nib file to open it in Interface Builder.

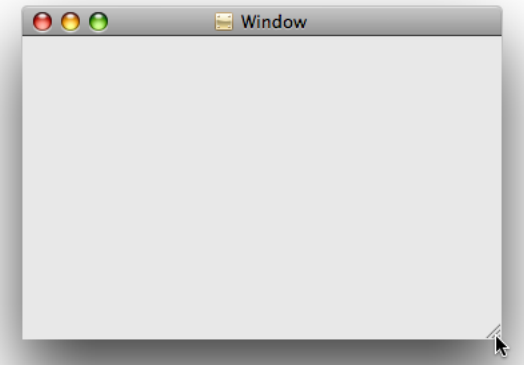
A default menu bar, the `MainMenu.xib` window, the Library, and an empty window titled Window appear when the nib file is opened.

nibs and xibs: Interface Builder stores object information in two forms. The xib file is analogous to a source file—it contains a detailed description of the interface specification, including various housekeeping information that Interface Builder uses while editing. At build time, the xib file is compiled down to a nib file, which is analogous to an object file. It contains only the information needed by your application at runtime. Because the xib file format is really just an editing format, this document will refer to both of these file types as nib files.

Resize the Window

Make the window smaller by dragging the bottom-right corner of the window inward, as shown in Figure 4-1.

Figure 4-1 Resizing a window manually

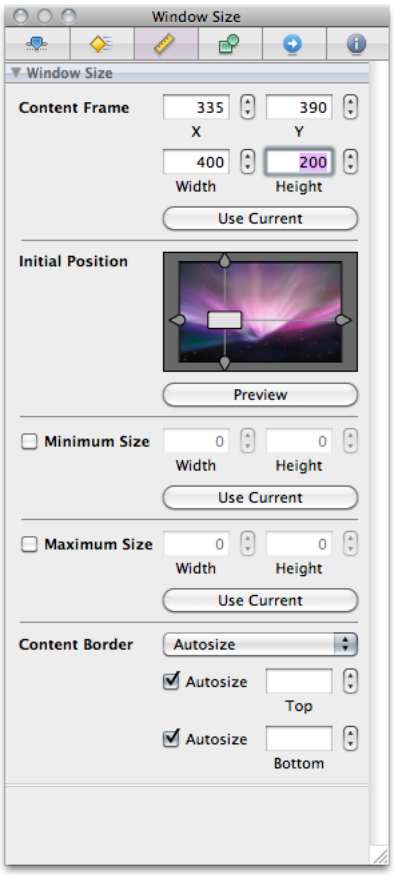


You can resize the window more precisely by using the Window Size inspector.

1. Select the window by clicking its titlebar or selecting its icon in the MainMenu.xib window.
2. Choose Tools > Inspector.
3. Click the Size icon, which looks like a yellow ruler.
4. In the Content Size & Position group, you can set the width, height, and default x and y coordinates of the window.

5. Use the width and height fields to enter a width of 400 and a height of 200, as shown in Figure 4-2.

Figure 4-2 Resizing a window with the inspector



6. In Initial Position, you see a graphical representation of your window and your screen. Drag the window to the top-left corner of the box for a traditional default window position.

Changing the Title of the Window

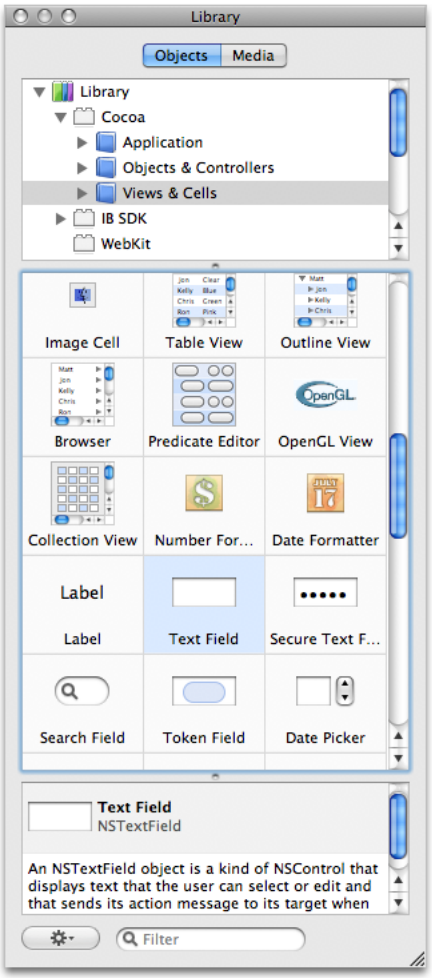
The default title of the window is Currency Converter, the name of your project. If you wish, you can change the title in the Attributes tab of the inspector. Currency Converter is a fine title for your window, so leave it as it is.

Add the Text Fields

Interface Builder's Library window contains user interface elements that you can drag into a window or menu to create an application's user interface.

1. If the Library is not already open, shown in Figure 4-3—choose Tools > Library.

Figure 4-3 Cocoa Views and Cells in the Interface Builder Library window



2. In the list at the top of the Library, open Library, then Cocoa, and select Views & Cells.

3. Notice how in Figure 4-3 the icons have labels. To display labels, choose View Icons & Labels from the gear menu at the bottom of the Library window. This will make it easier to locate the objects you are looking for.

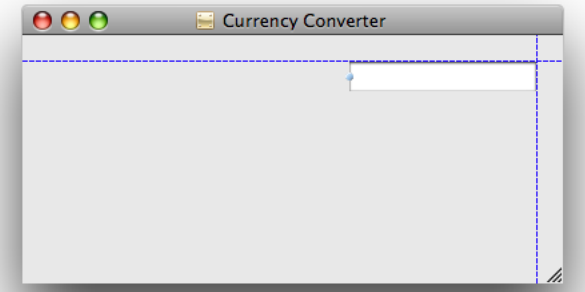
4. Place a text field in the Currency Converter window.

   Find the Text Field item (shown highlighted in Figure 4-3) and drag the Text Field object from the library to the top-right corner of the 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.

5. Increase the text field's size so 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 4-4.

Figure 4-4 Resizing a text field



6. Add 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.

   To duplicate the text field in the Currency Converter window:

   a. Select the text field, if it is not already selected.

   b. Choose Duplicate (Command-D) from the Edit menu. The new text field appears slightly offset from the original field.

   c. 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.

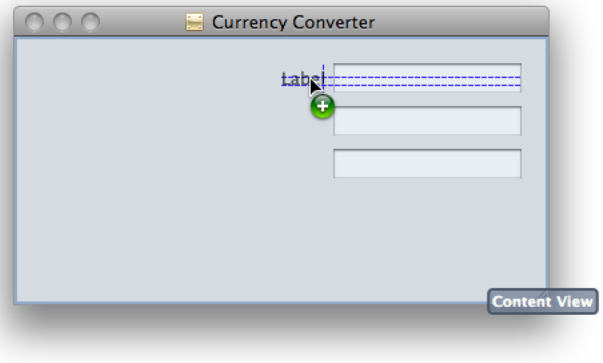
   d. To make the third text field, press Command-D again.

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

Assign Labels to the Fields

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

1. Drag a Label element onto the window from the Cocoa library.

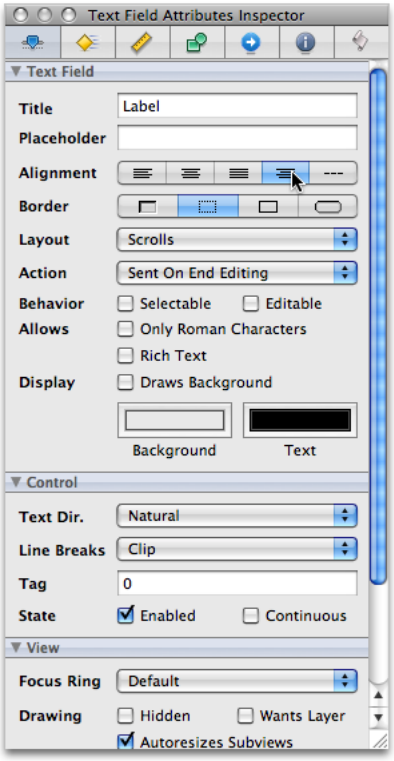


2. Make the label right aligned:

   With the label element selected, click the fourth button from the left in the Alignment area in the Text Field Attributes tab in the Inspector, as shown in Figure 4-5.

> **Note:** To open the inspector in Interface Builder, choose Tools > Inspector.

Figure 4-5 Right aligning a text label in Interface Builder

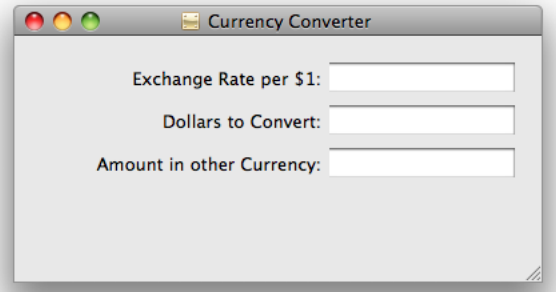


3. Enter `Exchange Rate per $1:` in the Title text field.

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

5. Expand the labels to the left so their entire titles are visible, as shown in Figure 4-6.

Figure 4-6 Text fields and labels in the Currency Converter window



Change the Text Field Attributes

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.

1. Select the third text field.

2. In the Inspector, navigate to the Text Field Attributes tab.

3. Click the Editable option to remove the checkmark, so users are not allowed to alter the contents of the text field. Leave the Selectable option checked so users can copy the contents of the text field to other applications.

Configure a Button

Providing a button to convert the currency amount creates a clear action for the user to take. To add a button to the Currency Converter window:

1. Drag the Push Button object from the library to the bottom-right corner of the window.

2. Double-click the button and change its title to Convert.

3. In the Key Equiv. section of the Button Attributes inspector, click the gray box. Now press Return. A return symbol should appear in the gray box. This makes the button respond to the Return key as well as to clicks.

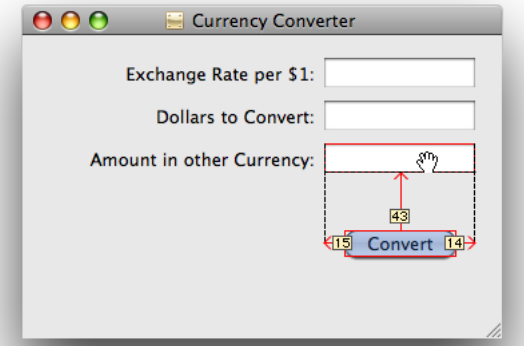
4. Align the button under the text fields.

    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, Interface Builder shows you the distance in pixels 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 4-7.

Figure 4-7 Measuring distances in Interface Builder



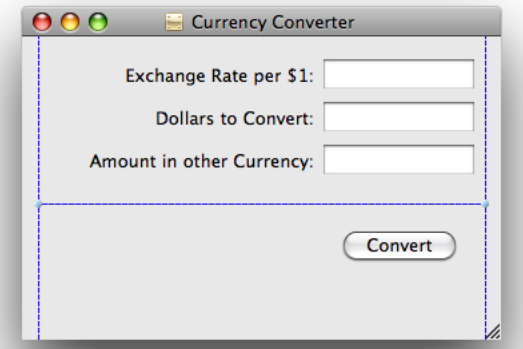
Add a Separator to Group Controls

The final interface for Currency Converter has a line separating the text fields and the button. To add the line to the Currency Converter window:

1. Drag a Horizontal Line object from the Library to the Currency Converter window.

2. Drag the endpoints of the line until the line extends across the window, as shown in Figure 4-8.

Figure 4-8 Adding a horizontal line to the Currency Converter window



3. Move the Convert button up until the layout guide appears below the horizontal separator, and shorten the window until the horizontal layout guide appears below the Convert button.

Refining the Layout and View Functionality

The Currency Converter window now contains all the necessary objects for it to function as designed. This section guides you through formatting the window and menu to make it more user friendly.

Aligning Objects in a Window

In order to make an attractive user interface, you should 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 submenu in the Layout menu provides various alignment commands and tools, including the Alignment window, which contains controls you can use to perform common alignment operations.

Finalize the Window Size

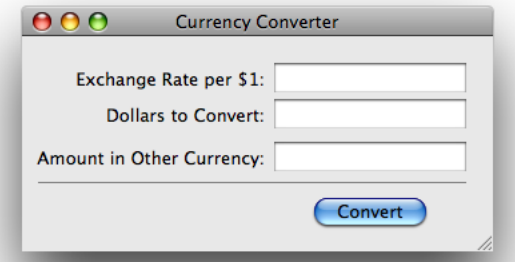
The Currency Converter interface is almost complete. The finishing touch is to resize the window so that all the user interface elements are centered and properly aligned to each edge. Currently, the objects are aligned only to the top and right edges.

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 labels.
2. Choose Layout > Size to Fit to resize all the labels to their minimum width.
3. Choose Layout > Alignment > Align Right Edges.
4. Drag the labels towards the left edge of the window, and release them when the layout guide appears.
5. Select the three text fields and drag them to the left, again using the guides to help you find the proper position.
6. Shorten the horizontal separator and move the button into position again under the text fields.
7. Make the window shorter and narrower until the layout guides appear to the right of the text fields.
8. Select the Window object.
9. Because you have made the window the perfect size for its components, you don't want the user to resize the window. In the Attributes tab of the Inspector, uncheck Resize under the Controls section.

At this point the application's window should look like Figure 4-9.

Figure 4-9 The Currency Converter final user interface in Interface Builder



Enable Tabbing Between Text Fields

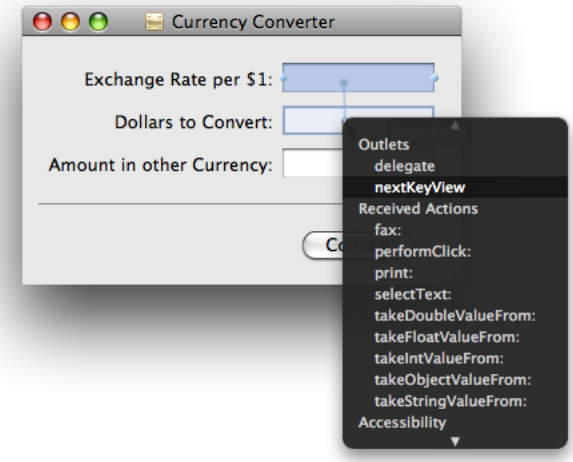
The final step in composing the Currency Converter user interface has more to do with behavior than with appearance. You want the user to be able to tab from the first editable field to the second, and back to the first. Many objects in the Interface Builder Library have an outlet named `nextKeyView`. This variable identifies the next object to receive keyboard events when the user presses the Tab key. A Cocoa application by default makes its "best guess" about how to handle text field tabbing, but you can improve the default behavior by explicitly setting the next key view according to the specific needs of your application. You set the next key view for an object by connecting the `nextKeyView` outlet.

To configure correct tabbing between text fields:

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 4-10. Press and hold the control key, then click on the Exchange Rate text field and drag the connection line to the Dollars to Convert text field. When the Dollars to Convert text field highlights, release the mouse button.

Figure 4-10 Connecting `nextKeyView` outlets in Interface Builder



3. Select `nextKeyView` under Outlets. This sets the next key view of the Exchange Rate text field (the object you dragged the connection from) to the Dollars to Convert text field (the object you dragged the connection to). At runtime, Cocoa uses the `nextKeyView` outlet to determine the next object to get keyboard focus after the Tab key is pressed.

4. Repeat the same procedure, dragging from the Dollars to Convert text field to the Exchange Rate text field.

Now that you've set up tabbing between text fields, you must tell Currency Converter which text field will be selected first. You do this by setting an initial first responder.

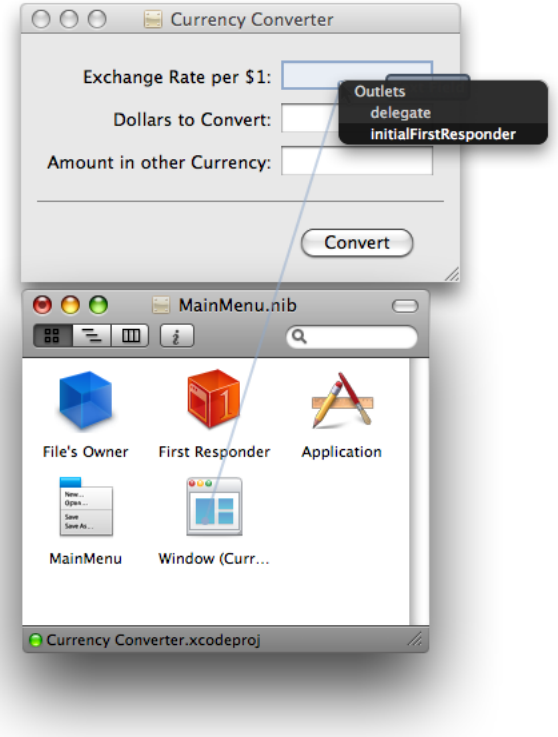
Set the First Responder Text Field

The `initialFirstResponder` outlet determines the default selected object when your application starts. If you do not set this outlet, the window sets a key loop and picks a default initial first responder for you.

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 4-11.

Figure 4-11 Setting the `initialFirstResponder` outlet in Interface Builder



2. Select `initialFirstResponder` under Outlets.

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 > Simulate Interface.
3. Try various user operations, such as tabbing, and cutting and pasting between text fields.
4. When finished, choose Quit Cocoa Simulator from the application menu to exit test mode.

What's Next?

In this chapter, you built the user interface for your application. You placed and organized all the various objects the user will interact with as they use the application, and you configured them to properly interact with each other. The next chapter will guide you through the creation of a controller that will connect the view you just created with the model you implemented in the previous chapter.

Bridging the Model and View: The Controller

Your model needs to be able to communicate with the view so it can understand what data is in the view's text fields. Your view needs to connect with the model so it can receive updates when calculations are performed whose results must be shown to the user. The controller layer of an application consists of one or more classes whose purpose is to communicate between the model and the view.

In this chapter, you will learn about the paths used for communication between objects and how to use Interface Builder to define those paths. You will also learn how to define the controller's behavior and how it can communicate between the model and the view using the paths you created.

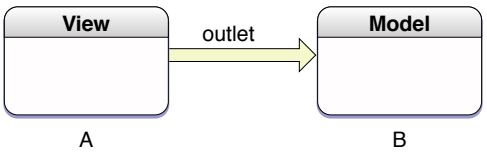
Paths for Object Communication: Outlets, Targets, and Actions

In Interface Builder, you use outlets and actions to specify the paths for messages traveling between the controller and other objects. 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 5-1 illustrates how an outlet in one object points to another object.

Figure 5-1 An outlet pointing from one object to another



An object 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, model objects (usually instances of custom classes), and even the application object itself. Outlets are special instance variables because they can be set in Interface Builder.

Outlets are declared as:

```
IBOutlet id variableName;
```

Note: `IBOutlet` is a null-defined macro that the C preprocessor removes at compile time. Interface Builder uses it to identify outlet declarations so that it can display them when connecting outlets visually.

Objects with `id` as their type are dynamically typed, meaning the class of the object is determined at runtime. You can use `id` as the type for any object. 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 of a specific class:

```
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 the ability of Interface Builder to initialize 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 parameter of the controller instance that receives the value from the exchange rate in the view would be initialized with a reference to the Exchange Rate text field object (see "Connect the ConverterController Instance to the Text Fields" (page 45) 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.

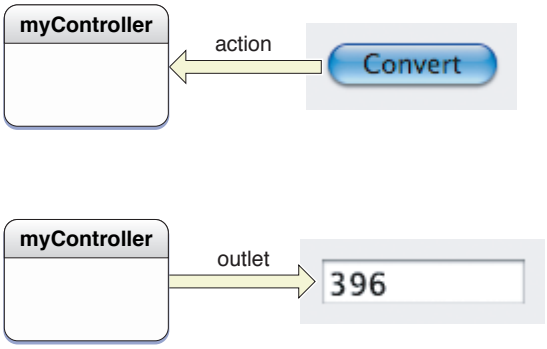
Target/Action in Interface Builder

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 the Application Kit 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 5-2 shows the interactions between the `ConverterController` class, the Convert button, and the Amount in Other Currency: field.

Figure 5-2 Relationships in the target-action paradigm



In the Actions column (in the Connections pane of the inspector), all action methods are defined by the class of the target object and known by Interface Builder. Interface Builder identifies action methods because their names follow this syntax:

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

Note: `IBAction`, like `IBOutlet`, is a null defined macro, that the C preprocessor removes at compile time. Interface Builder uses it to identify action declarations so it can display them when connecting actions visually.

Which Direction to Connect?

Usually the outlets and actions you connect belong to a custom subclass of `NSObject`. For these occasions, you need only to follow a simple rule to know 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 object you want to send the message to.
- To make an outlet connection, create the connection from the custom object to another object (another custom object or a 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 object with the outlet needs to find some other object, so the connection is from the object with the outlet to the other object. With actions, the control object needs to find the target, so the connection is from the control object to the custom object.

Defining the Controller Class

Interface Builder is a versatile tool for application developers. It enables you to not only compose the application's graphical user interface, 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 rest of the chapter shows how to define the `ConverterController` class and connect it to Currency Converter's user interface.

1. Select the Classes group in your project's Groups and Files list.
2. Choose File > New File.
3. Choose Objective-C Class and click Next.
4. Name the file `ConverterController.m`.
5. Make sure "Also create 'ConverterController.h'" is selected and click Finish.

Defining the Outlets for 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 Model" (page 19). The `Converter` class implements the conversion computation.

1. Open the ConverterController.h file for editing, if it's not already open.
2. Add the following lines between the brackets in the ConverterController interface file:

```
IBOutlet NSTextField *amountField;
IBOutlet NSTextField *dollarField;
IBOutlet NSTextField *rateField;
```

These lines declare three outlets, one for each of the three text fields in your interface. Notice that the three text field outlets are of type `NSTextField`. Because Objective-C is a dynamically typed language, it's fine to define all the outlets as type `id`. 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.

Declare the Controller Method

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. The `ConverterController` class must implement `convert:` so something will happen when a converter controller object receives the `convert:` message.

1. Add the declaration line for the convert: method to the interface file, as shown:

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

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

2. Save the file.

You might be asking yourself why the calculations aren't done here in the controller. It would obviously be faster without the extra layer of communication. It is entirely possible to design an application this way, but the MVC design pattern was created to enable portability and maintainability. In more complex applications, the underlying functionality may be operating system-dependent. If you make the controller have a method `convert:`, all the view needs to know is to call this method. From there, the various controllers written for the different operating systems can take care of calling the correct model functions. In this way, it's similar to writing an API for this particular application. Similarly, by factoring out the functionality into clearly distinct pieces, it's easy to remember what objects are responsible for what functionality.

Interconnecting the Controller with the View

You're now ready to connect the Currency Converter user interface and the `ConverterController` object to each other.

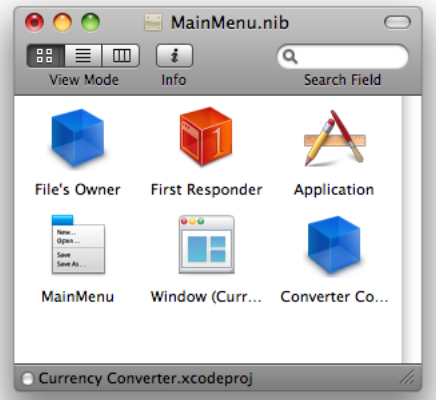
Add the ConverterController Class to Your Nib File

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. Add this class to your nib file.

1. In Interface Builder, choose File > Read Class Files.
2. Select the `ConverterController.h` file and click Open.
3. In the library, drag an object item into the `MainMenu.xib` window.
4. Select the new object and navigate to the Identity tab in the Inspector.

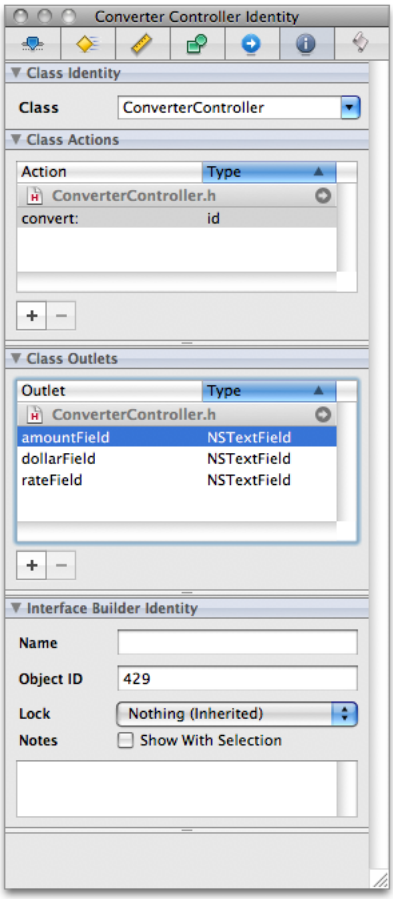
5. In the Class drop-down box, type `ConverterController` and press Return. The class actions and class outlets are filled in to match the outlets and actions you defined in the `ConverterController.h` file.

Figure 5-3 A newly instantiated instance of `ConverterController`



The result of these operations when the class is viewed in the object inspector in Interface Builder is shown in Figure 5-4.

Figure 5-4 Outlets and actions in the Converter Controller Identity inspector



Connect the ConverterController Instance to the Text Fields

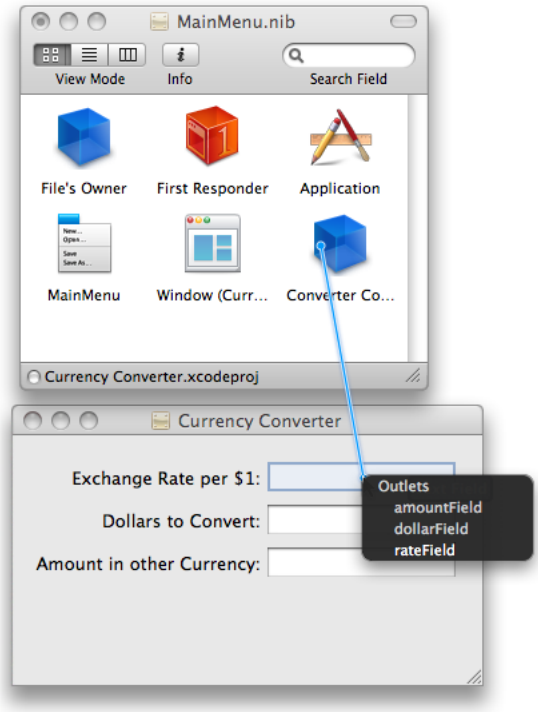
By connecting the `ConverterController` instance to specific objects in the interface, you initialize its outlets. The `ConverterController` object uses these outlets to get and set values in the user interface. To connect the instance to the user interface:

1. Control-drag a connection from the `ConverterController` instance to the Exchange Rate text field.

   Interface Builder displays the possible connections in a black box.

2. Select the outlet that corresponds to the first field, `rateField` as shown in Figure 5-5.

Figure 5-5 Connecting `ConverterController` to the `rateField` outlet



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

Connect the Convert Button to the Appropriate Methods

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. A black box will pop up similar to the one we used to connect outlets. Choose `convert:`.
3. Save the nib file.

Check Your Work

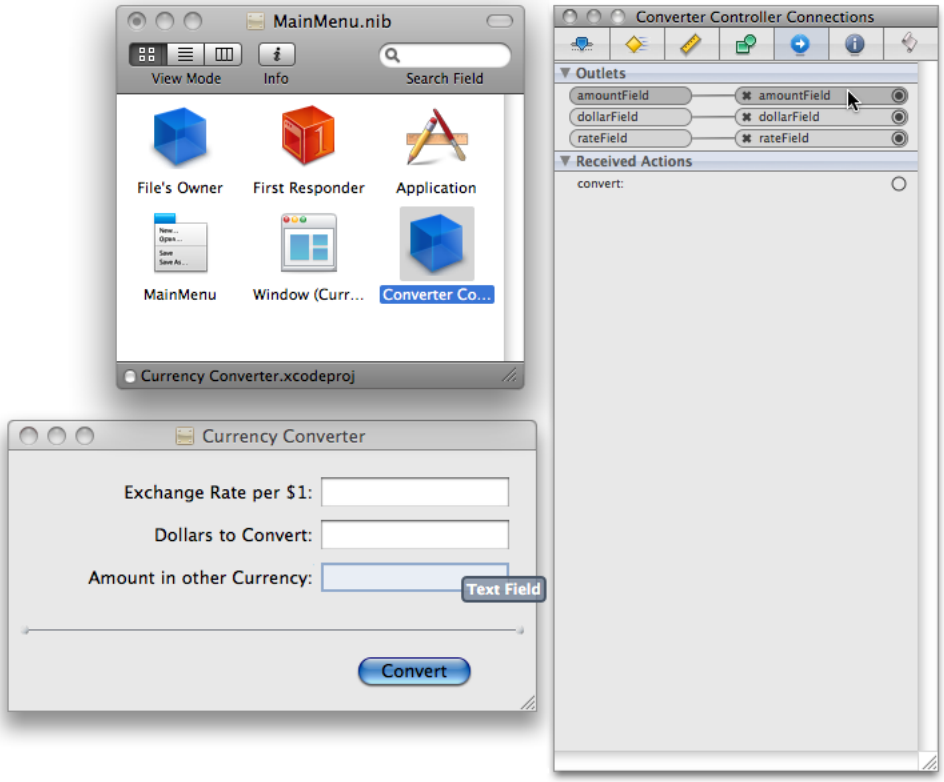
To make sure everything is done correctly:

1. Select the Converter Controller instance in the MainMenu.nib window.

2. Navigate to the Connections tab in the Inspector.

3. Make sure each outlet is connected to the correct text field by hovering the mouse over the connections as shown in Figure 5-6.

Figure 5-6 Checking the outlet connections



Connecting the Controller to the Model

Create an instance of the converter class inside the `ConverterController` class in Xcode.

1. In the Classes folder in the Groups and Files sidebar, double-click `ConverterController.h` to open it in an editor window.

2. Declare a pointer to a converter object by adding the following line to your code right after the outlets are declared, before the ending bracket:

```
Converter *converter;
```

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

3. Import `Converter.h` so `ConverterController` can instantiate a `Converter` object. Add the following line under the first import statement in `ConverterController.h`.

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

4. In the Classes group, double-click `ConverterController.m` to open this file in an editor window.

5. Add the definition of the `convert:` method, as shown in Listing 5-1, to `ConverterController.m`.

Listing 5-1 Definition of the `convert:` method in `ConverterController.m`

```
#import "ConverterController.h"
@implementation ConverterController
- (IBAction)convert:(id)sender {
    float amount;
    converter = [[Converter alloc]init];                                    // 1
    [converter setSourceCurrencyAmount:[dollarField floatValue]];           // 2
    [converter setRate:[rateField floatValue]];                             // 2
    amount =   [converter convertCurrency];                                 // 3

    [amountField setFloatValue:amount];                                     // 4
    [rateField selectText:self];                                            // 5
}
@end
```

The `convert:` method does the following:

1. Initializes a Converter object.

2. Sets the member variables of the `Converter` class to the values in the `rateField` and `dollarField` text fields.

3. Sends the `convertCurrency` message to the object pointed to by the `converter` pointer and gets the returned value.

4. Uses `setFloatValue:` to write the returned value to the Amount in Other Currency text field (`amountField`).

5. 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 variable `amount` holds a returned value.

Garbage Collection

You may be feeling a little uneasy about the following line being called every time the `convert:` method is called:

```
converter = [[Converter alloc]init];
```

This line allocates space in memory for a `Converter` instance and should be deallocated after you use it. You may notice that you didn't deallocate this instance.

The reason you can do this is because Objective-C 2.0 utilizes **garbage collection**. To enable garbage collection:

1. Choose Project > Edit Project Settings
2. Navigate to the Build tab
3. Set the value for Objective-C Garbage Collection to `Supported` under GCC 4.0 - Code Generation.

By supporting garbage collection, you don't have to worry about deallocating objects you instantiate. You can leave your code just the way it is and not have to worry about memory leaks.

More information about garbage collection can be found in *GNU C/C++/Objective-C 4.0.1 Compiler User Guide*.

What's Next?

You've now completed the implementation of Currency Converter. Notice how little code you had to write, given that your application now has a fully functional currency-converting system and a beautiful user interface. In the next chapter, you will learn how to build and run the application.

Building and Running Your Application

Now that you have written all the code necessary for your application to run and have built the user interface and connected the classes together, it's time to see whether your application compiles. In this chapter, you will learn how to run the process that builds your project into an application. You will also learn how to review errors that occurred while compiling.

Build the Application

To build the Currency Converter application:

1. In Xcode, choose File > Save All 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 52) for details.

Extra: Check Out the 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 the `ConverterController.m` file 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 Quick Help window appears with a summary of the reference documentation. Click the book icon to get to a complete description of the selected method. Read more in "Expanding on the Basics" (page 67).
3. Close the Quick Help 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 the Application

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.

1. Choose Build > Build and Go.
2. After the Currency Converter application launches, enter a rate and a dollar amount.
3. Click Convert.
4. Select the text in a text field and choose the Currency Converter > Services submenu.

   The Services menu lists other applications that can operate on the selected text.
5. Choose Currency Converter > Quit Currency Converter from the application menu to quit Currency Converter.

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.

1. To get an idea of the error-checking features of Xcode, introduce a mistake into the code and click Build again.

   You can now see that the left column of your code contains one or more error indicators.

   While the error indicator helps you understand the location of the error, you may want to examine the nature of the problem.
2. In the project window, click the Build Results tab.
3. Double click the error to open it in an editor window and display the error.
4. Fix the error in the code and build the application again. The errors go away, and the status bar indicates that the build was 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. You learned:

- To compose a graphical user interface (GUI) in Interface Builder

- To test a user interface in Interface Builder
- To specify a class' outlets and actions in Interface Builder
- To connect controller-instances to the user interface by using outlets and actions in Interface Builder
- To implement a model, a view, and a controller in Xcode
- To build applications and correct build errors in Xcode

What's Next?

In this chapter, you learned how to build and run your application. You also learned how to identify and resolve build errors. In the next chapter, you learn how to set properties such as your application's name and copyright information. You also learn how to give your application a custom icon.

Configuring Your Application

Now that you have your very own functional application, it's time to give it individual properties instead of the default properties. In this chapter, you learn about the `Info.plist` file, which is where you can change properties that govern your application's attributes. This chapter also describes the essential identification properties required of Mac OS X applications. It walks you through the process of configuring these properties in Currency Converter. You learn what changing these various properties does when viewing information about the application. Finally, you learn how to give an application its own icon to set it apart from other applications.

The Info.plist File

Mac OS X applications contain information to help distinguish them from one another. This information includes the application's primary and secondary version numbers, and the icon that 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 with the executable files and resources that make up an application, known as an 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*.

Basic Project Attributes

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.

- Without application identifiers, administrators would have to navigate to the location of each managed application, a relatively tedious task. 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.

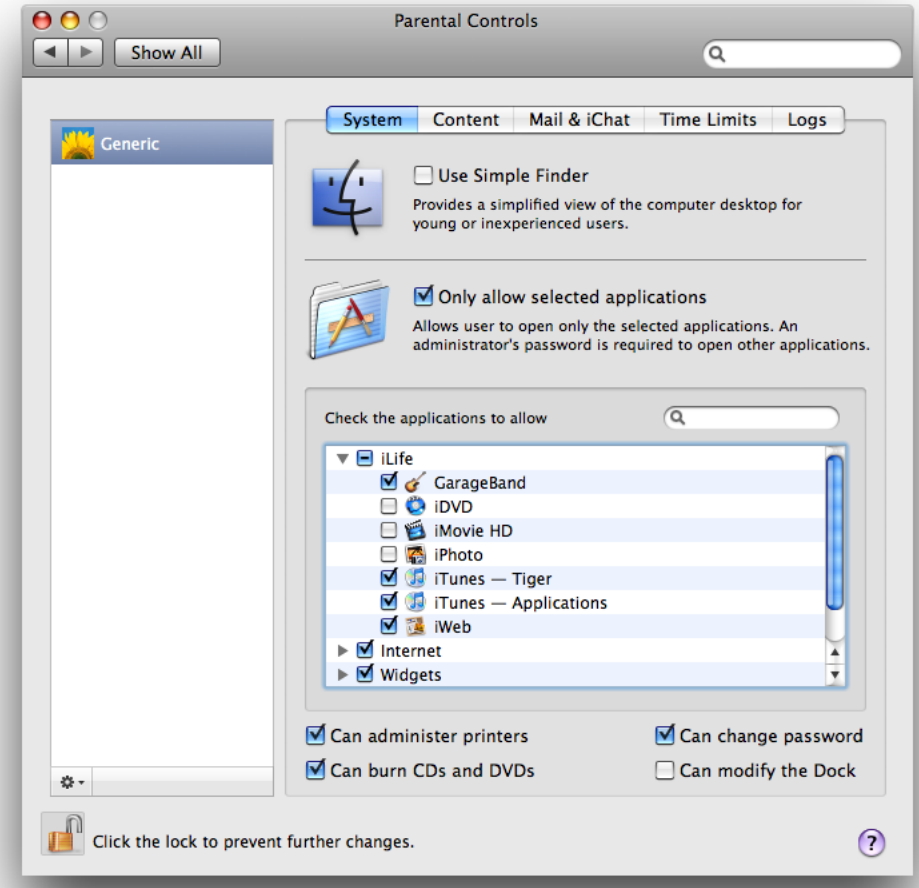
  The application-identifier property is specified with the `CFBundleIdentifier` key in the `Info.plist` file.

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 7-1.

Figure 7-1 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.

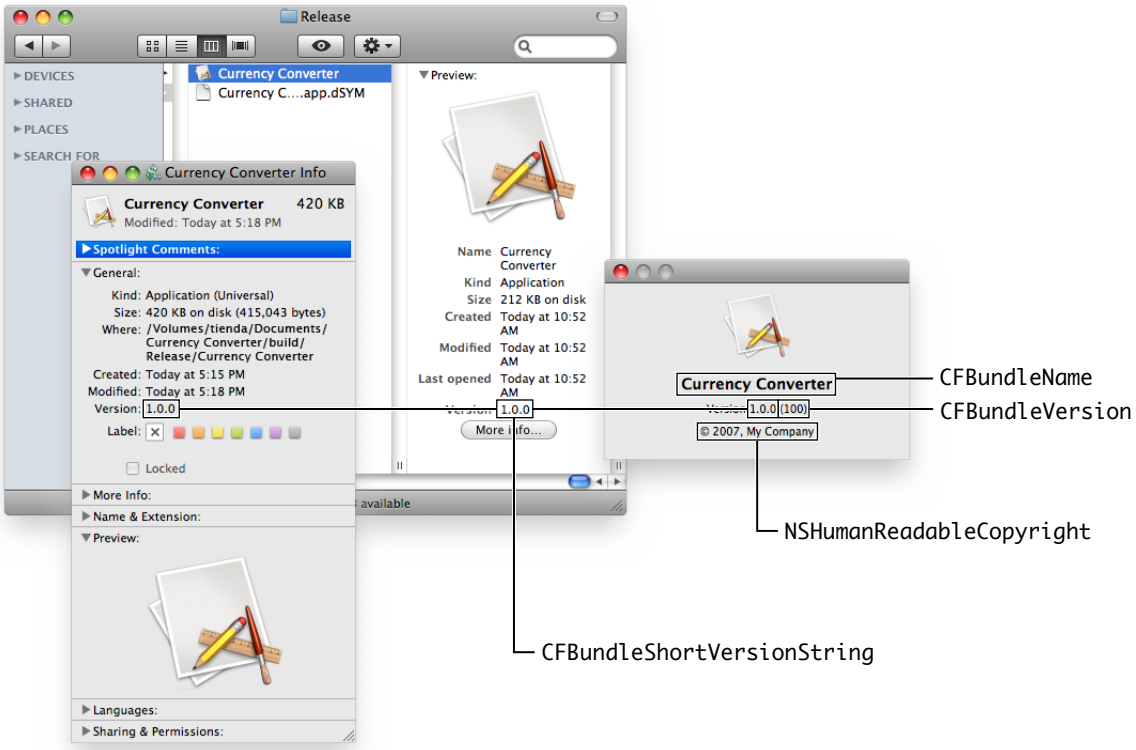
  The build version number is made up of a string of period-separated integers. Each integer must be equal to or greater than zero. For example, `55`, `1.2`, and `1.2.0.55`, are all valid build version numbers.

  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 7-2.

Figure 7-2 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, `© 2007 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: Xcode sets the application name to the project name when you create a new application project.

- 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*.

Specify the 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`.

Currency Converter's name property is set to the project name you entered in "Creating a Project in Xcode" (page 15), `Currency Converter`. Therefore, you don't need to change the value of this property.

First, remove the InfoPlist.strings file from the project. This file is used for internationalization, a subject outside the scope of this tutorial.

1. In the Currency Converter project window, select the Resources group in the Groups & Files list.
2. In the detail view, select the `InfoPlist.strings` file.
3. Choose Edit > Delete.
4. In the dialog that appears, click Also Move to Trash.

To set the application-identifier and build-version-number, follow these steps:

1. Open the Targets group in the project window and select the Currency Converter target.
2. Choose File > Get Info to view the Target Info.
3. Click the Properties tab to display the target's properties.
4. Enter `com.mycompany.CurrencyConverter` in the Identifier field to set the application identifier.
5. Enter `100` in the Build field to set the build-version-number to 100.

The release-version-number and copyright-text properties aren't exposed in the Properties tab. You can edit them, and other attributes stored in the Info.plist file, by opening the Info.plist file and editing it directly.

1. Click the Open Info.plist as File button at the bottom of the Properties tab.

   The Info.plist file is displayed in an editor, showing a list of keys on the left side and their associated values on the right side.

2. Set the release version number to `1.0.0`

   Find the "Bundle version string, short" entry in the editor and change its value to "1.0.0".

3. Set the copyright text to `© 2009, My Company`:

   a. Select a line of the Info.plist file and click the plus button at the right end of the line.

   b. Choose "Copyright (human-readable)" from the list of keys.

   c. Double-click the value cell on the right side of the table and enter the text "© 2009, My Company".

4. Save the `Info.plist` file.

To see how this affects your application:

1. Clean the project, and build and run the application.
2. Choose Currency Converter > About Currency Converter.

   Notice how the name, version number, and copyright information you entered in the `info.plist` file are displayed correctly here.

Note: If the information, such as the copyright information, is not the same, your `NSHumanReadableCopyright` value may be different from the one you entered in the `Info.plist` file. This will occur if there is an extra file called `InfoPlist.strings (English)`. This file is used for localization, that is, having different properties for an application based on the location it's designed for. In this case, any key values set in the English localization file will override the values set in the `Info.plist` file when compiling the English version of the project. To fix this issue, you can either delete the `InfoPlist.strings (English)` file, or open the file and change the value just as in the `Info.plist` file.

3. Quit Currency Converter.

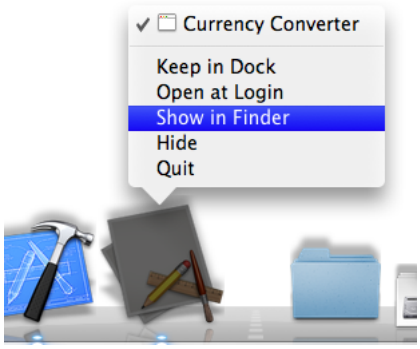
4. To see how application identifiers can be beneficial, execute the following command in a Terminal window:

```
> 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).

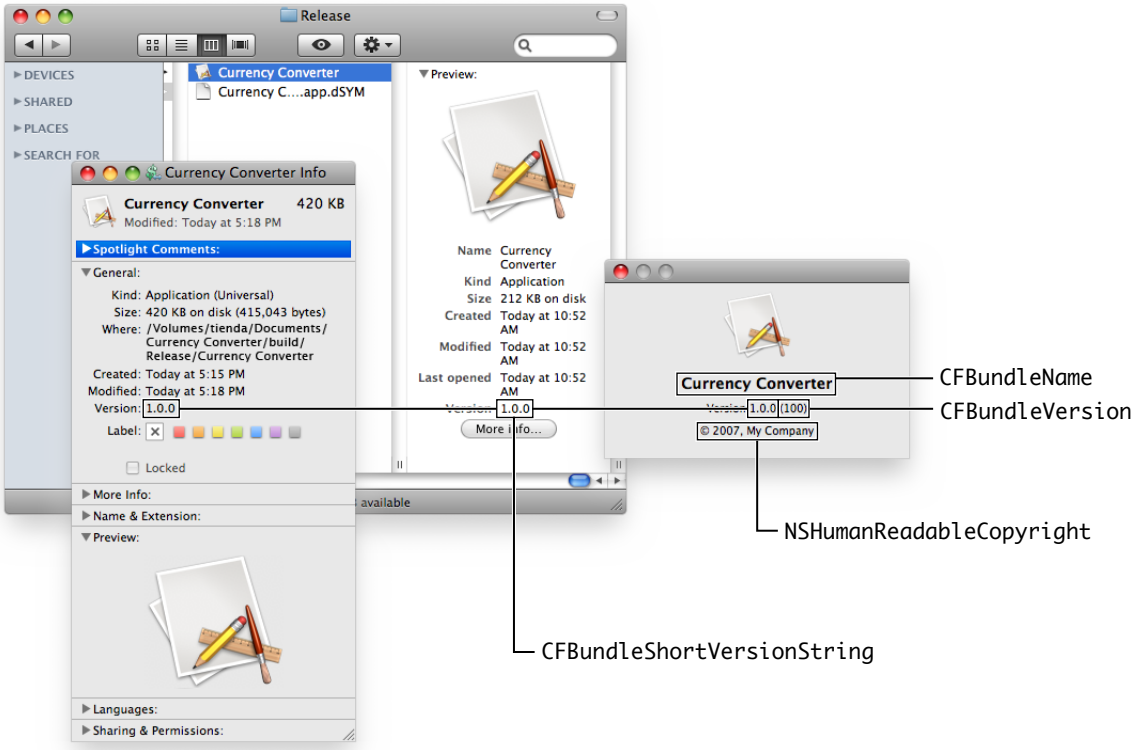
5. In the Dock, Control-click or click-and-hold the Currency Converter icon and choose Show In Finder from the shortcut menu, as shown in Figure 7-3.

Figure 7-3 Locating the application bundle from the Dock



The Finder opens a window shown in Figure 7-4, 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 parentheses, release version number, and copyright text (`NSHumanReadableCopyright`).

Figure 7-4 Application properties as seen by the user



6. Quit Currency Converter.

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

Create an Icon File

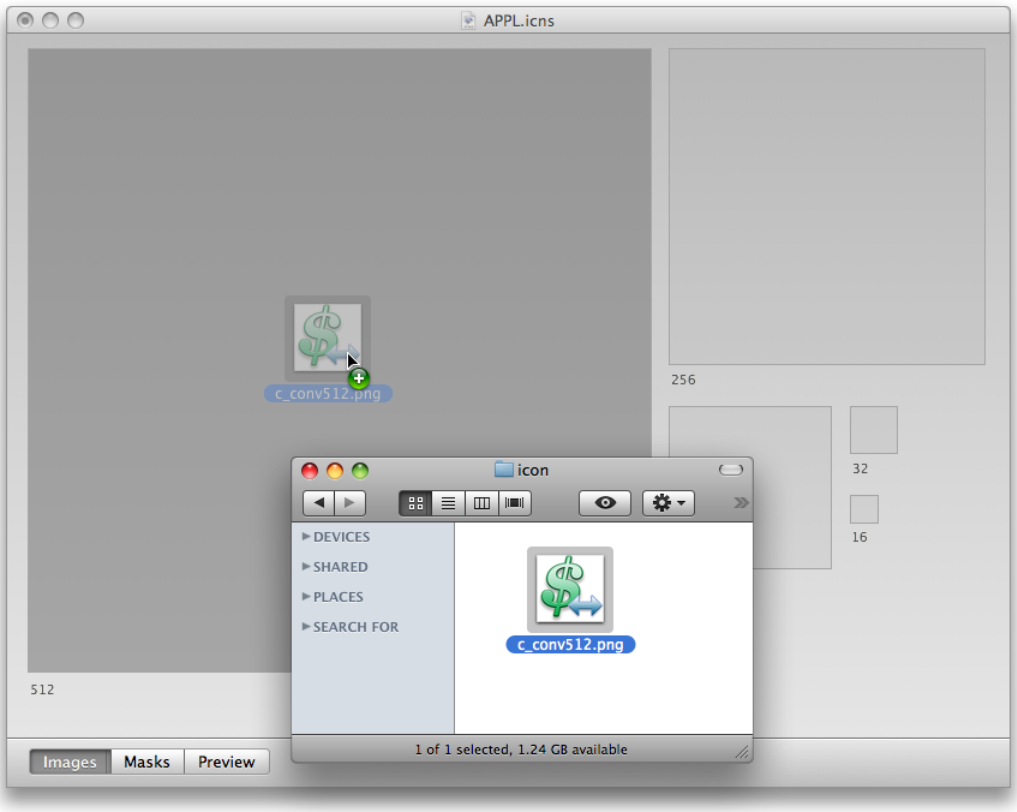
When you create a Cocoa application without specifying an icon for it, the Finder and the Dock assign it the generic application icon, as shown in Figure 7-4 (page 61). To make your applications more appealing to their users and to differentiate them from other applications, you should give your applications 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. To do this, you must have downloaded the companion file provided with this document.

Note: Before doing this, make sure you have downloaded the companion archive (`ObjCTutorial_companion.zip`). The archive contains the icon image you will be using for this section.

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 the image file that depicts the Currency Converter application icon.
3. Add the image file to the icon file.
    a. Drag `c_conv512.png` from the Finder window to the Large Image image well in the icon file editor, as shown in Figure 7-5.

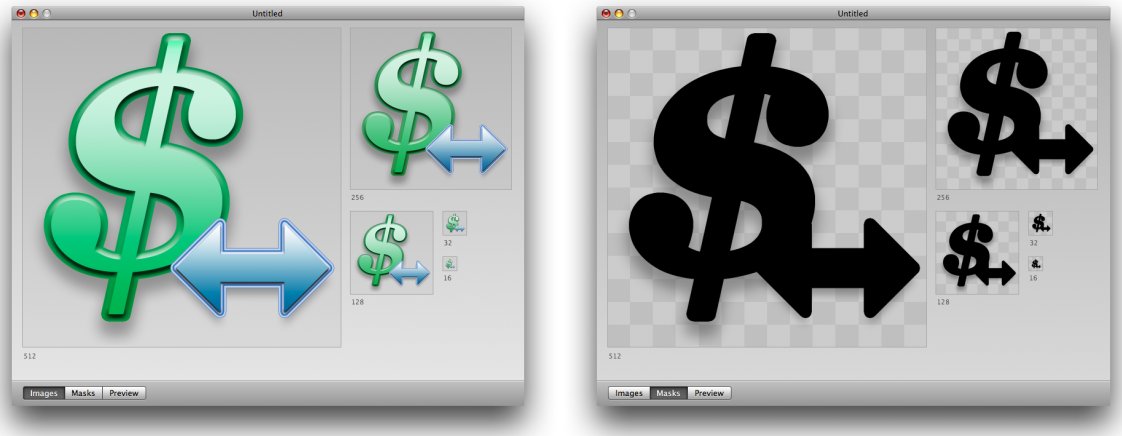
Figure 7-5 Dragging `c_conv512.png` to the icon file editor



    b. A dialog asks if you would like to copy the image to other sizes. Choose "Copy to all smaller sizes" and press Import. This automatically scales the 512 x 512 image to the smaller sizes.
    c. The hit masks for all sizes are automatically extracted. Hit masks are a bitmapping of the locations in which the image will respond when it is clicked.

d. The icon file editor should look like Figure 7-6.

Figure 7-6 Icon file editor with icon images and icon masks 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.

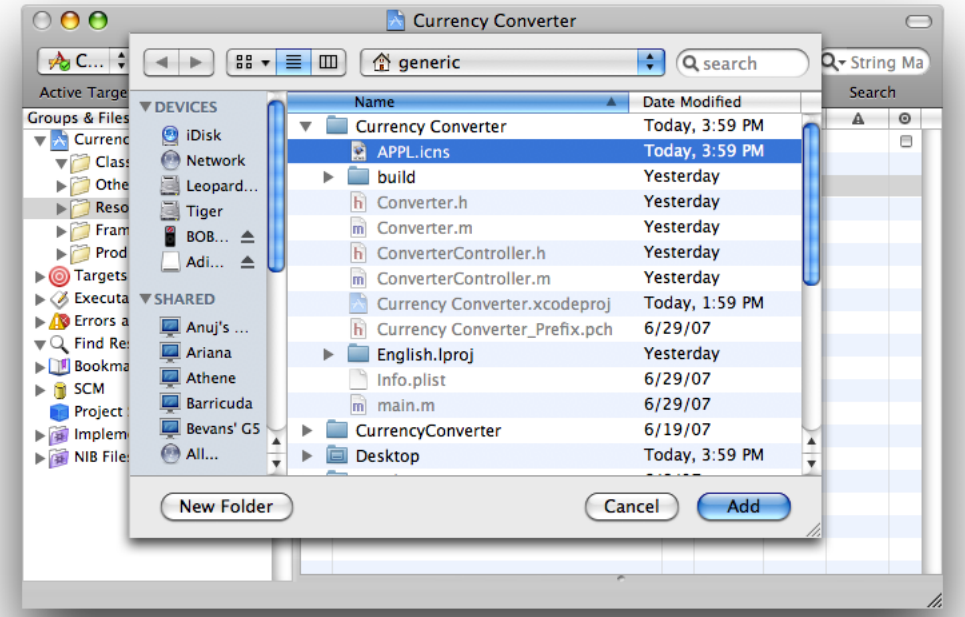
To add the icon to the project:

1. In the Currency Converter project window, select the Resources group in the Groups & Files list.

2. Choose Project > Add to Project.

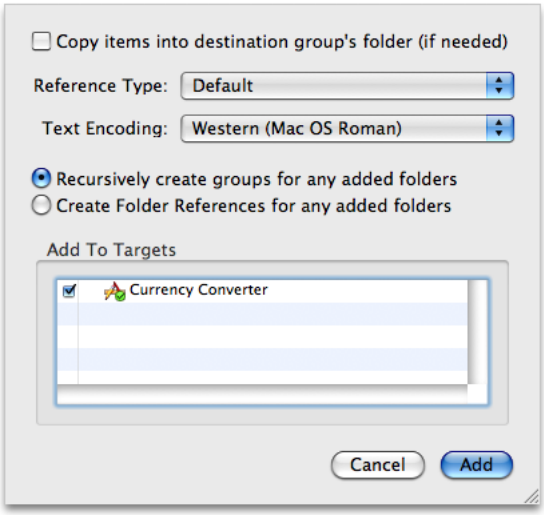
3. In the dialog that appears, select the `APPL.icns` file in the Currency Converter project directory, and click Add, as shown in Figure 7-7.

Figure 7-7 Selecting the icon file to add to the Currency Converter project



4. In the dialog that appears next, shown in Figure 7-8, click Add.

Figure 7-8 Specifying project file-addition options



5. Finally, specify the name of the icon file in the Icon File field of the target properties.

To see the icon in your project:

1. Quit Currency Converter if you have not done so already.
2. Choose Build > Clean.
3. Click "Build and Go" to build and run the application.
4. Currency Converter now has a distinguishing icon, shown in Figure 7-9.

Figure 7-9 Currency Converter sporting an elegant icon

Configuring your applications appropriately is essential for providing a good experience to your customers.

What's Next?

Now that your project is complete, you may want to learn about what separates Cocoa from other frameworks. You probably noticed how much you can do with the application you built without programming any of those features. In the next chapter, you learn about many of the features Cocoa supplies and how to take advantage of them.

Expanding on the Basics

Cocoa was designed to make application development simple and to take a lot of the uninteresting, repetitive work out of application development. Many common features of applications that users may take for granted are quite difficult to program. Cocoa implements a large selection of these features so you can spend your time on what makes your application special, instead of implementing baseline features. This chapter describes those integrated components of Cocoa. You may be surprised how many classes and features come packaged with Cocoa to maximize your productivity.

For Free with Cocoa

The simplest Cocoa application, 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 the Interface Builder 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 you hadn't configured the Currency Converter window in Interface Builder to remove the resize box, you could resize it now. You 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 the Currency Converter 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 Overview* 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 application 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. A few of these tools are described next.

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 the Xcode 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 Control or Meta keys. For information on key bindings, see About Key Bindings in *Text Input Management*.

Objective-C Quick Reference Guide

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 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; `nil` 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's 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` compiler directive before the declaration.

Document Revision History

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

Date	Notes
2009-08-03	Updated for Xcode 3.2 and fixed several minor bugs.
2007-10-31	Updated for Mac OSX v10.5. Added Objective-C 2.0 content such as garbage collection and declared properties.
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” 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”.
	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” to instruct that the `#import "Converter.h"` code line has to be added to `ConverterController.h`.
	Updated the introduction and “Creating the Currency Converter User Interface” to specify the development environment required to successfully complete the tasks described in this document.
	Updated “Paths for Object Communication: Outlets, Targets, and Actions” 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” to explain why the `converter` outlet cannot be typed.
	Corrected “Declaration of the convertCurrency:atRate: method in Converter.h” by moving the method declaration after the right curly brace.
	Corrected “Definition of the convertCurrency:atRate: method in Converter.m” by including a tag number in the code line with the right brace.

Date	Notes
	Corrected “Definition of the convert: method in ConverterController.m” 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.