Cocoa Tutorial for Java Programmers

(Legacy)

Cocoa > Java

2006-10-03

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

Simultaneously published in the United States and Canada.

Contents

Figures and Listings

Introduction to Cocoa Tutorial for Java Programmers

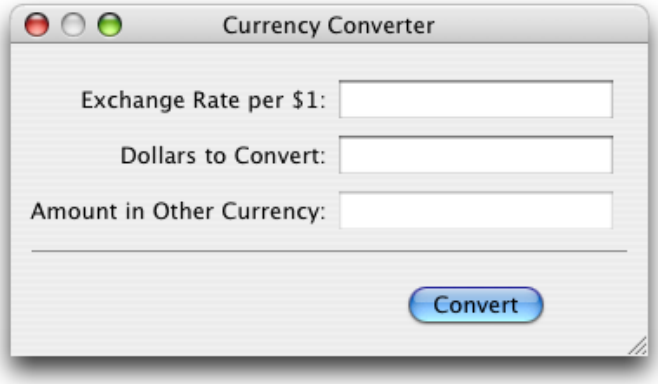
Important: The Java API for Cocoa is deprecated in Mac OS X version 10.4 and later. You should use the Objective-C API instead. For a tutorial on using Cocoa with Objective-C, see *Cocoa Application Tutorial*.

This document introduces the Cocoa application environment using the Java language and teaches you how to leverage Apple's development tools 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 Java programming but does not assume you have previous experience with Cocoa or Xcode Tools.

This document is intended for Java programmers interested in developing Cocoa applications. Keep in mind, however, that Java is not Cocoa's native language. To develop Cocoa applications that you intend to release to end users, you must use Objective-C. No Java interfaces for new Cocoa features will be added to Mac OS X versions after 10.4. Therefore, features added to Cocoa in subsequent versions Mac OS X will not be available to Cocoa applications developed using Java.

This document shows how to build Currency Converter, an application that converts a dollar amount to an amount in another currency, given the rate of that currency relative to the dollar. The main window of the finished application is shown in Figure I-1.

Figure I-1 The Currency Converter main window



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

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

- Create an Xcode project
- Create a graphical user interface using Interface Builder

- Create a custom subclass of a Cocoa framework class
- Connect an instance of your custom subclass to the user interface
- Build an application and correct problems

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

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

Organization of This Document

This document consists of the following chapters:

- Creating the Currency Converter Project and User Interface (page 11) guides you through the development of the Currency Converter user interface.
- Implementing Currency Converter (page 39) shows how to define the custom behavior of the application.
- Building Currency Converter (page 43) explains how to build the application.
- Expanding on the Basics (page 47) explains some of the behavior Cocoa applications get by default.
- Adopting Objective-C (page 51) lists the basic steps Java developers need to perform to adopt Objective-C.

This document also contains a revision history.

See Also

These documents provide detailed information on Cocoa development:

- *Cocoa Fundamentals Guide* describes the Cocoa application environment.
- *Getting Started with Cocoa* provides a road map for learning Cocoa.
- *The Objective-C 2.0 Programming Language* introduces Objective-C and describes the Objective-C runtime system, which leads to much of Cocoa's dynamic behavior and extensibility.

- *Apple Human Interface Guidelines* explain how to lay out user interface elements to provide a pleasant user experience.

Creating the Currency Converter Project and User Interface

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

Creating the Currency Converter Project

Every Cocoa application starts life as a **project**. A project is a repository for all the elements that go into the application, such as source code files, frameworks, libraries, the application's user interface, sounds, and images. You use the Xcode application to create and manage your project.

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

Open Xcode

To open Xcode:

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

Figure 1-1 The Xcode application icon

The first time you start Xcode, it asks you a few setup questions. The default values should work for the majority of users.

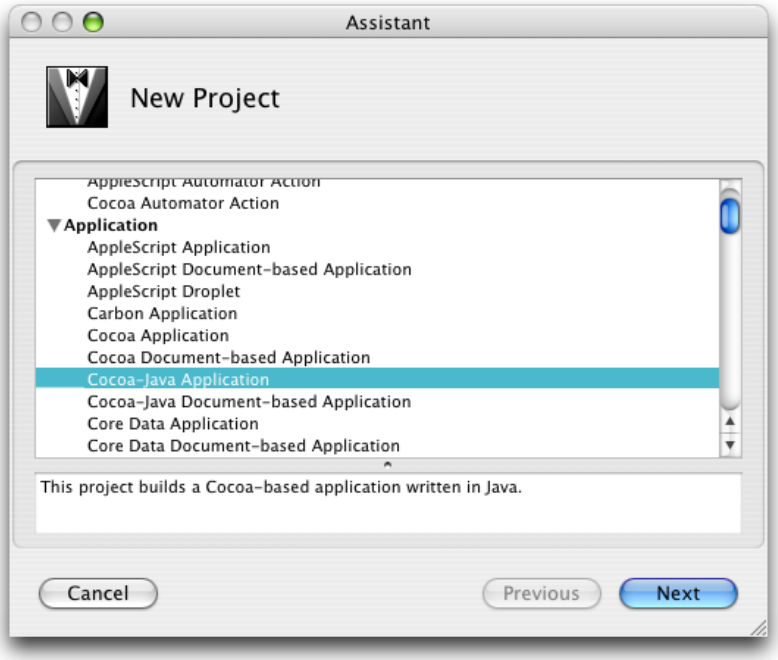
Choose the New Project Command

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

Choose a Project Type

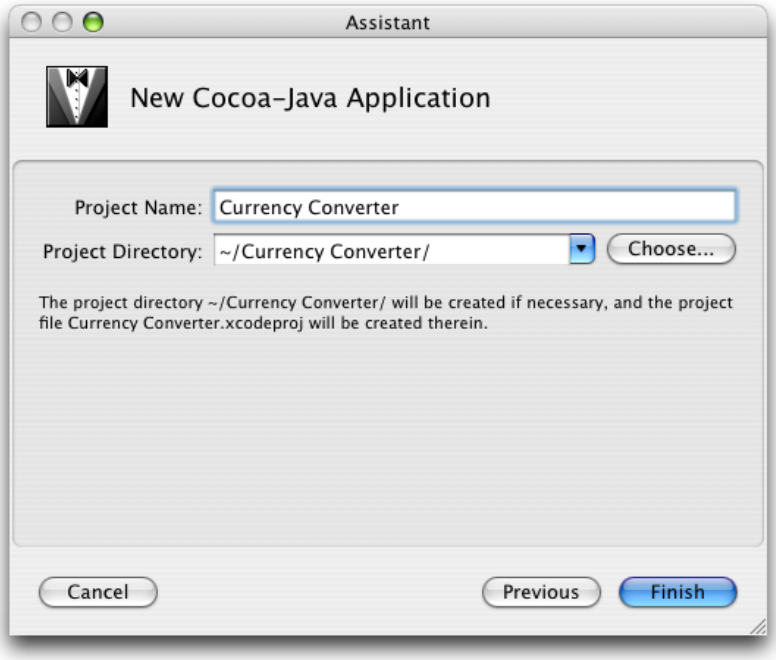
Xcode can build several types of applications, including everything from Carbon and Cocoa applications to Mac OS X kernel extensions and Mac OS X frameworks. For this tutorial, select Cocoa-Java Application and click Next, as shown in Figure 1-2.

Figure 1-2 Xcode New Project Assistant



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

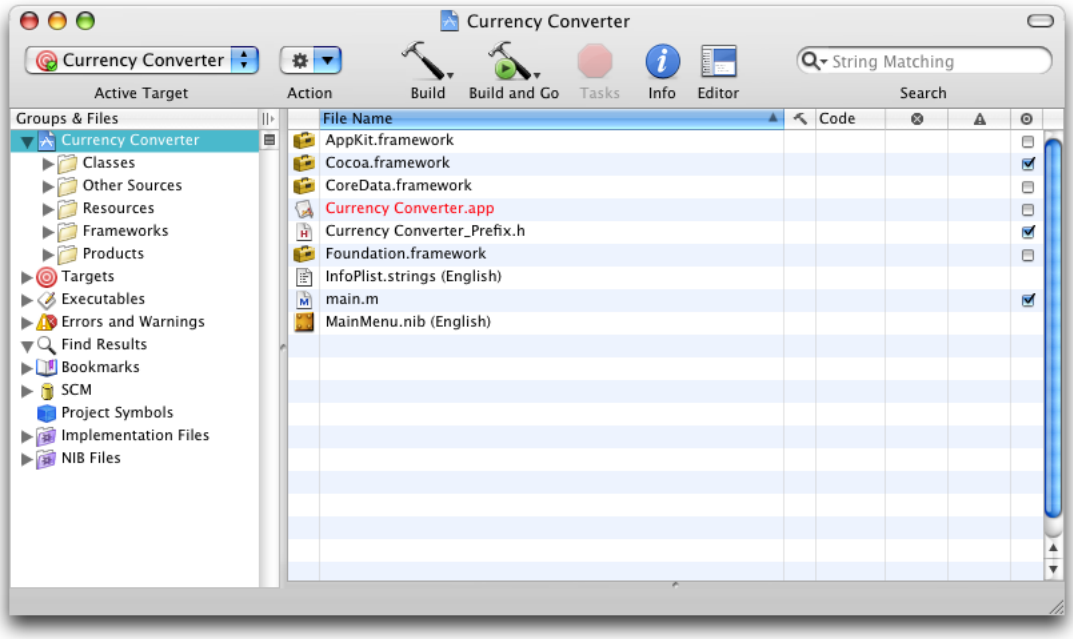
Figure 1-3 Entering a project's name and choosing its location in Xcode New Project Assistant



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

When you click Finish, Xcode creates the project's files and displays the project window, shown in Figure 1-4.

Figure 1-4 The new Currency Converter project in Xcode



The Groups & Files list is comprised of all the source files, images, and other resources that make up a project. These files are grouped in the project group, the first item in the Groups & Files list; this group is named after the project. The project's files are grouped into subgroups, such as Classes, Other Sources, Resources, and so on, as shown in Figure 1-4. These groups are very flexible in that they do not necessarily reflect either the on-disk layout of the project or the way the build system handles it. They are purely for organizing your project. The groups created by Xcode should be suitable for most developers, but you can rearrange them however you like.

These are the groups Xcode sets up for Cocoa-Java applications:

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

Other Sources. This group contains the `main.m` file, which implements the `main` routine. This routine starts the application. (You shouldn't have to modify this file.) This group also contains `Currency Converter_Prefix.h`. This "prefix header" helps Xcode reduce the compilation time of C-based source files. This is not important for this tutorial.

Resources. This group contains the nib files and other resources that specify the application's user interface. What Is a Nib File? (page 15) describes nib files.

Frameworks. This group contains the frameworks (which are similar to packages) 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 column—allow you to sort the project's files using custom rules in a way similar to using smart playlists in iTunes.

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

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

Executables. Contains all the the executable products your project creates.

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

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

`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 directories containing localized resources, such as `Dutch.lproj`.

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

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

Creating the Currency Converter User Interface

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

What Is a Nib File?

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

- **Archived objects.** Also known in object-oriented terminology as "flattened" or "serialized" objects—meaning that the object has been encoded in such a way that it can be saved to disk (or transmitted over a network connection to another computer) and later restored to memory. Archived objects contain information such as their size, location, and position in the object hierarchy. At the top of the hierarchy of archived objects is the File's Owner object, a proxy object that points to the actual object that owns the nib file (typically, the one that loaded the nib file from disk).
- **Images.** Image files that you drag and drop on the nib file window or on an object that can accept them (such as a button or image view).
- **Class references.** Interface Builder can store the details of Cocoa objects and objects that you place into static palettes, but it does not know how to archive instances of your custom classes since it doesn't have access to the code. For these classes, Interface Builder stores a proxy object to which it attaches your custom class information.
- **Connection information.** Information about how objects within the class hierarchies are interconnected. Connector objects special to Interface Builder store this information. When you save a document, its connector objects are archived in the nib file along with the objects they connect.

Open the Main Nib File

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

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

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

Windows in Cocoa

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

NSWindow and the Window Server

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

Two interacting systems create and manage Cocoa windows. A window is created by the Window Server. The Window Server is a process that uses the internal window management portion of Quartz (the low-level drawing system) to draw, resize, hide, and move windows using Quartz graphics routines. The Window Server also detects user events (such as mouse clicks) and forwards them to applications.

The window that the Window Server creates is paired with an object supplied by the Application Kit framework (AppKit). The object supplied is an instance of the NSWindow class. Each physical window in a Cocoa program is managed by an instance of NSWindow or a subclass of it. For information on AppKit, see The Cocoa Frameworks 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 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) that come below it in the hierarchy. The NSWindow distributes events to views in the hierarchy and regulates coordinate transformations among them.

Another rectangle, the frame rectangle, defines the outer boundary of the window and includes the title bar and the window's controls. Cocoa uses the bottom-left corner of the frame rectangle as the origin for the base coordinate system, unlike Carbon and Classic applications, which use the top-left corner. Views draw themselves in coordinate systems transformed from (and relative to) this base coordinate system.

Key and Main Windows

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

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

An application has one main window, which can often have key status as well. The main window is the principal focus of user actions for an application. Often user actions in a modal key window (typically a panel such as the Font window or an Info window) have a direct effect on the main window.

Resize the Window

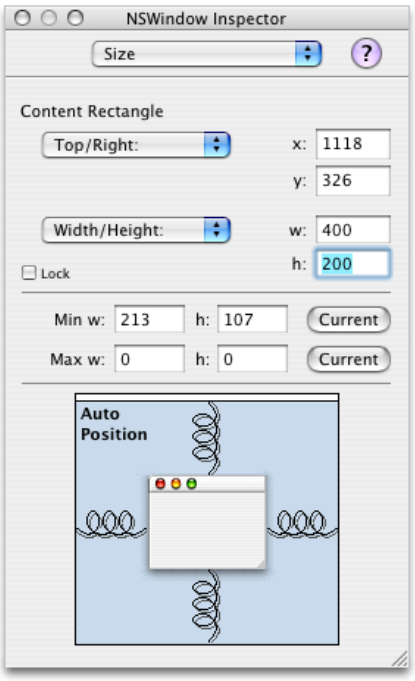
Make the window smaller by dragging the bottom-right corner of the window inward, as shown in Figure 1-5.

Figure 1-5 Resizing a window manually

You can resize the window more precisely in the Size pane in the NSWindow inspector.

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

Figure 1-6 Resizing a window with the NSWindow inspector



Set the Window's Title and Other Attributes

Set other attributes for the window in the NSWindow inspector:

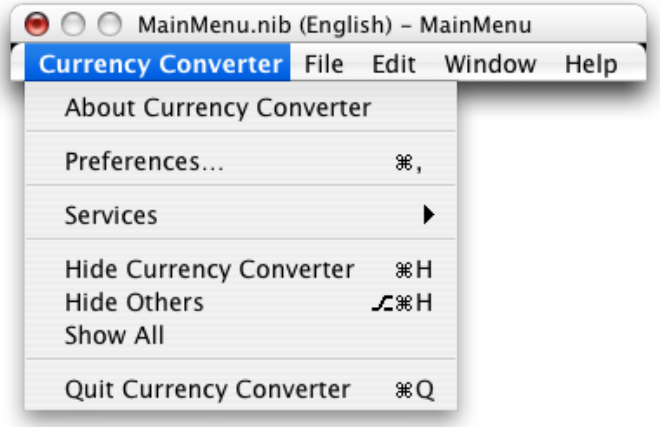
1. Choose Attributes from the inspector pop-up menu and change the window's title to "Currency Converter". Press Return to lock in the change.
2. Verify that the "Visible at launch time" option is selected.
3. Deselect the Zoom option in the "Title bar controls" group.

Set the Application Name in the Menu

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

1. In the MainMenu window, double-click NewApplication, and press the Space bar.
2. In the NSMenuItem inspector, enter `Currency Converter` in the Title text field and press Return.

3. In the MainMenu window, click Currency Converter, double-click Quit NewApplication, and type `Quit Currency Converter`.

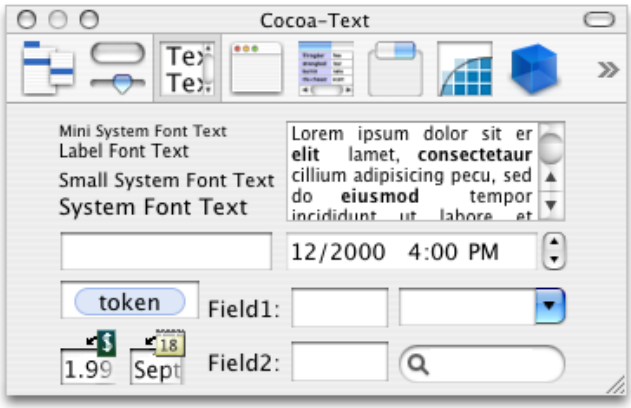


4. Click Help and replace “New Application Help” with “Currency Converter Help”.

Configure a Text Field

The Interface Builder palette window contains several user-interface elements or controls that you can drag into a window or menu to create an application’s user interface. You open the Interface Builder palette window—shown in Figure 1-7—by choosing Tools > Palettes > Show Palettes.

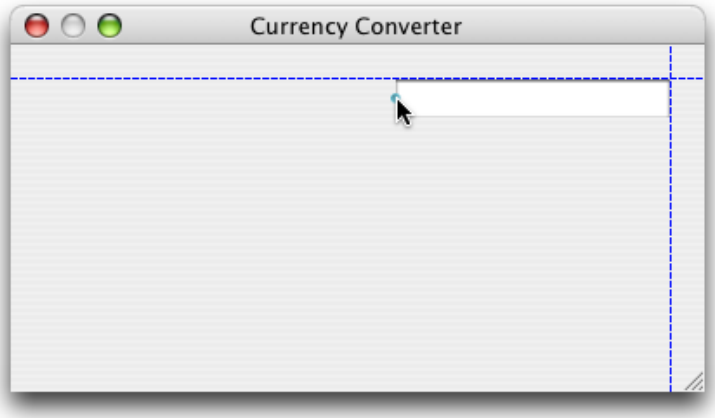
Figure 1-7 Cocoa text controls in the Interface Builder palette window



1. Click the text toolbar item in the palette window (shown as the third toolbar item in Figure 1-7) and drag a text field object to the top-right corner of the Currency Converter window. Notice that Interface Builder helps you place objects according to the Apple human interface guidelines by displaying layout guides when an object is dragged close to the proper distance from neighboring objects or the edge of the window.

2. Increase the text field's size so that it's about 50% 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 1-8.

Figure 1-8 Resizing a text field



Currency Converter needs two more text fields, both the same size as the first. There are two options: You can drag another text field from the palette to the window and make it the same size as the first one; or you can duplicate the text field already in the window.

Duplicate an Object

To duplicate the text field in the Currency Converter window:

1. Select the text field, if it is not already selected.
2. Choose Duplicate (Command-D) from the Edit menu. The new text field appears slightly offset from the original field.
3. Position the new text field under the first text field. Notice that the layout guides appear and Interface Builder snaps the text field into place.
4. To make the third text field, press Command-D. Notice that Interface Builder remembered the offset from the previous Duplicate command and automatically applied it to the newly created text field.

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

Change the Attributes of a Text Field

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

1. Select the third text field.

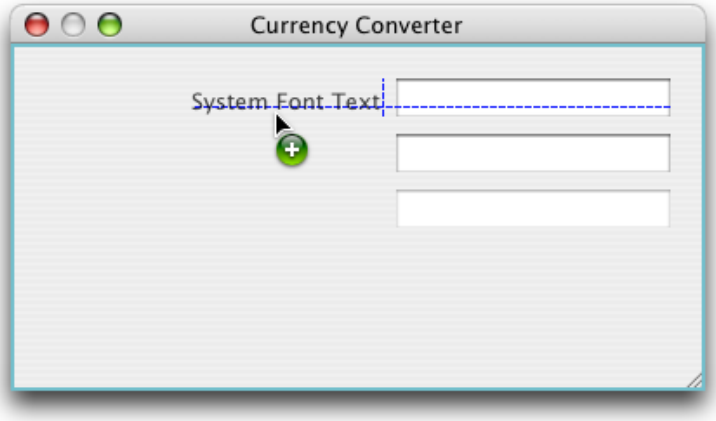
2. In the NSTextField inspector, choose Attributes from the pop-up menu.

3. Deselect the Editable option so that users are not allowed to alter the contents of the text field. Make sure the Selectable option is selected so that users can copy and paste the contents of the text field to other applications.

Assign Labels to the Fields

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

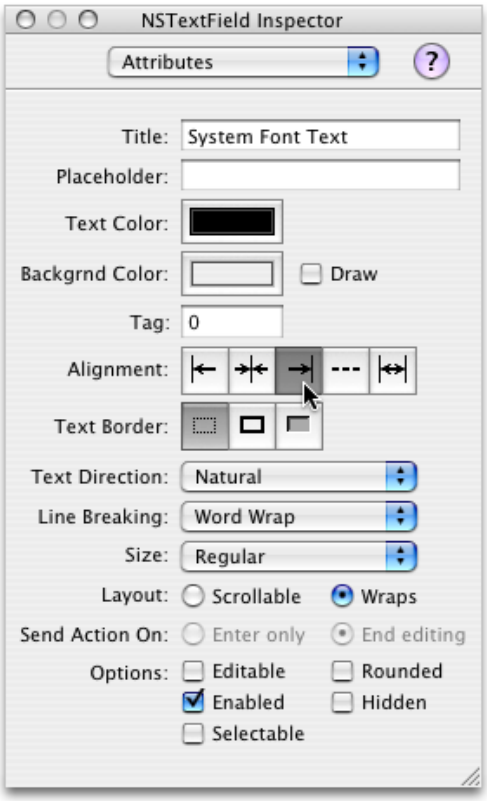
1. Drag a System Font Text element onto the window from the Cocoa Text palette.



2. In the NSTextField inspector, enter `Exchange Rate per $1:` in the Title text field.

3. Make the text label right aligned. With the System Font Text element selected, click the third button from the left in the Alignment area in the inspector, as shown in Figure 1-9.

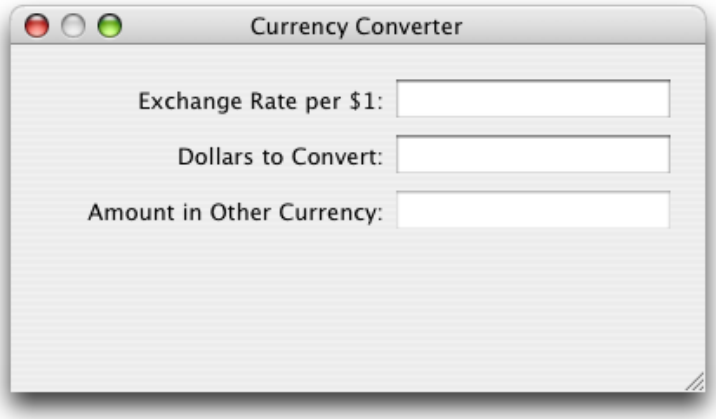
Figure 1-9 Right-aligning a text label in Interface Builder



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

5. Align the new text labels as shown in Figure 1-10. You may need to expand the text labels so that their entire titles are visible.

Figure 1-10 Aligned text fields and labels in Interface Builder



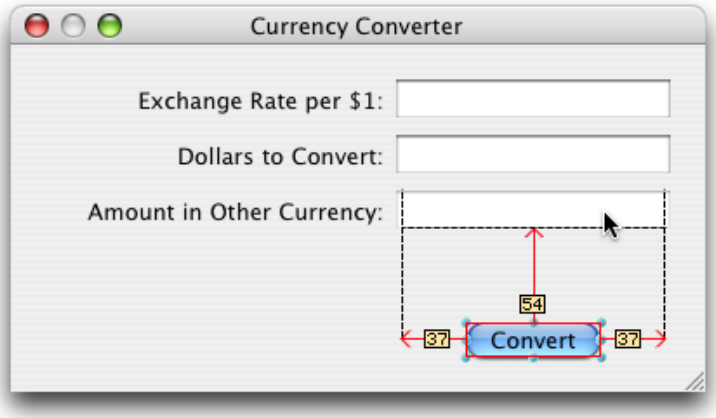
Configure a Button

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

1. Drag the Button element from the Cocoa Controls palette to the bottom-right corner of the window.
2. Double-click the button and change its title to "Convert".
3. Choose Attributes from the NSButton inspector pop-up menu and then choose Return from the Key Equiv pop-up menu. This makes the button respond to the Return key as well as clicks.
4. Align the button under the text fields:
    a. Drag the button downward until the layout guide appears and then release it.
    b. With the button still selected, hold down the Option key. If you move the pointer around, Interface Builder shows you the distance from the button to the object 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 1-11.

Figure 1-11 Measuring distances in Interface Builder



Add a Horizontal Decorative Line

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

1. Drag a horizontal line element from the Cocoa Controls palette to the Currency Converter window.
2. Drag the endpoints of the line until the line extends across the window, as shown in Figure 1-12.

Figure 1-12 Adding a horizontal line to the Currency Converter window

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

Interface Layout and Object Alignment

In order to make an attractive user interface, you must be able to visually align interface objects in rows and columns. “Eyeballing” the alignments can be very difficult; and typing in x/y coordinates by hand is tedious and time consuming. Aligning Aqua 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, the Apple user interface guidelines say that a push button should be 20 pixels tall, but you actually need a frame of 32 pixels for both the button and its shadow. The layout rectangle is what you must align. You can view the layout rectangles of objects in Interface Builder using the Show Layout Rectangles command (Command-L) 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 arrow keys (with the grid off, the selected objects move one pixel)
- Using a reference object to put selected objects in rows and columns
- Using the built-in alignment functions
- Specifying origin points in the Size pane in the inspector

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

Finalize the Window Layout

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

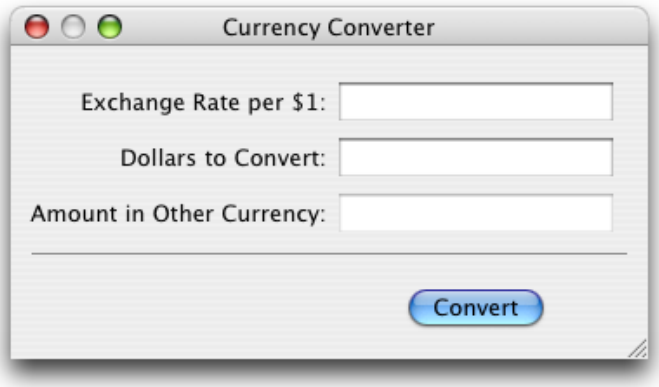
Perform these steps to finalize the Currency Converter window:

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

8. Make the window shorter and narrower until the layout guides appear to the right of the text fields and below the Convert button.

At this point the application's window should look like Figure 1-13.

Figure 1-13 Currency Converter's final user interface in Interface Builder



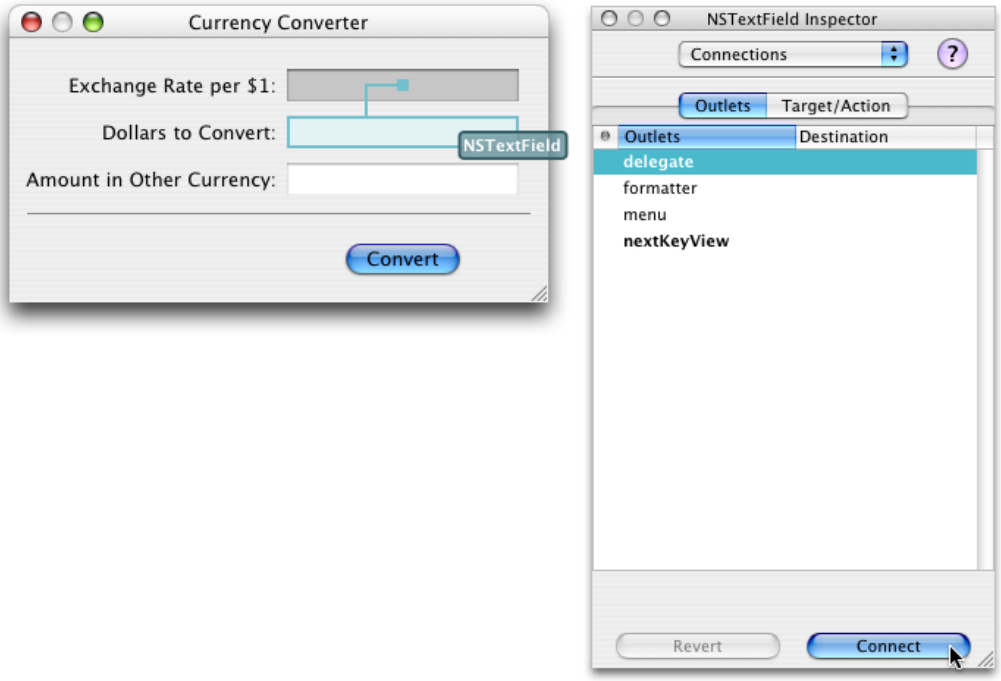
Enable Tabbing Between Text Fields

The final step in composing the Currency Converter user interface has more to do with behavior than with appearance. You want the user to be able to tab from the first editable field to the second, and back to the first. Many objects in Interface Builder's palettes have an outlet named `nextKeyView`. This variable identifies the next object to receive keyboard events when the user presses the Tab key (or the previous object when Shift-Tab is pressed). A Cocoa application by default makes its "best guess" about how to handle text field tabbing, but this guess often produces unexpected results. If you want correct interfield tabbing, you must connect fields through the `nextKeyView` outlet:

1. Select the Exchange Rate text field.

2. Control-drag a connection from the Exchange Rate text field to the Dollars to Convert text field, as shown in Figure 1-14.

Figure 1-14 Connecting `nextKeyView` outlets in Interface Builder



3. In the inspector for the Dollars to Convert text field click Outlets, select `nextKeyView`, and click Connect. The `nextKeyView` outlet identifies the next object to respond to events after the Tab key is pressed.

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

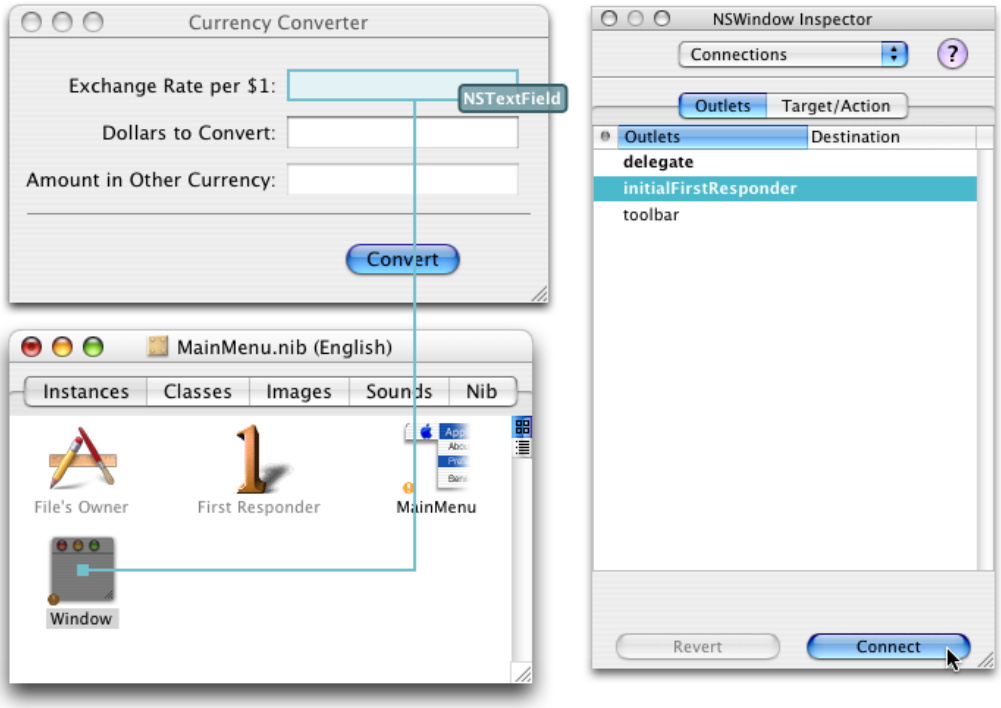
Set the First Responder for the Currency Converter Window

In Enable Tabbing Between Text Fields (page 26), you set up the key view loop using Interface Builder, establishing connections between the `nextKeyView` outlets of the two text fields. Now you must set the window's `initialFirstResponder` outlet to the text field that you want selected when the window is first displayed onscreen. If you do not set this outlet, the window sets a key loop and picks a default initial first responder for you (not necessarily the same as the one you would have specified).

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

1. Control-drag a connection from the Window instance in the `MainMenu.nib` window to the Exchange Rate text field, as shown in Figure 1-15.

Figure 1-15 Setting the `initialFirstResponder` outlet in Interface Builder



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

The Currency Converter user interface is now complete.

Test the Interface

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

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

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

Defining the ConverterController Class

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

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

Classes and Objects

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

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

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

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

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

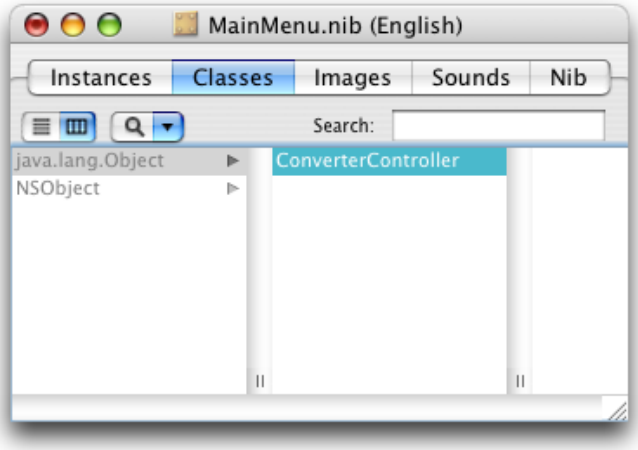
Implicit in the notion of a taxonomy is inheritance, a key property of classes. Classes exist in a hierarchical relationship to one another, with a subclass inheriting behavior and data structures from its superclass, which in turn inherits from its superclass.

Specify the ConverterController Class

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

1. In the `MainMenu.nib` window, click Classes.
2. In the leftmost column of the browser, select `java.lang.Object` and press Return to create a `java.lang.Object` subclass called MyObject.
3. Type `ConverterController` to rename MyObject, and press Return. Figure 1-16 shows the result of this operation.

Figure 1-16 Subclassing `java.lang.Object`



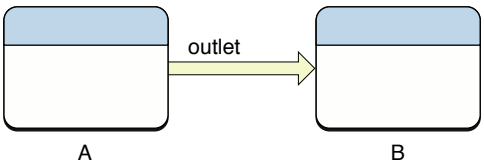
Paths for Object Communication: Outlets, Targets, and Actions

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

Outlets

An **outlet** is an instance variable that identifies an object. Figure 1-17 illustrates how an outlet in one object points to another object.

Figure 1-17 An outlet pointing from one object to another



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

An outlet can reference any object in an application: user-interface objects such as text fields and buttons, windows and dialogs, instances of custom classes, and even the application object itself. What distinguishes outlets is their relationship to Interface Builder.

Outlets are declared as:

```
public Object variableName;
```

You might notice that though the outlet type is labeled `id` in Interface Builder, the variable in the source file is declared with the type `Object`. This is simply a consequence of the Java specification—since Java is a strongly typed language, the object cannot directly act as a Cocoa `id` object (which represents a dynamically typed object).

Since `Object` refers to an arbitrary object type, you can—and should, in most cases—statically type outlets with the appropriate class:

```
IBOutlet NSButton myButton;
```

Xcode helps you remember what objects are Interface Builder outlets by appending a comment to the declaration:

```
IBOutlet NSButton myButton; /* IBOutlet */
```

This comment is added automatically if the outlet is originally declared in Interface Builder. If you add the outlet explicitly to your class definition, it is good practice to add this comment yourself. It helps you distinguish between interface outlets and other noninterface objects in the future.

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

At application load time, the instance variables that represent outlets are initialized to point to the corresponding target. For example, the `rateField` of the ConverterController instance would be initialized with a reference to the Exchange Rate text field object (see Connect the ConverterController Class to the Text Fields (page 35) 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.

You are free to use all Java language features in your code, but in rare situations, you may need to know something about the Cocoa implementation and its implications for Java code.

The Cocoa classes are implemented in Objective-C. When you use a Cocoa class from Java, you are using a Java "wrapper" class for the Objective-C class of the same name. An instance of such a class is actually an Objective-C instance. In this example, you are using the NSTextField Java class that wraps the Objective-C NSTextField class.

Java uses automatic garbage collection to manage dynamic memory; Objective-C, on the other hand, uses a retain-release reference-counting mechanism that is only semiautomatic. You will almost never have to concern yourself with the interaction between these two mechanisms if you take note of the following advice: Be sure that an object has been assigned to a instance variable before you use it as a target, a delegate, or an NSTableView item.

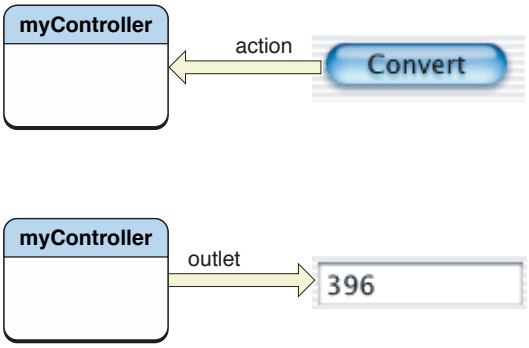
Target/Action in Interface Builder

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

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

For example, when a user clicks the Convert button in the Currency Converter window, the button gets the required information from its cell and invokes the `convert` method on the target outlet object, which is an instance of the custom class ConverterController. Figure 1-18 shows the interactions between the ConverterController class, the Convert button, and the Amount in Other Currency field.

Figure 1-18 Relationships in the target-action paradigm



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

```
public void myAction(Object sender)
```

Here, it looks for the argument `sender`.

Which Direction to Connect?

Usually the outlets and actions that you connect belong to a custom subclass of `java.lang.Object`. 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 instance you want to send the message to.
- To make an outlet connection, create the connection from the custom instance to another object (another instance or user-interface element) in the application.

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

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

Define the User Interface and Model Outlets of the ConverterController Class

ConverterController needs to communicate with the user-interface elements in the Currency Converter window. It must also communicate with an instance of the Converter class, defined in Defining the Converter Class (page 37). The Converter class implements the conversion computation.

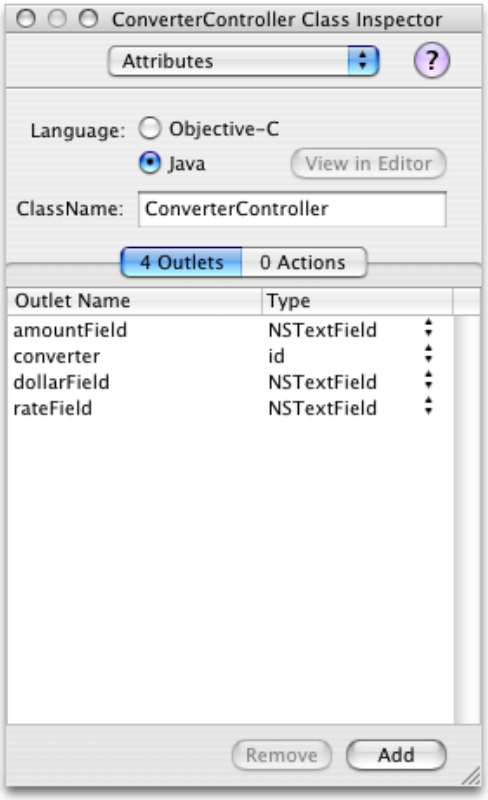
To add the outlets required by the ConverterController class:

1. Select ConverterController in the Classes pane in the `MainMenu.nib` window.
2. Choose Add Outlet to ConverterController from the Classes menu, or:
    a. Choose Attributes from the inspector pop-up menu.
    b. Click 0 Outlets.
    c. Click Add.
3. Name this outlet `rateField` and press Return.
4. Since the `rateField` outlet is still selected, all you have to do to create more outlets is press Return. Do this once to create the `dollarField` outlet, and again for the `amountField` outlet.
5. Add another outlet named "converter". This is the outlet ConverterController will use to communicate with the Converter instance. (The Converter class is defined later in this chapter.)

Notice the Type column in the table of outlets. By default, the type of outlets is set to `id`. It rarely works to leave it as `id`—Java is a statically typed language, and the Java compiler will not compile your program if it invoked any methods implemented by Cocoa classes on an object declared as `id`. You may sometimes get lucky—for example, if you do not call any of the object's intended class methods—but it's a good idea to get into the habit of setting the types for outlets. (This practice is not required when using Objective-C but improves the application's performance.) Change the type of the three outlets to `NSTextField` by choosing it from the pop-up menus currently set to `id`.

The result of these operations is shown in Figure 1-19.

Figure 1-19 Outlets and actions in the ConverterController inspector



Define the Actions of the ConverterController Class

ConverterController has one action method, `convert`. When the user clicks the Convert button, the `convert` method is invoked on the target object, an instance of ConverterController. "Action" refers both to a message sent to an object when the user clicks a button or manipulates some other control object and to the method that is invoked. To add the `convert` method to ConverterController:

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

Interconnecting the ConverterController Class and the User Interface

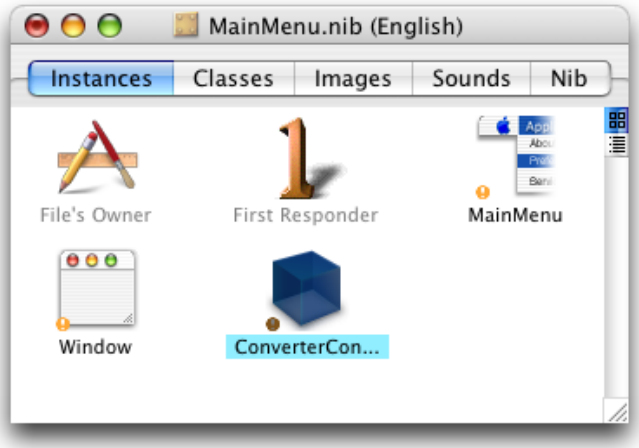
The following sections show how to connect the Converter Controller user interface and the ConverterController class to each other.

Create an Instance of the ConverterController Class

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

1. Select ConverterController in the Classes pane in the `MainMenu.nib` window.
2. Choose Instantiate ConverterController from the Classes menu. The instance appears in the Instances pane as shown in Figure 1-20. Notice that the instance has a yellow badge with an exclamation point next to it. This means the instance contains unconnected outlets.

Figure 1-20 A newly instantiated ConverterController instance



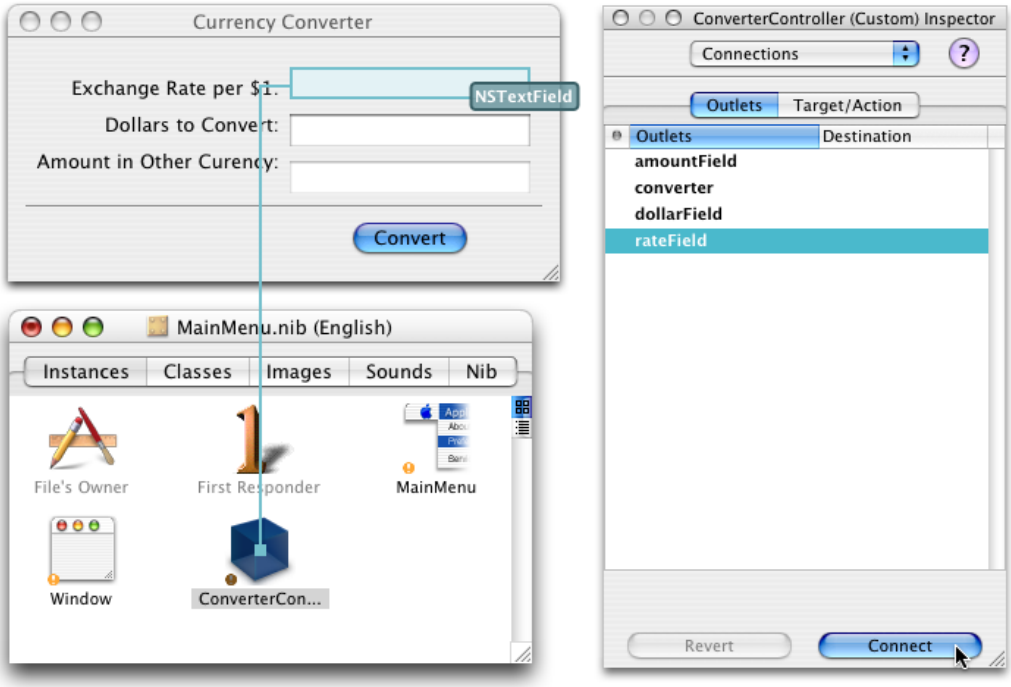
Connect the ConverterController Class to the Text Fields

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

1. In the Instances pane in the nib file window, Control-drag a connection from the ConverterController instance to the Exchange Rate text field.
2. Interface Builder displays the Connections pane of the inspector. Select the outlet that corresponds to the first field, `rateField`.

3. Click Connect, as shown in Figure 1-21.

Figure 1-21 Connecting ConverterController to the `rateField` outlet



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

Connect the Convert Button to the ConverterController convert Action Method

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

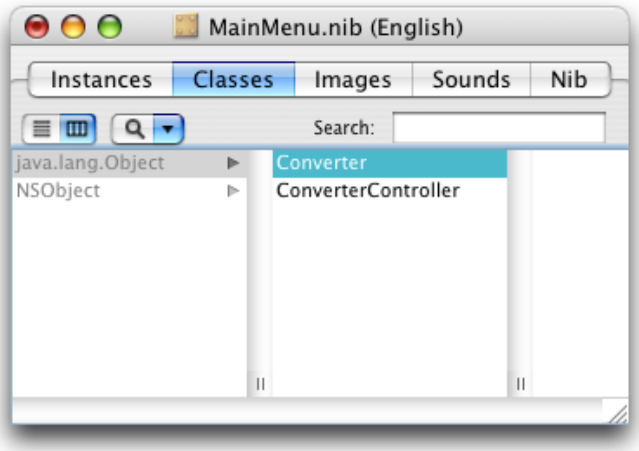
1. Control-drag a connection from the Convert button to the ConverterController instance in the nib file window.
2. In the Connections pane in the nib file window, make sure the Target/Action pane is displayed.
3. Select `convert()` in the Actions in ConverterController list and click Connect.
4. Save the nib file.

Defining the Converter Class

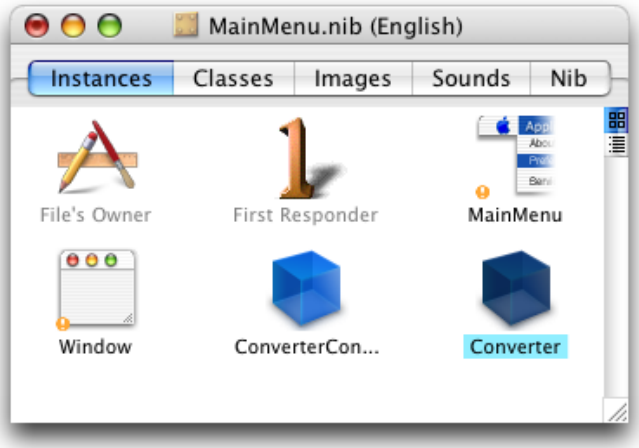
While connecting ConverterController's outlets, you probably noticed that one outlet remains unconnected: `converter`. This outlet identifies an instance of the Converter class in the Currency Converter application, but this instance doesn't exist yet.

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

1. In the Classes pane in the nib file window, create a subclass of `java.lang.Object` named "Converter". See Specify the ConverterController Class (page 29) for details.

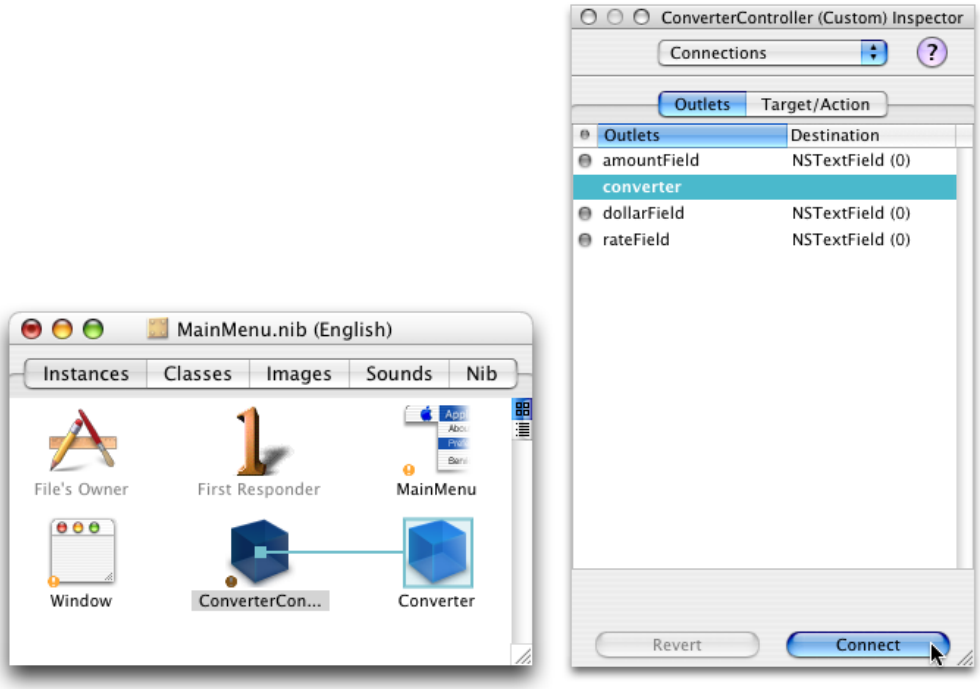


2. Instantiate the Converter class. See Create an Instance of the ConverterController Class (page 35) for details.



3. Connect the `converter` outlet of the ConverterController instance to the Converter instance, as shown in Figure 1-22.

Figure 1-22 Connecting `ConverterController.converter` to the Converter instance



4. Save the nib file.

Implementing Currency Converter

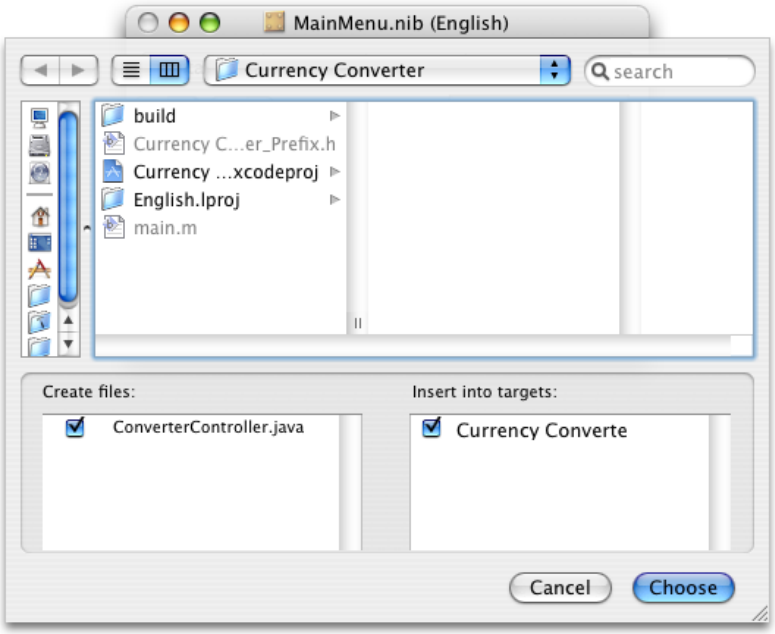
In Creating the Currency Converter Project and User Interface (page 11), you created the skeleton and the interface for the Currency Converter application. This chapter guides you through defining the custom behavior of the application and in the process teaches you the final steps essential to implementing a Cocoa application.

Generate the Source Files

To generate the source files for the Currency Converter application:

1. In the Classes pane in the nib file window, select the ConverterController class.
2. Choose Classes > Create Files for ConverterController.
3. Make sure `ConverterController.java` in "Create files" and Currency Converter in "Insert into targets" are selected.

   If Currency Converter in "Insert into targets" is not selected, you may not have saved the nib file earlier. Save the file and go to step 2.



4. Click Choose.

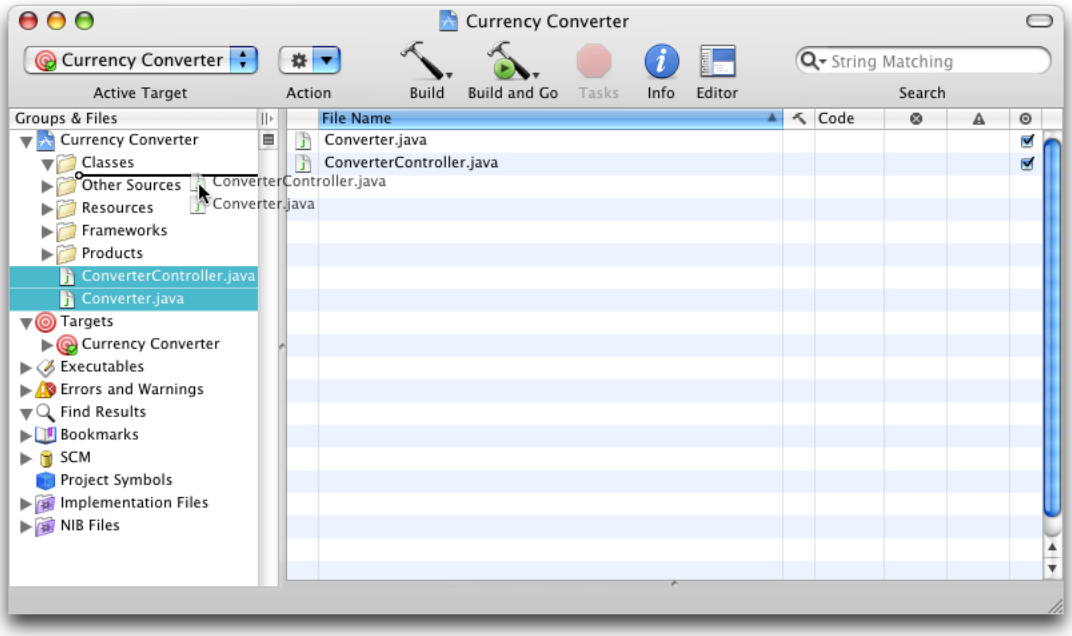
5. Repeat steps 1 through 4 for the Converter class.
6. Save the nib file.

Quit Interface Builder.

Place the Implementation Files in the Appropriate Group

When Interface Builder adds the `.java` files to the Currency Converter project, it puts them into the main Groups & Files list. Although this location is perfectly acceptable—the build system doesn't care where the files are—they are classes of your project and should be organized accordingly. To put these files in the Classes subgroup:

1. In the project window in Xcode, select the two files in the project group in the Groups & Files list.
2. Drag the files to the Classes group.



Implement the Currency Converter Classes

The Converter class needs a method to perform the conversion computation. To add this method to `Converter.java`:

1. In the Groups & Files list in the project window, select the Classes group.
2. In the detail view, double-click `Converter.java`. Xcode opens the file in an editor window.

3. Insert the tagged lines in Listing 2-1 into the `Converter.java` file.

Listing 2-1 Implementation of the Converter class

```
/* Converter */
import com.apple.cocoa.foundation.*;
import com.apple.cocoa.application.*;
public class Converter {
    public float convertDollars(float dollars, float rate) {       // 1
        return dollars * rate;                                     // 2
    }                                                              // 3
}
```

The `convertDollars` method multiplies the two arguments and returns the result.

Now, update the empty implementation of the `convert` method in `ConverterController.java` that Interface Builder generated for you by changing or adding the tagged lines in Listing 2-2 to the file.

Listing 2-2 Implementation of the ConverterController class

```
import com.apple.cocoa.foundation.*;
import com.apple.cocoa.application.*;

public class ConverterController {
    public NSTextField amountField; /* IBOutlet */
    public Converter converter; /* IBOutlet */                     // 1
    public NSTextField dollarField; /* IBOutlet */
    public NSTextField rateField; /* IBOutlet */

    public void convert(Object sender) { /* IBAction */
        float dollars, rate, amount;                               // 2
        dollars = dollarField.floatValue();                        // 3
        rate = rateField.floatValue();                             // 4

        amount = converter.convertDollars(dollars, rate);          // 5

        amountField.setFloatValue(amount);                         // 6
        rateField.selectText(this);                                // 7
    }
}
```

Notice that line 1 changes the type of the `converter` instance variable from `Object` to `Converter`. This is possible here because the Converter class is now defined in the project (when the ConverterController class was defined in Interface Builder, the Converter class didn't exist). You could have also done this by explicitly setting the type of the `converter` outlet to Converter within Interface Builder after defining the Converter class. This is an important step because leaving `converter`'s type as `Object` prevents the program from accessing the methods that the Converter class implements. (The Objective-C language doesn't have this limitation.)

The `convert` method does the following:

- Gets the floating-point values typed into the Exchange Rate and Dollars to Convert text fields (lines 3 and 4).
- Invokes the `convertDollars` method and stores the result (line 5).

- Uses `setFloatValue` to write the returned value in the Amount in Other Currency text field (line (line 6).
- Invokes `selectText` on the rate field (line 7). This selects any text in the text field or, if there is no text, places the insertion point so the user can enter text into the text field.

You've now completed the implementation of Currency Converter. Are you surprised how little code you had to write, given that your application now has a fully functional converting system and a beautiful Aqua user interface? Building Currency Converter (page 43) shows how to build and run the application.

Building Currency Converter

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

Overview of the Build Process

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

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

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

Xcode also copies nib files, sound files, image files, and other resources from the project directory to the appropriate locations in the application bundle. An application bundle is a directory that contains the application executable and the resources needed by that executable. This directory appears as a single file in the Finder; it can be double-clicked to launch the application.

Build the Currency Converter Application

To build the Currency Converter application:

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

The status bar at the bottom of the project window indicates the status of the build. When Xcode finishes—and encounters no errors along the way—it displays "Build succeeded" in the status bar. If there are errors, however, you need to correct them and start the build process again. See Correct Build Errors (page 44) for details.

Look Up Documentation

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

1. Open `ConverterController.java` in an editor window.
2. Option–double-click the word `setFloatValue` in the code. (Hold down the Option key and double-click the word.) The Developer Documentation window appears with a list of relevant method names in its detail view. The Java language version of `setFloatValue` in NSCell is highlighted in the detail view, and its associated HTML-formatted documentation appears in the content pane. This reference-library access system provides a fast way to get to reference material. Read more in Expanding on the Basics (page 47).
3. Close the Developer Documentation window.
4. Command–double-click the same word. A pop-up menu with a list of method names appears.
5. Choose `[NSCell setFloatValue]`. This time, Xcode jumps to the method declaration in the associated Objective-C header file.
6. Close the header file.

Run Currency Converter

Your hard work is about to pay off. Since you haven't edited any code since the last time you built the project, the application is ready to run. Notice that the Build and Build and Run toolbar items are pull-down menus containing other useful commands. Click and hold the Build and Run toolbar item and choose Run.

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

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

Correct Build Errors

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

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

1. Open `ConverterController.java`.
2. Delete the semicolon after the `selectText` call in the `convert` method.

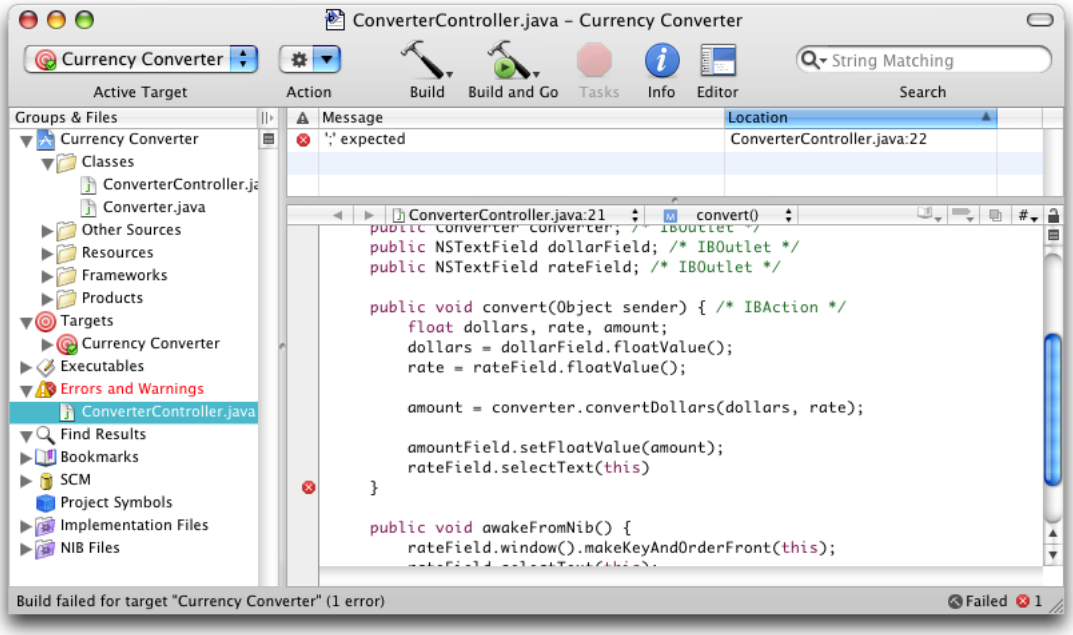
3. Click the Build toolbar item.

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

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

Select the file in the Errors and Warnings group to display the error. Xcode displays information about the error in the detail view, as shown in Figure 3-1.

Figure 3-1 Identifying build errors



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

Great Job!

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

- Composing a graphical user interface (GUI) in Interface Builder
- Testing a user interface in Interface Builder
- Specifying a class's outlets and actions in Interface Builder
- Connecting controller-instances to the user interface via using outlets and actions in Interface Builder

- Implementing a model class in Xcode
- Building applications and correcting build errors in Xcode

Expanding on the Basics

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

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

Application and Window Behavior

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

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

1. Click the Edit menu. Its items appear and disappear when you release the mouse button, as with any application menu.
2. Click the minimize button. Click the window's icon in the Dock to get the application back.
3. Click the close button. The Currency Converter window disappears. Quit the application and launch it again.

Controls and Text

The buttons and text fields in the Currency Converter window come with many built-in behaviors. Notice that the Convert button pulses as is the default for Aqua buttons that respond to Return key presses. Click the Convert button. Notice how the button is highlighted momentarily.

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

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

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

Menu Commands

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

Currency Converter needs only a few commands: the Quit and Hide commands and the Edit menu's Copy, Cut, and Paste commands. You can delete the unwanted commands if you wish. However, you can 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." 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 data.

Document Management

Many applications create and manage repeatable, semiautonomous 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.

See *Document-Based Applications Overview* 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.
- **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 custom views that draw their own content and respond to user actions. To assist you in this, Cocoa provides classes and functions for drawing, for example, the NSBezierPath class.

Internationalization

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

Editing Support

You can get several utility windows (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 a simple Interface Builder procedure, Cocoa automates simple printing of views that contain text or graphics. When a user starts a print action, an appropriate dialog helps to configure the print process. The output is WYSIWYG.

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

Help

You can very easily create context-sensitive help for your application using the Interface Builder inspector. When the pointer hovers over an object for which you've specified help content, a small window appears with the information you provided.

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 with a pencil, a brush, an 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.

Adopting Objective-C

Objective-C is a powerful, yet simple programming language. It's as simple as C but with the advantages that object orientation, dynamic typing, and late binding provide. Cocoa is a technology that facilitates the creation of powerful and innovative applications with relatively little custom code. Cocoa allows programmers to be very productive by implementing many of the user-interface behaviors that Mac OS X users expect in their applications. It also provides easy-to-use programming interfaces to system-level resources, such as memory, files, and interapplication-communication facilities. Cocoa uses Objective-C as its native language to take advantage of its speed, object orientation, and extensibility.

This document introduced you to Cocoa using Java, a language familiar to you. However, to create applications that can take advantage of all the benefits Cocoa provides, you must implement your applications using Objective-C.

In many ways, Objective-C and Java are very similar: They are both object-oriented languages with a C-based syntax. Although moving from Java to Objective-C presents a few challenges, the rewards are many, including:

- A small learning curve for Java programmers
- Fast prototyping of full-featured applications
- Developing cutting-edge user interfaces with few lines of code
- Using system resources through simple and consistent programming interfaces

The following sections explain the basic steps Java developers need to perform to adopt Objective-C.

Learn the Objective-C Language

The Objective-C language is based on the C language, which has a very simple syntax. The Java language borrows heavily from C's syntax; therefore, it should be easy for Java programmers to familiarize themselves with the syntax used in Objective-C.

These are some of the salient differences between Java and Objective-C:

- Java source files are compiled into bytecode object files and archives that can be used in more than one platform. Objective-C source files are compiled into machine-language object files. These files contain code tailored to a specific platform. Therefore, to build an Objective-C program targeted at more than one platform, each source file must be compiled once for each of the targeted platforms.
- C uses **header files**. Header files declare the interface that a module's client uses to interact with the corresponding implementation (a set of object files or libraries). The separation of public interface from implementation allows developers to conceive and release the interface to a module before implementing it. Objective-C maintains this separation.
- Like C, Objective-C has no automatic garbage collection. You must explicitly release the memory occupied by unused objects. Fortunately, Cocoa provides a mechanism you can use to facilitate this task. See Learn Memory Management in Cocoa (page 52) for details.

- Objective-C is a late-binding language. That is, the compiler doesn't bind method invocations to the corresponding implementations at compile time. Instead, the Objective-C runtime binds method invocations to the appropriate implementation at runtime. This eliminates the need to cast objects to the type that implements a particular method and allows the development of generic code.

For an in-depth discussion of Objective-C, see *The Objective-C 2.0 Programming Language*.

Learn the Cocoa Class Hierarchy

Just as Java programmers must learn the hierarchy of several Java packages—such as `java.lang`, `java.util`, and `java.awt` to develop applications—aspiring Cocoa programmers must familiarize themselves with the Foundation and Application Kit (AppKit) frameworks, Cocoa's main frameworks.

Cocoa applications use the Foundation framework to work with essential data types, such as strings, numbers, and dates, and to interact with system resources. They use the AppKit framework to present their user interface to the user and respond to user events, such as button clicks and menu choices.

For more information about the Cocoa frameworks, see The Cocoa Frameworks in *Cocoa Fundamentals Guide*.

Learn Memory Management in Cocoa

Garbage collection in Java frees programmers from having to release the memory used by objects that are not referenced by other parts of a running application. However, in general, programmers have little control over the garbage collector. Although the garbage collector attempts to perform its duties during an application's idle time, sometimes its activities may adversely impact an application's performance. Certain Java virtual machines, such as HotSpot, contain very efficient garbage collectors and may allow programmers or system administrators to configure them for specific usage patterns. But this approach might negate to some level the advantages of worry-free object instantiation.

The Objective-C language doesn't have a garbage collector. But Cocoa uses a mechanism based on reference counting to determine when it's safe to free the memory occupied by an object no longer in use. This mechanism, however, requires programmer intervention. When you create an object or receive an object from a routine that creates it but otherwise doesn't "own" it and you want the object to persist for an arbitrary amount of time, you tell Cocoa that you want to hold on to (or retain) the object. When you're done with the object, you tell Cocoa that you want to let go of (or release) the object. Through these messages, Cocoa increments or decrements a counter that indicates how many references to the object exist. When the reference count reaches zero, Cocoa disposes of the object and releases the memory it occupies. The advantage of this approach over Java's garbage collection is that programmers have more control over when objects are disposed of.

For details on memory management in Cocoa, see *Memory Management Programming Guide for Cocoa*.

Document Revision History

This table describes the changes to *Cocoa Tutorial for Java Programmers*.

Date	Notes
2006-10-03	Deprecated the Cocoa-Java tutorial.
	See *Cocoa Application Tutorial* for an introduction to Cocoa using Objective-C.
2005-07-07	Updated to specify the purpose of Cocoa-Java technology in application development. Revised tutorial using Xcode 2.1. Changed title from "Developing Cocoa Applications: A Tutorial."
	Added Adopting Objective-C (page 51).
	Shortened introduction by listing specific documents in See Also (page 8).
	Added instructions for adding the `converter` outlet to the ConverterController class in Interface Builder to Defining the Converter Class (page 37).
2004-02-12	Complete overhaul of the tutorial for Xcode and the current state of Java within it. The tutorial is now the Currency Converter, to provide parity with the Objective-C tutorial.
2003-05-03	Revision history was added to existing document. It will be used to record changes to the content of the document.

Document Revision History