

Currency Converter Tutorial

This tutorial will teach you the basics of Cocoa application development. You'll learn how to use Apple's development tools and the Objective-C language to build object-oriented applications that leverage the power of the Cocoa APIs.

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.

Currency Converter converts a dollar amount to an amount in another currency, given the rate of that currency relative to the dollar. Here's what it looks like:



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

Currency Converter Tutorial

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

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

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

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

Along the way, you'll also learn how to design an application using a common object-oriented design paradigm.

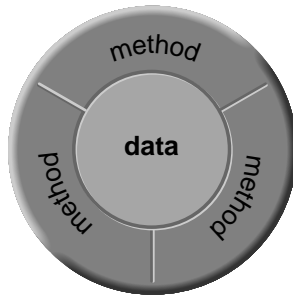
For additional background on Object oriented programming and Objective-C, see *Inside Cocoa: Object-Oriented Programming and the Objective-C Language*.

Designing the Currency Converter Application

An object-oriented application should be based on a design that identifies the objects of the application and clearly defines their roles and responsibilities. You normally work on a design before you write a line of code. You don't need any fancy tools for designing many applications; a pencil and a pad of paper will do.

Object Notation

When designing an object-oriented application, it is often helpful to graphically depict the relationships between objects. This book depicts objects graphically like this:



Though it might look a bit like a jelly donut, or a lifesaver, or a slashed tire—this symbol illustrates data encapsulation, the essential characteristic of objects. An object consists of both data and procedures for manipulating that data. Other objects or external code cannot access that data directly, but must send messages to the object requesting its data.

An object's procedures (called methods) respond to the message and may return data to the requesting object. As the symbol suggests, an object's methods do the encapsulating, in effect mediating access to the object's data. An object's methods are also its interface, articulating the ways in which the object communicates with the world outside it.

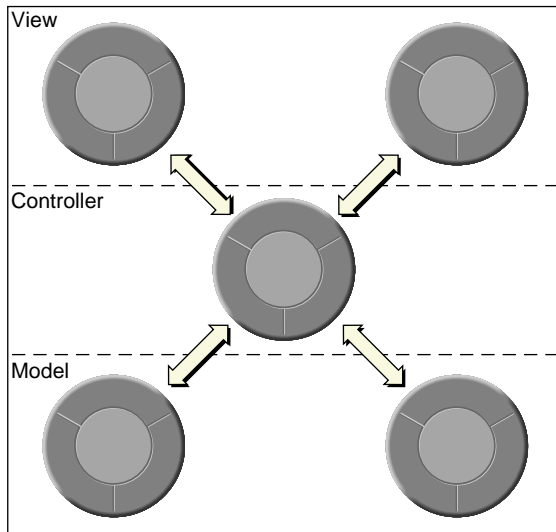
The donut symbol also helps to convey the modularity of objects. Because an object encapsulates a defined set of data and logic, you can easily assign it to particular duties within a program. Conceptually, it is like a functional unit—for instance, "Customer Record"—that you can move around on a design board; you can then plot communication paths to and from other objects based on their interfaces.

See the appendix "Object Oriented Programming" for a fuller description of data encapsulation, messages, methods, and other things pertaining to objects.

The Model-View-Controller (MVC) Paradigm

Currency Converter is an extremely simple application, but there's still a design behind it. This design is based upon the Model-View-Controller paradigm, the model behind many designs for object-oriented programs. This design paradigm aids in the development of maintainable, extensible, and understandable systems.

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



Model Objects

This type of object represents special knowledge and expertise. Model objects hold data and define the logic that manipulates that data. For example, a Customer object, common in business applications, is a Model object. It holds data describing the salient facts about a customer and has access to algorithms that access and calculate new data from those facts. A more specialized Model class might be one in a meteorological system called Front; objects of this class would contain the data and intelligence to represent weather fronts. Model objects are not directly displayed. They often are reusable, distributed, persistent, and portable to a variety of platforms.

View Objects

A View object in the paradigm represents something visible on the user interface (a window, for example, or a button). A View object is “ignorant” of the data it displays. The Application Kit usually provides all the View objects you need: windows, text fields, scroll views, buttons, browsers, and so on. But you might want to create your own View objects to show or represent your data in a novel way (for example, a graph view). You can also group View objects within a window in novel ways specific to an application. View objects, especially those in kits, tend to be very reusable and so provide consistency between applications.

Controller Object

Acting as a mediator between Model objects and View objects in an application is a Controller object. There is usually one per application or window. A Controller object communicates data back and forth between the Model objects and the View objects. It also performs all the application-specific chores, such as loading nib files and acting as window and application delegate. Since what a Controller does is very specific to an application, it is generally not reusable even though it often comprises much of an application’s code. (This last statement does not mean, however, that Controller objects cannot be reused; with a good design, they can.)

Because of the Controller’s central, mediating role, Model objects need not know about the state and events of the user interface, and View objects need not know about the programmatic interfaces of the Model objects. You can make your View and Model objects available to others from a palette in Interface Builder.

Hybrid Models

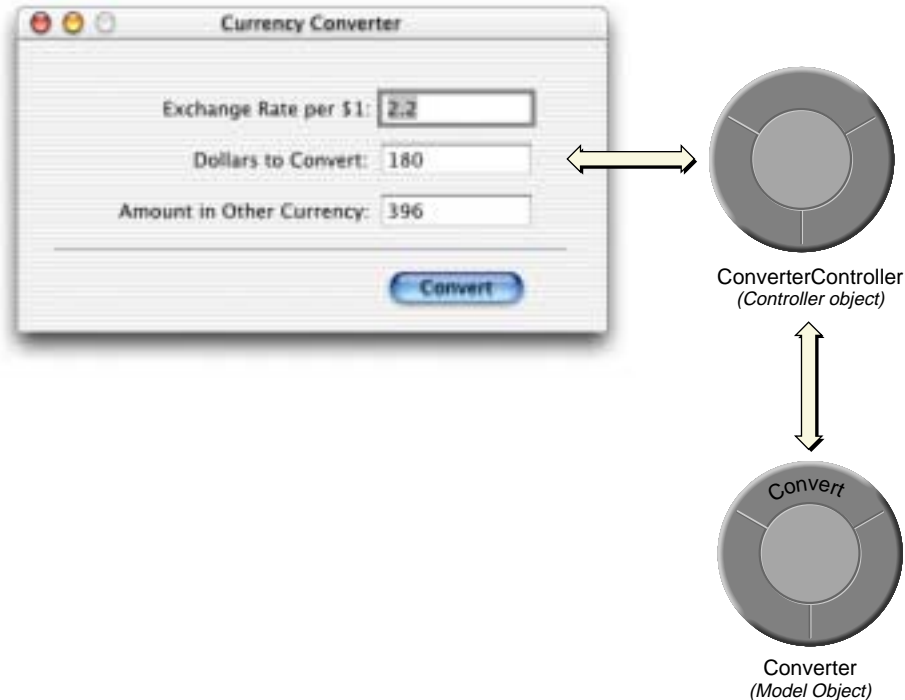
MVC, strictly observed, is not advisable in all circumstances. Sometimes it’s best to combine roles. For instance, in a graphics-intensive application, such as an arcade game, you might have several View objects that merge the roles of View and Model. In some applications, especially simple ones, you can combine the roles of Controller and Model; these objects join the special data structures and logic of Model objects with the Controller’s hooks to the interface.

MVC in Currency Converter’s Design

Currency Converter consists of two custom objects (Model and Controller) and a user interface (View), implemented using collection of ready-made Application Kit objects. A Converter object is responsible for computing a currency amount and

Currency Converter Tutorial

returning that value. Between the user interface and the Converter object is a controller object, ConverterController. ConverterController coordinates the activity between the Converter object and the UI objects.



The ConverterController class assumes a central role. Like all controller objects, it communicates with the interface and with model objects, and it handles tasks specific to the application. ConverterController gets the values that users enter into fields, passes these values to the Converter object, gets the result back from Converter, and puts this result in a field in the interface.

The Converter class merely computes a value from two arguments passed into it and returns the result. As with any model object, it could also hold data as well as provide computational services. Thus, objects that represent customer records (for example) are akin to Converter. By insulating the Converter class from application-specific details, the design for Currency Converter makes it more reusable.

This design for Currency Converter is intended to illustrate a few points, and so may be overly designed for something so simple. It is quite possible to have the application's controller class, `ConverterController`, perform the computation and do without the `Converter` class.

Creating The Currency Converter Project

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

There are three basic steps to create a Cocoa project:

“Open Project Builder” (page 7)

“Choose the New Project Command” (page 8)

“Select Project Type” (page 8)

Open Project Builder

To open Project Builder:

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



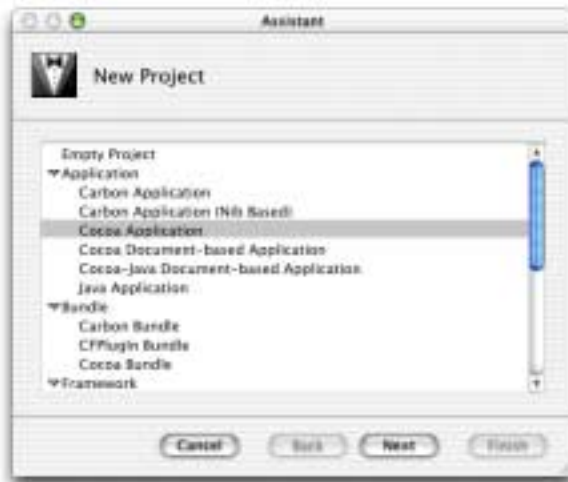
The first time you start Project Builder, you'll be asked a few setup questions. The default values should work for the majority of users.

Choose the New Project Command

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

Select Project Type

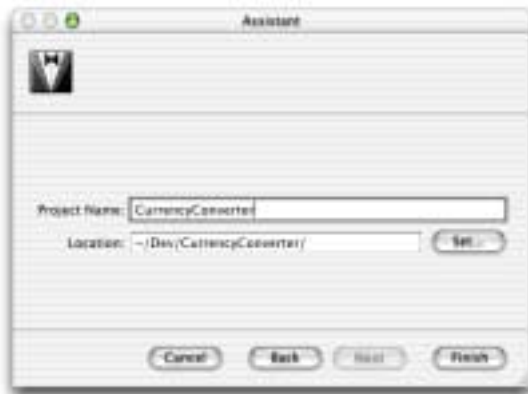
Project Builder can build many different types of applications, including everything from Carbon and Cocoa applications to Mac OS X kernel extensions and Mac OS X frameworks. For this tutorial, select Cocoa Application and click Next.



1. Using the file-system browser, navigate to the directory where you want your project to be.

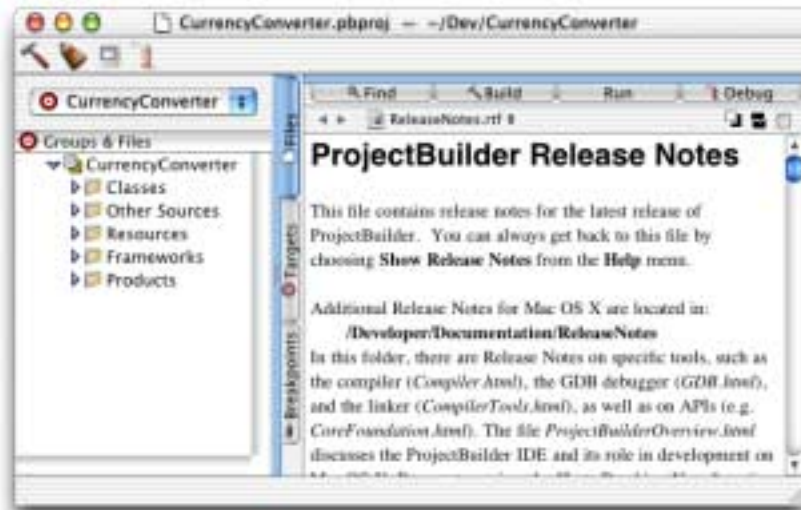
C H A P T E R 1

Currency Converter Tutorial



2. Type the name of the project in the Name field. For the current project, type the name “CurrencyConverter.”
3. Click Finish.

When you click Finish, Project Builder creates and displays a project window.



Currency Converter Tutorial

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

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

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

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

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

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

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

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

`English.lproj`

A directory containing resources localized to your preferred language. In this directory are nib files automatically created for the project. For more information on nib files, see the developer documentation for Interface Builder.

`main.m`

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

`CurrencyConverter.pbproj`

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

Once a project has been created, it's a good idea to have Project Builder index the project. During indexing Project Builder stores all symbols of the project (classes, methods, globals, etc.) on disk. This allows Project Builder to access project-wide information quickly. Indexing is indispensable to such features as name completion and Project Find.

Creating the Currency Converter Interface

Creating a functioning graphical interface for Currency Converter involves only twelve code-free steps:

- “Open the Main Nib File” (page 12)
- “Resize the Window” (page 14)
- “Set the Window’s Title and Attributes” (page 16)
- “Place a Text Field, Resize and Initialize It” (page 16)
- “Duplicate an Object” (page 17)
- “Change the Attributes of a Text Field” (page 18)
- “Assign Labels to the Fields” (page 18)
- “Add a Button to the Interface and Initialize It” (page 20)
- “Add a Horizontal Decorative Line” (page 20)
- “Align the Text Fields and Labels” (page 21)
- “Enable Tabbing Between Text Fields” (page 23)
- “Test the Interface” (page 24)

This section of the tutorial guides you through these steps, explaining interesting and important aspects of Cocoa programming along the way.

Nib Files

Every application with a graphical user interface has at least one nib file. The main nib file is loaded automatically when an application launches and contains the application menu and generally 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) and later restored in memory. Archived objects contain information such as their size, location, and position in the object hierarchy. At the top of the hierarchy of archived objects is the File’s Owner object, a proxy object that points to the actual object that owns the nib file (typically the one that loaded the nib file from disk).

Images Image files that you drag and drop over the nib file window or over an object that can accept them (such as a button or image view).

Class References Interface Builder can store the details of Cocoa objects and objects that you palettize (static palettes), but it does not know how to archive instances of your custom classes since it doesn’t have access to the code. For these classes, Interface Builder stores a proxy object to which it attaches class information.

Connection Information Information about how objects within the object hierarchy are interconnected. Connector objects special to Interface Builder store this information. When you save the document, connector objects are archived in the nib file along with the objects they connect.

Open the Main Nib File

1. Locate MainMenu.nib in the Resources group in Project Builder.
2. Double-click to open it. This will open Interface Builder and bring up the nib file.

A default menu bar and window titled “My Window” will appear when the application is launched.

Windows in Cocoa

A window in Cocoa looks very similar to windows in other user environments such as Windows or Mac OS 9. It is a rectangular area on the screen in which an application displays things such as controls, fields, text, and graphics. Windows can be moved around the screen and stacked on top of each other like pieces of paper. A typical Cocoa window has a title bar, a content area, and several control objects.

NSWindow and the Window Server

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

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

The window that the Window Server creates is paired with an object supplied by the Application Kit: an instance of the `NSWindow` class. Each physical window in a Cocoa program is managed by an instance of `NSWindow` (or subclass).

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

Application, Window, View

In a running Cocoa application, `NSWindow` objects occupy a middle position between an instance of `NSApplication` and the views of the application. (A view is an object that can draw itself and detect user events.) The `NSApplication` object keeps a list of its windows and tracks the current status of each. Each window, on the other hand, manages a hierarchy of views in addition to its window.

At the “top” of this hierarchy is the content view, which fits just within the window’s content rectangle. The content view encloses all other views (its subviews), which come below it in the hierarchy. The `NSWindow` distributes events to views in the hierarchy and regulates coordinate transformations among them.

Another rectangle, the frame rectangle, defines the outer boundary of the window and includes the title bar and the window’s controls. Cocoa uses the lower-left corner of the frame rectangle defines the window’s location relative to the screen’s coordinate system and establishes the base coordinate system for the views of the window. This is different from Carbon and Classic applications which use the upper left corner as the origin of the coordinate system. Views draw themselves in coordinate systems transformed from (and relative to) this base coordinate system.

Key and Main Windows

Windows have numerous characteristics. They can be on-screen or off-screen. On-screen windows are “layered” on the screen in tiers managed by the Window Server. On-screen windows also can carry a status: key or main.

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

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

Resize the Window

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

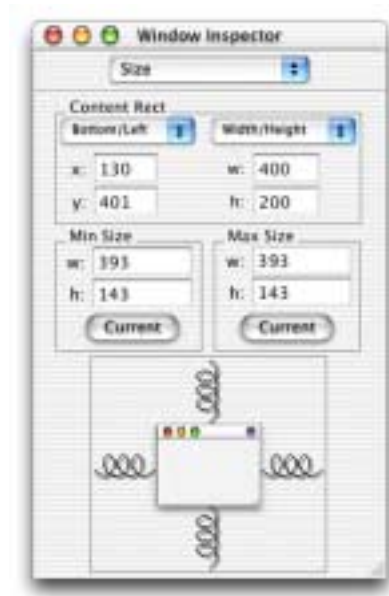
C H A P T E R 1

Currency Converter Tutorial



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

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



Set the Window's Title and Attributes

While the Inspector is open, set other attributes for the window.

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

Place a Text Field, Resize and Initialize It

1. Select the Views palette in the Cocoa objects palette. If you don't see the Cocoa objects palette, select Palettes from the Tools menu to bring it forward.

Currency Converter Tutorial



2. Drag a text field object (the empty field in the leftmost column) onto the Currency Converter window.
3. Resize the text field by grabbing a handle and dragging in the direction you want it to grow.



Currency Converter needs two more text fields, both the same size as the first. You have two options: you can drag another object from the palette and make it the same size; or you can duplicate the first object.

Duplicate an Object

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

Currency Converter Tutorial

2. Choose Duplicate (Command-D) from the Edit menu. The new text field appears slightly offset from the original field.
3. Reposition the new text field under the first text field.
4. To make the third text field, type Command-D again. Notice that IB remembered the offset from the previous Duplicate command and automatically applied it to the newly created text field.

Change the Attributes of a Text Field

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

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

Assign Labels to the Fields

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

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



C H A P T E R 1

Currency Converter Tutorial

2. Make the text right aligned; with the Message Text object selected, click on the third button from the left in the Alignment area of the Inspector.



3. Duplicate the text label twice and enter the text for each as shown:



Add a Button to the Interface and Initialize It

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

1. Drag the button object from the Views palette and put it in the lower-right portion of the window.
2. Double click the title of the button to select its text label and change the title to “Convert”.
3. Make the button the same size as the text fields. Either drag its handles to make it larger, or select the text field, and then the button, and then use the Same Size function in the Layout menu.
4. Choose Attributes in the NSButton Inspector, and then choose Return from the Key pop-up menu. This will give the button the capacity to respond to the Return key as well as to mouse clicks.

Add a Horizontal Decorative Line

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

1. Drag a horizontal separator object from the Views palette onto the interface. It’s located just beneath the Box object in the lower right hand corner of the Views palette.
2. Drag the endpoints of the line until the line extends across the window.



Aqua Layout and Object Alignment

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

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

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

Align the Text Fields and Labels

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

- Dragging objects with the mouse
- Pressing arrow keys (with the grid off, the selected objects move one pixel)
- Using a reference object to put selected objects in rows and columns
- Using the built-in alignment functions
- Specifying origin points in the Size display of the Inspector panel
- Using a grid

For Currency Converter, use the built-in alignment functions technique.

1. Choose Show Layout Rects from the Layout menu.

C H A P T E R 1

Currency Converter Tutorial

2. Choose Alignment from the Tools menu. This brings up the Alignment palette.



3. Select the three text field objects by dragging a selection box around them.
4. In the Alignment palette, choose Right/Bottom from the Edge popup menu.
5. In the Alignment palette, click the third button from the left in the top row. As suggested by the icon on the button, this aligns the selected objects by their right edges.
6. In the Alignment palette, choose Center from the Edge popup menu.
7. Select the first text field label “Exchange Rate per \$1:” and its corresponding text field.
8. In the Alignment palette, click the second button from the left in the top row. As suggested by the icon on the button, this aligns the selected objects by their centers.
9. Repeat steps 6 and 7 for the other label/text field pairs.
10. Use what you’ve learned to align the text fields and the Convert button.

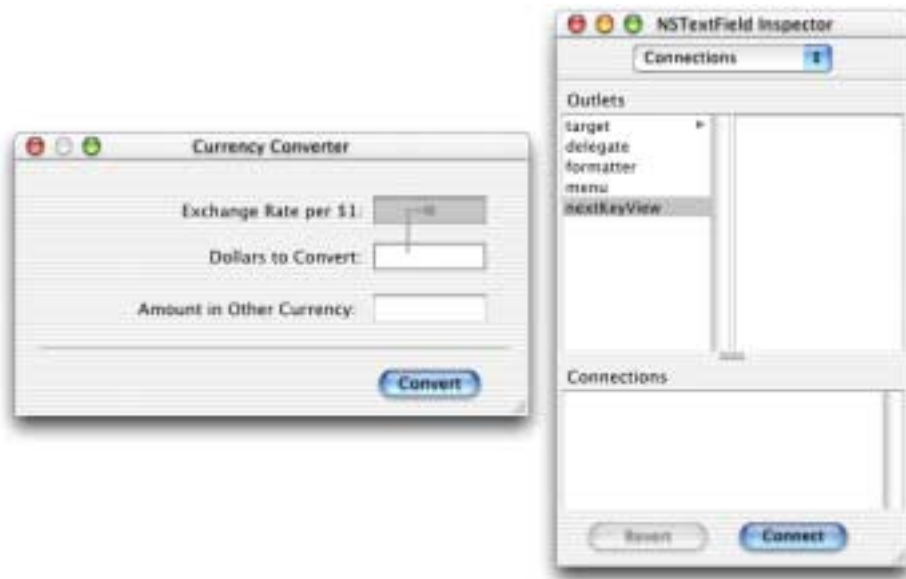
At this point, your window should look like this:



Enable Tabbing Between Text Fields

The final step in composing the Currency Converter interface has more to do with behavior than with appearance. You want the user to be able to tab from the first editable field to the second, and back to the first. Many objects in Interface Builder's palettes have an instance variable named `nextKeyView`. This variable identifies the next object to receive keyboard events when the user presses the Tab key (or the previous object if Shift-Tab is pressed). If you want inter-field tabbing, you must connect fields through the `nextKeyView` variable.

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



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

Test the Interface

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

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

For Free with Cocoa

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

Application and Window Behavior

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

Reactivate Currency Converter by clicking on its window. Then move the window around by its title bar. Here are some other tests you can make:

1. Click the Edit menu. Its items appear and disappear when you release the mouse button, as with any application menu.
2. Click the miniaturize button. Click the window’s icon in the Dock get the application back.
3. Click the close button, and the Currency Converter window disappears. (Choose Quit from the main menu and re-enter test mode to get the window back.)

If we hadn’t configured Currency Converter’s window in Interface Builder to remove the resize box, we could resize it now. We could also have set the auto-resizing attributes of the window and its views so that the window’s objects would resize proportionally to the resized window or would retain their initial size (see Interface Builder Help for details on auto-resizing).

Controls and Text

The buttons and text fields of Currency Converter come with many built-in behaviors. Notice that the Convert button “throbs” as is the default for Aqua buttons associated with the Return key. Click the Convert button. Notice how the button is highlighted momentarily.

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

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

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

Menu Commands

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

Currency Converter needs only a few commands: the Quit and Hide commands and the Edit menu’s Copy, Cut, and Paste commands. You can delete the unwanted commands if you wish. However, you could also add new ones and get “free” behavior. An application designed in Interface Builder can acquire extra functionality with the simple addition of a menu or menu command, without the need for compilation. For example:

- The Font submenu adds behavior for applying fonts to text in `NSText` objects, like the one in the scroll view object in the DataViews palette. Your application gets the Font panel and a font manager “for free.”
- The Text submenu allows you to align text anywhere text is editable, and to display a ruler in the `NSText` object for tabbing, indentation, and alignment.

- Many objects that display text or images can print their contents as PostScript data. Later you'll learn how to add the Print menu

Document Management

Many applications create and manage repeatable, semi-autonomous objects called documents. Documents contain discrete sets of information and support the entry and maintenance of that information. A word-processing document is a typical example. The application coordinates with the user and communicates with its documents to create, open, save, close, and otherwise manage them.

File Management

An application can use the Open panel, which is created and managed by the Application Kit, to help the user locate files in the file system and open them. It can also use the Save panel to save information in files. Cocoa also provides classes for managing files in the file system (creating, comparing, copying, moving, and so forth) and for managing user defaults.

Communicating With Other Applications

Cocoa gives an application several ways to exchange information with other applications:

- **Pasteboard** The pasteboard is a global facility for sharing information among applications. Applications can use the pasteboard to hold data that the user has cut or copied and may paste into another application.
- **Services** Any application can access the services provided by another application, based on the type of selected data (such as text). An application can also provide services to other applications such as encryption, language translation, or record-fetching.
- **Drag-and-drop** If your application implements the proper protocol, users can drag objects to and from the interfaces of other applications.

Custom Drawing and Animation

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

Localization

Cocoa provides API and tool support for localizing the strings, images, sounds, and nib files that are part of an application.

Editing Support

You can get several panels (and associated functionality) when you add certain menus to your application's menu bar in Interface Builder. These “add-ons” include the Font panel (and font management), the Color panel (and color management), and, although it's not a panel, the text ruler and the tabbing and indentation capabilities the Text menu brings with it.

Formatter classes enable your application to format numbers, dates, and other types of field values. Support for validating the contents of fields is also available.

Printing

With just a simple Interface Builder procedure, Cocoa automates simple printing of views that contain text or graphics. When a user clicks the control, an appropriate panel 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—known as “tool tips”—for your application using the Inspector in Interface Builder. After you've entered the tool tip text, the user can then hold the cursor over an object on the application's interface and a small window will appear containing concise information on the object.

Plug-in Architecture

You can design your application so that users can incorporate new modules later on. For example, a drawing program could have a tools palette: pencil, brush, eraser, and so on. You could create a new tool and have users install it. When the application is next started, this tool appears in the palette.

Defining the Classes of Currency Converter

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

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

“Specify a Subclass” (page 29)

“Define the Outlets of the Class” (page 36)

“Define the Actions of the Class” (page 37)

Specify a Subclass

You must go to the Classes display of the nib file window to define a class. Once there, the first thing you must do is select the **superclass**, the class your new **subclass** will inherit from. Let's start with the ConverterController class.

1. Select the Classes display of the `MainMenu.nib` window.
2. ConverterController doesn't need to inherit any complex behavior, so select NSObject from the list of classes (It's at the very top of the list.). This is the superclass your new subclass will inherit from. NSObject will provide ConverterController with everything necessary to function as a Cocoa object.



3. Select Subclass from the Classes menu.
4. Type “ConverterController” to replace the text “MyObject”.

Now your class is established in the hierarchy of classes within the nib file.

Classes and Objects

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

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

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

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

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

A class, on the other hand, implements the behavior of all of its instances in a running program. The donut symbol used to represent objects is a bit misleading here, because it suggests that each object contains its own copy of code. This is fortunately not the case; instead of being duplicated, this code is shared among all current instances in the program.

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

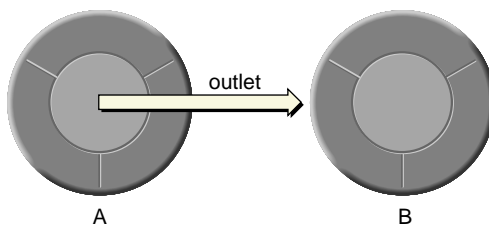
See the appendix, “Object-Oriented Programming,” for more on these and other aspects of classes.

Paths for Object Communication: Outlets, Targets, and Actions

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

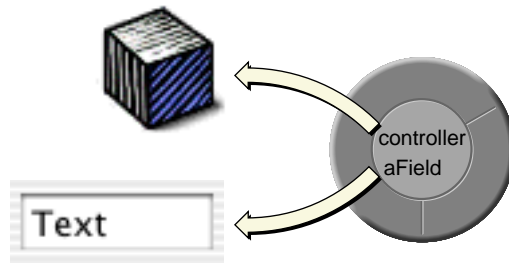
Outlets

An outlet is an instance variable that identifies an object.



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

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



Outlets are declared as:

```
IBOutlet id variableName;
```

You can use `id` as the type for any object; objects with `id` as their type are dynamically typed, meaning that the class of the object is determined at run time.

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

```
IBOutlet NSButton* myButton;
```

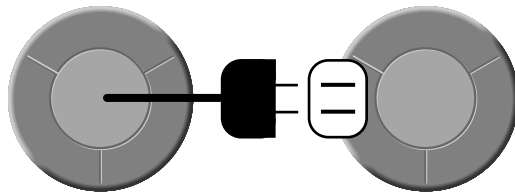
Interface Builder can “recognize” outlets in code by their declarations, and it can initialize outlets. You usually set an outlet's value in Interface Builder by drawing connection lines between objects. There are ways other than outlets to reference objects in an application, but outlets and Interface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

When You Make a Connection in Interface Builder

As with any instance variable, outlets must be initialized at run time to some reasonable value—in this case, an object's identifier (`id` value). Because of Interface Builder, an application can initialize outlets when it loads a nib file.

When you make a connection in Interface Builder, a special connector object holds information on the source and destination objects of the connection. (The source object is the object with the outlet.) This connector object is then stored in the nib file. When a nib file is loaded, the application uses the connector object to set the source object's outlet to the `id` value of the destination object.

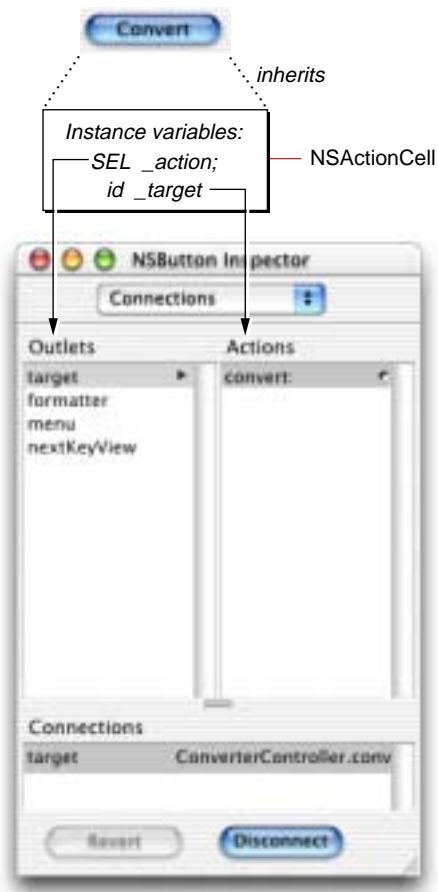
It might help to understand connections by imagining an electrical outlet (as used in the Classes display of the nib file window) embedded in the destination object. Also picture an electrical cord extending from the outlet in the source object. Before the connection is made the cord is unplugged and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



Target/Action in Interface Builder

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

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



For example, when a user clicks the Convert button of Currency Converter, the button gets the required information from its cell and sends the message `convert:` to the target outlet, which is an instance of your custom class `ConverterController`.

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

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

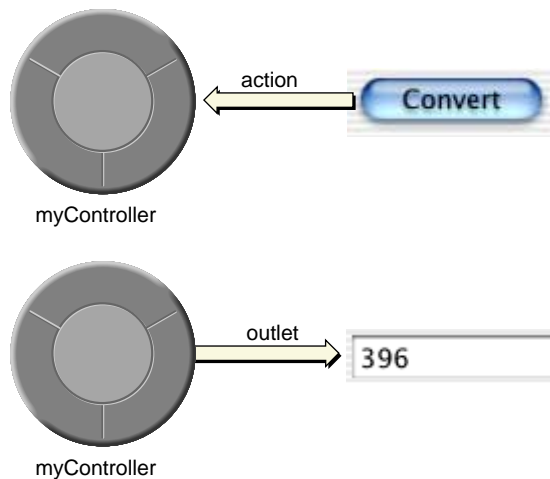
It looks in particular for the argument `sender`.

Which Direction to Connect?

Usually the outlets and actions that you connect belong to a custom subclass of NSObject. For these occasions, you need only follow a simple rule to know which way to draw a connection line in Interface Builder. Draw the connection in the direction that messages will flow:

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

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

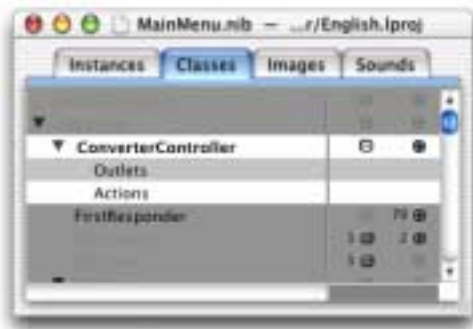


Another way to clarify connections is to consider who needs to find whom. With outlets, the custom object needs to find some other object, so the connection is from the custom object to the other object. With actions, the control object needs to find the custom object, so the connection is from the control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

Define the Outlets of the Class

Classes created in Interface Builder hold objects as outlets typed as **id**. Objects in applications often hold outlets as part of their data so they can send messages to the objects referenced by the outlets. An outlet lets you keep track of or manipulate something in the interface.

1. Select ConverterController in the Classes window.
2. Click the electrical-outlet icon to the right of the class.



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

ConverterController needs to access the text fields of the interface, so you've just provided outlets for that purpose. But ConverterController must also communicate with the Converter class (yet to be defined). To enable this communication, add an outlet named `converter` to ConverterController.

Define the Actions of the Class

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

1. Choose Add Action from the Classes menu.
2. Type the name of the method, `convert`. IB adds the “:” for you.

Connecting ConverterController to the Interface

Generate an Instance of the Class

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

1. Select Instantiate from the Classes menu. The instance will appear in the Instances view as shown highlighted below.



Currency Converter Tutorial

When you instantiate a class (that is, create an instance of it), Interface Builder switches to the Instances display and highlights the new instance, which is named after the class.

In fact, the Instantiate command does not generate a true instance of ConverterController, but creates a stand-in object used for establishing connections. When the nib file's contents are unarchived, Interface Builder will create true instances of these classes and use the proxy objects to establish the outlet actions connections.

Connect the Custom Class to the Interface

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

1. In the Instances display of the nib file window, Control-drag a connection line from the ConverterController instance to the first text field. When the instance is outlined, release the mouse button.

C H A P T E R 1

Currency Converter Tutorial



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



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

Connect the Interface Controls to the Class's Actions

To receive action messages from the user interface – to be notified, for example, when users click a button – you must connect objects that emit those messages to `CurrencyConverter`. The procedure for connecting actions is similar to that for outlets, but with one major difference. When you connect an action, always start the connection line from the *control object* (such as a button, text field, or form) that sends the action message. The instance to which you are connecting—in this case `ConverterController`—is the *target* outlet of the control object.

1. Control-drag a connection from the Convert button to the `ConverterController` instance in the nib file window. When the instance is outlined, release the mouse button.
2. In the Connections display, make sure **target** in the Outlets column is selected.

Currency Converter Tutorial

3. Select **convert** in the Actions column.
4. Click the Connect button.
5. Save the CurrencyConverter.nib file.

Define the Converter Class

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

Define the Converter class. This should be pretty easy because Converter, as you might recall, is a model class within the Model-View-Controller paradigm. Since instances of this type of class don't communicate directly with the interface, there is no need for outlets or actions. Here are the steps to be completed:

1. In the Classes display, make Converter a subclass of NSObject.
2. Instantiate the Converter class.
3. Make an outlet connection between ConverterController and Converter. Hint: Control-drag from the ConverterController instance to the Converter instance.
4. Save CurrencyConverter.nib.

Implementing the Classes of Currency Converter

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

Generate the Source Files

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

C H A P T E R 1

Currency Converter Tutorial

1. Go to the Classes display of the nib file window.
2. Select the ConverterController class.
3. Choose Create Files in the Classes menu.

Interface Builder then displays a dialog box.



1. Verify that the check boxes in the Create column next to the .h and .m files are selected.
2. Verify that the check box next to “Insert into Project Builder” is selected.
3. Click the Choose button.
4. Repeat for the Converter class
5. Save the nib file.

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

Objective C Quick Reference

The Objective-C language is a superset of ANSI C with special syntax and run-time extensions that make object-oriented programming possible. Objective-C syntax is uncomplicated, but powerful in its simplicity. You can mix standard C with Objective-C code.

The following summarizes some of the more basic aspects of the language. See “Inside Cocoa: Object-Oriented Programming and the Objective-C Language” for additional details. Also, see “Object-Oriented Programming” in the appendix for explanations of key terms.

Messages and Method Implementations

Methods are procedures implemented by a class for its objects (or, in the case of class methods, to provide functionality not tied to a particular instance). Methods can be public or private; public methods are declared in the class’s header file (see above). Messages are invocations of an object’s method that identify the method by name.

Message expressions consist of a variable identifying the receiving object followed by the name of the method you want to invoke; enclose the expression in brackets.

```
[anObject doSomethingWithArg:this];
```

As in standard C, terminate statements with a semicolon.

Messages often result in values being returned from the invoked method; you must have a variable of the proper type to receive this value on the left side of an assignment.

```
int result = [anObj calcTotal];
```

You can nest message expressions inside other message expressions. This example gets the window of a form object and makes the returned `NSWindow` object the receiver of another message.

```
[[form window] makeKeyAndOrderFront:self];
```

A method is structured like a function. After the full declaration of the method comes the body of the implementing code enclosed by braces.

Currency Converter Tutorial

Use `nil` to specify a `null` object; this is analogous to a `null` pointer. Note that some Cocoa methods do not accept `nil` as an argument.

A method can usefully refer to two implicit identifiers: `self` and `super`. Both identify the object receiving a message, but they affect differently how the method implementation is located: `self` starts the search in the receiver's class whereas `super` starts the search in the receiver's superclass. Thus,

```
[super init];
```

causes the `init` method of the superclass to be invoked.

In methods you can directly access the instance variables of your class' instances. However, accessor methods are recommended instead of direct access, except in cases where performance is of paramount importance.

Declarations

Dynamically type objects by declaring them as `id`:

```
id myObject;
```

Since the class of dynamically typed objects is resolved at run time, you can refer to them in your code without knowing beforehand what class they belong to. Type outlets and objects in this way if they are likely to be involved in polymorphism and dynamic binding.

Statically type objects as a pointer to a class:

```
NSString *mystring;
```

You statically type objects to obtain better compile-time type checking and to make code easier to understand.

Declarations of instance methods begin with a minus sign (`-`), a space after the minus sign is optional.

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

Put the type of value returned by a method in parentheses between the minus sign (or plus sign) and the beginning of the method name. (See above example.) Methods that return nothing should have a return type of `void`.

Currency Converter Tutorial

Method argument types are in parentheses and go between the argument's keyword and the argument itself:

```
- (id)initWithName:(NSString *)name andType:(int)type;
```

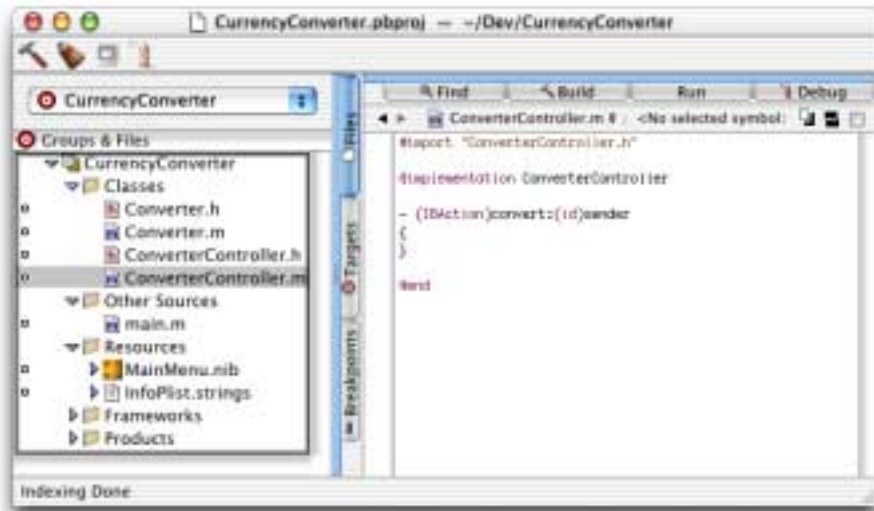
Be sure to terminate all declarations with a semicolon.

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

Examine an Interface (header) File in Project Builder

When Interface Builder adds the source files to the Currency Converter project, the source files appear in the Other Sources group, and the header files appear in the top level CurrencyConverter group. Since these files are class implementations, add them to the Classes group.

1. Click Project Builder's main window to activate it.
2. Select ConverterController.m and Converter.m in the project browser.
3. Drag them into the Classes group.
4. Select ConverterController.h and Converter.h in the project browser.
5. Drag them into the Classes group.



Add a method declaration

You can add instance variables or method declarations to a header file generated by Interface Builder. This is commonly done, but it isn't necessary in ConverterController's case. But we do need to add a method to the Converter class that the ConverterController object can invoke to get the result of the computation. Let's start by declaring the method in Converter.h.

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

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

@end
```

Currency Converter Tutorial

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

Now you need to update both implementation files.

Implement Currency Converter’s Classes.

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

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

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

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

1. Next update the “empty” implementation of the `convert:` method that **Interface Builder** generated in `ConverterController.m`.

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

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

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

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

Currency Converter Tutorial

```
}
```

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

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

The `convert:` method does the following:

- Gets the floating-point values typed into the rate and dollar-amount fields.
- Invokes the `convertAmount: atRate:` method and gets the returned value.
- Uses `setFloatValue:` to write the returned value in the Amount in Other Currency text field (`totalField`).
- Sends `selectText:` to the rate field; this selects any text in the field or, if there is no text, inserts the cursor so the user can begin another calculation.

Each line of the `convert:` method, excluding the declaration of floats, is a message. The “word” on the left side of a message expression identifies the object receiving the message (called the “receiver”). These objects are identified by the outlets you defined and connected. After the receiver comes the name of the method that the sending object (called the “sender”) wants to invoke. Messages often result in values being returned; in the above example, the local variables `rate`, `amt`, and `total` hold these values.

Implement the `awakeFromNib:` method

Before you build the project, add a small bit of code to `ConverterController.m` that will make life a little easier for your users. When the application starts up, you want Currency Converter’s window to be selected and the cursor to be in the “Exchange Rate per \$1” field. We can do this only after the nib file is unarchived, which establishes the connection to the text field `rateField`. To enable set-up operations like this, `awakeFromNib` is sent to all objects when unarchiving concludes. Implement this method to take appropriate action.

1. Type in the following code

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


Currency Converter Tutorial

You've seen the `selectText: message` before, in the `convert: implementation`; it selects the text in the text field that receives the message, inserting the cursor if there is no text.

The `makeKeyAndOrderFront: message` does as it says: It makes the receiving window the key window and puts it before all other windows on the screen. This message also nests another message, `[rateField window]`. This message returns the window to which the text field belongs, and the `makeKeyAndOrderFront: method` is then sent to this returned object.

Build, Debug, and Run CurrencyConverter

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

What Happens When You Build An Application

By clicking the Build button in Project Builder, you run the build tool. By default, the build tool is jam, but it can be any build utility that you specify as a project default in Project Builder. The build tool coordinates the compilation and linking process that results in an executable file. It also performs other tasks needed to build an application.

The build tool invokes the compiler, passing it the source code files of the project. Compilation of these files (Objective-C, C++, and standard C) produces machine-readable object files for the architecture or architectures specified for the build.

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

The build tool also copies nib files, sound, images, and other resources from the project to the appropriate localized or non-localized locations in the application package. An application package is a directory that contains the application executable and the resources needed by that executable. This directory appears as a single file in the Finder that can be double-clicked to launch the application.

Build the Project

You begin builds from the Project Build panel.

1. Save source code files and any changes to the project.
2. Click the Build button on the main window.



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

Debug the Project

Of course, rare is the project that is flawless from the start. Project Builder is likely to catch some errors when you first build your project. To see the error-checking features of Project Builder, introduce a mistake into the code.

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

Help on Development Tools

The Project Builder and Interface Builder applications provide Tool Tips, short descriptions of parts of the interface that briefly appear when the mouse pointer hovers over those areas.

Project Builder and Interface Builder also provide comprehensive task-based Help, accessible from the Help menu.

Help on APIs

Project Builder gives you several ways to get information on Cocoa APIs when you're developing an application.

Project Find

The Project Find panel allows you to search for definitions of, and references to, classes, methods, functions, constants, and other symbols in your project. Since it is based on project indexing, searching is quick and thorough, and leads directly to the relevant code. Searches can be textual or regular expression based.

Reference Documentation Lookup

If the results of a search using Project Find include Cocoa symbols, you can easily get related reference documentation that describes that symbol. See the online help for instructions on the use of this feature.

Frameworks

Under Frameworks in the project browser, you can browse the header files and documentation related to Cocoa frameworks within Project Builder. The Application Kit and Foundation frameworks always are included by default for Cocoa projects.

Cocoa Technical Documentation

Most Cocoa programming documentation is located in `/Developer/Documentation/Cocoa`.

Reference

Cocoa API reference documentation includes specifications of classes, protocols, functions, types, and constants. This documentation is located in the appropriate Cocoa frameworks, except for information that is common to all frameworks. Use AppleHelp and go to Developer Center -> Cocoa.

Development Tools Reference.

Covers the compiler, the debugger, and other tools; it's located in `/Developer/Documentation/DeveloperTools`.

Tasks and Concepts

Discovering Cocoa: A Developer Tutorial (this manual).

“Programming Languages”: an on-line resource for programming languages, especially Objective-C in both “classic” and “modern” syntaxes.

“Topics in Cocoa Programming” contains concepts and programming procedures.

Run Currency Converter

Congratulations! You've just created your first Cocoa application. All you have to do is click the Run button in Project Builder (immediately to the left of the debug button) to launch the application. If you want, you can locate CurrencyConverter in the Finder (in the build subdirectory of the CurrencyConverter PB project), launch it, and try it out.

Enter some rates and dollar amounts and click Convert. Also, select the text in a field and choose Services sub-menu in the Application menu; this menu now lists the other applications that can do something with the selected text.

Of course, the more complex an application is, the more thoroughly you will need to test it. You might discover errors or shortcomings that necessitate a change in overall design, in the interface, in a custom class definition, or in the implementation of methods and functions.

Although it's a simple application, Currency Converter still introduced you to many of the concepts, tools, and skills you'll need to develop Cocoa applications. Let's review what you've learned:

- Composing a graphical user interface (GUI) with Interface Builder
- Testing the interface
- Designing an application using the Model-View-Controller paradigm
- Specifying a class's outlets and actions
- Connecting the class instance to the interface via its outlets and actions

C H A P T E R 1

Currency Converter Tutorial

- Class implementation basics
- Building an application and error resolution

C H A P T E R 1

Currency Converter Tutorial