Xcode Quick Tour Guide

Tools > Xcode

2006-11-07

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

Simultaneously published in the United States and Canada.

Contents

Introduction Introduction to Xcode Quick Tour Guide 7

Organization of This Document 7
See Also 8

Chapter 1 Creating a Project 9

Creating an Xcode Project 9
The Project Window 11
  The Toolbar and Status Bar 11
  Groups & Files 12
Editing Project Files 12
  Using Interface Builder 13
  Using the Source Editor 18
Building the Application 21
Running the Application 22
  Compile-Time Errors 23
  Runtime Debugging 24
Summary 27

Chapter 2 Finding Technical Information 29

Searching For API Documentation 29
  Searching From the Documentation Window 30
  Searching From a Source Editor Window 31
Searching For All Relevant Documentation 31
Following Links 32
Finding Information in Headers 33
  Finding Framework Headers 33
  Finding Symbol Declarations 35
Downloading ADC Reference Library Updates 36
Summary 37

Chapter 3 Designing a User Interface 39

Creating the Converter Interface 39
  Getting Started 40
  Setting the Window's Attributes 41

Adding the Fahrenheit Control 43
Adding the Celsius Control 44
Adding Static Text Labels 44
Adding the Convert Button 45
Aqua Layout and Object Alignment 46
Finishing the Window Layout 47
Testing the Interface 48
Implementing the Converter Application 48
Summary 49

Chapter 4 Using Fix and Continue 51

Configuring the Project 51
Patching the Running Application 51
Summary 54

Appendix A Converter Source Code 55

Document Revision History 59

Figures and Listings

Chapter 1 Creating a Project 9

Figure 1-1 The Xcode application icon 9
Figure 1-2 The New Project Assistant 10
Figure 1-3 Hello project window 11
Figure 1-4 The project window toolbar 11
Figure 1-5 Adding class source files to a project from Interface Builder 14
Figure 1-6 The Interface Builder `NSWindow` inspector 15
Figure 1-7 Adding a user interface element to a window in Interface Builder 16
Figure 1-8 Selecting the class of a user interface element in Interface Builder 17
Figure 1-9 Specifying the autosizing behavior of a user interface element in Interface Builder 18
Figure 1-10 Assigning source files to the appropriate group 19
Figure 1-11 The code completion pop-up menu 20
Figure 1-12 The Debug build configuration in the Hello target Info window 22
Figure 1-13 Main window for the Hello application 23
Figure 1-14 Error and warning messages 24
Figure 1-15 Setting a breakpoint 25
Figure 1-16 The Debug window 26
Figure 1-17 A breakpoint action 27
Listing 1-1 Initial Implementation of the `HelloView` class 19
Listing 1-2 Implementation of the `drawRect:` method 20

Chapter 2 Finding Technical Information 29

Figure 2-1 Using the API-reference search option in Xcode 30
Figure 2-2 Using the full-text search option in Xcode 32
Figure 2-3 Searching for headers in a framework group 34
Figure 2-4 Using the Open Quickly command 35
Figure 2-5 The Open Quickly dialog 35
Figure 2-6 Header search results 36
Figure 2-7 Reference Library update dialog 37

Chapter 3 Designing a User Interface 39

Figure 3-1 The Converter user interface 39
Figure 3-2 Editing windows in Interface Builder 41
Figure 3-3 Attributes of the Converter window 42

Figure 3-4 Converter window before adding controls 43
Figure 3-5 Adding an editable text field 43
Figure 3-6 Converter window after adding temperature controls 44
Figure 3-7 Converter window after adding static text 45
Figure 3-8 Measuring distances in Interface Builder 46
Figure 3-9 Layout rectangles in Interface Builder 47

Chapter 4 Using Fix and Continue 51

Figure 4-1 The default drawing behavior of Sketch 52
Figure 4-2 The new drawing behavior of Sketch 53

Appendix A Converter Source Code 55

Listing A-1 Converter source code 55

Introduction to Xcode Quick Tour Guide

Xcode Tools is the developer tools package for Mac OS X. This package includes an integrated suite of software development tools, including compilers and applications, together with an extensive set of programming libraries and interfaces. The centerpiece of these tools is the Xcode application, which provides an elegant, powerful user interface for creating and managing software development projects in Mac OS X. (Elsewhere in this document, the name Xcode refers to the Xcode application.)

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

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

This document gives you a hands-on introduction, in the form of four short tutorials, to the Xcode application and some of its companion tools.

To get the most out of these tutorials, you should already be familiar with C programming and the Mac OS X user interface. You don't need any previous experience with Mac OS X software development.

Organization of This Document

This document contains the following chapters:

- "Creating a Project" (page 9) provides an introduction to some of the basic features in Xcode.
- "Finding Technical Information" (page 29) shows how to quickly find and display technical information in Apple's developer documentation library.
- "Designing a User Interface" (page 39) demonstrates how to use Interface Builder to design a user interface for a Carbon application.
- "Using Fix and Continue" (page 51) describes how to use Xcode's patching facility to modify a running application during a debugging session.

This document also contains an appendix and a revision history.

See Also

- For information about all aspects of software development in Mac OS X, visit the Apple Developer Connection website at http://developer.apple.com.
- If you're interested in developing applications using Cocoa, see *Cocoa Application Tutorial*.
- To learn more about Xcode, see *Xcode User Guide*.
- For an overview of the software development tools in Xcode, see *Getting Started with Tools*.
- For tips on converting CodeWarrior projects into Xcode projects, see *Porting CodeWarrior Projects to Xcode*.
- If you want an introduction to Mac OS X system architecture and technologies, see *Mac OS X Technology Overview*.
- The principles and conventions of Mac OS X user interface design are covered in *Apple Human Interface Guidelines*.

Creating a Project

Every software product starts out as a project. A project is the repository for all the elements used to design and build your product—including source files, user interface specifications, sounds, images, and links to supporting frameworks and libraries.

Xcode is a great application for creating and managing projects. Xcode can be used to build a wide variety of software products, ranging from Carbon and Cocoa applications to kernel extensions, libraries, and Mac OS X frameworks.

This short tutorial shows how to create an Xcode project for a Cocoa application called Hello that prints "Hello, World!" inside a window. Along the way, you get a chance to explore some of the basic features of Xcode.

Important: Before continuing, make sure your development environment meets the requirements specified in the introduction.

Creating an Xcode Project

Xcode includes a set of built-in project templates configured for building specific types of software products. When creating a project, you can save time by starting with the appropriate template.

To create an Xcode project for the Hello application, using a template:

1. Find Xcode in the `/Developer/Applications` directory, and double-click its icon (see Figure 1-1). The first time you use Xcode, it asks a few setup questions. The default values should work for the majority of users.

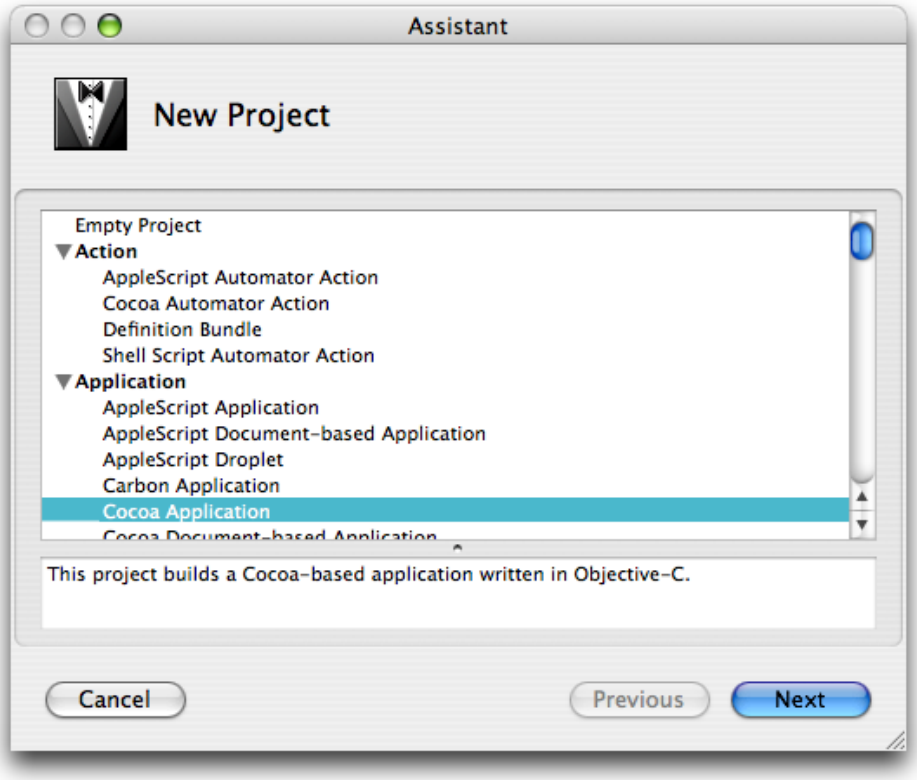
Figure 1-1 The Xcode application icon

2. Choose File > New Project. New Project Assistant appears.

If you're curious, browse through the list of templates to see the variety of software products Xcode can build.

3. Select the Cocoa Application template as shown in Figure 1-2, and click Next.

Figure 1-2 The New Project Assistant

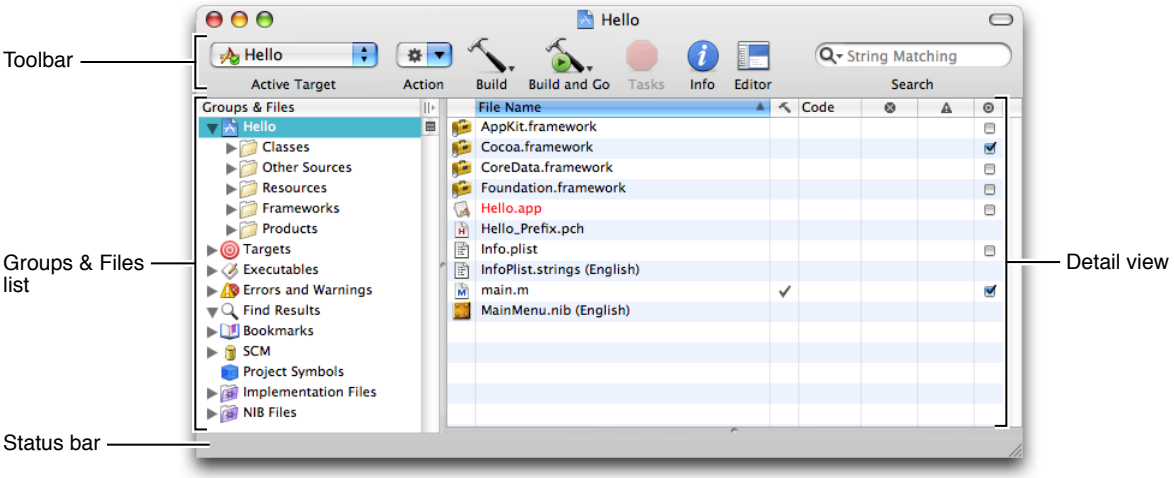


4. Type `Hello` in the Project Name field.

5. Click Choose and navigate to the location where you want the Xcode application to create the project folder.

6. Click Finish. The Xcode application creates the project and displays the project window, as shown in Figure 1-3.

Figure 1-3 Hello project window



The Project Window

The project window is the control center for an Xcode project. This section briefly describes the components of the project window.

The Toolbar and Status Bar

The project window toolbar contains buttons and other controls you can use to perform common operations. Figure 1-4 shows the default toolbar.

Figure 1-4 The project window toolbar

- The Active Target pop-up menu selects the active build target.
- The Action pop-up menu displays actions you can perform on the currently selected item. You get the same menu when you Control-click an item.
- Build buttons initiate actions such as building, cleaning, running, and debugging.

> **Note:** Toolbar buttons with drop-down triangles have additional commands associated with them. Pressing one of these buttons allows you to choose a different command from a pop-up menu.

- The Tasks button allows you to stop any operation in progress.
- The Info button opens an Info window to examine and edit groups, files, and targets in your project.

- The Editor button shows or hides the source editor in the project window.
- The Search text field filters the items currently displayed in the detail view.

Located at the bottom of the project window is the status bar, which displays status messages for the project, such as whether a build is successful.

Groups & Files

The Xcode application uses various groups to organize the files and information in a project. These groups are visible in the Groups & Files list, shown in Figure 1-3 (page 11). To see the contents of a group, select the group or click its disclosure triangle.

Groups are flexible—they don't need to reflect the actual structure of the project directory or the way the build system views the files. Their purpose is to help you organize your project and quickly find project components and information. You can customize some of the default groups in the list and define groups of your own.

Groups with plain icons are static groups. The items in a static group stay put until you move them. Groups with gears or other fancy icons are dynamic groups, called **smart groups**, that show items with a particular characteristic. The items in a smart group can change as you perform various actions in your project.

Here are some of the basic groups:

- The project group organizes all the components needed to build your product. This group is always listed first, and its name is the same as the project.
- Inside the project group are subgroups that contain your project's source files, resource files, frameworks, and products.
- The Targets group contains all the build targets in your project. A build target is a blueprint for building a product from a set of specified files and libraries.
- Errors and Warnings is a smart group that contains any error and warning messages generated during the most recent build.

To the right of the Groups & Files list is the detail view. The detail view is a flattened list of all the items that are currently selected in the Groups & Files list. You can quickly search and sort the items in the detail view, gaining rapid access to important information in your project.

Editing Project Files

This section shows how to modify the behavior of the application to print "*Hello, World!*" in its main window.

To implement this new behavior, you:

- Create the `HelloView` class as a subclass of `NSView`
- Add a `HelloView` user interface element to the Hello application window

- Implement the `HelloView -drawRect:` method to draw the textual greeting

The following sections describe these tasks in detail.

Using Interface Builder

In Cocoa, all drawing is done in objects known as **views.** The `NSView` class implements the basic functionality of a view. Subclasses of `NSView`, such as `NSButton` and `NSTextView`, add functionality tailored for specific user interface objects.

You use Interface Builder to design user interfaces. To familiarize yourself with the task of creating user interfaces with Interface Builder, you create a subclass of `NSView` and add an instance of this class to the application window:

1. Open `MainMenu.nib` in Interface Builder:

   In the Groups & Files list, select the project group and double-click `MainMenu.nib` in the detail view.

2. Create the `HelloView` class:

   a. In the `MainMenu.nib` window, click Classes.

   b. In the tree view, select `NSObject` > `NSResponder` > `NSView`.

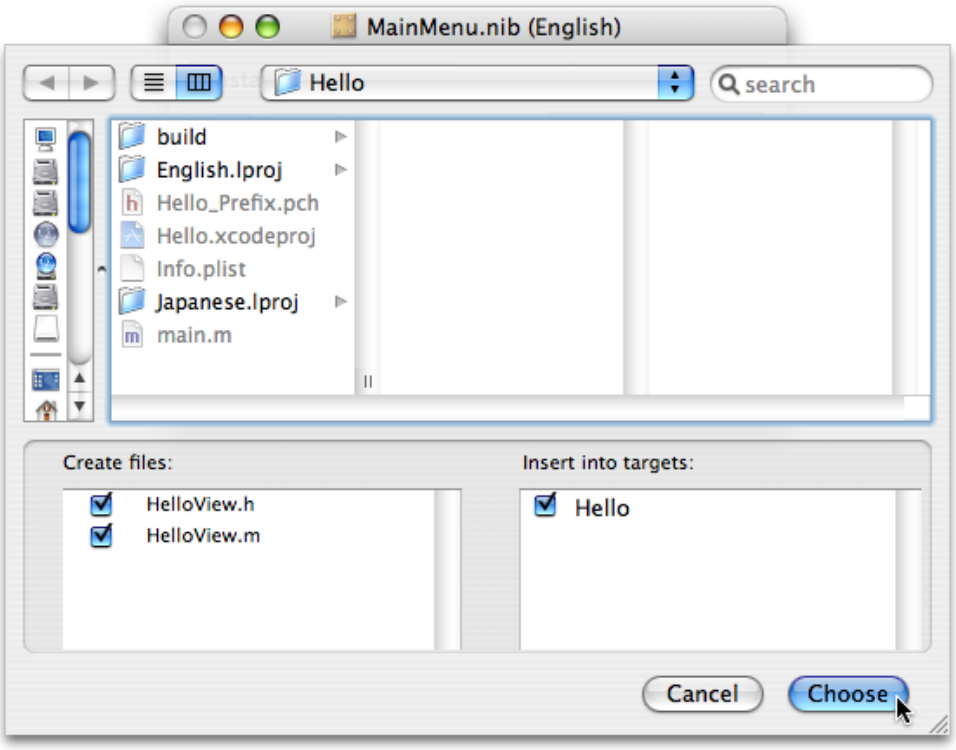
   c. Choose Classes > Subclass `NSView`.

   d. Change `MyView` to `HelloView` and press Return.

   e. Choose Classes > Create Files for `HelloView`.

In the dialog that appears (Figure 1-5), click Choose.

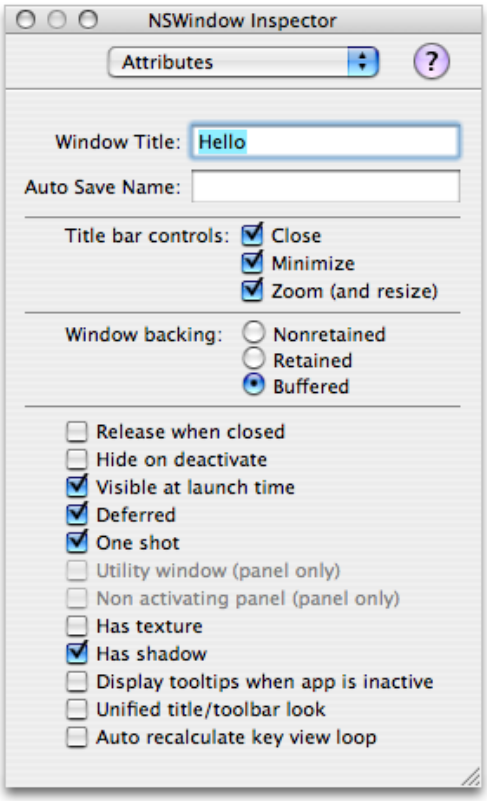
Figure 1-5 Adding class source files to a project from Interface Builder



3. Change the title of the application window:
    a. Click inside the Window window.
    b. Choose Tools > Show Inspector.

c. Enter `Hello` in the Window Title text field and press Return. Figure 1-6 shows the `NSWindow` inspector.

Figure 1-6 The Interface Builder `NSWindow` inspector

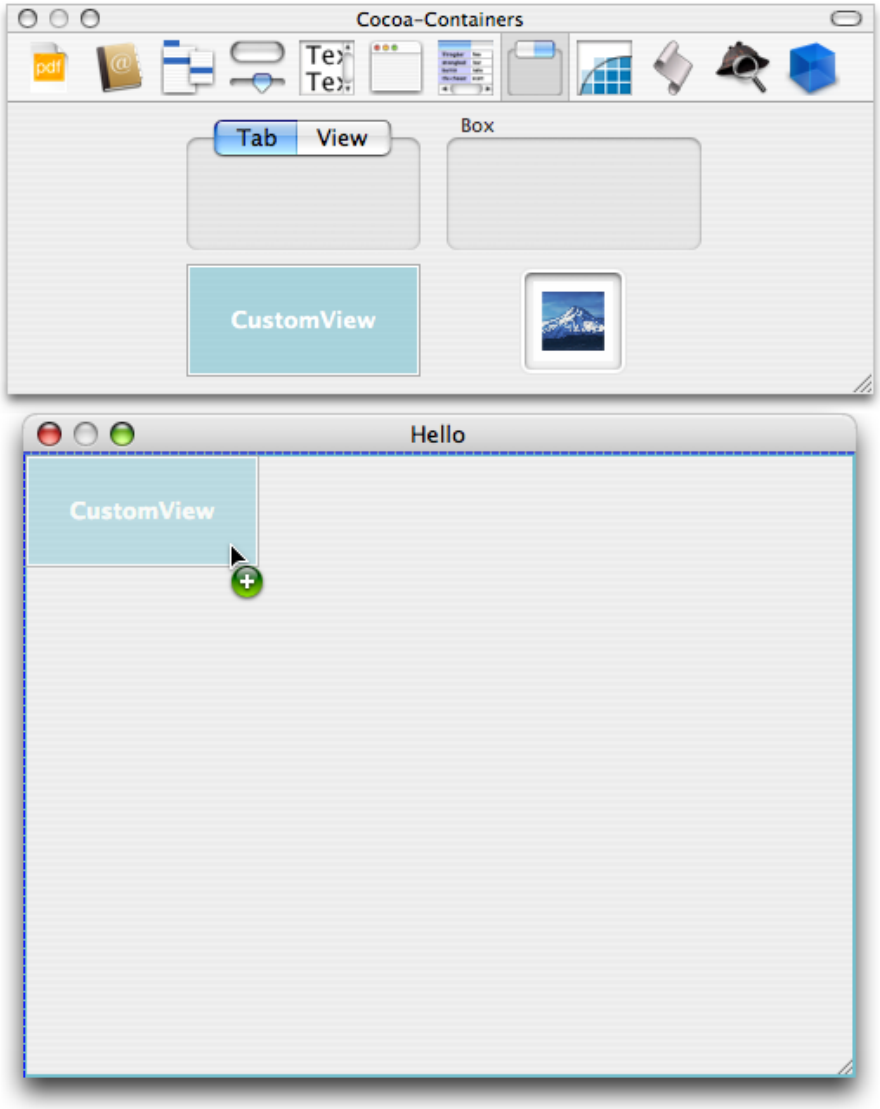


4. Add a `HelloView` element to the Hello window:

   a. Choose Tools > Palettes > Show Palettes.

b. Select the Cocoa-Containers palette and drag the Custom View element to the Hello window, as shown in Figure 1-7.

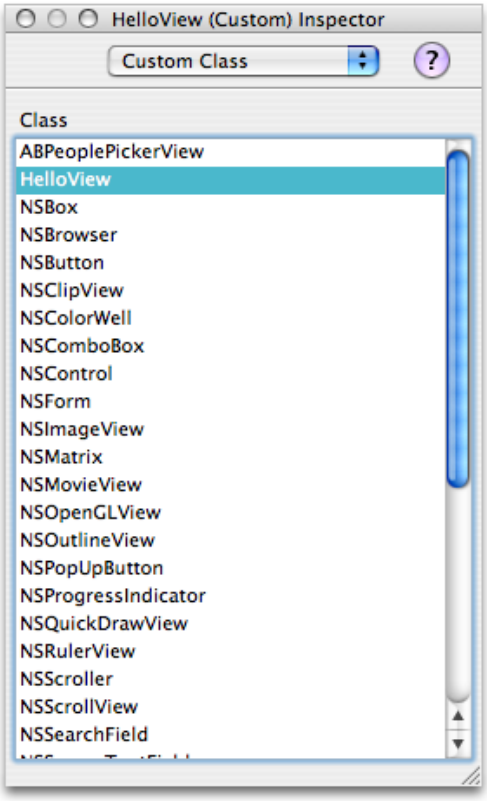
Figure 1-7 Adding a user interface element to a window in Interface Builder



c. Resize the `CustomView` element so that it occupies the entire content area of the Hello window.

d. In the `NSView` inspector, choose Custom Class from the pop-up menu and select `HelloView` in the Class list, as shown in Figure 1-8.

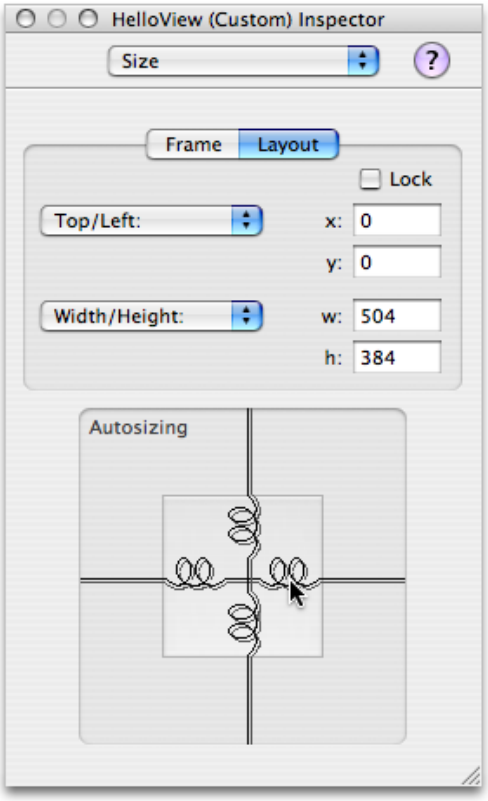
Figure 1-8 Selecting the class of a user interface element in Interface Builder



e. Choose Size from the pop-up menu.

f. In the Autosizing area, click the vertical and horizontal lines in the inner square, so they change to springs, as shown in Figure 1-9.

Figure 1-9 Specifying the autosizing behavior of a user interface element in Interface Builder



Save the nib file and quit Interface Builder.

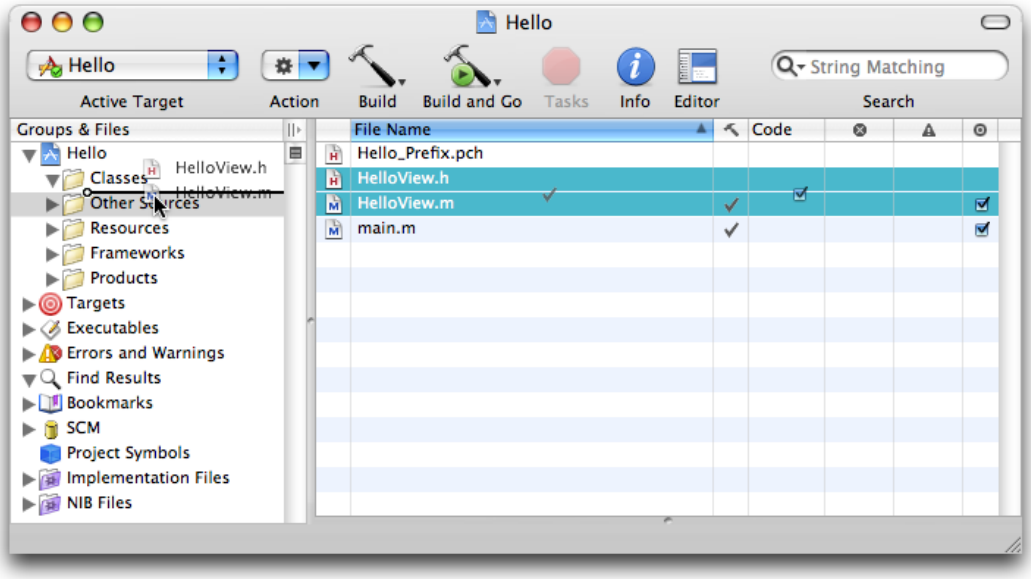
Using the Source Editor

Xcode has a built-in source editor that supports multiple buffers and windows. For convenience, a source file can be edited in a separate editor or directly inside the project window.

To edit the source code for the Hello project:

1. Move the `HelloView` source files to the Classes group, as shown in Figure 1-10.

Figure 1-10 Assigning source files to the appropriate group



2. Open `HelloView.m` in an editor by double-clicking it in the detail view. The file should look like Listing 1-1.

Listing 1-1 Initial Implementation of the `HelloView` class

```
#import "HelloView.h"

@implementation HelloView

- (id)initWithFrame:(NSRect)frameRect
{
    if ((self = [super initWithFrame:frameRect]) != nil) {
        // Add initialization code here
    }
    return self;
}

- (void)drawRect:(NSRect)rect
{
}

@end
```

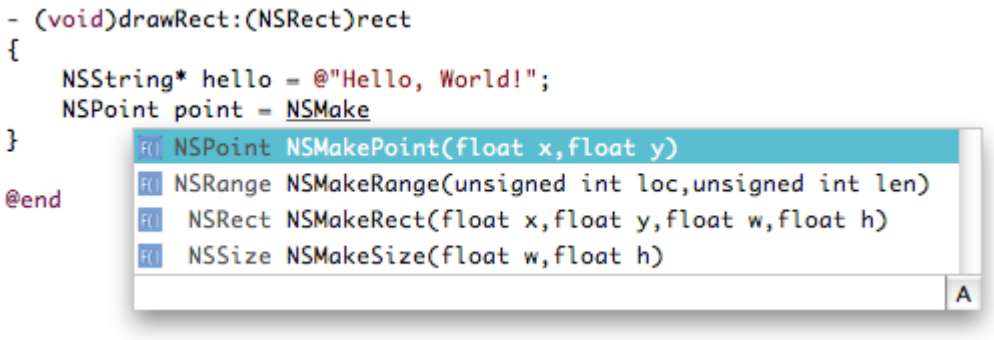
3. Insert these code lines as the body of the `drawRect:` method:

```
NSString* hello = @"Hello, World!";
NSPoint point = NSMake
```

4. Press Escape. The code completion pop-up menu appears.

The menu contains the symbols Xcode knows about whose name start with "NSMake," as shown in Figure 1-11.

Figure 1-11 The code completion pop-up menu



This is an example of **code completion**. When Xcode underlines the text you've typed, it has compiled a list of symbols whose name starts with the letters you've typed so far. You can continue typing or press Escape to view Xcode's suggestions.

5. Complete the `point` assignment expression:

    a. Choose `NSPoint NSMakePoint(float x, float y)` from the code-completion pop-up menu.

    b. Replace `<#float x#>` with `15`.

    c. Replace `<#float y#>` with `75`.

    d. Add a semicolon (`;`) to the end of the code line.

6. Finish entering the implementation of `drawRect:`, shown in Listing 1-2.

Listing 1-2 Implementation of the `drawRect:` method

```
- (void) drawRect:(NSRect) rect
{
    NSString* hello = @"Hello, World!";
    NSPoint point = NSMakePoint(15, 75);
    NSMutableDictionary* font_attributes = [NSMutableDictionary new];
    NSFont* font = [NSFont fontWithName:@"Futura-MediumItalic" size:42];
    [font_attributes setObject:font forKey:NSFontAttributeName];

    [hello drawAtPoint:point withAttributes:font_attributes];

    [font_attributes release];
}
```

7. Save your changes by choosing File > Save.

Building the Application

When you initiate a build, Xcode begins a process that starts with the source files in your project directory and ends with a complete product. Along the way, Xcode performs various tasks such as compiling source files, copying resource files, linking object files, and so forth.

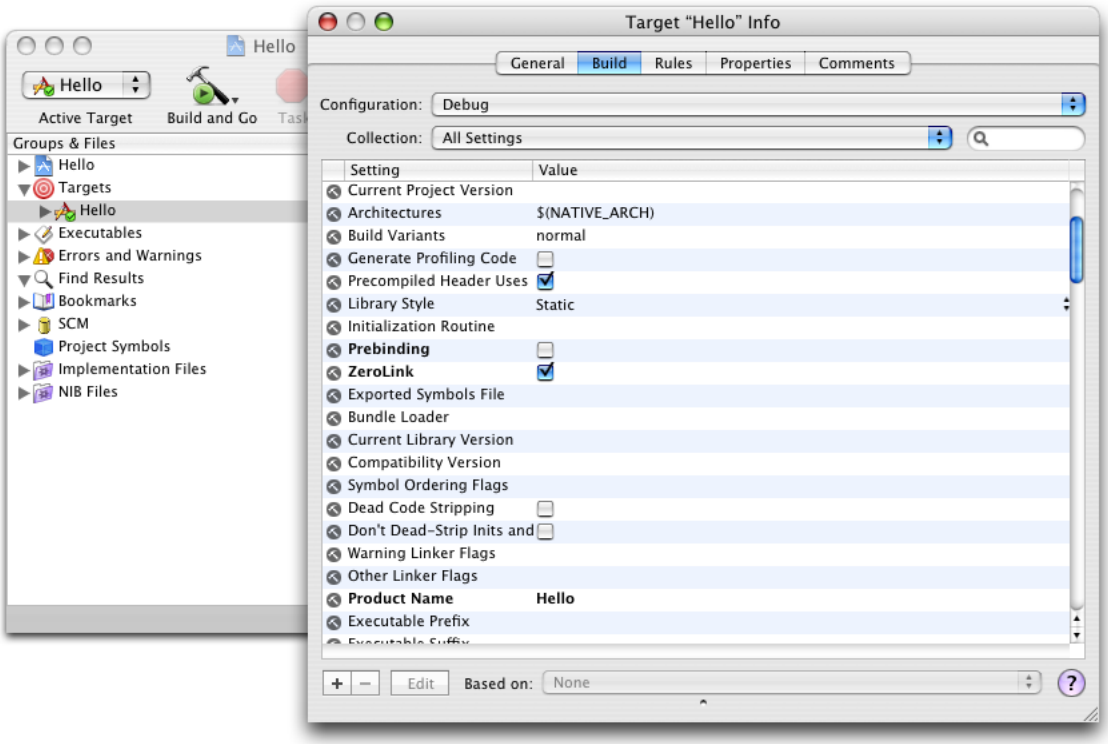
One of these tasks is to copy nib files, sounds, images, and other resources from the project to the appropriate places inside the application bundle. An **application bundle** is a directory that contains the application executable and the resources needed by that executable. This directory appears in the Finder as a single file that can be double-clicked to launch the application.

The build system in Xcode handles the complex process of creating a finished product based on build settings and rules specified in each of the project's targets. A target represents a single product, and Xcode supports multiple targets in a project.

Each target can specify one or more sets of build setting specifications, called **build configurations**. Targets are preconfigured with two build configurations, Debug and Release. The Debug build configuration specifies build settings that generate products containing information that is useful during development, such as debug symbols. The Release build configuration specifies build settings appropriate for products that are ready for regular use. For example, the ZeroLink build setting specifies whether the generated executable file is a self-contained application or a stub. A self-contained application file can be launched by itself but takes longer to build. An application stub requires Xcode to run, but its build time is minimized. By default, the Release build configuration turns ZeroLink off. The Debug build configuration turns ZeroLink on.

Figure 1-12 shows some of the build settings defined in the Debug build configuration of the Hello target. You can see a target's build configurations by selecting the target in the Targets group in the Groups & Files list in the project window, choosing File > Get Info, and clicking Build in the target Info window.

Figure 1-12 The Debug build configuration in the Hello target Info window



At the project level, you specify the name of the build configuration targets must use when you start a build. You choose the **active build configuration** from the Project > Set Active Build Configuration menu. As each target is built, it uses the build configuration whose name matches the project's active build configuration. Xcode then places the target's product in a folder inside the project's build folder named after the build configuration used to create the product.

For more information on ZeroLink and build configurations, see LinkingBuild Configurations, respectively, in *Xcode User Guide*.

To build the Hello application:

1. From the Project > Set Active Build Configuration menu, choose Release.
2. From the Build menu, choose Build.

When Xcode finishes building the product (and if it doesn't encounter any errors along the way), it displays "Build succeeded" in the status bar in the project window. See "Compile-Time Errors" (page 23) for details on handling build errors.

Running the Application

Now you have an application that's ready to run. Xcode places the application bundle in a location specified in your project's settings—in this case, in a folder named Release inside your project folder.

To verify that the application runs:

1. Choose Debug > Run Executable.
2. Verify that the application opens a window, displays "Hello,World!" and waits for user interaction.

Figure 1-13 Main window for the Hello application

3. Close the window by clicking the window's close button or typing Command-W.
4. Type Command-Q to quit.

You may also run the application from the Finder by opening the Release folder in the project's build directory, and double-clicking Hello.

Compile-Time Errors

Projects are rarely flawless from the start. By introducing a mistake into the source code for the Hello project, you can discover the error-checking features in Xcode.

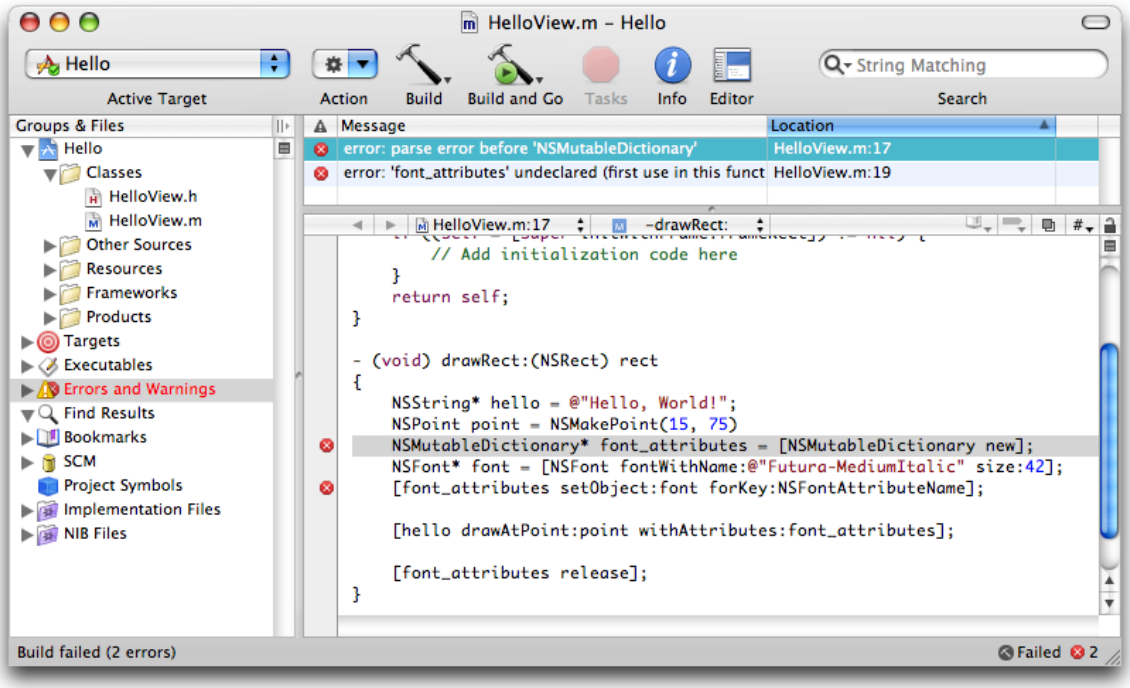
To see how error checking works:

1. Open `HelloView.m` in either the project window or a separate editor window.
2. Remove the semicolon from the `point` definition code line, creating a syntax error.
3. Choose File > Save to save the modified file.

4. Choose Build > Build. As expected, this time the build fails.

5. To view the error and warning messages in the project window, select Errors and Warnings in the Groups & Files list, as shown in Figure 1-14. The messages are displayed in the detail view and in the gutter of the editor window.

Figure 1-14 Error and warning messages



6. Fix the error in the source file, save, and build again. Notice that the error messages from the previous build have been cleared.

Runtime Debugging

Xcode provides a graphical user interface for GDB, the GNU source-level debugger. Debugging is an advanced programming topic that's beyond the scope of this tutorial, but it's useful to try out the debug command to see how it works. Before you can debug the Hello application, you need to set Debug as the active build configuration. Choose Project > Set Active Build Configuration > Debug.

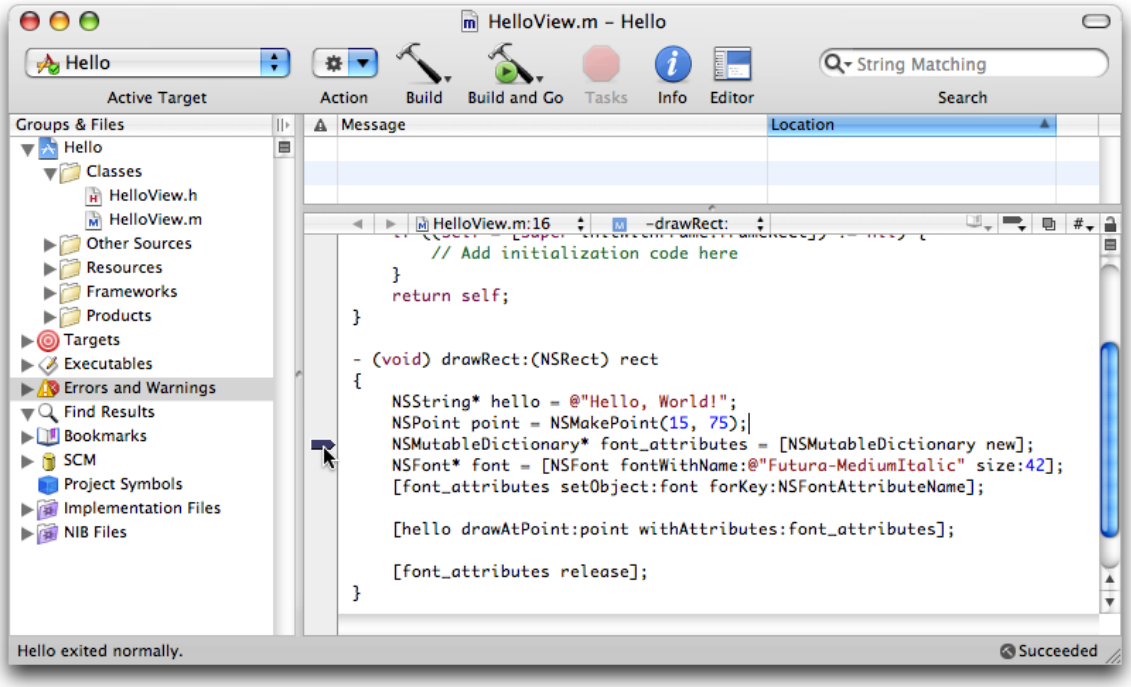
To set a breakpoint and step through a block of code:

1. Open the `HelloView.m` file. As before, you can open the file inside the project window or in a separate editor window.

2. Find the line that defines the `font_attributes` variable.

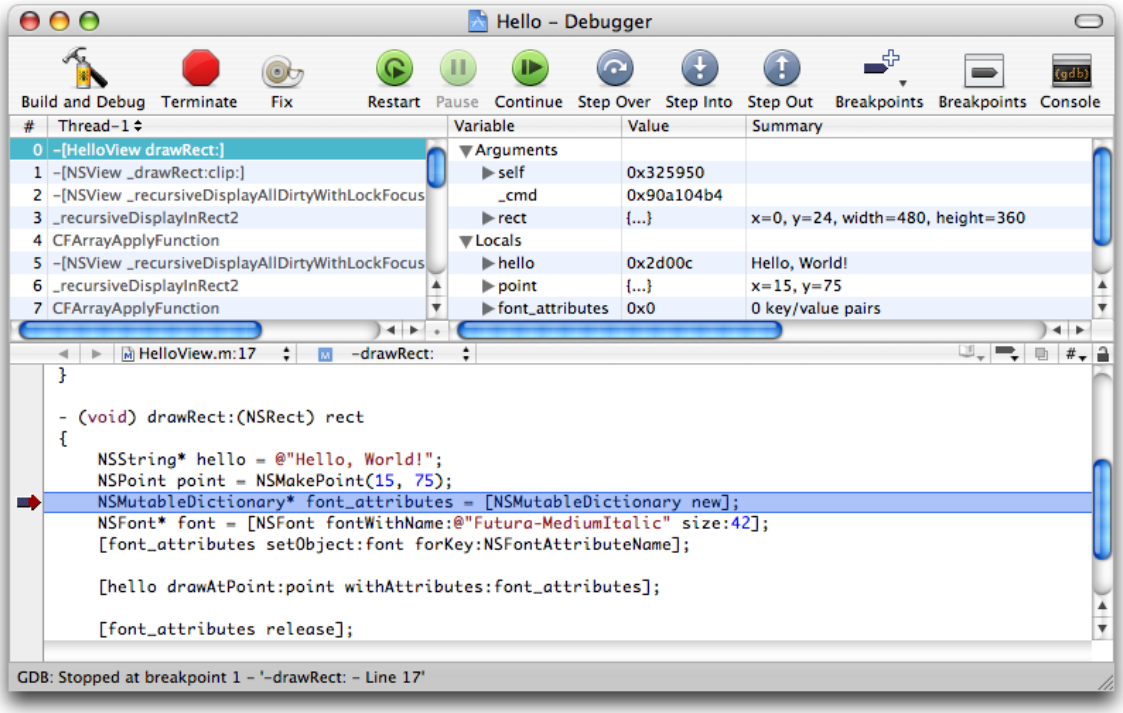
3. Set a breakpoint by clicking in the column to the left of the code line, as shown in Figure 1-15.

Figure 1-15 Setting a breakpoint



4. Choose Debug > Debug Executable. Xcode opens the debugger window and runs the application in debug mode, pausing at the breakpoint you set (see Figure 1-16).

Figure 1-16 The Debug window



5. Using the Debug > Step Over command, begin stepping through the code. As each line of code executes, you can examine the program's state. The value of a variable is sometimes drawn in red to indicate that the value was modified in the last step.

   Notice that the debugger pauses *before* executing the indicated line. After each pause, you can add additional breakpoints or use the Debug > Restart command to terminate the application and start a new debug session.

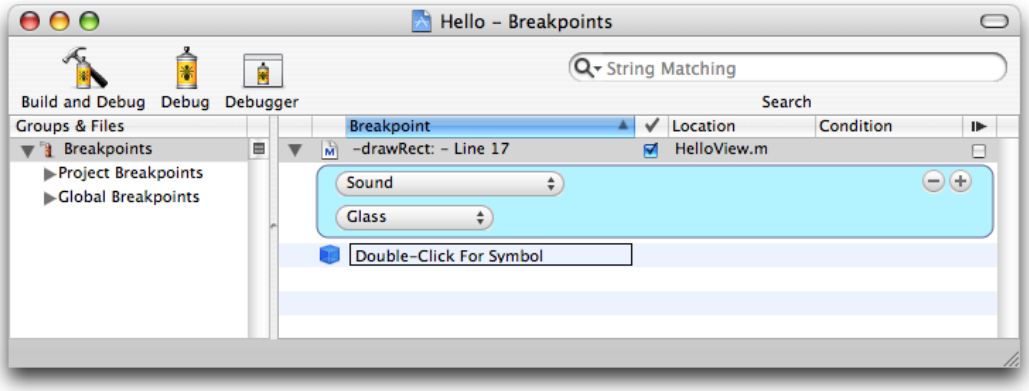
Normally, Xcode just stops the execution of the program when it encounters a breakpoint. By specifying **breakpoint actions**, you make Xcode perform other actions, such as logging output to the console.

To add a breakpoint action to the breakpoint you added earlier:

1. Choose Debug > Breakpoints. Xcode displays the Breakpoints window.
2. In the detail view in the Breakpoints window, click the triangle to the left of the breakpoint you added earlier.
3. Click the "add" (+) button that appears below the breakpoint.
4. From the first pop-up menu that appears below the breakpoint, choose Sound.
5. From the second pop-up menu, choose your preferred breakpoint sound.

Figure 1-17 shows a breakpoint action that plays the Glass sound when the breakpoint is reached.

Figure 1-17 A breakpoint action



Now, Xcode plays a sound—in addition to stopping the program—when execution reaches the breakpoint.

For more information on breakpoints, see Controlling Execution of Your Code in *Xcode User Guide*.

Summary

This tutorial provided a brief introduction to the Xcode application. It showed how to use Xcode to manage projects, build products, enter source code using code completion, find and correct build errors, and debug an application using breakpoints and breakpoint actions.

Finding Technical Information

Finding technical information about an unfamiliar technology or API is an important and often time-consuming activity for software developers. Xcode includes an array of features to help you quickly find technical information as you work on your project.

Xcode provides two main ways for getting you to the information you seek: focused API search and text-based search.

- **API Search** is the fastest way to get to reference documentation from an editor window or in the Documentation window. "Searching For API Documentation" (page 29) describes this method.
- **Full-Text Search** provides a way to find conceptual and reference documentation on a particular subject. With this method you can look through the set of documents that deal with the area in which you're interested. "Searching For All Relevant Documentation" (page 31) delves into this search method.

Apple publishes an extensive library of Mac OS X technical information, called **ADC Reference Library**. This library is included in the Xcode Tools package.

The documentation window in Xcode provides a consistent, intuitive user interface and a wide range of search options for viewing the HTML-based documentation in the ADC Reference Library. It's worth noting that Xcode uses the same HTML rendering engine used in Safari, Apple's web browser.

This chapter shows you can use the Xcode features that allow you to find technical information on a subject or API symbol. (For in-depth details, see Finding Information in a Project in *Xcode User Guide*.) This chapter also explains how to view the header files in Mac OS X frameworks and how to update the ADC Reference Library on your computer.

Searching For API Documentation

Xcode makes it easy to find the API documentation for any Apple-defined symbol—including functions, classes, methods, data types, and constants.

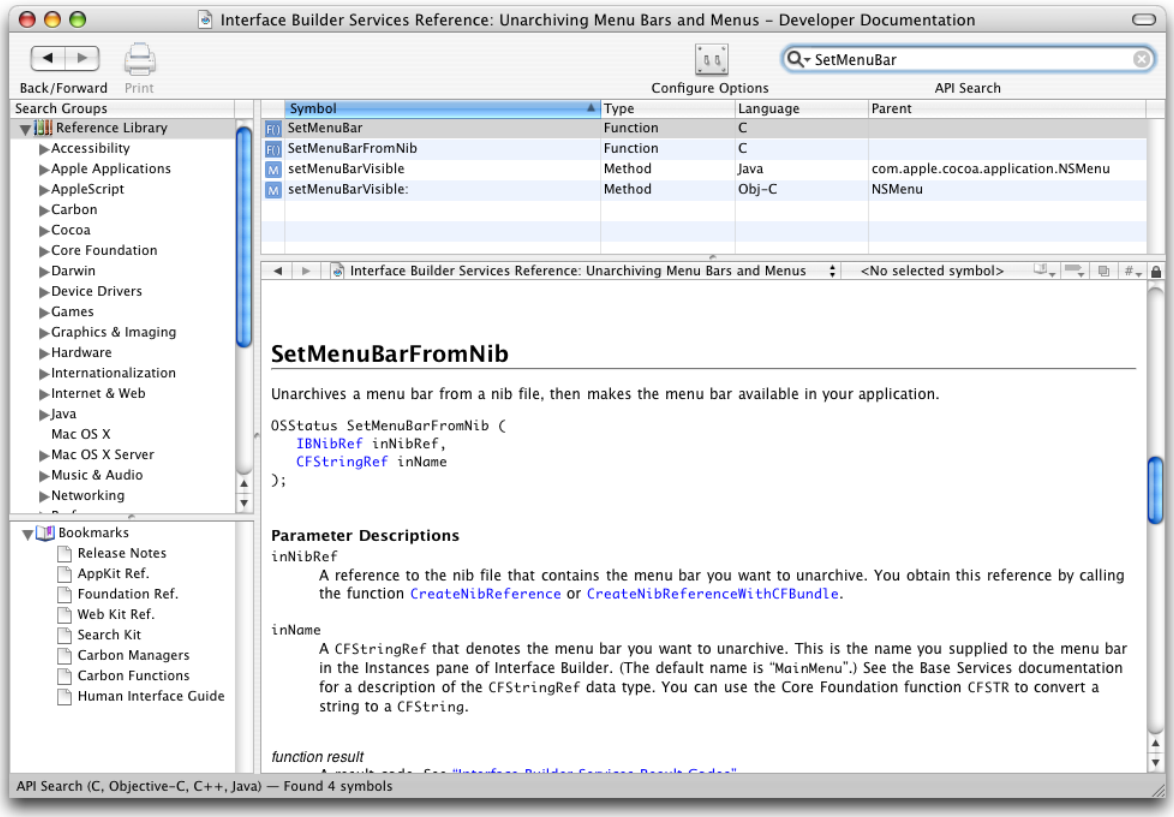
Searching From the Documentation Window

The search field in the documentation window allows you to display a list of all known symbols that match a particular search string. As you type in the search string, Xcode dynamically builds and displays the list.

To find API documentation using a search string:

1. Launch Xcode and open the documentation window.
2. In the Search Groups list, select Reference Library or Carbon to define the scope of the search.
3. From the pull-down menu in the search field, choose API Search.
4. Click in the search field and begin typing the name of the function `SetMenuBarFromNib`. Notice that each time you type a character, the list of matching symbols is reduced in size.
5. Stop typing when the list of matching symbols is reduced to just a few entries.
6. Select the entry for `SetMenuBarFromNib`. As shown in Figure 2-1, Xcode finds and displays the API documentation for this function in the content pane.

Figure 2-1 Using the API-reference search option in Xcode



Searching From a Source Editor Window

As a convenience, Xcode can also locate the API documentation for a symbol directly from a source editor window.

1. Open the Hello project window.
2. Open the source file `main.c` in an Xcode editor window.
3. Scroll down and find the call to the `ShowWindow` function.
4. Option–double-click this function—that is, hold down the Option key and double-click the name of the function. Xcode opens the documentation window and displays reference information about this function.

Searching For All Relevant Documentation

The Full-Text Search mode in the documentation window lets you search the reference library for documents relevant to a particular search string.

To search all technical documentation:

1. Launch Xcode and choose Help > Documentation. Xcode opens the window and displays the main documentation page.

   Note: After the first time you use the documentation window, Xcode remembers the last page you were viewing the next time you bring up the documentation window.

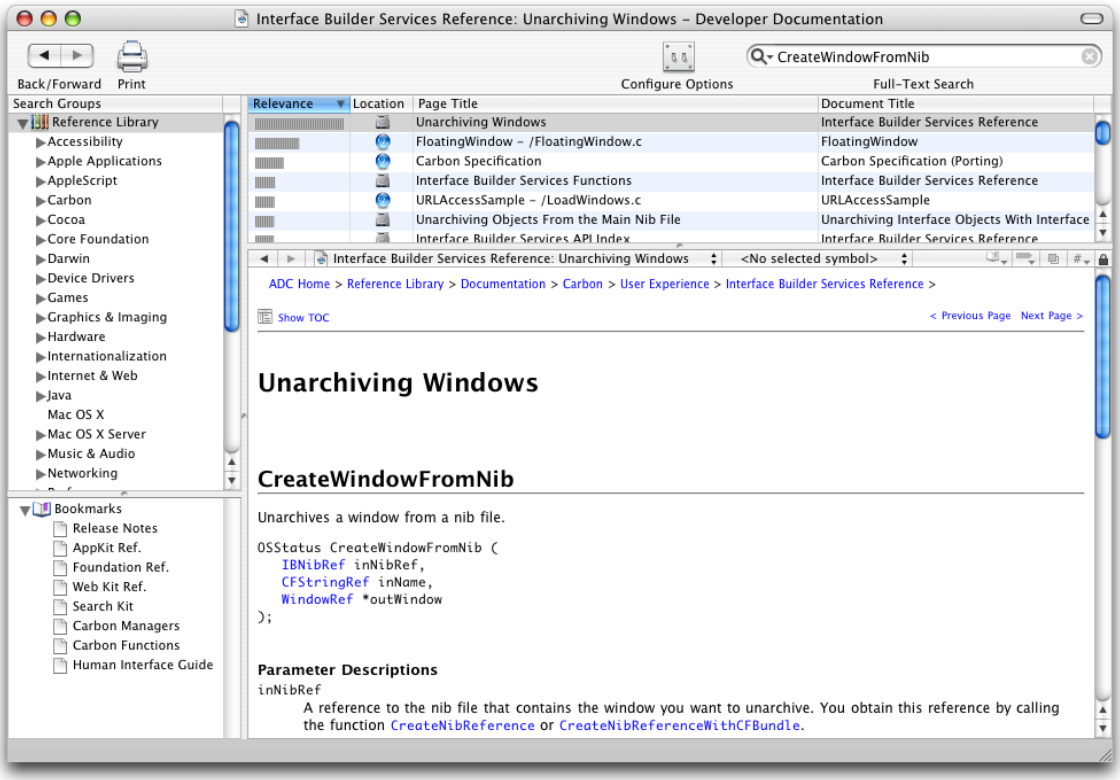
2. Make sure the Reference Library group in the Search Groups list is selected.

   This selection tells Xcode to search the entire library. You can narrow the search scope by selecting specific categories in the list.

3. From the pull-down menu in the search field, choose Full-Text Search.
4. Type the word `CreateWindowFromNib` and press Return. Xcode finds all documents that contain this word and lists them in order of relevance.

5. Select one of the items listed in the search results. Xcode displays this page of documentation in the viewing pane, as shown in Figure 2-2.

Figure 2-2 Using the full-text search option in Xcode



Note: Reference Library items whose location icon is a blue globe instead of a hard disk are not located in your computer. When you click them, Xcode sends the corresponding URL to your web browser, which displays the webpage containing the item.

Following Links

The document in Figure 2-2 (page 32) contains a number of hypertext links to other pages in the library. By convention, links are displayed as blue text.

There are several ways to use these links:

1. Click any link in the viewing pane of the documentation window. Xcode displays the linked page in the same viewing pane. To return to the previous page, click the left arrow button above the viewing pane.

2. Command-click the same link. Now Xcode displays the linked page in a new Xcode window.

3. Option-click the link. This time, Xcode sends the page's URL to your web browser.

Finding Information in Headers

As you work on an Xcode project, sometimes it's useful to study the declarations and comments in a Mac OS X framework header. Typically, you already know the name of the header or the name of one of its symbols.

Finding Framework Headers

If you know the name of a header but not its location, you can use Xcode to locate the header in a framework included in your project.

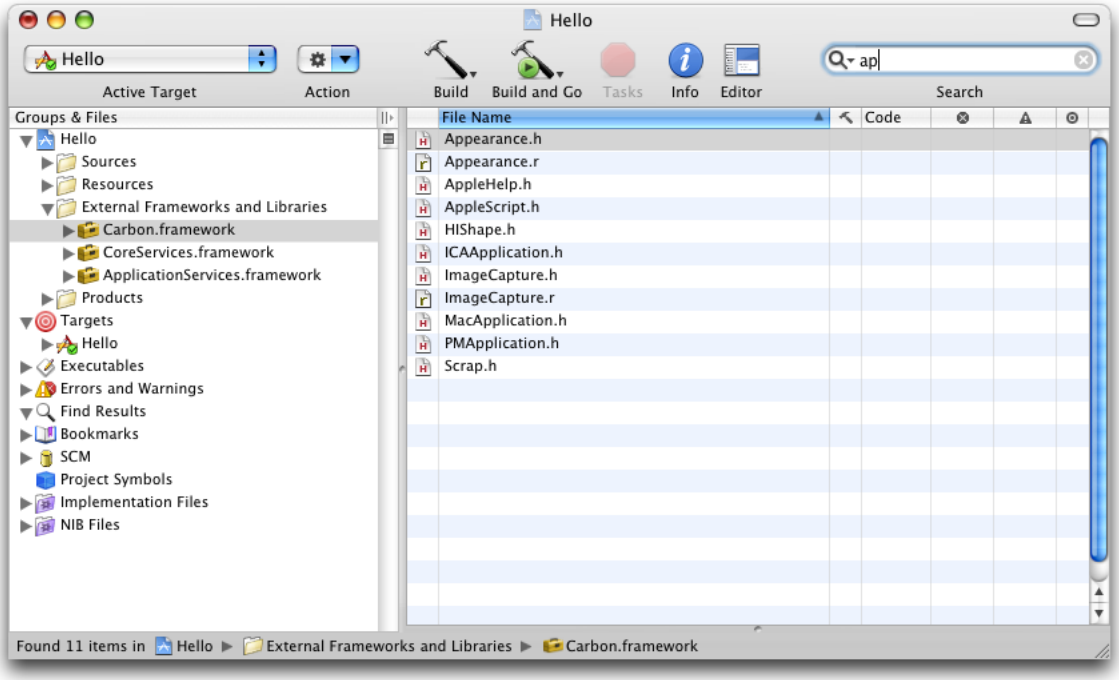
For example, to locate a header in the Carbon framework:

1. In the project window, use the disclosure triangles to open the project group. The groups inside are called **source groups**. Source groups contain references to actual files somewhere on your hard disk.

2. Open up the source group named External Frameworks and Libraries, and select `Carbon.framework`. Xcode lists the header files contained in this framework in the detail view.

3. Click in the project window search field and slowly type a few letters—for example, “ap”. Notice how Xcode filters the list of header files dynamically as you type each letter, as shown in Figure 2-3.

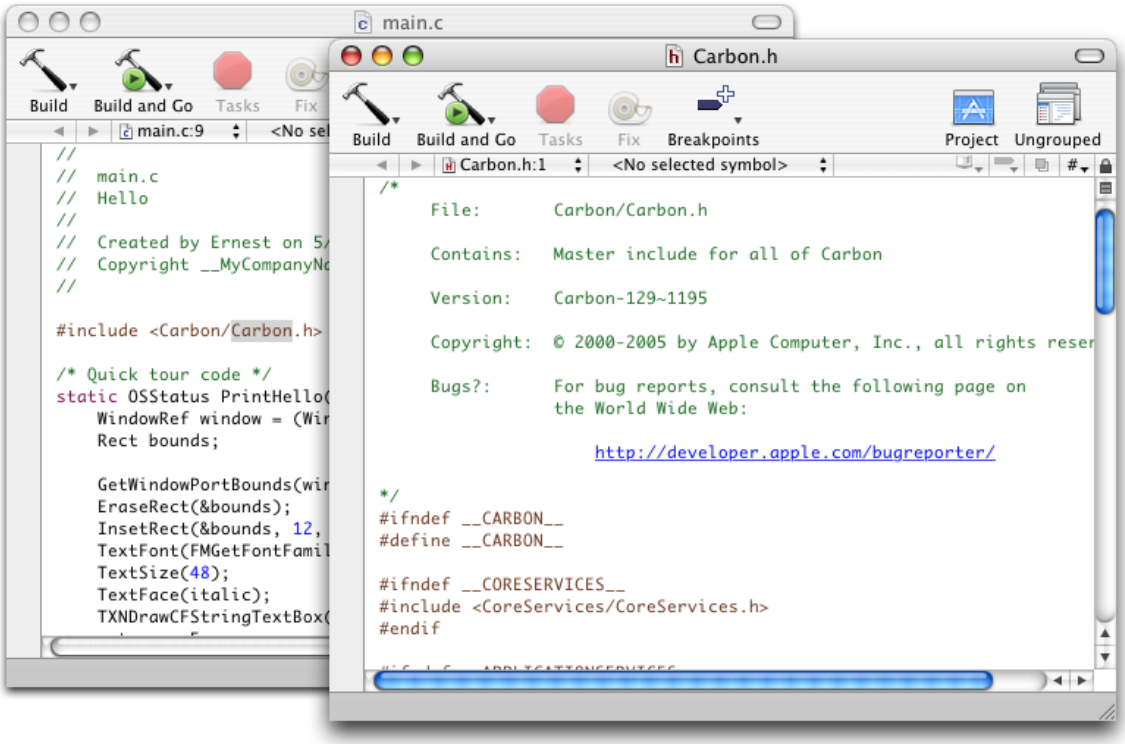
Figure 2-3 Searching for headers in a framework group



Another way to find headers is to use Xcode’s Open Quickly command. When the insertion point is on the name of a header in a source editor, you can press Shift–Command-D to bring up a source editor window with the corresponding header file. Xcode looks for the header file in the standard header locations.

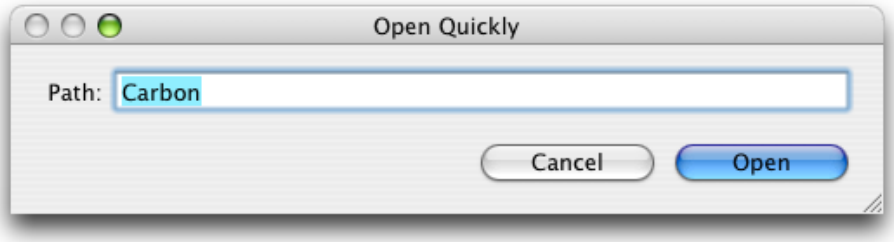
Figure 2-4 shows Open Quickly in action.

Figure 2-4 Using the Open Quickly command



If you're not editing a file or the insertion point is not over the name of a header, Xcode displays the Open Quickly dialog, as shown in Figure 2-5. You enter the name of header in the Path text field and click Open.

Figure 2-5 The Open Quickly dialog



Finding Symbol Declarations

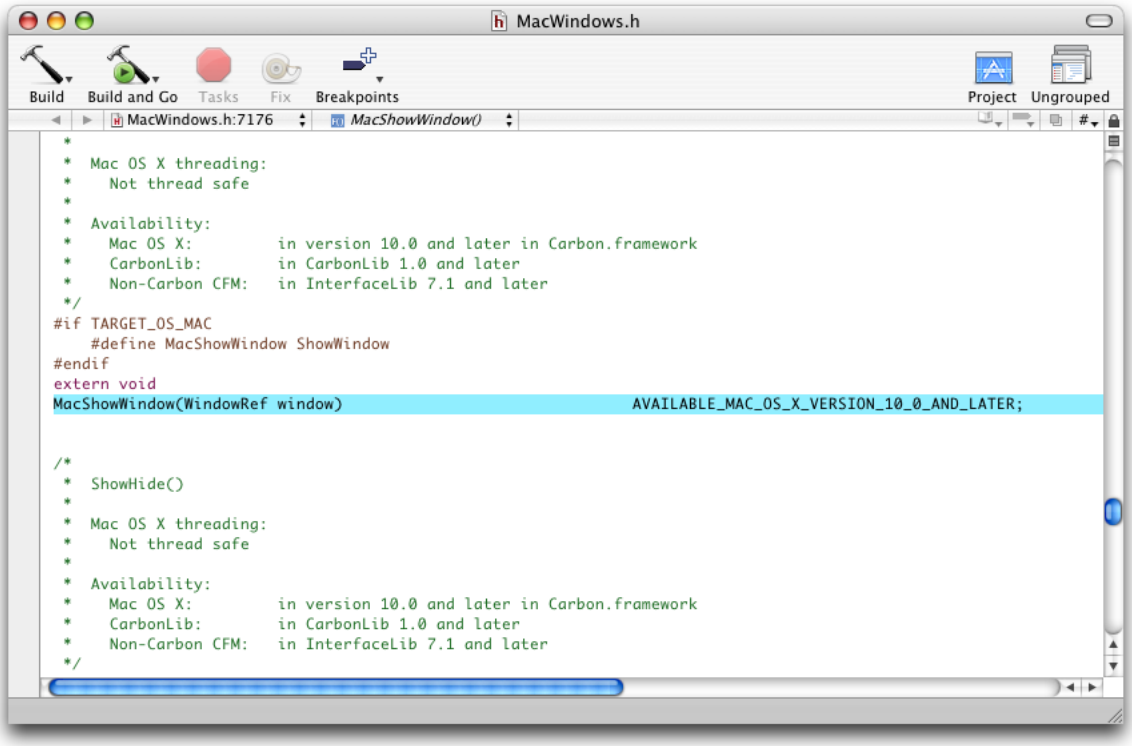
In "Searching From a Source Editor Window" (page 31), you learned how to find the API documentation for any Apple-defined symbol in your source code. Using a similar technique, you can also find the declaration for a symbol.

For example, to find the declaration for `ShowWindow`:

1. From the Hello project window, open the source file `main.c`.

2. Find the line where the function `ShowWindow` is called.
3. Command–double-click this function—that is, hold down the Command key and double-click the name of the function. Xcode opens the header file `MacWindows.h` and displays the declaration for `ShowWindow`, as shown in Figure 2-6.

Figure 2-6 Header search results

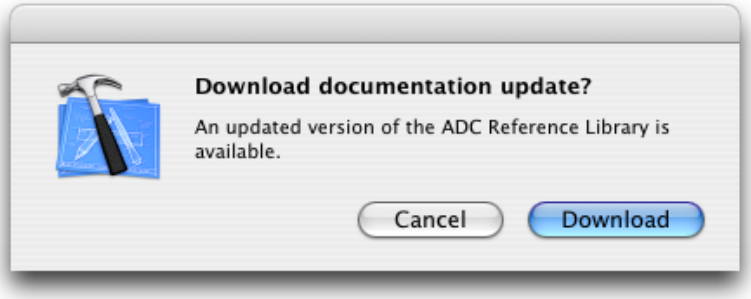


Downloading ADC Reference Library Updates

The ADC Reference Library is a repository of documents, code examples, technical notes, and release notes about development using Apple technologies. You can access this library at http://developer.apple.com as well as in Xcode. However, the library gets updated more regularly than new Xcode versions are released.

To keep your version of the ADC Reference Library up to date, Xcode periodically checks for updates to the library's content. When an update is available, Xcode displays the dialog shown in Figure 2-7. You can also check for updates any time by clicking Check Now in the Documentation pane in the Xcode Preferences window.

Figure 2-7 Reference Library update dialog



When you click Download in the documentation-update dialog, you're taken to the Developer Connection sign-in webpage in your web browser. If you're an ADC member, sign in and follow the download instructions. Otherwise, you must become an ADC member before you can download ADC Reference Library updates. ADC membership is available for free.

Summary

This tutorial provided an overview of the Xcode features that allow you to find technical information contained in the ADC Reference Library. It showed how to search for conceptual documentation as well as API documentation from the documentation window and from a source editor window. This tutorial also showed how to locate the header files that declare a symbol used in a source file. Finally, this tutorial described the process of updating your local ADC Reference Library content when a new version of the library is published by Apple.

Designing a User Interface

Interface Builder is Apple's graphical editor for designing user interfaces. Interface Builder makes it easy to design an interface that:

- Adheres to the Aqua layout guidelines
- Takes advantage of the newest technologies in Carbon and Cocoa

This tutorial shows how to use Interface Builder to design the user interface for a Carbon application called Converter. After you're finished, you can incorporate your design into a working application.

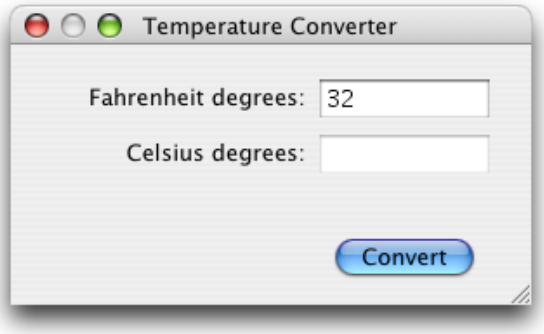
Note: If you're planning to use Cocoa, you should read *Cocoa Application Tutorial*.

Creating the Converter Interface

Converter is a simple application that converts temperatures from Fahrenheit degrees to Celsius degrees.

The user interface for Converter is displayed in a single window. Figure 3-1 shows what the application's user interface looks like after you've completed this tutorial.

Figure 3-1 The Converter user interface



To convert a temperature, you tab into the “Fahrenheit degrees” text field and enter a value. When you press Return or click the Convert button, Converter displays the result in the “Celsius degrees” text field, which is read only. To quit the application, you can either click the close button or type Command-Q.

Getting Started

When you use Interface Builder to design a user interface, your work is saved in a file with the `.nib` extension. At runtime, an application uses the contents of this file to construct and display its user interface, typically inside a window.

Xcode includes a template project that you can use to create an Xcode project for the Converter application. This template includes a nib file that defines a default user interface.

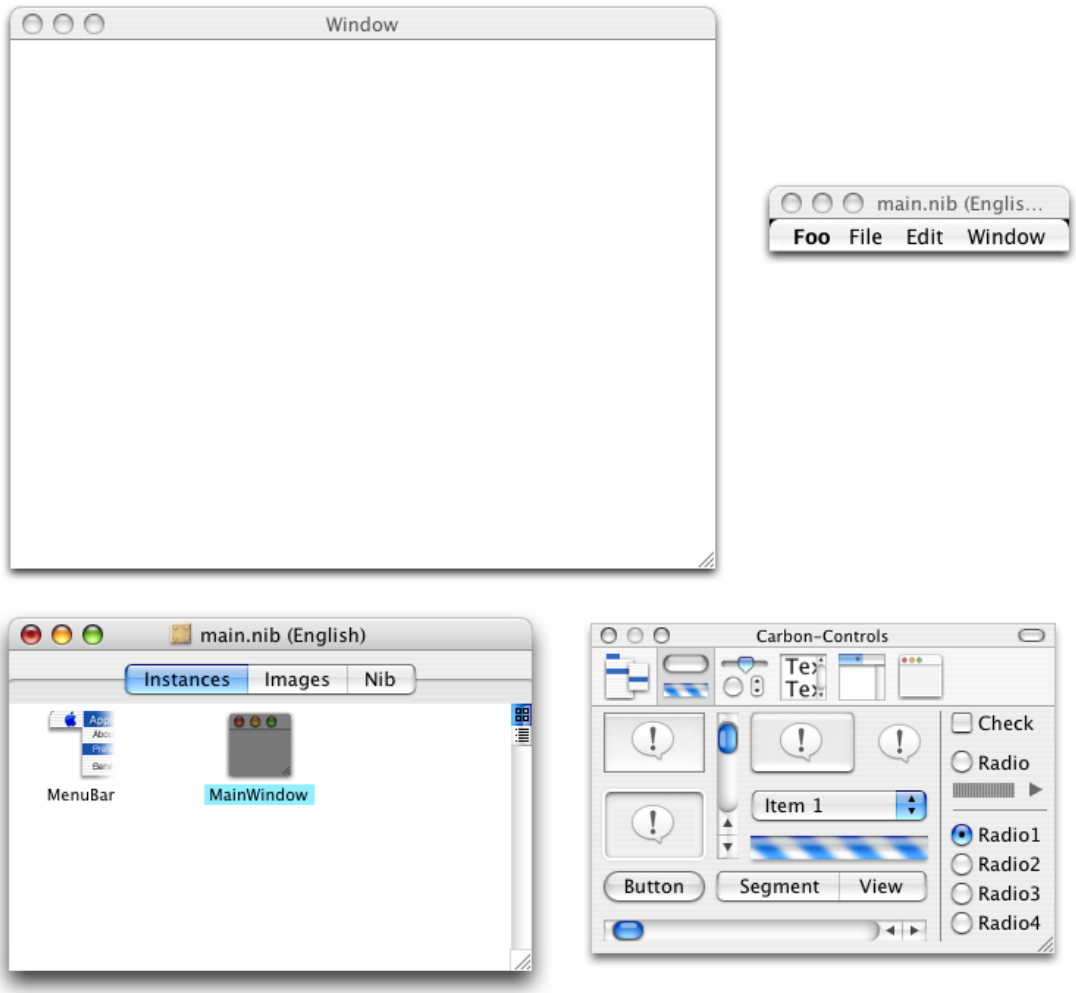
To create the Converter project and open its nib file:

1. Use the procedure described in “Creating an Xcode Project” (page 9) to create a project for a Carbon application. Give the project the name Converter.

2. In the Converter project window, locate and double-click the `main.nib` file.

As shown in Figure 3-2, Interface Builder launches and presents a set of windows. The `main.nib` window with the Instances, Images, and Nib tabs represents the nib file. The MenuBar and MainWindow items in this window represent the application’s menu bar and main window. The window titled “Window” corresponds to MainWindow. In it you place the controls that define the main window’s user interface. The window with a menu bar corresponds to MenuBar. You use this window to customize the application’s main menu. The palette window contains several controls, including menu items, which you drag into the empty window and the menu bar window to create the application’s user interface.

Figure 3-2 Editing windows in Interface Builder



Setting the Window's Attributes

To set the attributes for the Converter window:

1. Click inside the main window to select it.
2. Choose Tools > Show Inspector (Shift–Command-I).

   This command opens an Interface Builder window called **inspector** (see Figure 3-3). The inspector window lets you view and change the attributes of any object in a nib file. In Interface Builder, there's only one inspector, and it displays the attributes for the currently selected object.
3. From the pop-up menu at the top of the inspector, choose Attributes. Interface Builder displays the Attributes pane of the Converter (MainWindow) inspector.
4. Change the Converter window's title to "Temperature Converter", and press Return to make the change appear in the window.

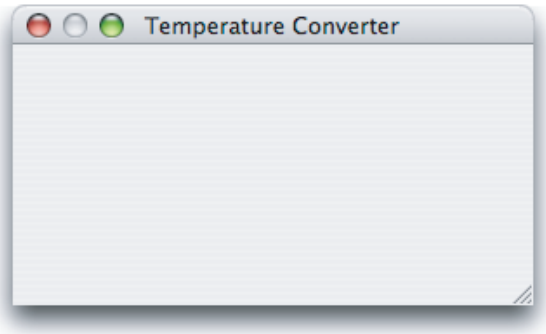
5. Make sure Window Class is set to Document.
6. From the Theme Brush menu, choose Movable Modal. This theme gives the interior part of the window an Aqua look.
7. Deselect the Resize and Zoom options—there's no need to support window zooming and resizing controls here.
8. Deselect Compositing since this tutorial doesn't use HIView functionality. At this point, the window attributes should look like those in Figure 3-3.

Figure 3-3 Attributes of the Converter window

9. From the inspector pop-up menu, choose Size.
10. In the Content Rectangle group, set the window's width to 300 and its height to 150. There are two ways to resize a window in Interface Builder—by direct manipulation, or by specifying the exact dimensions as you have done here. These dimensions are not exactly correct, but they're a good first approximation. You set the exact dimensions later, after adding the window's user interface elements.

The main window should now resemble the window in Figure 3-4.

Figure 3-4 Converter window before adding controls



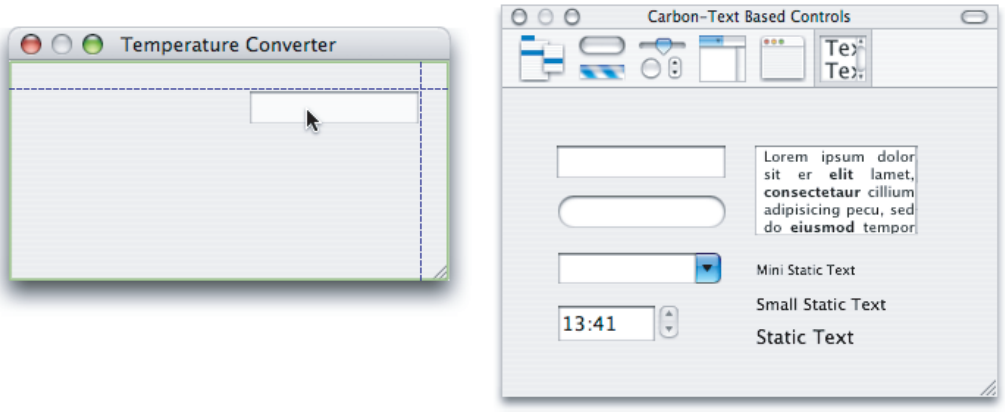
The inspector is used throughout this tutorial, so it makes sense to leave it open.

Adding the Fahrenheit Control

To add the Fahrenheit control:

1. In the Interface Builder palette window, select Carbon Text-Based Controls. You can see the names of the control groups by placing the pointer over them in turn.
2. Drag the editable text field (EditText) from the palette to the main window, as shown in Figure 3-5.

Figure 3-5 Adding an editable text field



Notice that Interface Builder helps you place objects according to the Aqua guidelines by displaying pop-up guides when an object is dragged close to the proper distance from neighboring objects or the edge of the window.

3. In the Converter window, select the editable text field, and choose Attributes from the inspector pop-up menu.
4. In the Title field, type `32` and press Return to apply the value. This serves as the default Fahrenheit temperature.

5. Ensure the text field is still selected. From the inspector pop-up menu, choose Control.
6. All Carbon controls have a 4-byte application signature and a unique integer identifier. For the signature of this control, enter `DEMO`. For the ID, enter `128`.
7. From the inspector pop-up menu, choose Size. Set the width to 92.
8. Back in the main window, adjust the position of the text field as needed.

Adding the Celsius Control

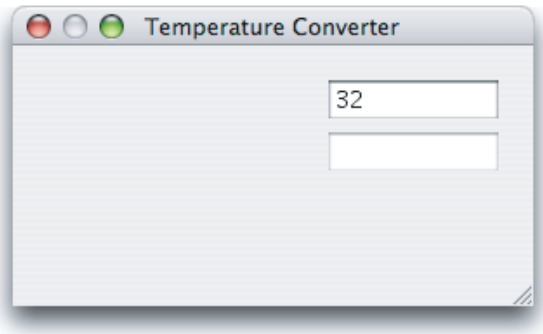
Converter needs a second control to display the Celsius temperature, the same size as the first.

To duplicate the Fahrenheit control:

1. In the Converter window, select the editable text field.
2. Choose Edit > Duplicate (Command-D). Interface Builder adds a new text field, slightly offset from the original.
3. Position the new text field below the original. Allow the guides to assist you by snapping the second field into place.
4. With the new field selected, go to the inspector and choose Attributes from the pop-up menu. Remove the title—a default Celsius temperature is not needed.
5. From the inspector menu, choose Control. Set ID to `129`, and deselect the Enabled option to make this a read-only control.

The Converter window should now resemble the window in Figure 3-6.

Figure 3-6 Converter window after adding temperature controls



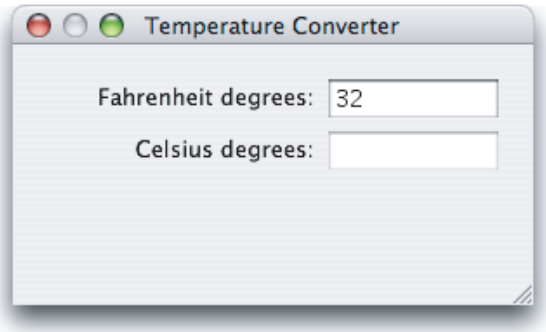
Adding Static Text Labels

Controls without labels would be confusing, so you're going to add these labels now. Carbon uses static text fields for this purpose.

1. Drag a static text field from the Carbon Text-Based Controls palette to the Converter window. For now, you can place it in the empty area below the temperature controls.
2. With the static text field selected, go to the inspector and choose Attributes from the pop-up menu. Set the title to "Fahrenheit degrees:" and set the Justification attribute to Right.
3. Now choose Size from the pop-up menu, and set the width of this field to 150.
4. Back in the Converter window, position the static text field to the left of the first editable text field. Allow the guides to assist you by snapping it into place.
5. Make a duplicate static text field. Give it the title "Celsius degrees:" and position it to the left of the second editable text field.

The main window should now resemble the window in Figure 3-7.

Figure 3-7 Converter window after adding static text



Adding the Convert Button

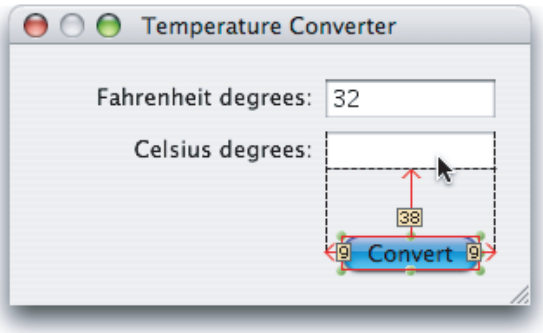
The Converter application is designed to convert a Fahrenheit temperature when the user presses Return or clicks the Convert button.

To add the Convert button:

1. Drag the Button element from the Carbon Controls palette to the bottom-right portion of the Converter window.
2. With the button selected, choose Attributes from the inspector menu and set the button's title to "Convert".
3. Set the Button Type attribute to Default. In a user interface with a default button, pressing the return key simulates a button click.
4. From the inspector menu, choose Control. Set the signature to `DEMO`, the ID to `130`, and the command code to `Conv`. When the user clicks the Convert button or presses the return key, a Carbon event with this command code is sent to the application.
5. In the inspector, choose Size from the main pop-up menu and set the button width to 80.

6. Return to the main window, and align the button under the temperature controls. Drag the button downward until the Aqua guide appears and release the mouse. With the button still selected, hold down the Option key. If you move the cursor around, Interface Builder shows the distance from the button to the object that you've indicated with the cursor. With the Option key still down and the cursor over the Celsius text field, use the arrow keys to nudge the button horizontally to the exact center of the text fields (see Figure 3-8).

Figure 3-8 Measuring distances in Interface Builder



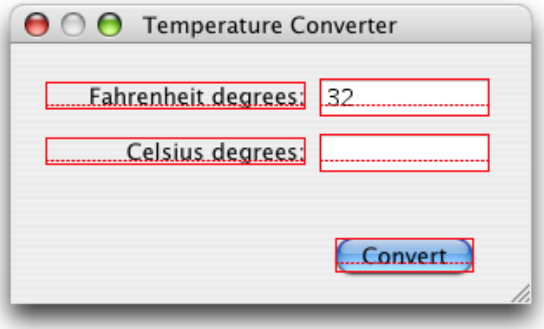
Aqua Layout and Object Alignment

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

Because interface objects have shadows, they don't visually align correctly if you align the edges of the frames (as in Mac OS 9). For example, the Aqua 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.

To view the layout rectangles in a window, choose Show Layout Rectangles from the Layout menu (Command-L). Figure 3-9 shows the layout rectangles in the Converter window.

Figure 3-9 Layout rectangles in Interface Builder



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

- Dragging objects with the mouse in conjunction with the Aqua guides
- Pressing the arrow keys (with the grid off, the selected objects move 1 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 of the inspector
- Using horizontal or vertical guides

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

Finishing the Window Layout

The Converter user interface is almost complete. The finishing touch is to align the controls with respect to the top-left corner of the window and then resize the window. You keep using the automated Aqua guides, along with a few layout commands.

1. Select all the controls in the main window, either by dragging a rectangle around them or by typing Command-A.
2. Choose Layout > Group (Command-G) to group them together. Interface Builder uses a stippled rectangle to indicate the object group.
3. Drag the group (using any control as a handle) to the top-left corner of the window. Use the Aqua guides to help you find the proper position relative to the edges of the window.
4. Choose Layout > Ungroup (Shift-Command-G).
5. If necessary, move the button into position again under the temperature controls.
6. Now resize the window, using the guides to give you the proper distance from the text fields (on the right) and the Convert button (on the bottom).

At this point, the main window should look like Figure 3-1 (page 39).

Testing the Interface

The Converter interface is now complete. Interface Builder provides a simulator that lets you test your interface without having to write one line of code.

To test your interface:

1. Choose File > Save to save your work.
2. Choose File > Test Interface to run the simulator.
3. Try tabbing into the Fahrenheit field, entering a value, and cutting and pasting the value.
4. Make sure the Convert button is pulsating, indicating that it's the window's default button.
5. Note that the screen position of the window in Interface Builder is used as the initial position for the window when Converter is launched.
6. When finished, choose Carbon Simulator > Quit (Command-Q).

Implementing the Converter Application

Appendix A, "Converter Source Code" (page 55), contains the source code to implement the Converter application. The source code is also available in this document's companion-file package. Because this tutorial is not designed to teach you Carbon programming, it doesn't provide an explanation of the code.

To add this source code to the project:

1. In the Finder, copy `main.c` to `main.c.backup`.
2. In Xcode, open `main.c` in an editor window and replace its contents of with the source code in Listing A-1 (page 55).
3. Save the file.
4. Build and run the application to make sure it works properly.

To make Converter a solid Mac OS X application, clearly there's more work to be done. If you're interested in Carbon programming, here are a few ideas to improve and extend this application:

- Edit the main menu bar using Interface Builder. The default Carbon application menu contains items that don't apply here and should be removed.
- Add range checking for the Fahrenheit temperature control. Absolute zero is around –459.6 degrees Fahrenheit.
- Add conversion from Celsius to Fahrenheit.

Summary

This tutorial showed how to implement the user interface for a simple Carbon application using Interface Builder. First, it described how to lay out a window's user interface elements. Second, it showed how to configure the user interface elements to help users work with the application. Finally, this tutorial demonstrated how to test an application's user interface before implementing its logic.

Using Fix and Continue

Fix and Continue makes it possible to modify an executable image at debug time. You can see the results of your modification without interrupting or restarting the debugging session.

This feature is especially useful when it takes a lot of time to reach a particular state of execution while debugging your application. Rather than recompiling your project and starting a new debugging session, you can make minor changes to your code, fix your executable image, and see the results of your changes immediately. For detailed information on Fix and Continue, see Modifying Your Code Using Fix and Continue in *Xcode User Guide*.

This tutorial uses an example project named Sketch, which implements a simple drawing program built using the Cocoa framework. You're going to change the way Sketch draws shapes by changing the source code and using the Fix command in Xcode to patch the running program.

You don't need to be familiar with Objective-C to work through this tutorial.

Configuring the Project

To configure the Sketch project to use the Fix command:

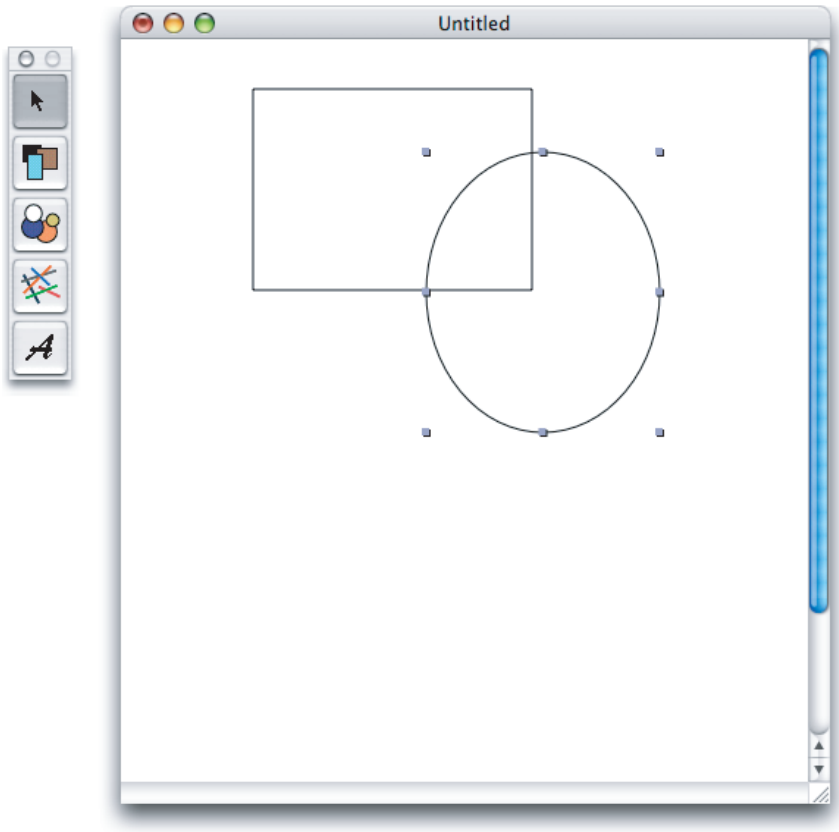
1. Make a copy of the Sketch project, which is located in `/Developer/Examples/AppKit`. To avoid confusion, place your copy somewhere inside your home directory.
2. In the Finder, drag your copy of the Sketch project folder onto the Xcode application, which is located in `/Developer/Applications`. Notice that Xcode finds and opens the project file inside this folder.
3. Make sure the active build configuration is Debug. From the Project menu, choose Set Active Build Configuration > Debug.
4. From the Build menu, choose Build (Command-B) to build the target.

Patching the Running Application

1. To start a debugging session, choose Debug > Debug Executable (Option–Command-Y).

2. Click in the Sketch drawing window, select a drawing tool (rectangle or circle), and draw a few objects.

Figure 4-1 The default drawing behavior of Sketch



3. Return to the project window and select the Sketch group.
4. Open the source file named `SKTGraphic.m`.
5. In the source editor, locate the `-(id)init` method. It looks like this:

```
- (id)init {
    self = [super init];
    if (self) {
        _document = nil;
        [self setBounds:NSMakeRect(0.0, 0.0, 1.0, 1.0)];
        [self setFillColor:[NSColor whiteColor]];
        [self setDrawsFill:NO];
        [self setStrokeColor:[NSColor blackColor]];
        [self setDrawsStroke:YES];
        [self setStrokeLineWidth:1.0];
        _origBounds = NSZeroRect;
        _gFlags.manipulatingBounds = NO;
    }
    return self;
}
```

6. Change some of the values.

   For example, change:

```
[self setFillColor:[NSColor whiteColor]];
[self setDrawsFill:NO];
[self setStrokeColor:[NSColor blackColor]];
[self setDrawsStroke:YES];
[self setStrokeLineWidth:1.0];
```

   to:

```
[self setFillColor:[NSColor yellowColor]];
[self setDrawsFill:YES];
[self setStrokeColor:[NSColor blackColor]];
[self setDrawsStroke:YES];
[self setStrokeLineWidth:8.0];
```

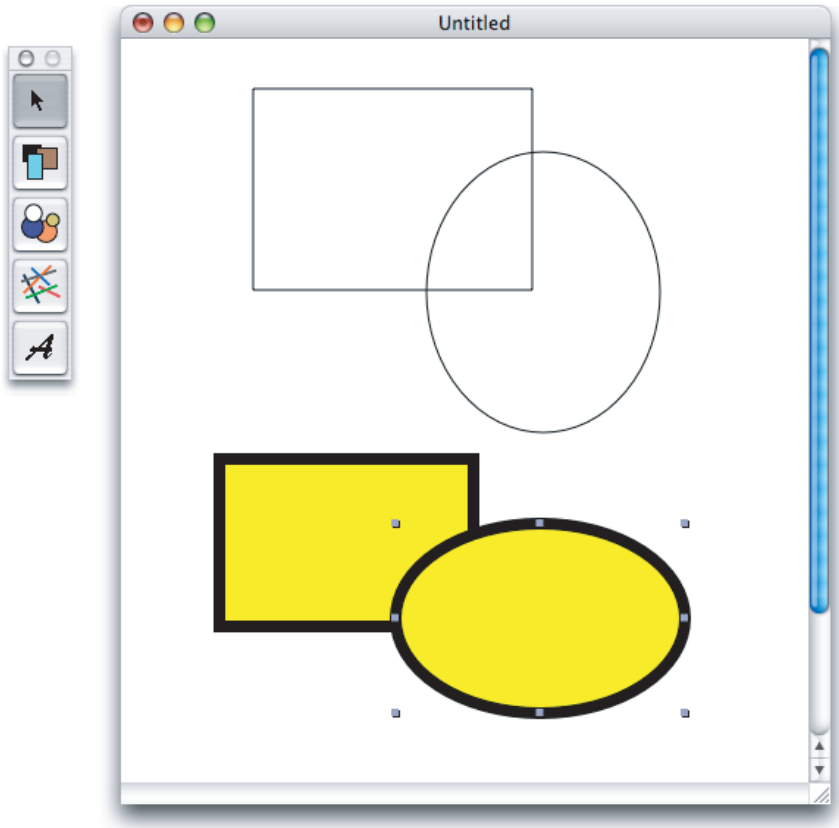
7. Save your changes.

8. Choose Debug > Fix to modify the running program.

9. Return to the Sketch drawing window and draw some more objects.

   You should see a dramatic change in the appearance of the objects you draw, as illustrated in Figure 4-2.

Figure 4-2 The new drawing behavior of Sketch



Summary

This tutorial showed how to use Fix and Continue to patch changes into a running application.

Converter Source Code

Listing A-1 contains the source code for the Converter application described in "Designing a User Interface" (page 39). This file is also available in this document's companion-file package.

Listing A-1 Converter source code

```
// File: main.c
#include <Carbon/Carbon.h>

static OSStatus Convert(WindowRef window)
{
    const ControlID kFahrenheit = { 'DEMO', 128 };
    const ControlID kCelsius = { 'DEMO', 129 };

    ControlRef fahrenheit, celsius;
    CFStringRef fahrenheit_text, celsius_text;
    double fahrenheit_value, celsius_value;
    OSStatus err = noErr;

    err = GetControlByID(window, &kFahrenheit, &fahrenheit);
    require_noerr_quiet(err, CantGetControl);

    err = GetControlByID(window, &kCelsius, &celsius);
    require_noerr_quiet(err, CantGetControl);

    err = GetControlData(fahrenheit, 0, kControlEditTextCFStringTag,
        sizeof(CFStringRef), &fahrenheit_text, NULL);
    require_noerr(err, CantGetControlData);

    fahrenheit_value = CFStringGetDoubleValue(fahrenheit_text);
    celsius_value = (fahrenheit_value - 32.0) * 5.0 / 9.0;

    celsius_text = CFStringCreateWithFormat(
        NULL, NULL, CFSTR("%g"), celsius_value);

    err = SetControlData(celsius, 0, kControlEditTextCFStringTag,
        sizeof(CFStringRef), &celsius_text);

    CFRelease (celsius_text);
    require_noerr (err, CantSetControlData);

    DrawOneControl(celsius);
```

```
    CantSetControlData:
    CantGetControlData:
    CantGetControl:
    return err;
}

static OSStatus MainWindowCommandHandler(
    EventHandlerCallRef nextHandler, EventRef event, void *userData)
{
    HICommand command;
    WindowRef window = (WindowRef) userData;
    OSStatus err = noErr;

    err = GetEventParameter(event, kEventParamDirectObject,
        typeHICommand, NULL, sizeof(HICommand), NULL, &command);
    require_noerr (err, CantGetParameter);

    switch (command.commandID)
    {
        case 'Conv':
            Convert (window);
            break;
        default:
            err = eventNotHandledErr;
            break;
    }

CantGetParameter:
    return err;
}


static OSStatus MainWindowEventHandler(
    EventHandlerCallRef nextHandler, EventRef event, void *userData)
{
    OSStatus err = noErr;

    switch (GetEventKind(event))
    {
        case kEventWindowClosed:
            QuitApplicationEventLoop();
            break;
        default:
            err = eventNotHandledErr;
            break;
    }

    return err;
}

static OSStatus CreateMainWindow()
{
    IBNibRef nibRef;
    WindowRef window;
    OSStatus err;

    const EventTypeSpec commands[] = {
```

```
        { kEventClassCommand, kEventCommandProcess }
    };

    const EventTypeSpec events[] = {
        { kEventClassWindow, kEventWindowClosed }
    };

    err = CreateNibReference(CFSTR("main"), &nibRef);
    require_noerr(err, CantGetNibRef);

    err = SetMenuBarFromNib(nibRef, CFSTR("MenuBar"));
    require_noerr(err, CantSetMenuBar);

    err = CreateWindowFromNib(nibRef, CFSTR("MainWindow"), &window);
    require_noerr(err, CantCreateWindow);

    DisposeNibReference(nibRef);

    err = InstallWindowEventHandler (window,
            NewEventHandlerUPP (MainWindowEventHandler),
            GetEventTypeCount(events), events,
            window, NULL);

    require_noerr(err, CantInstallHandler);

    err = InstallWindowEventHandler(window,
            NewEventHandlerUPP (MainWindowCommandHandler),
            GetEventTypeCount(commands), commands,
            window, NULL);

    require_noerr(err, CantInstallHandler);

    ShowWindow(window);

CantInstallHandler:
CantCreateWindow:
CantSetMenuBar:
CantGetNibRef:

    return err;
}

int main(int argc, char* argv[])
{
    OSStatus err = CreateMainWindow();
    RunApplicationEventLoop();
    return err;
}
```

APPENDIX A

Converter Source Code

Document Revision History

This table describes the changes to *Xcode Quick Tour Guide*.

Date	Notes
2006-11-07	Reorganized the "Finding Technical Information" chapter to emphasize API Search in the Xcode editor.
2006-07-24	Based Hello, World example on Cocoa instead of Carbon.
2006-01-10	Specified Xcode Tools version requirements.
	Updated the introduction and "Creating a Project" (page 9) to specify the development environment required to successfully complete the tasks described in this document.
2005-09-08	Updated introduction to clarify how to install Xcode Tools.
	Corrected screenshot in Figure 3-3 (page 42).
2005-06-06	Updated for Mac OS X v10.4. Changed title from "A Quick Tour of Xcode."
	Added information on code completion, build configurations, breakpoint actions, and Open Quickly in "Creating a Project" (page 9).
	Added ADC Reference Library–update information in "Finding Technical Information" (page 29).
2003-08-28	First public version of this document.
2003-06-20	Released as a preliminary document for WWDC 2003.

Document Revision History