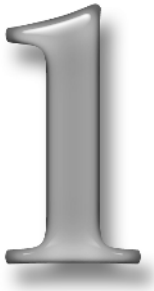

"Hello, World!" Version 48,345,093.1

In This Chapter

Greetings, programmer!

In this chapter, you will take your first steps on the long, event-filled journey into the world of Macintosh OS X software development — unless, of course, you've already done some Macintosh OS X software development, in which case these won't be your first steps now, will they? If so, think of this chapter as an encore telecast. (Yes, we're mixing metaphors; we can do that because we have recently renewed our poetic licenses.)

The Mac has had a long and often deserved reputation for being a difficult platform for which to develop. At different times, the difficulties have been due to different things. For instance, the first Macs were difficult to program simply because the Mac didn't *have* any tools that let programmers program for it: You needed an Apple Lisa (raise your hands if you remember those... okay, put them down; it's not like we can see you out there) wired up to a Mac and running a Macintosh development kit. Then, of course, there was the matter of the Macintosh toolbox, those sets of specialized routines and functions built into the early Mac ROMs, which programmers had to learn to understand and use, and which were documented in a heap of very heavy and continually updated loose-leaf binders. Wrapping one's head around those was not a project for a rainy afternoon.

But, some 20 years into the Macintosh era, things have gotten better... not that you don't need to learn a bunch of stuff (hey, if programming were easy, everyone would be doing it), but the programming environment is cheap (free, in fact, which is at the lower limit of cheap), and the documentation, instead of bowing your bookshelves, is now at your fingertips as you code. And, of course, the tools are a lot — and we mean, a *whole* lot — better.

Apple collectively calls these tools *Xcode*. And with them, we're going to write yet another version of that staple of introductory programming classes, the "Hello, World!" application. In the process, you'll get a quick preview of the power and, dare we say it, utter coolness of twenty-first-century Mac programming. So let's get ready to rumble...

YOU *HAVE* INSTALLED XCODE, HAVEN'T YOU?

Yes, Apple really does provide this wonderful Xcode tool chest free of charge (well, free if you don't count the fact that you have to have Tiger to make use of it and Tiger isn't exactly free). Unlike most of the other really cool toys Apple includes with Tiger, such as iTunes and iChat, Xcode doesn't get installed automatically — you have to do that yourself. So, slip your Tiger DVD back into your Mac and look for a folder named Xcode — you'll probably have to scroll down to find it.

1. **Open the Xcode folder and double-click the Xcode.mpkg file to present the very familiar Installer dialog (shown in Figure 1-1).**

2. **Click Continue to see the Software License Agreement.** Click Continue again and agree to the license agreement (after reading every last clause and condition and consulting with the friendly legal adviser that you have on retainer for just such occasions... or not), and then click Continue yet another time.

3. **Choose a destination volume, which must be a Tiger start-up volume.**

4. **Click Continue, click Install, and then type an Administrator password as shown in Figure 1-2.** (If you're not running from an Administrator account, you'll need to enter an Administrator account's user name in the Name field.)

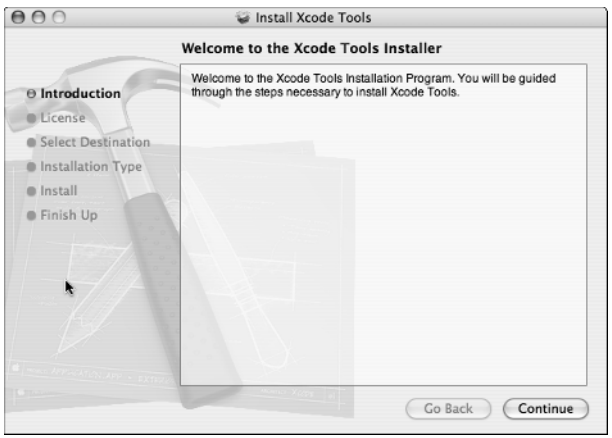

Figure 1-1
Xcode's installer looks very familiar

Figure 1-2
You need Administrator access to install Xcode

Then, sit back, listen to your iPod, and wait while the Installer creates a Developer folder at the root level of your start-up volume and fills it with all the bright and shiny Xcode goodies. When the Installer finishes its tasks, click the Close button. In the Finder, you'll see a Developer directory such as the one shown in Figure 1-3.

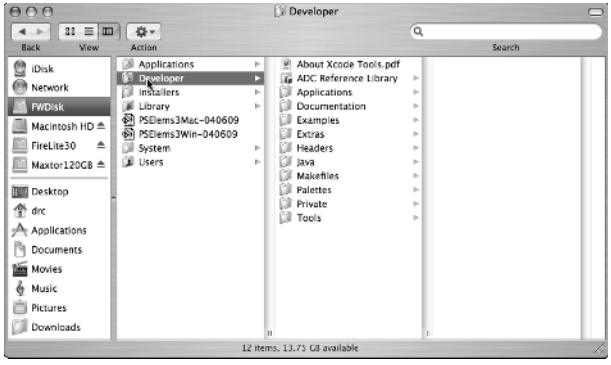

Figure 1-3
When all is said and done...

Now that you have Xcode installed, your creative urges are welling, but when you double-click Xcode (it's in Developer ➜ Applications), you'll find that you still need to configure Xcode, and the Xcode Assistant (see Figure 1-4) will hold your hand through the setup process (which is really quite a trick, when you stop to consider that the assistant has no hands). In other words, there are a few more steps on this meandering path.

Figure 1-4
We're installed; let's get set up

5. **Click Next.** The Assistant, as shown in Figure 1-5, asks you a couple of questions about how you want to build your projects. First, you can have Xcode place your builds' results in the project directory or in another directory you specify. Second, if your product is complex enough to build intermediate elements (subprojects, if you will), you can decide whether they should be stored where the final build goes or in another location that you specify. Of course, the explanation at the bottom of the Assistant's window told you all of this already — and if you didn't follow all of what we and the window just told you, you should probably accept the default settings; Apple (usually) chooses defaults that make sense.

Figure 1-5
Tell Xcode where you want to store the things you build

6. **Click Next.** The Assistant now inquires whether Xcode should keep track of all your open windows when you close a project or quit from Xcode so that reopening the project will reopen all your windows just as they were when you closed the project. You can see this really simple step in Figure 1-6.

Figure 1-6
Do you like to pick up where you left off when you reopen a project?

7. **Click Finish.**

If you want to see what's new in this version of Xcode, choose Help ➜ Show Release Notes, and Xcode opens a documentation window, as shown in Figure 1-7, displaying the Release Notes for your Xcode version.

THE PROJECT ASSISTANT HAS A NICE BOW TIE!

All righty. Xcode is installed, and you've fired it up once just to make sure that it works. So let's actually do something with it. Like, say, create a new project...

Note

Right now we're just going to blast through this whole building-a-program business step by step without (much) in-depth explanation. In later parts of the book we'll tell you more — much more — about what's going on when you perform actions like those that follow. For example, you'll see figures suspiciously similar to the ones that follow in the next chapter when we give you a slo-mo replay of the New Project Assistant. Here, however, we focus just on getting something done.

1. **Choose New Project from the File menu.** It's easy to find this command because it's right at the top of the File menu (see Figure 1-8). When you choose New Project, you get the New Project Assistant (see Figure 1-9), which, like most Mac OS X assistants, is represented by a headless torso wearing a bow tie. You'll see this dude a lot; even though he doesn't have a head (or other extremities), he can be quite helpful.

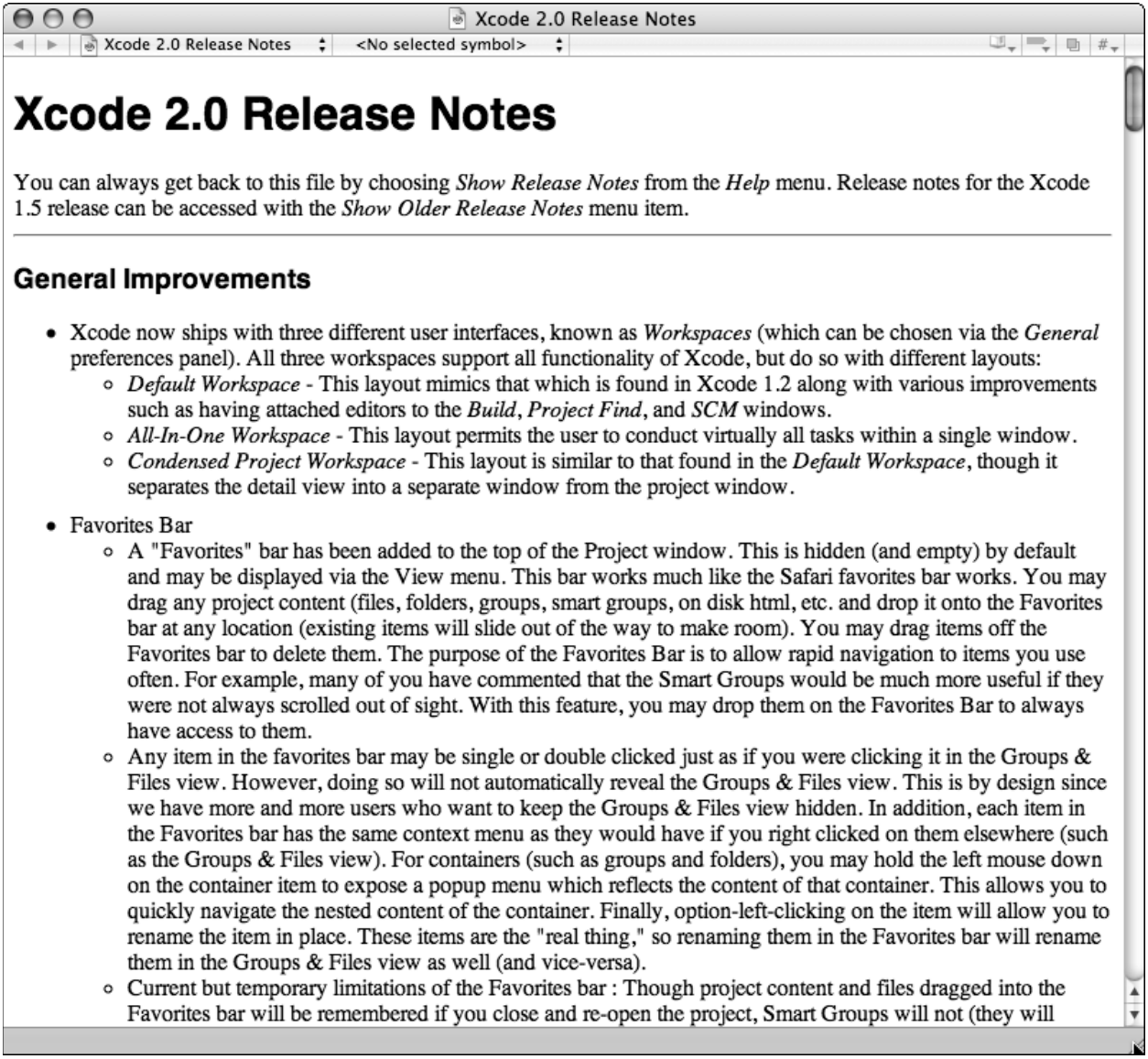

Figure 1-7
The 411 on what's new in this version of Xcode

Figure 1-8
It's easy to start a new Xcode project...

2. **Click Cocoa Application and then click Next.** The Project Assistant dude is not done with you; he wants to know what you want to call the project and where you want to save it (see Figure 1-10).

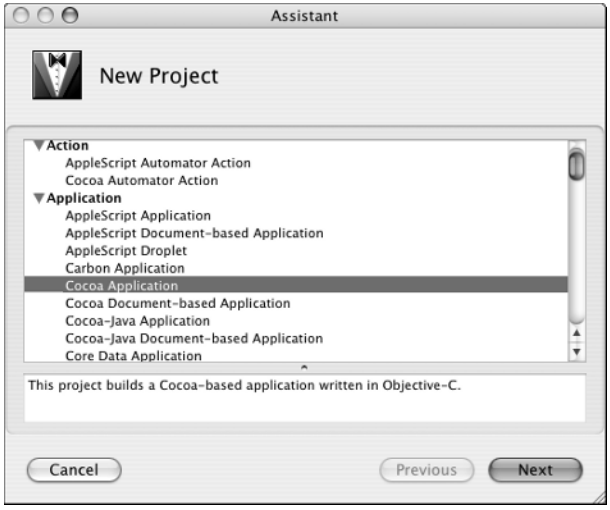

Figure 1-9
...just let the New Project Assistant do its stuff

Figure 1-10
Give the new project a name and place to live

3. **Type a project name in the Project Name field.** As you type, the Assistant dude fills out the second field, which is going to be the complete Unix-style path (that is, the same way you type paths in the Finder's Go to Folder dialog) to where your project is stored. If you don't mess with the second field, your project ends up in your Home directory in a folder that has the project name you typed.

4. **Click Finish.**

And lo! There's your new project, in a project window (see Figure 1-11) that's chock-full (what *is* a chock and how full can you get one, anyway?) of all sorts of things you don't understand... yet. But you will; oh, you will! (Cue the evil laughter sound effect.)

At this very instant, you have a complete application ready to build and run if you like. Now, we're not saying you *should* build your project, but if you *did* (say, for example, by clicking that inviting Build and Go icon at the top of the project window), Xcode would compile, link, and then launch your new, utterly useless-but-very-cool-nonetheless application — which is already sophisticated enough to show you a (blank) window and a set of standard menus.

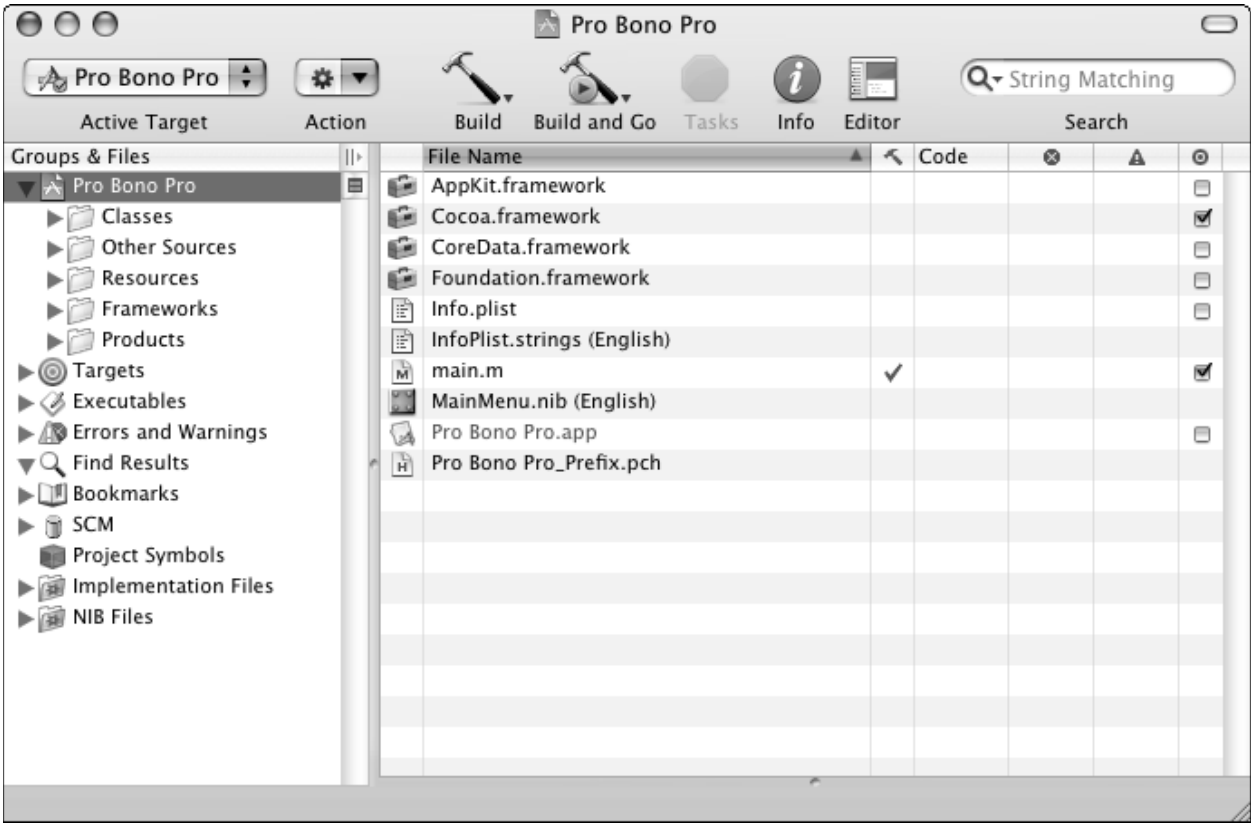

Figure 1-11
A new project, untouched by human hands

Tip

The standard menus in your application include a program menu with a Quit command — you'll probably want to use this command if you disregarded our half-hearted advice and built your project in order to see what would happen.

GETTING IN YOUR FACE

If you're like just about every other programmer we ever met, you are probably itching to write some code. We suggest, however, that you soothe your itchy patches with some Benadryl or hydrocortisone cream, because we're not going to be typing any code yet. There's a lot to get out of the way, first — for example, it wouldn't be a bad idea to decide just what it is that our program is going to do.

Because our program is the traditional Hello, World! program, it should, as all such programs do, display the message, "Hello, world!" Because it's a Mac program, it should display this message in a window, and, because Mac programs are user friendly, it should let the user click something to produce the message. In short, we need a window, a button, and a text display area.

If you build and run the project we just created, you'll discover that the default Cocoa application already provides a window. So that's one down. Next, we need to add the button and the text display area to our project's window.

You may be thinking: Aha! *Now* we're going to write some code. Nope. Still not yet. Instead, we're going to build our program's interface graphically with Xcode's Interface Builder application and let *it* do most of the coding for us. But don't worry: Eventually you'll get to type some code yourself. Honest. Maybe even a *whole line* of it...

1. **Double-click the MainMenu.nib file shown in the project window.** This file appears in the right pane of your project window (see Figure 1-11); it was generated automatically when you created the project. Double-clicking it opens Interface Builder, the Xcode application that lets you assemble the standard elements of Mac OS X's graphical user interface by using palettes and drag-and-drop techniques. (Chapter 9 covers this mind bogglingly cool tool in greater depth.)

When Interface Builder finishes launching, you see something like Figure 1-12: a window named Window, a window named Cocoa-Menus, and a window named MainMenu.nib, all floating over your project window. The Window window is your application's main window. The Cocoa-Menus window is a palette that contains user interface elements you can add to your project; its name changes depending on the types of user interface elements displayed in it. The MainMenu.nib window shows the interface elements contained in your project; currently, it shows that your project contains a window element, a menu element, and a couple of other necessary things.

Note

The user interface elements shown in Interface Builder are, technically, instances of classes that are predefined as part of Mac OS X's Cocoa framework. If you know something about object-oriented programming, you'll know what we're talking about; if not, you should bone up on the basic concepts a little: Mac OS X is *very* object-oriented.

2. **Click the Controls icon in the palette.** This icon, shown here, appears at the top of the interface elements palette, second from the left. When you click the icon, the palette's name changes to Cocoa-Controls, and the palette displays a set of control widgets that you can click and drag into the interface you are building.

Tip

When you point at an element in Interface Builder's Cocoa interface elements palette, a tool tip appears

that identifies the Cocoa class to which the element belongs. You may notice, for example, that many of the different-looking buttons appearing in the Cocoa-Controls palette actually belong to the NSButton class, a class that is defined in the Cocoa frameworks. Chapter 7 describes how you can consult documentation describing the frameworks and their classes at any time as you work. (Okay, that's really two tips here — we're generous.)

3. **Click and drag the button labeled Button from the top left of the Cocoa-Controls palette to the application's window and release the mouse.** This is how you add interface elements to your application (see Figure 1-13). If, at this moment, you were to save your work (and guess what? ⌘+S will do that very thing) and then build your application again, the button would appear in the application's main window exactly where you put it in Interface Builder.

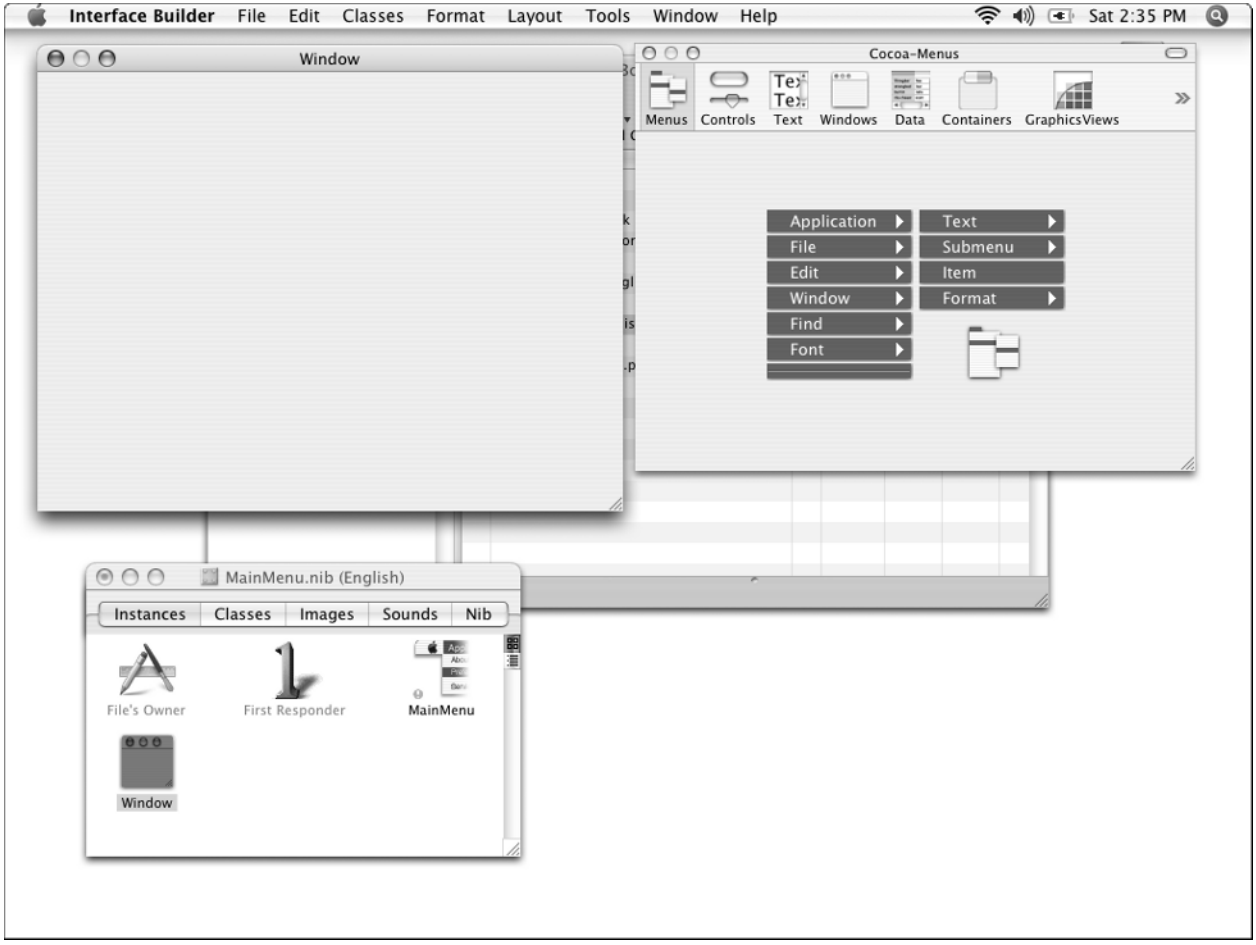

Figure 1-12
The program's user interface as seen in Interface Builder

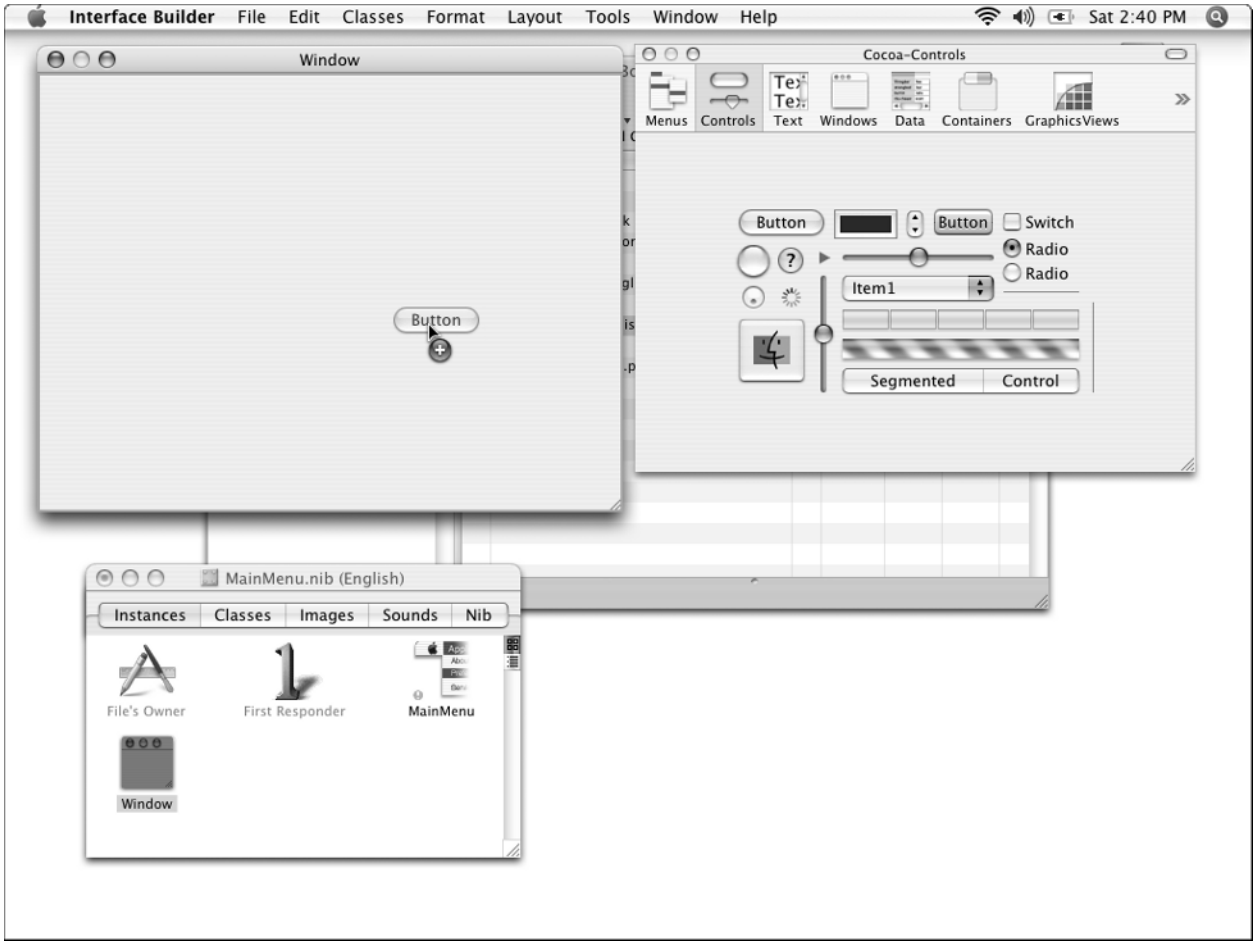

Figure 1-13
Adding an interface element to the application's main window

4. **Click and drag one of the button's handles on its left or right side to make it wider.**

5. **Double-click the button's label to select it, and then type a new label for the button.** Figure 1-14 shows you how we adjusted the button's size and location and the label we gave it; feel free to modify your button to suit your own aesthetic sensibilities — it's easy to do and fun for the whole family.

6. **Click the text icon in the palette.** This icon, shown here, appears at the top of the interface elements palette, third from the left. When you click the icon, the palette's name changes to Cocoa-Text, and the palette displays a set of text widgets, which, just like the controls, you can drag into the interface you are building.

A SMALL PACKAGE OF VALUE WILL COME TO YOU, SHORTLY

All of the button's characteristics that you have (and will) set in this tutorial are saved in the MainMenu.nib file that belongs to your project, and that file becomes part of the application's resources when you build your project. For you Mac old-timers, this arrangement is conceptually somewhat like the old resource fork of days gone by, but architecturally it's quite different. Modern Mac applications (like the one that we are making here) are usually stored in special directories, called *packages*, that just happen to *look* like files. Inside of packages live various directories and files, including the .nib files containing the application's interface elements.

You can easily see inside of packages in the Finder: Simply Control + click the file (or right-click for you multibutton mousers out there) and select Show Package Contents from the contextual menu. Like magic, a new Finder window opens and you can go traipsing through the package to your heart's content. If you find a .nib file, you can double-click it and it will open in Interface Builder — which we really must tell you is both an exceedingly exciting and a dangerously foolhardy thing to do with any application that you care about in any way. Using this trick, you can walk on the wild side, destroying various applications' interfaces with Interface Builder, just like Mac old-timers could do with ResEdit. The Old and True Rules apply: Only mess around with copies, and hack responsibly.

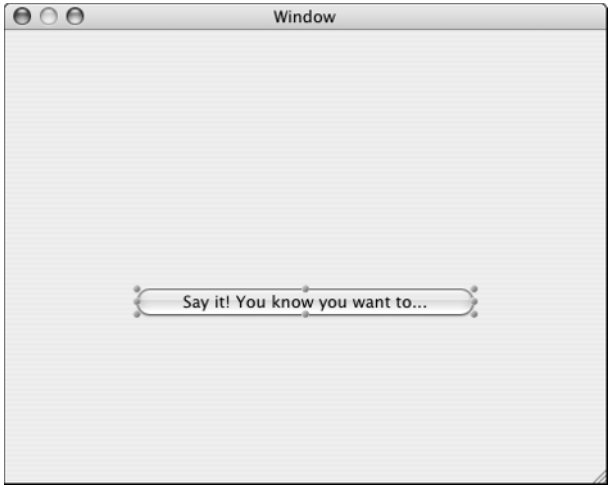

Figure 1-14
A resized and relabeled button now adorns the interface

7. **Click and drag a blank NSTextField object from the Cocoa-Text palette to the application's window and release the mouse.** This field, like the button you previously added, is now part of the application's main window (see Figure 1-15), and it is where your app is going to display its Hello, World message. (If you can't find it, the NSTextField object is the recessed-looking blank text field on the left side of the Cocoa-Text palette as shown in Figure 1-15.)

8. **Click and drag one of the field's handles on its left or right side to make it wider.** You want to make the field wide enough to display the Hello, World message. You can also click the field and drag it where you think it looks best in the window.

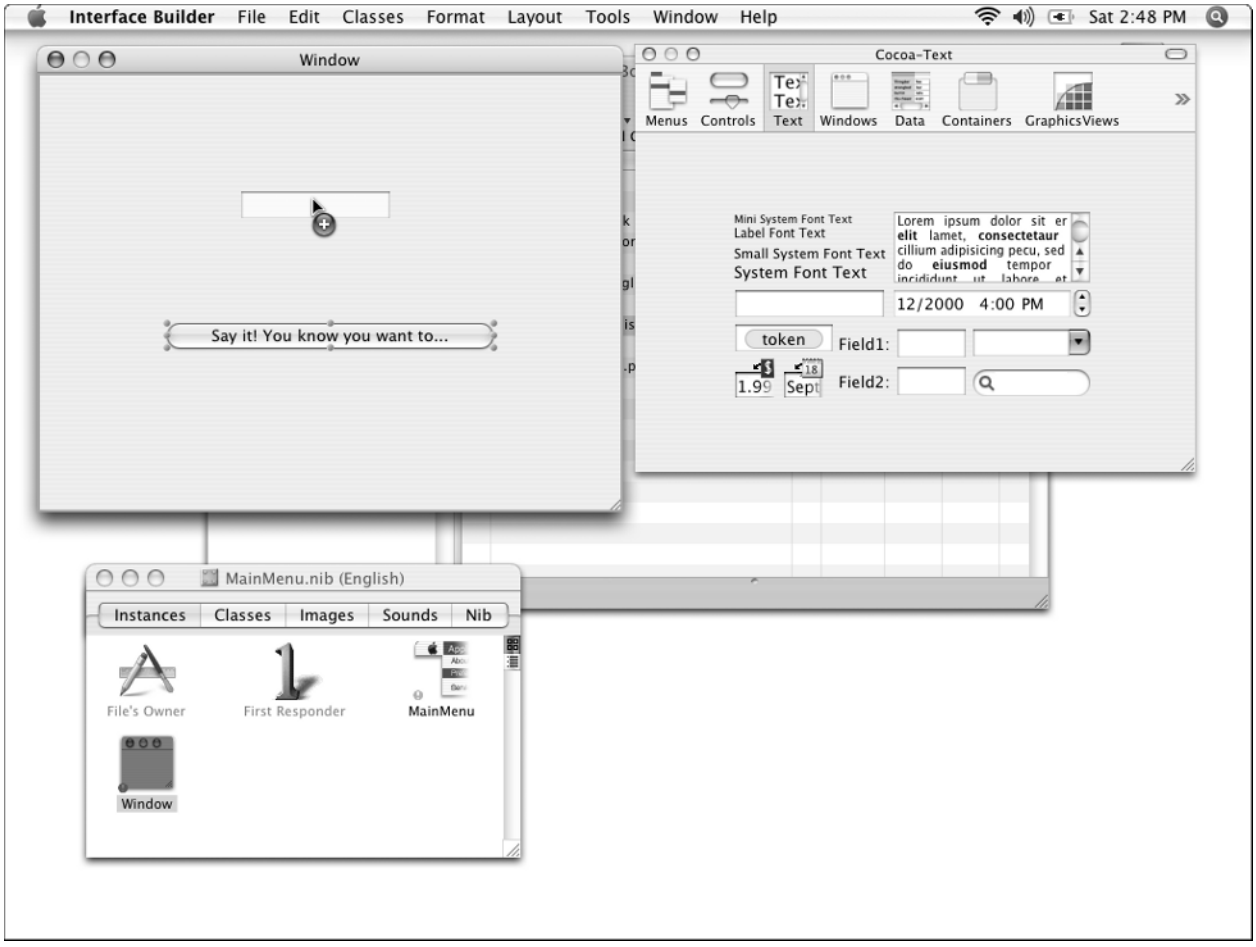

Figure 1-15
Toss in a text field to display the Hello, World message

9. **(Optional) Double-click the field to put an insertion point into it, and then type some text for the field to display.** The field in your application's main window initially will contain this text when the application runs. If you like, you can also use the Font and Text submenus in Interface Builder's Format menu to style the field's text however you want. You can see our version of the field in Figure 1-16.

10. **Save your work using either ⌘+S or Save from the File menu.**

You now have all the elements of the app's user interface in place. Were you to build and run the app right now, you'd have a fully functional nonfunctional program; that is, the button, field, and window would appear, but they wouldn't do anything very interesting. Wiring them up is our next mission.

THE N WORDS

So far, you've seen a couple of interface elements that represent certain Cocoa framework classes: The button you dragged into your window, for example, is an NSButton, and the text field you dragged is an NSTextField. So, what's up with this "NS" stuff?

Those letters are a historical holdover. The classes in the Cocoa framework were originally developed at Steve Job's previous computer company, NeXT, which Apple bought in the late 1990s. These classes, as well as much of the Cocoa framework, were part of the NeXTStep class libraries, and the NS, in case you haven't guessed by now, stands for "NeXTStep." Interface Builder itself, in fact, is a direct descendent of the interface-building tool from the NeXTStep system. One might be tempted to think that Cocoa is simply NeXTStep with a candy coating. One might be right.

Figure 1-16
The Hello, World! app's text display field gets some style and content

HOOKING UP

Right out of the wrapper, Cocoa buttons and text fields are pretty smart: A Cocoa button "knows" when it has been clicked and can even tell another object, "Hey, I've been clicked!" and a Cocoa text field can easily be told to display any given piece of text. So here's what we have to figure out next: how to translate our button's "Hey, I've been clicked!" message into a message that tells our text field what to display. There are lots of possible answers (that's what makes programming the enjoyable, creative time-sink that it is); the approach we will take is the Model-View-Controller approach that all the best authorities in the field consider to be The Right Approach to object-oriented programming (and we're not about to argue with all the best authorities in the field without a really good reason, such as a cash payment or a free week on Maui).

This is what you need to do: Create a new type of object to control the interaction between our button and our field. This controller object will listen to the button, and, when the button tells the controller object that it's been clicked, the controller object will tell the text field what to display. Setting this sort of thing up with Interface Builder is pretty easy.

And, no, we're *still* not going to be writing any code yet, but we're getting closer.

1. **Click the Classes tab in the MainMenu.nib window. Scroll all the way to the left, and click NSObject (see Figure 1-17).** NSObject is the Cocoa class from which all other Cocoa classes descend. We want our new controller class to be a Cocoa class, so this is the perfect place to start.

Note

The NSObject class, like other Cocoa classes, appears in gray in the MainMenu.nib window in order to indicate that it is a standard class and not one that you

THE ABCS OF MVC IN OOP

The Model-View-Controller (MVC) way of thinking about object-oriented programming (OOP) is just a way to reduce the brain-cramp that can occur when you design large, complicated programs that have a whole slew of interrelated functions and data. The idea is pretty simple: Conceptually divide the object classes between those that the user sees and touches (the view), the underlying structure (including data structures and the routines to manipulate them) that the view represents (the model), and the code that manages interactions between the view and the model (the controller).

Clear as mud, right? In our Hello, World! application, it breaks down this way. The View is the window, text field, button, and menus that the user sees and can manipulate. The Model is the underlying idea that a textual message is displayed in response to some action by the user — the particular appearance of the text field that displays the message or of the button that the user clicks (or even if it *is* a button-click that triggers the message) is irrelevent (in fact, our Hello, World! application's model is pretty barren, lacking any data structures or data manipulating routines). The Controller is the program logic that monitors the trigger condition and then changes the text message when the condition is met.

By separating things out this way, we can change the controller's capabilities (for example, what it tells the field to display) without touching the button or the field. We can change where the button and field appear in the window, as well as their sizes, shapes, and so on, without touching the controller logic. We can devise additional ways of creating the text-changing trigger conditions (that is, changing the model) without touching the view or the controller logic. And we can reduce brain-cramp in the process. Everybody wins.

have created and can modify. This color also makes the class name hard to see when you select it, as we have in Figure 1-17. Nice one, Apple.

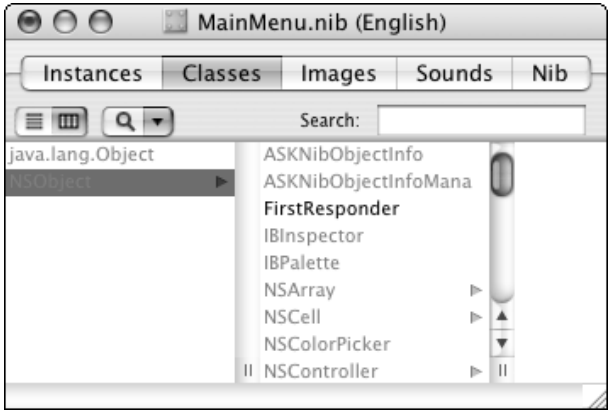

Figure 1-17
Select a base class...

2. **Select Subclass NSObject from the Classes menu (see Figure 1-18).** A new entry appears in the column to the right of NSObject in the MainMenu.nib window: MyObject. Its name is selected, ready for you to change it to something more appropriate to its function (see Figure 1-19).

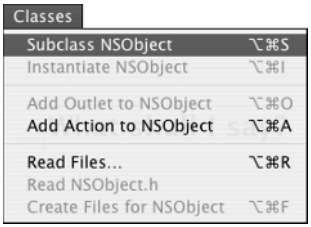

Figure 1-18
... subclass it...

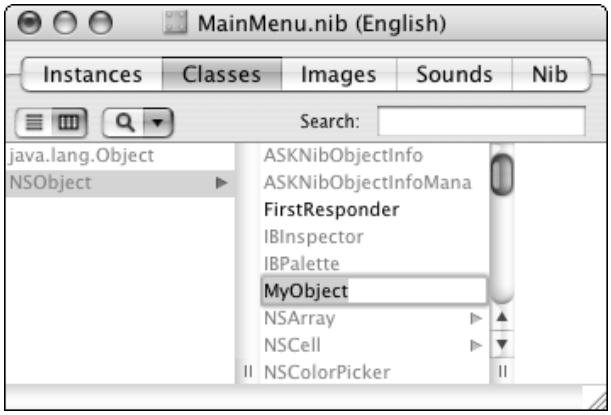

Figure 1-19
...and give it a name

3. **Type a new name for the class.** We called ours HelloController; by convention, class names begin with an uppercase letter.

4. **Control-click (or right-click) the new class and select Add Outlet to HelloController (or whatever it was that you named your class) from the contextual menu.** A Class Info window opens showing you the attributes of your new class, with a new outlet selected so you can change its name (see Figure 1-20).

Note

An *outlet* in Cocoa/Interface Builder parlance is really nothing more than an instance variable that gets added to a class. Interface Builder uses outlets to establish connections between an application's interface objects so they can exchange messages. The outlet we're creating here eventually will be connected to the text field in our app's main window so the controller object can tell the text field to change the text that it's showing.

5. **Give the outlet a name.** We called ours helloSayer. We could have called it Fred. We didn't. Also note that, by convention, outlet names begin with a lowercase letter.

Figure 1-20
Like so many of us, our controller needs an outlet

6. **Click the small pop-up widget beside the outlet's type and select NSTextField from the pop-up scrolling list (see Figure 1-21).** Why? Because the app's text field is an NSTextField, and we want our controller to be able to communicate with it.

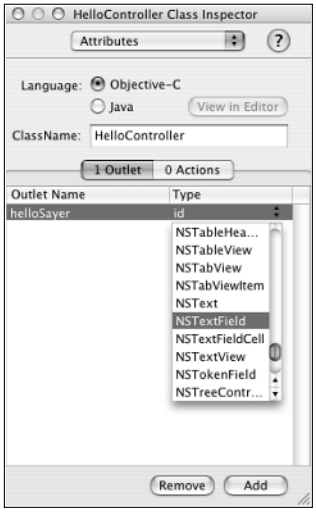

Figure 1-21
Picking an outlet type

7. **Click the 0 Actions tab in the HelloController Class Inspector window.** You see a list of actions that your controller class can take... actually, you *don't* see a list of actions, because your controller class doesn't have any yet. But you can see where the list would be if there were any items in it.

8. **Click the Add button at the bottom of the HelloController Class Inspector window.** A new action appears with its name selected so you can change it. Also note that the 0 Actions tab in the Inspector window changes to say 1 Action. That's right, Interface Builder knows how to count.

9. **Give your action a suitable name.** We named ours sayTheThang: (see Figure 1-22). Note that action names, by convention, begin with a lowercase letter. They also *must* end with a colon... but if you don't supply one, Interface Builder will. It knows the rules.

Figure 1-22
An action hero is born

Note

An *action* is nothing more than the name of a method (or of a member function for you C++ fans out there). When a user does something to an interface object, such as clicking it, the object can respond by sending an *action message* to another object; this message becomes, through the magic of the Cocoa runtime environment, a method call with a single parameter (specifically, the ID of the sender). The action we have created here is the action message that our HelloController will receive from the app's button when the button is clicked... once, that is, we establish a connection between the button and the HelloController with Interface Builder. We'll do that shortly. In step 15, in fact.

10. **Select Create Files for HelloController from the Classes menu (the name of the class in the command will match the name you gave yours; see Figure 1-23).** A sheet descends from your MainMenu.nib window (see Figure 1-24).

Figure 1-23
Free code...

11. **Make sure that the checkboxes for both the class's header file (ending with .h) and the implementation file (ending with .m) in the bottom-left pane of the sheet are checked, and that the files are being inserted into the right target (that is, your project), as shown in the bottom-right pane of the sheet, and then click Choose.** Interface Builder magically creates the two source files that define your class and adds them into the Other Sources folder in your Xcode project window (see Figure 1-25).

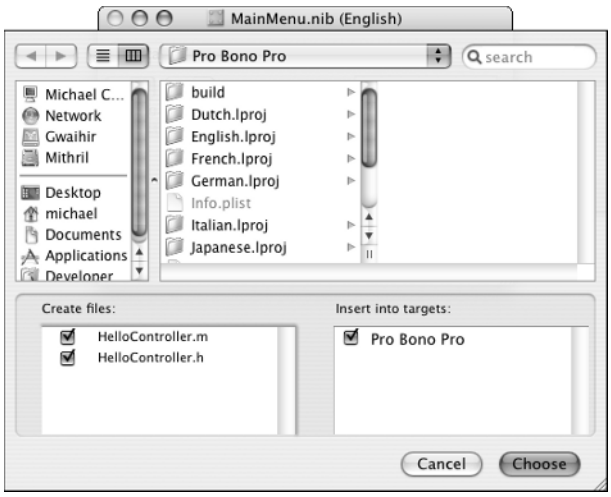

Figure 1-24
... is just a click or two away

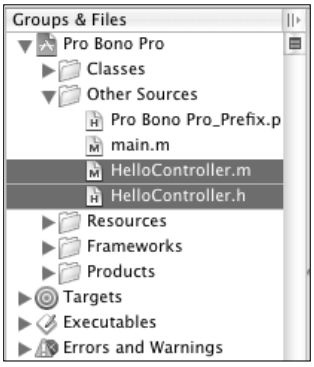

Figure 1-25
This source is a new source, of course, of course

12. **Choose Classes ➜ Instantiate HelloController (you may need to select the class first in the Classes tab of the MainMenu.nib window).** This creates an actual instance of the class in your nib. You'll need this.

13. **Click the Instances tab of the MainMenu.nib window.** The window displays your controller object amid its collection of interface objects (see Figure 1-26). A small white exclamation mark in a yellow disk appears beside the object, which is Interface Builder's polite way of saying that the object has some yet-to-be-connected outlets.

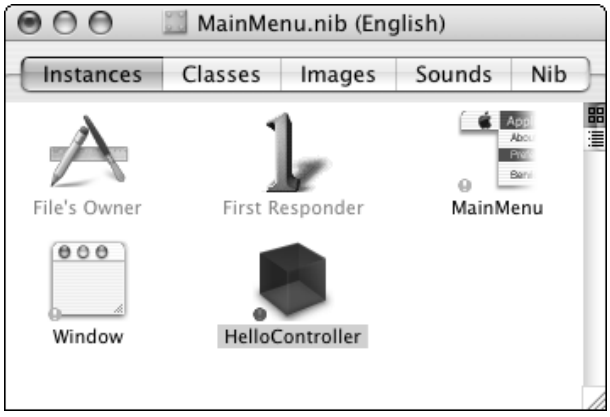

Figure 1-26
The controller, ready to be wired up

Tip

Hover your mouse over the exclamation mark to see a tool tip that tells you exactly what Interface Builder thinks is the "problem" with your interface object.

14. **While holding down the Control key, click and drag from the controller object in the MainMenu.nib window to the text field in the Window window, and then release the mouse.** A line stretches from the controller to the field (see Figure 1-27), indicating that you've established a connection between the two objects, and the Inspector window shows the controller object's outlet list.

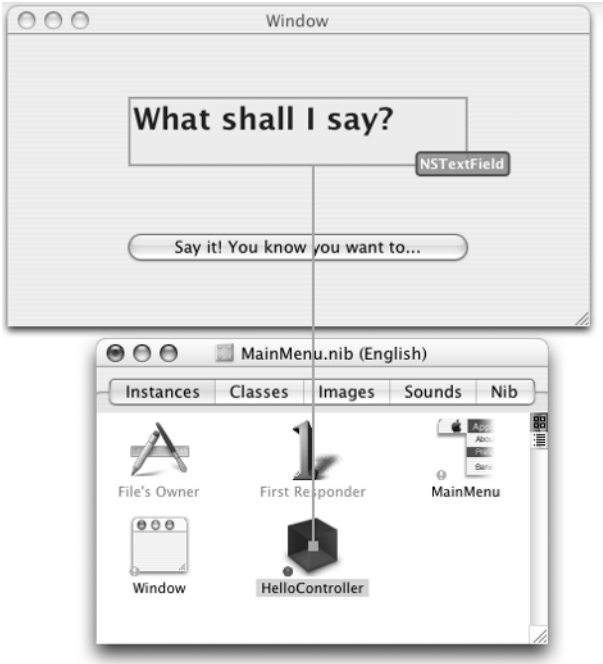

Figure 1-27
The controller is connected to the text field

15. **In the Inspector window, click to select the outlet you created in steps 4 through 6 (if it's not already selected), and then click the window's Connect button.** A gray button appears to the outlet's left and the outlet's destination appears to its right (see Figure 1-28). In other words, the instance variable that corresponds to your controller object's outlet now contains a reference to the text field (or, rather, it will when the application is launched and the Cocoa runtime environment rifles through the .nib file looking for things to hook up).

Figure 1-28
Destination confirmed; enjoy your trip

16. **While holding down the Control key, click and drag from the button in the Window window to the controller object in the MainMenu.nib window, and then release the mouse.** A line now stretches from the button to the controller object (see Figure 1-29), and the Inspector window shows the button's Target/Action list, which contains the controller object's only available Target/Action, sayTheThang.

17. **Click the action listed in the Inspector window and then click the window's Connect button to complete the connection.**

18. **Save your work using either ⌘+S or Save from the File menu.**

And that's it for the Interface Builder portion of our day's entertainment. You've wired up all of your app's interface objects, generated source files to play with, and now, finally, at long last, you can get ready to scratch your coding itch. We hope it was worth the wait.

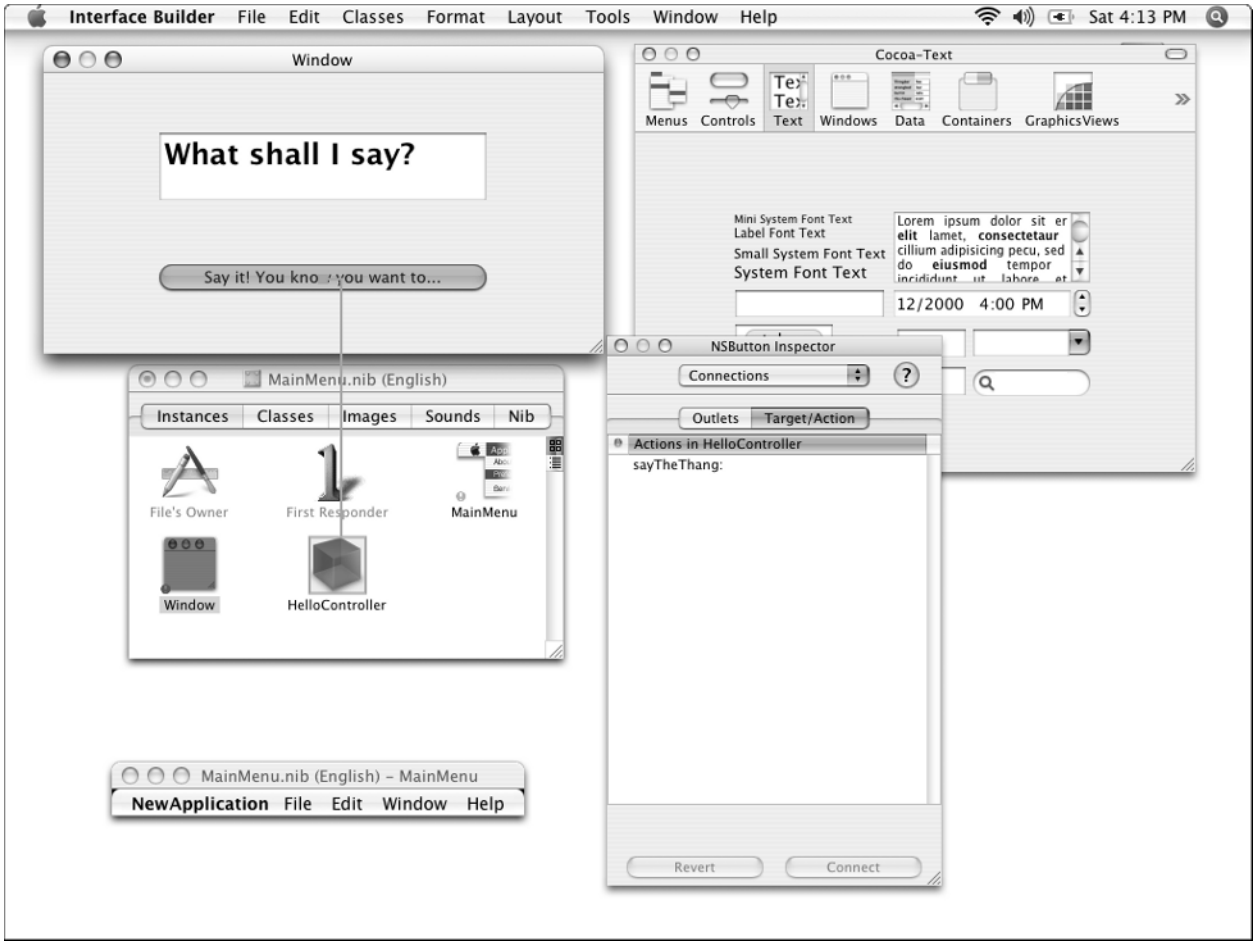

Figure 1-29
The button is connected to the controller

A LITTLE TYPECASTING

Back when we generated source files from our MainMenu.nib file, Interface Builder produced two files, which you saw in Figure 1-25. If you named your controller class HelloController like we did, you now have a header file, HelloController.h, and an implementation file, HelloController.m. Because our project is a Cocoa application, Interface Builder produced these files in Objective-C, an object-oriented version of the C programming language, which is Cocoa's preferred language (although you can use the Cocoa framework from Java, C++, and even a weird and wacky language blend called Objective-C++). Let's take a look at these two files, see what needs changing, and make the changes.

Note

Objective-C is a very simple superset of standard C; it's easy to pick up the basics in just a few hours, especially if you are familiar with other object-oriented programming languages. Xcode's extensive built-in documentation includes the guide, *The Objective-C Programming Language*, which can get you up to speed with the language rather quickly. To find out how to use Xcode's documentation resources, you can flip to Chapter 7.

First, we check out the header file:

1. **Switch back over to the Xcode application if you haven't already.**
2. **Double-click your controller class's header file (for example, HelloController.h) in the project window.** It should appear in the big right-hand pane of your project window; it can also be found in the Other Sources folder in the project window's left-hand pane, as shown in Figure 1-25. An Xcode editing window opens (see Figure 1-30).

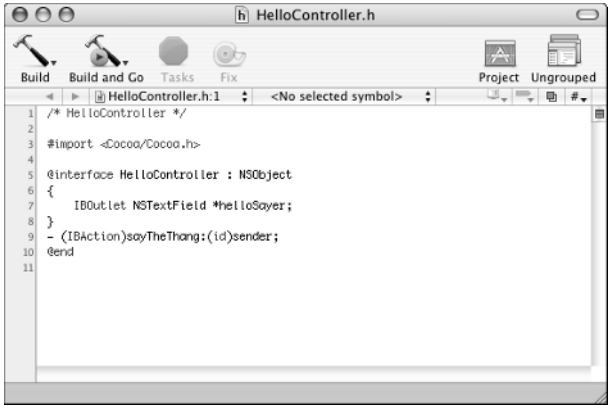

Figure 1-30
Go to the header of the class

The header file is pretty short and sweet, containing

- a comment that identifies the source;
- an #import compiler directive to bring in all the Cocoa stuff (which is much like a #include directive that avoids some recursion issues);
- an @interface statement that lets you, the compiler, and the rest of the known universe know that the HelloController is a subclass of NSObject;
- a sole instance variable, helloSayer, which is the NSTextField outlet you created in Interface Builder (the IBOutlet type qualifier lets Interface Builder and Xcode coordinate with each other as you develop your app) — note that Objective-C instance variables are included between curly braces; and
- a method declaration, sayTheThang, which corresponds to the action you added to your controller class in Interface Builder

There's nothing you need to change here. All is as it should be. You can close the editing window.

Now let's look at the class's implementation source file. Here's where you are going to write some code (Quick! Alert the media!).

1. **Double-click the class's implementation file (for example, HelloController.m) in the project window.** Like the header file, it should appear in the big right-hand pane of your project window and in the Other Sources folder in the project window's left-hand pane. An editing window opens (see Figure 1-31).

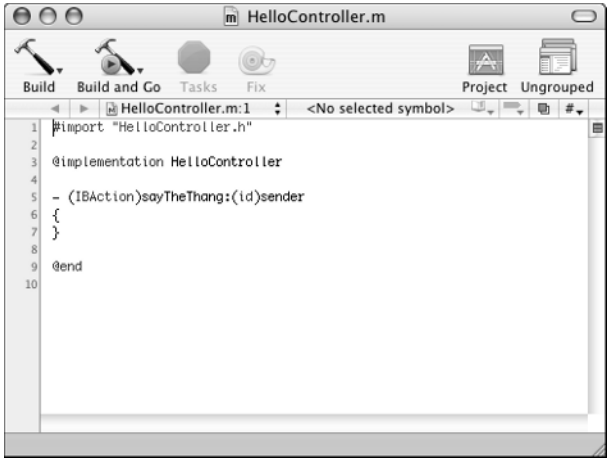

Figure 1-31
A class method, ready to edit

2. **Following the open brace, type the following line of code exactly.** (Well, almost exactly — if you called your outlet something other than `helloSayer`, you should type the name you used instead.) The result should look like Figure 1-32:

```
[helloSayer setStringValue:@"Hello,
world!!!"];
```

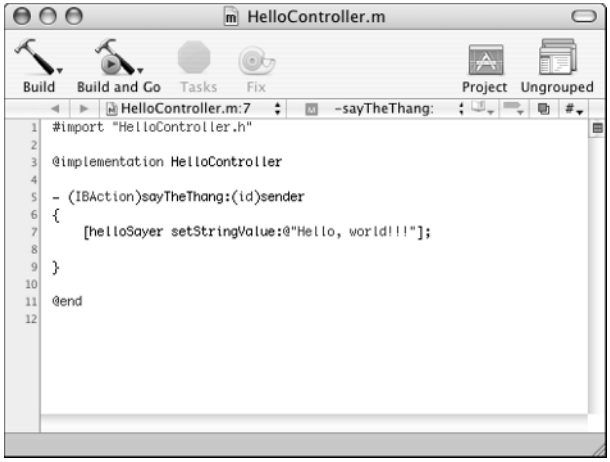

Figure 1-32
A class method, finished and ready to compile

IN CASE YOU ARE WONDERING...

The code you typed is an Objective-C message, which is sent to the object represented by the `helloSayer` instance variable. The message is `setStringValue`. This message (a "message" is Objective-C terminology for what other object-oriented languages might refer to as a "method call" or "member function call") takes a single parameter: the literal string, "Hello, world!!!" (its literalness is indicated by the @ in front of it). In Objective-C, messages are enclosed in brackets to differentiate them from plain, ordinary, C function calls, which Objective-C supports as well (it is, after all, a superset of C).

3. **Save your changes (choose Save from the File menu, or press ⌘+S).**

And that's it. That's *all* the code you have to write to make your application say "Hello, world!!!" when you click the button. Now wasn't *that* worth waiting for?

DO YOU FEEL LUCKY? WELL DO YOU?

Finally, you need to build the program and run it.

If you come from old-school Unix programming, you may be wondering about things like build files, targets, compiler options, and stuff like that there. Don't worry: All that stuff is still around, and if you have a burning desire to get at it, you'll find out how to do so later in this book (say, for example, Chapters 11 and 12). But you don't have to if you don't want to because Xcode tries to handle all of it for you, and it does a pretty good job, too. The only thing you have to do is push a button.

Oh, yes, you *do* need to know which button, don't you? Okay, um, let's see. Right. It's that Build and Go button at the top of your editing window (we may have mentioned it before... well, we're mentioning it again). Click it.

As Xcode indexes, compiles, builds, and links your application, you should see something similar to Figure 1-33 — note the progress indicator and status messages that appear at the bottom of the window. The build process may take a few seconds the first time you build your project, because Xcode has to bring together all the pieces that make up your application (which includes things like the Cocoa framework, which is no small thing), but subsequent builds (say, if you need — or want — to make any minor changes) won't take quite as long.

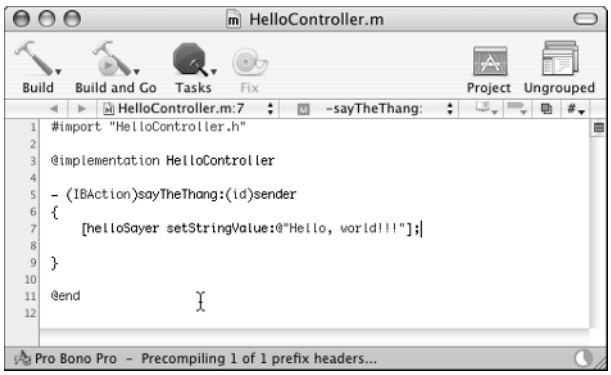

Figure 1-33
Once I built a Mac app, made it run...

Assuming you have typed everything correctly in the one line of code that you had to type, the application launches, its menu bar appears, and the window you designed with Interface Builder shows up. Click the window's button (see Figure 1-34) and see the "Hello, world!!!" message.

Figure 1-34
You don't have to shout

Woo, as they say, hoo! You've written a Macintosh OS X application. Have a cookie. Oh, and don't forget to quit the application when you're finished admiring what a fine job you've done (use the free Cocoa-supplied Quit command on the application's menu).

THE LESSONS OF HELLO, WORLD!

And what have we learned today, kids? That a copy of Xcode comes free inside each colorfully wrapped container of Mac OS X Tiger. That Xcode provides project templates that make it very easy for you to get started creating a program. That Xcode's Interface Builder application lets you do a lot of your user interface design work using simple drag-and-drop techniques instead of requiring you to create carefully handcrafted code. That the Xcode code editor lets you compile the code that you *do* have to write without switching to another mode or application. And, finally, that cranking out a sophisticated application with a fully functional, Aqua-fresh graphical user interface is pretty easy to do.

At least, *we* thought it was.