Chapter 2

Let's Get Started

Many books would start off by giving you a lot of philosophy. This would be a waste of precious paper at this point. Instead, I am going to guide you through writing your first Cocoa application. Upon finishing, you will be excited and confused…and ready for the philosophy.

Our first project will be a random number generator application. It will have two buttons labeled Seed random number generator with time and Generate random number. There will be a text field that will display the generated number. This is a simple example that involves taking user input and generating output. At times, the description of what you are doing and why will seem, well, terse. Don't worry—we will explore all of this in more detail throughout this book. For now, just play along.

Figure 2.1 shows what the completed application will look like.

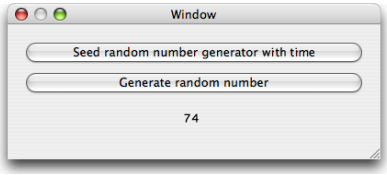

Figure 2.1 Completed Application

In Xcode

Assuming you have installed the developer tools, you will find Xcode in `/Developer/Applications/`. Drag the application to the dock at the bottom of your screen; you will be using it a lot. Launch Xcode.

As mentioned earlier, Xcode will keep track of all the resources that go into your application. All these resources will be kept in a directory called the *project directory*. The first step in developing a new application is to create a new project directory with the default skeleton of an application.

Create a New Project

Under the File menu, choose New Project.... When the panel appears (see Figure 2.2), choose the type of project you would like to create: Cocoa Application. Notice that there are many other types of projects available as well.

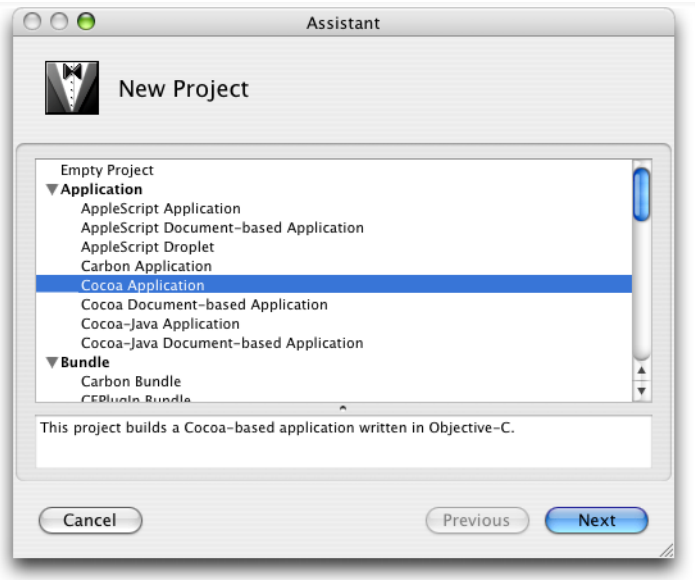

Figure 2.2 Choose Project Type

In this book, we will discuss the following major types of projects:

Application: A program that creates windows.

Tool: A program that does not have a graphical user interface. Typically, a tool is a command-line utility or a daemon that runs in the background.

Bundle or Framework: A directory of resources that can be used in an application or tool. A bundle is dynamically loaded at runtime. An application typically links against a framework at compile time.

For the project name, type in RandomApp, as in Figure 2.3. Application names are typically capitalized. You can also pick the directory into which your project directory will be created. By default, your project directory will be created inside your home directory. Click the Finish button.

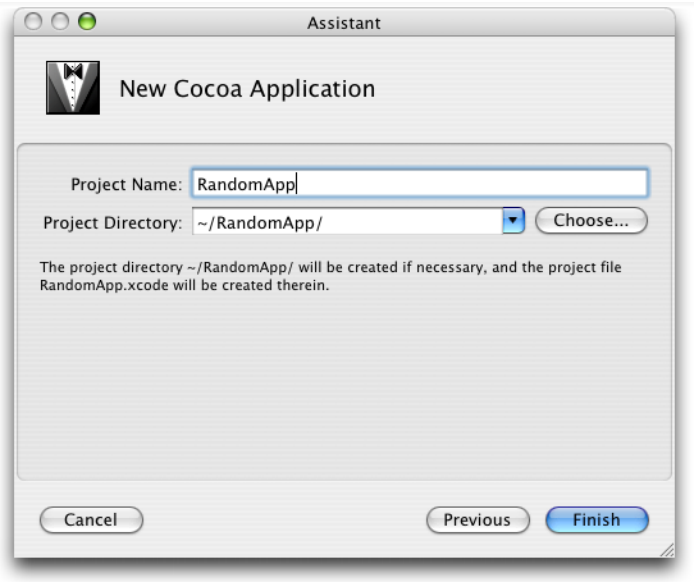

Figure 2.3 Name Project

A project directory will be created for you, with the skeleton of an application inside it. You will extend this skeleton into the source for a complete application and then compile the source into a working application.

Looking at the new project in Xcode, you will see an outline view on the left side of the window. Each item in the outline view represents one type of information that might be useful to a programmer. Some items are files, others are messages like compiler errors or find results. For now, you will be dealing with editing files, so open the item that says RandomApp to see folders that contain the files that will be compiled into an application.

The skeleton of a project that was created for you will actually compile and run. It has a menu and a window. Click on the toolbar item with the hammer and green circle to build and run the project as shown in Figure 2.4.

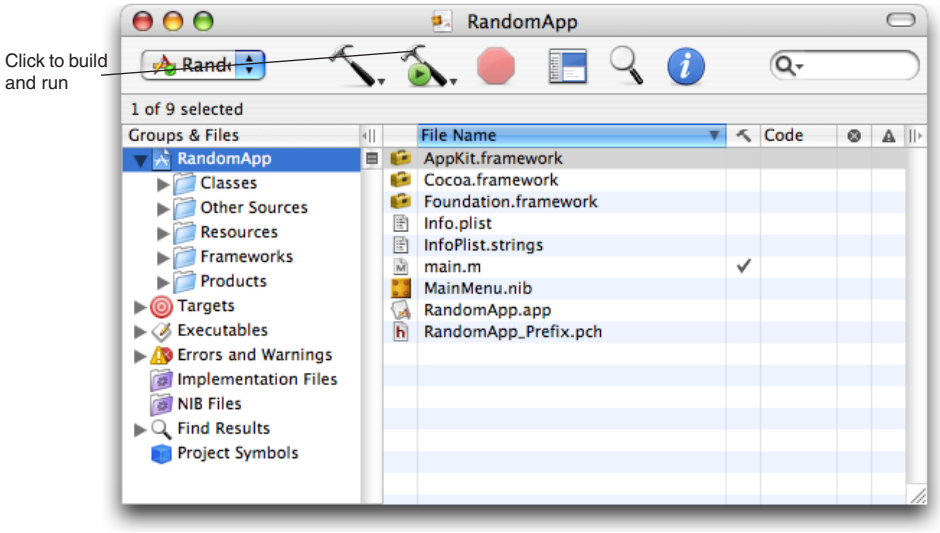

Figure 2.4 Skeleton of a Project

While the application is launching, you will see a bouncing icon in the dock. The name of your application will then appear in the menu. This means that your application is now active. The window for your application may be hidden by another window. If you do not see your window, choose Hide Others from the RandomApp menu. You should see an empty window as shown in Figure 2.5.

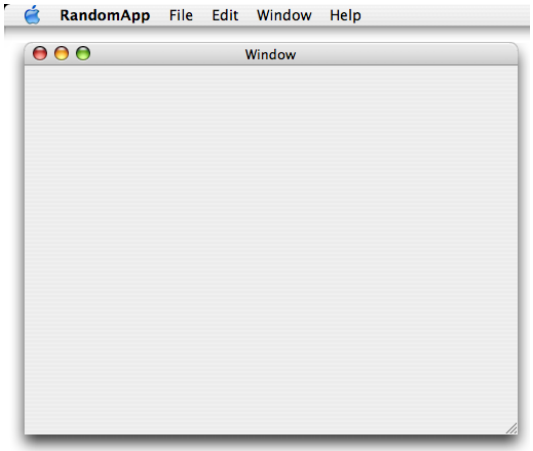

Figure 2.5 Running the Project

It doesn't do much, but notice that it is already a fully functional application. Printing even works. Quit RandomApp and return to Xcode.

The main Function

Select `main.m` by single-clicking on it. If you double-click on the filename, it will open in a new window. Because I deal with many files in a day, this tends to overwhelm me rather quickly, so I use the single-window style. Click on the Editor toolbar item to split the window and create an editor view. The code will appear in the editor view (Figure 2.6).

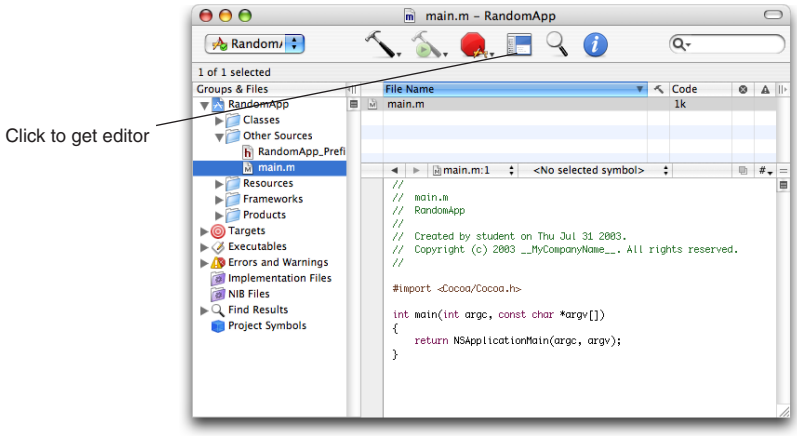

Figure 2.6 main() Function

You will almost never modify `main.m` in an application project. The default **`main()`** simply calls **`NSApplicationMain()`**, which in turn loads the user interface objects from a *nib file*. Nib files are created with Interface Builder. (Trivia: "NIB" stands for "NeXT Interface Builder"; "NS" stands for "NeXTSTEP.") Once your application has loaded the nib file, it simply waits for the user to do something. When the user clicks or types, your code will be called automatically. If you have never written an application with a graphical user interface before, this change will be startling to you: The user is in control, and your code simply reacts to what the user does.

In Interface Builder

In the outline view under Resources, you will find a nib file called `MainMenu.nib`. Double-click on it to open the nib in Interface Builder. Lots of windows will appear, so this is a good time to hide your other applications. In the Interface Builder menu, you will find Hide Others.

Interface Builder allows you to create and edit user interface objects (like windows and buttons) and save those objects into a file. You can also create instances of your custom classes and make connections between those instances and the standard user interface objects. When users interact with the user interface objects, the connections you have made between them and your custom classes will cause your code to be executed.

The Standard Palettes

The palette window (Figure 2.7) is where you will find user interface widgets that can be dragged into your interface. For example, if you want a button, you can drag it from the palette window. Notice the row of buttons at the top of the palette window. As you click the buttons, the various palettes will appear. In Chapter 27, you will learn to create your own palettes.

Figure 2.7 Palette Window

The Blank Window

The blank window (Figure 2.8) represents an instance of the **`NSWindow`** class that is inside your nib file.

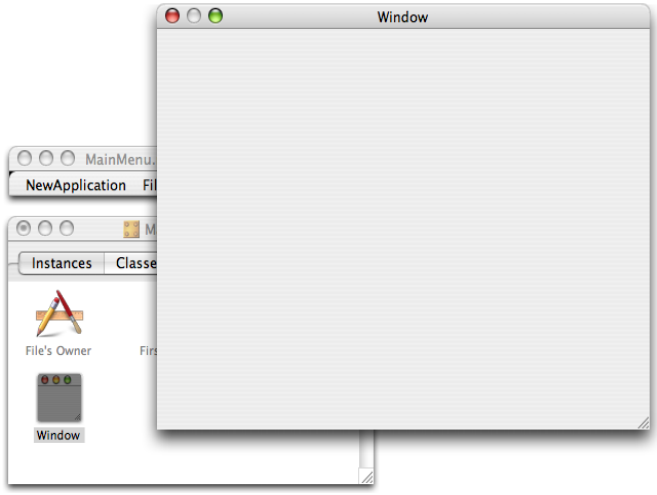

Figure 2.8 Blank Window

As you drop objects from the palettes onto the window, they will be added to the nib file. After you have created instances of these objects and edited their attributes, saving the nib file is like "freeze-drying" the objects into the file. When the application is run, the nib file will be read and the objects will be revived. The cool kids say, "The objects are *archived* into the nib file by Interface Builder and *unarchived* when the application is run."

Lay Out the Interface

I am going to walk you through it, but keep in mind that your goal is to create a user interface that looks like Figure 2.9.

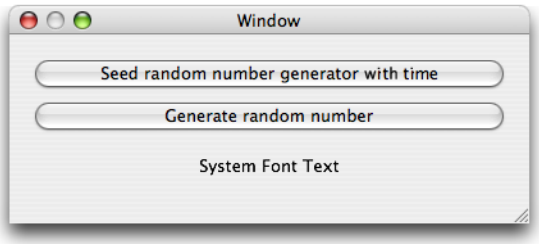

Figure 2.9 Completed Interface

Drag a button from the palette window (as shown in Figure 2.10) and drop it onto the blank window.

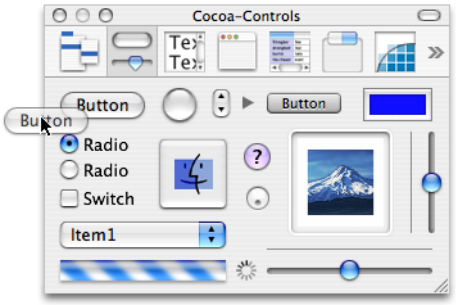

Figure 2.10 Dragging a Button

Double-click on the button to change its title to Seed random number generator with time.

Drag another button out, and relabel it Generate random number. Drag out the text field that says System Font Text (as shown in Figure 2.11) and drop it on the window.

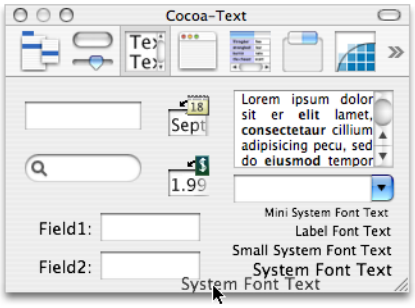

Figure 2.11 Dragging a Text Field

Make the window smaller.

The text field should be as wide as possible. Drag the left and right sides of the text field toward the sides of the window. Notice that blue lines appear when you are close to the edge of the window. These guides are intended to help you conform to Apple's GUI guidelines (Figure 2.12).

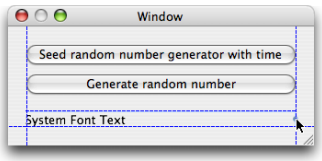

Figure 2.12 Resize Text Field

To make the text field center its contents, you will need to use the Info Panel (also known as the "Inspector"). Select the text field, and choose Show Info from the Tools menu. Click on the center justify button (Figure 2.13).

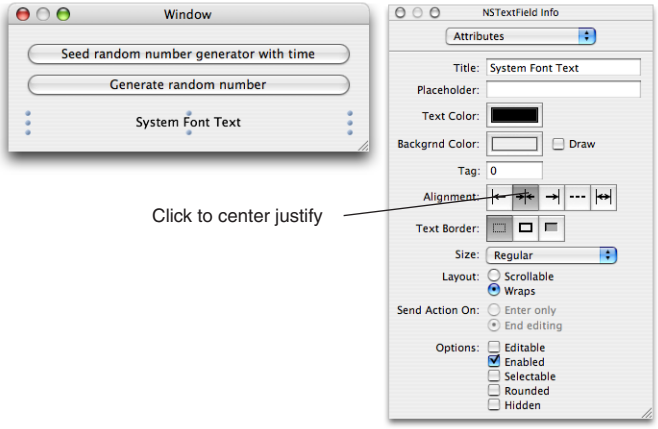

Figure 2.13 Center Justify Text Field

The Doc Window

In your nib file, some objects (like buttons) are visible, and others (like your custom controller objects) are invisible. The icons that represent the invisible objects appear in the *doc window* (Figure 2.14).

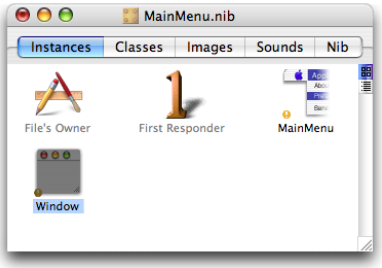

Figure 2.14 The Doc Window

In the doc window (the one entitled MainMenu.nib), you will see icons representing the main menu and the window. First Responder is a fictional object, but it is a very useful fiction. It will be fully explained in Chapter 18. File's Owner in this nib is the **`NSApplication`** object for your application. The **`NSApplication`** object takes events from the event queue and forwards them to the appropriate window. We will discuss File's Owner in depth in Chapter 9.

Create a Class

The doc window also has a simple class browser that you can use to create a skeleton of your custom class. Click on the Classes tab and select **`NSObject`** (Figure 2.15). In the Classes menu, choose Subclass NSObject. Rename the new class **Foo**. Interface Builder now knows that you intend to create a subclass of **`NSObject`** called **Foo**. **`NSObject`** is the root class for the entire Objective-C class hierarchy. That is, all objects in the framework are descendants of **`NSObject`**.

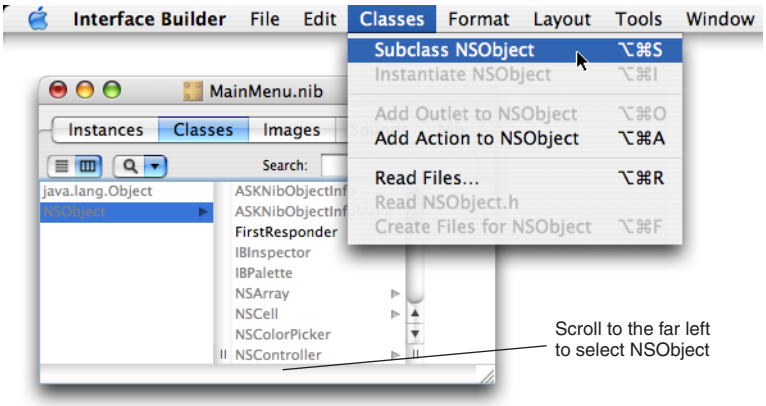

Figure 2.15 Foo Is a Subclass of NSObject

Class names, by convention, are capitalized.

Next, you will add instance variables and methods to your class. Instance variables that are pointers to other objects are called *outlets*. Methods that can be triggered by user interface objects are called *actions*. If you select the **Foo** class and bring up the inspector (use the Show Info menu item to activate the inspector), you will see that your class doesn't have any outlets or actions yet (Figure 2.16).

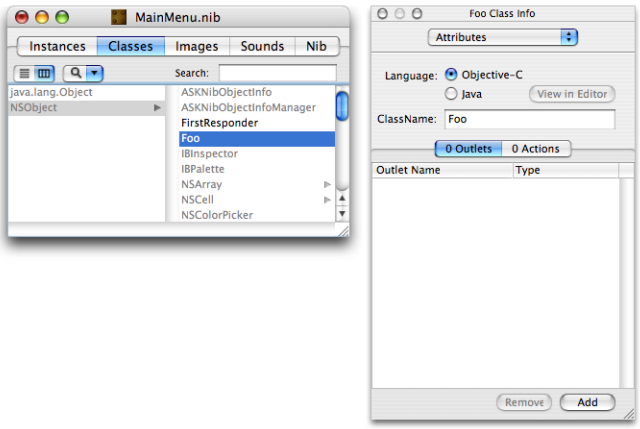

Figure 2.16 View Outlets and Actions

To add an outlet, select the Outlets tab and click Add. Rename the new outlet textField. You can set the type of the pointer in a pop-up. Here `textField` will be a pointer to an **`NSTextField`** object. Set its type using the pop-up as shown in Figure 2.17.

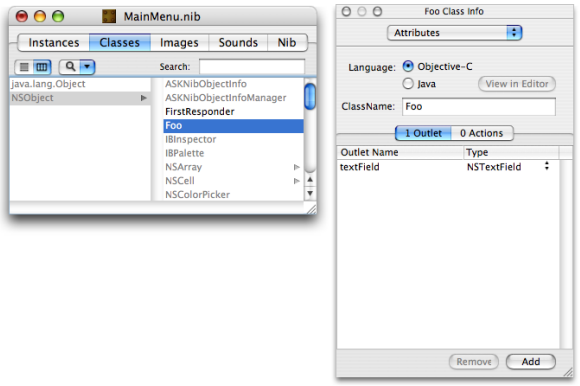

Figure 2.17 Create an Outlet

To add an action, select the Actions tab and click Add. Rename the new action seed. (When you press Enter, it will add a colon to the end of the action name. Thus **`seed:`** is the actual name of the method that will be created.) Add a second action, and name it generate: (Figure 2.18).

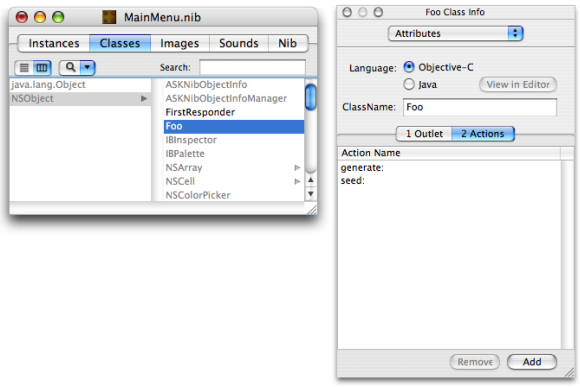

Figure 2.18 Create Two Actions

By convention, the names of methods and instance variables start with lowercase letters. If the name would be multiple words in English, each new word is capitalized—for example, `favoriteColor`.

Now you will create the files for the class **Foo**. In Objective-C, every class is defined by two files: a header file and an implementation file. The header file, also known as the interface file, declares the instance variables and methods your class will have. The implementation file actually defines what those methods do.

Under the Classes menu, choose Create files for Foo... (Figure 2.19).

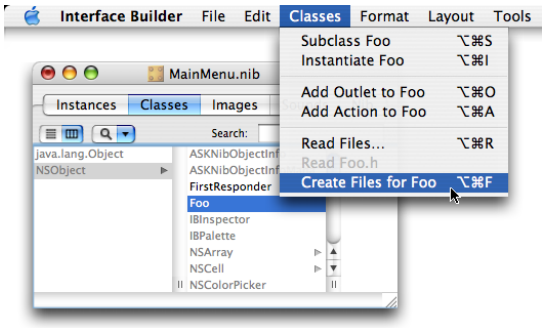

Figure 2.19 Create Files

A save panel will appear. The default location (your project directory) is perfect. Save `Foo.h` (the header file) and `Foo.m` (the implementation file) there. Note that the files are being added to your RandomApp project (Figure 2.20).

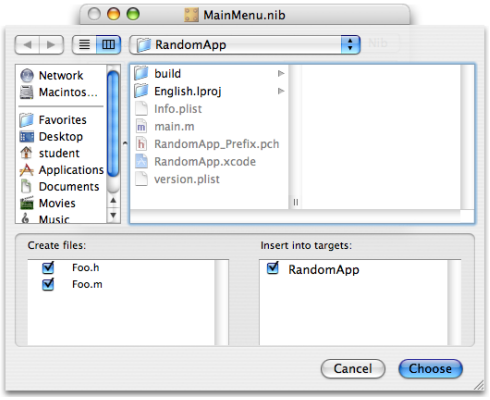

Figure 2.20 Choose a Location for the Files

Create an Instance

Next, you will create an instance of the class **Foo** in your nib file. Select **Foo** in the class browser and choose Instantiate Foo from the Classes menu (Figure 2.21).

Interface Builder will take you back to the Instances tab, where you will see a symbol representing your instance of **Foo** (Figure 2.22).

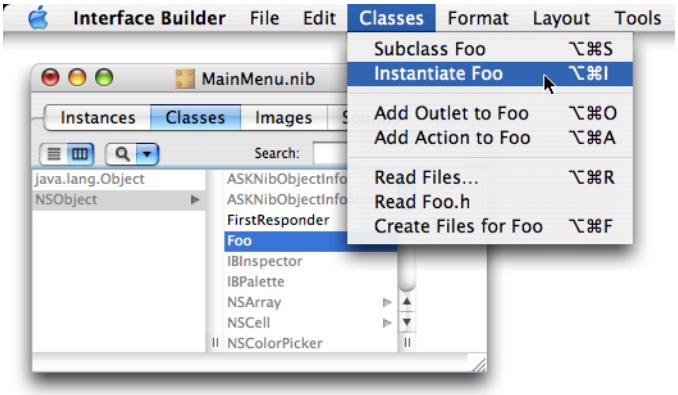

Figure 2.21 Create an Instance of Foo

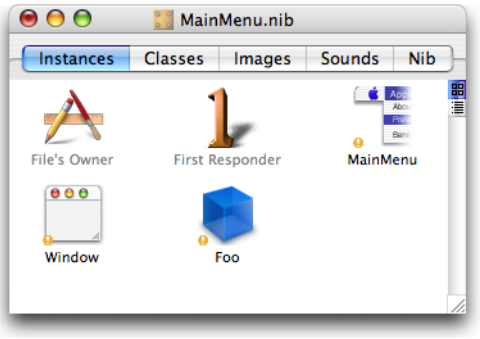

Figure 2.22 An Instance of Foo

Make Connections

A lot of object-oriented programming has to do with which objects need to know about which other objects. Now you are going to introduce some objects to each other. Cocoa programmers would say, "We are now going to set the outlets of our objects." To introduce one object to another, you will control-drag from *the object that needs to know* to the *object it needs to know about*. The object diagram in Figure 2.23 shows which objects need to be connected in your example.

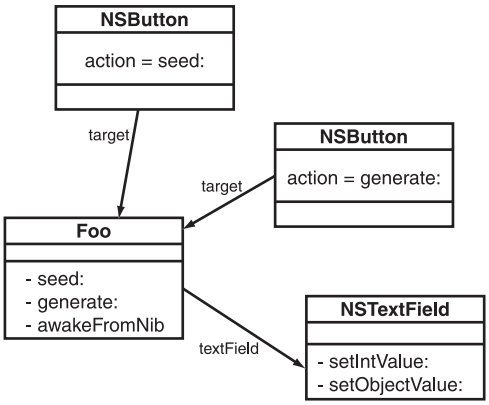

Figure 2.23 Object Diagram

You will set **Foo**'s `textField` instance variable to point to the **`NSTextField`** object on the window that currently says System Font Text. Control-drag from the symbol that represents your instance of **Foo** to the text field. The inspector panel will then appear. Choose `textField` in the view on the left, and click Connect. You should see a dot appear next to `textField` (Figure 2.24).

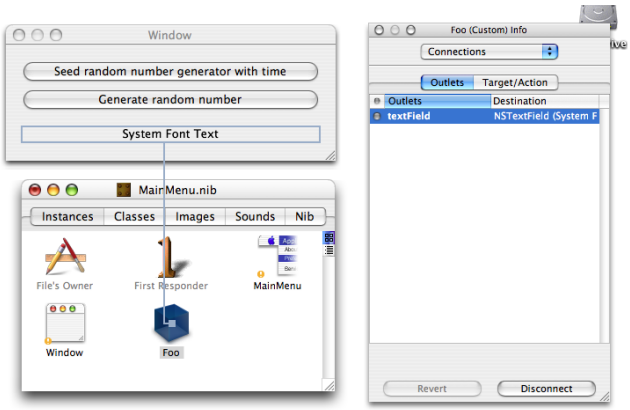

Figure 2.24 Set the textField Outlet

This step is all about pointers: You have just set the pointer `textField` in your **Foo** object to point to the text field.

Now you will set the Seed button's `target` outlet to point to your instance of **Foo**. Furthermore, you want the button to trigger **Foo**'s **`seed:`** method. Control-drag from the button to your instance of **Foo**. Choose the `Target/Action` tab in the inspector and select **`seed:`**. Click the Connect button to complete the connection (or double-click the word **`seed:`**). You should see a dot appear next to **`seed:`** (Figure 2.25).

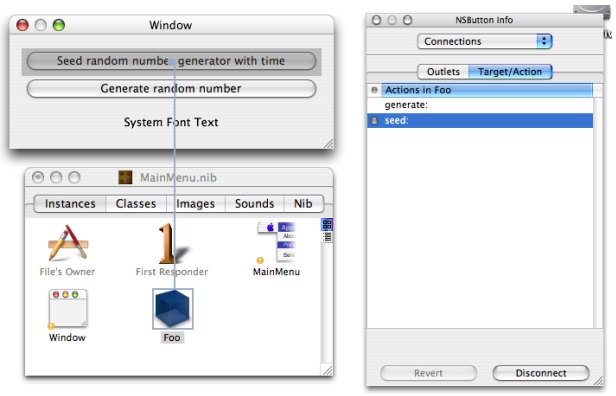

Figure 2.25 Set the Target and Action of the Seed Button

Similarly, you will set the Generate button's `target` instance variable to point to your instance of **Foo** and set its action to the **`generate:`** method. Control-drag from the button to **Foo**. Choose **`generate:`** in the Target/Action view. Double-click on the action name (**`generate:`**) to complete the connection. Note the appearance of the dot (Figure 2.26).

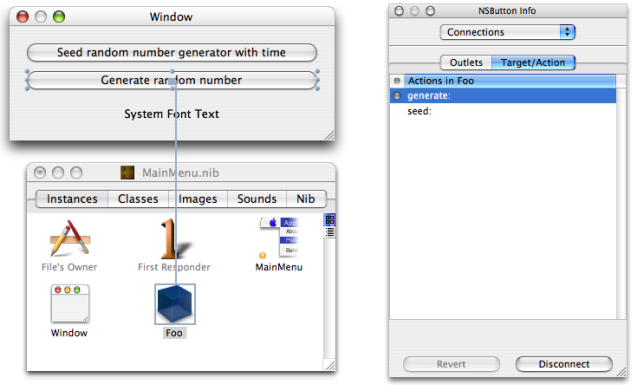

Figure 2.26 Set the Target and Action of the Generate Button

You are done with Interface Builder, so save the file and hide the application. Click on the Xcode icon in the dock to bring Xcode to the front.

Back in Xcode

In Xcode, you will see that `Foo.h` and `Foo.m` have been added to the project. Most programmers would put these files under the Classes group (Figure 2.27). Drag the files into the Classes group if they aren't there already.

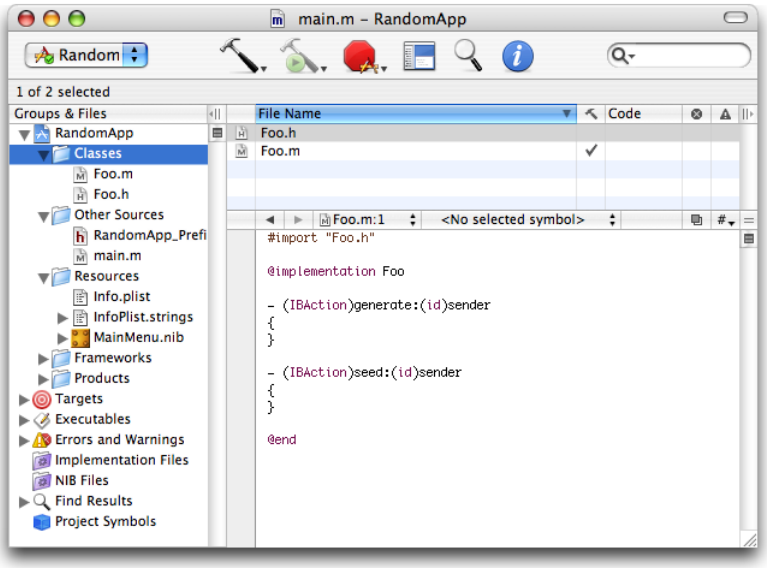

Figure 2.27 The New Class in Xcode

If this is the first time that you are seeing Objective-C code, you may be alarmed to discover that it looks quite different from C++ or Java code. The syntax may be different, but the underlying concepts are the same. For example, in Java a class would be declared like this:

```
import com.megacorp.Bar;
import com.megacorp.Baz;

public class Rex extends Bar implements Baz {
...methods and instance variables...
}
```

This says, "The class **Rex** inherits from the class **Bar** and implements the methods declared in the **Baz** interface."

The analogous class in Objective-C would be declared like this:

```
#import <megacorp/Bar.h>
#import <megacorp/Baz.h>

@interface Rex : Bar <Baz> {
...instance variables...
}
...methods...
@end
```

If you know Java, Objective-C really isn't so strange. Note that like Java, Objective-C allows only single inheritance; that is, a class has only one superclass.

Types and Constants in Objective-C

Objective-C programmers use a few types that are not found in the rest of the C world.

- `id` is a pointer to any type of object.
- `BOOL` is the same as `char`, but is used as a Boolean value.
  `YES` is 1.
  `NO` is 0.
- `IBOutlet` is a macro that evaluates to nothing. Ignore it. (`IBOutlet` is a hint to Interface Builder when it reads the declaration of a class from a `.h` file.)
- `IBAction` is the same as `void`. It also acts as a hint to Interface Builder.
- `nil` is the same as `NULL`. We use `nil` instead of `NULL` for pointers to objects.

Look at the Header File

Click on `Foo.h`. Study it for a moment. You should see that it declares **`Foo`** to be a subclass of **`NSObject`**. Instance variables are declared inside the curly braces.

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

@interface Foo : NSObject
{
   IBOutlet NSTextField *textField;
}
- (IBAction)generate:(id)sender;
- (IBAction)seed:(id)sender;
@end
```

`#import` is similar to the C preprocessor's `#include`. However, `#import` ensures that the file is included only once.

Notice that the declaration of the class starts with `@interface`. The `@` symbol is not used in the C programming language. To minimize conflicts between C code and Objective-C code, Objective-C keywords are prefixed by `@`. Here are a few other Objective-C keywords: `@end`, `@implementation`, `@class`, `@selector`, and `@encode`.

In general, you will find entering code easier if you turn on syntax-aware indention. In Xcode's Preferences, select the Indentation pane. Check the box labeled Syntax-aware indenting, as shown in Figure 2.28.

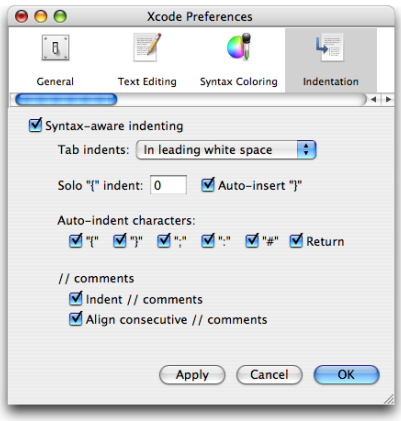

Figure 2.28 The New Class in Xcode

Edit the Implementation File

Now look at `Foo.m`. It contains the implementations of the methods. In C++ or Java, you might implement a method something like this:

```
public void increment(Object sender) {
    count++;
    textField.setIntValue(count);
}
```

In English, you would say, "**`increment`** is a public instance method that takes one argument that is an object. The method doesn't return anything. The method increments the `count` instance variable and then sends the message **`setIntValue()`** to the `textField` object with `count` as an argument."

In Objective-C, the analogous method would look like this:

```
- (void)increment:(id)sender
{
    count++;
    [textField setIntValue:count];
}
```

Objective-C is a very simple language. It has no visibility specifiers: All methods are public, and all instance variables are protected. (Actually, there are visibility specifiers for instance variables, but they are rarely used. The default is protected, and that works nicely.)

In Chapter 3, we will explore Objective-C in all its beauty. For now, just copy the methods:

```
#import "Foo.h"

@implementation Foo

- (IBAction)generate:(id)sender
{
    // Generate a number between 1 and 100 inclusive
    int generated;
    generated = (random() % 100) + 1;

    // Ask the text field to change what it is displaying
    [textField setIntValue:generated];
}

- (IBAction)seed:(id)sender
{
    // Seed the random number generator with the time
    srandom(time(NULL));
    [textField setStringValue:@"Generator seeded"];
}

@end
```

(Remember that `IBAction` is the same as `void`. Neither method returns anything.)

Because Objective-C is C with a few extensions, you can call functions (such as **`random()`** and **`srandom()`**) from the standard C and Unix libraries.

Build and Run

Your application is now finished. To build and run the application, click on the hammer/green circle toolbar item (Figure 2.29). If your app is already running, the toolbar item will be disabled; quit your app before trying to run it again.

If your code has an error, the compiler's message indicating a problem will appear at the view in the upper-right corner. If you click on the message, the erroneous line of code will be selected in the view on the lower right. In Figure 2.29, the programmer has forgotten a semicolon.

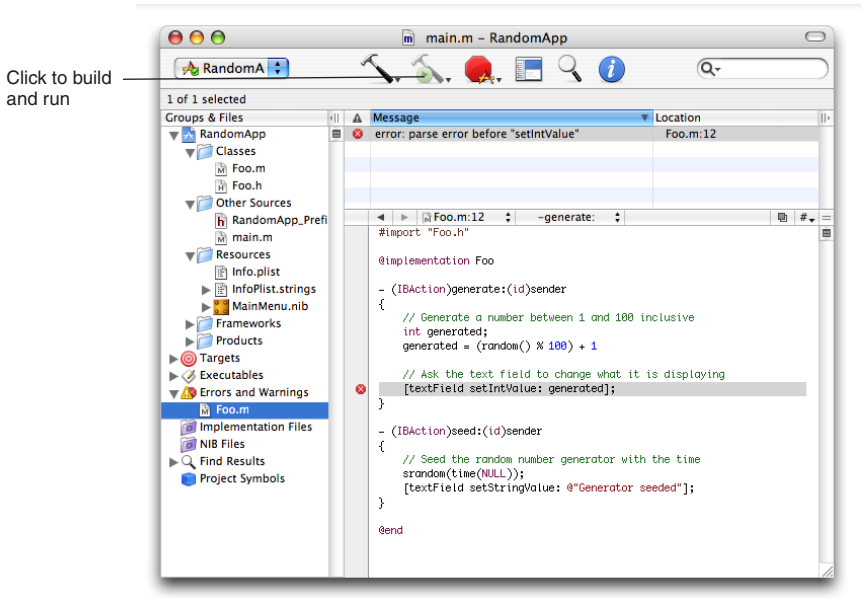

Figure 2.29 Compiling

Launch your application. Click the buttons and see the generated random numbers. Congratulations—you have a working Cocoa application.

awakeFromNib

Notice that your application is flawed: When the application first starts, instead of anything interesting, the words System Font Text appear in the text field. Let's fix that problem. You will make the text field display the time and date that the application started.

The nib file is a collection of objects that have been archived. When the program is launched, the objects are brought back to life before the application handles any events from the user. Notice that this mechanism is a bit unusual—most GUI builders generate source code that lays out the user interface. Instead, Interface Builder allows the developer to edit the state of the objects in the interface and save that state to a file.

After being brought to life but before any events are handled, all objects are automatically sent the message **`awakeFromNib`**. You will add an **`awakeFromNib`** method that will initialize the text field's value.

Add the **awakeFromNib** method to `Foo.m`. For now, just type it in. You will understand it later on. Briefly, you are creating an instance of **NSCalendarDate** that represents the current time. Then you are telling the text field to set its value to the new calendar date object:

```
- (void)awakeFromNib
{
    NSCalendarDate *now;
    now = [NSCalendarDate calendarDate];
    [textField setObjectValue:now];
}
```

The order in which the methods appear in the file is not important. Just make sure that you add them after `@implementation` and before `@end`.

You will never have to call **awakeFromNib**; it gets called automatically. Simply build and run your application again. You should now see the date and time when the app runs (Figure 2.30).

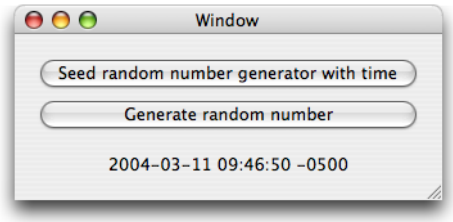

Figure 2.30 Completed Application

In Cocoa, a lot of things (like **awakeFromNib**) get called automatically. Some of the confusion that you may experience as you read this book will come from trying to figure out which methods you have to call and which will get called for you automatically. I'll try to make the distinction clear.

Documentation

Before this chapter wraps up, you should know where to find the documentation, as it may prove handy if you get stuck while doing an exercise later in the book. The online developer documentation is kept in the directory `/Developer/Documentation/`. The easiest way to get to it is by choosing Show Documentation Window from Xcode's Help menu (Figure 2.31).

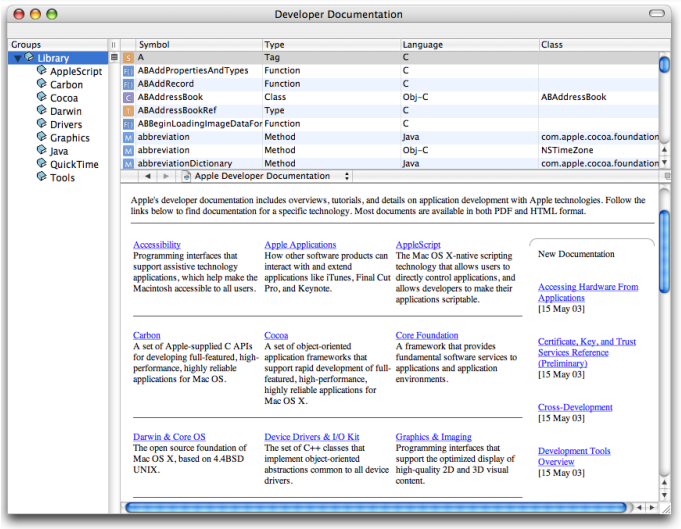

Figure 2.31 The Documentation

What Have You Done?

You have now gone through the steps involved in creating a simple Cocoa application:

- Create a new project.
- Lay out an interface.
- Create custom classes.
- Connect the interface to your custom class or classes.
- Add code to the custom classes.
- Compile.
- Test.

Let's briefly discuss the chronology of an application: When the process is started, it runs the **`NSApplicationMain`** function. The **`NSApplicationMain`** function creates an instance of **`NSApplication`**. A global variable called `NSApp` points to that instance of **`NSApplication`**. `NSApp` reads the main nib file and unarchives the objects inside. The objects are all sent the message **`awakeFromNib`**. Then `NSApp` checks for events. The timeline for these events appears in Figure 2.32.

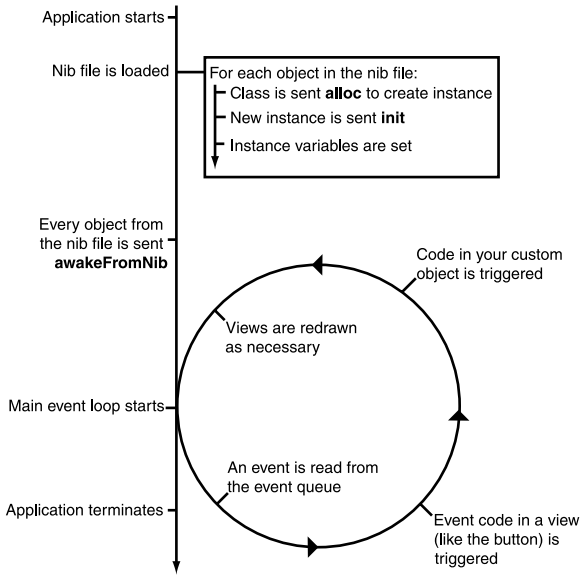

Figure 2.32 A Timeline

When the window server receives an event from the keyboard and mouse, it puts the event data into the event queue for the appropriate application, as shown in Figure 2.33. `NSApp` reads the event data from its queue and forwards it to a user interface object (like a button), and your code gets triggered. If your code changes the data in a view, the view is redisplayed. Then `NSApp` checks its event queue for another event. This process of checking for events and reacting to them constitutes the *main event loop*.

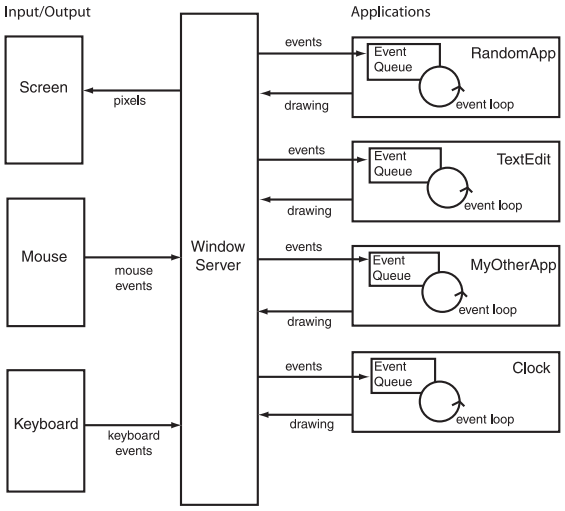

Figure 2.33 The Role of the Window Server

When the user chooses Quit from the menu, `NSApp` is sent the **`terminate:`** message. This ends the process, and all your objects are destroyed.

Puzzled? Excited? Move on to the next chapter so we can fill in some blanks.