Your First Mac App

Developer

Contents

Introduction

This tutorial shows how to create a simple application for Mac OS X and introduces essential aspects of the development process.

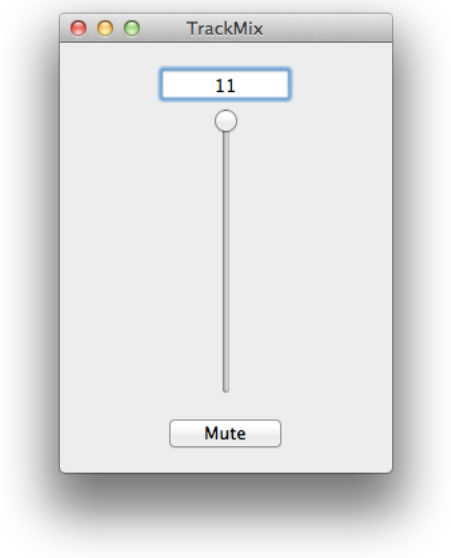
The goal is not to create a finely polished application, or even to get an application onscreen as quickly as possible, but to illustrate the Three T's:

- **Tools:** How you create and manage a project using Xcode
- **Technologies:** How you make your application respond to user input using standard user interface controls
- **Techniques:** The fundamental design patterns and techniques that underlie all Mac OS X development

To provide some context for the application: Imagine you want to write an application that simulates a recording console in a recording studio.

To modify the volume of each track, you have a large collection of sliders. To allow the user to enter a value directly, you provide a text field. And to allow the user to set the volume to zero quickly, you provide a mute button.

To understand how to create such an application, you might start with a simpler version managing just a single track, which has a text field, a slider, and a button—and to begin with you wouldn't actually manage any sound output. Your prototype might thus contain just a single window, like the one below. If you type a value into the text field, the value of the slider updates; if you change the value of the slider, the number in the text field updates. Clicking Mute sets both values to zero.



You should read this document if you are just starting to develop applications for Mac OS X. You must, though, already have some familiarity with the basics of computer programming in general and with Objective-C language in particular. If you haven't used Objective-C before, read *Learning Objective-C: A Primer*.

Important To follow this tutorial, you must have installed the Mac OS X SDK and developer tools available from the Mac Dev Center.

This document describes the tools that are available for Mac OS X Lion—check that your version of Xcode is 4.2.

This tutorial does not discuss issues beyond basic application development; in particular, it does not describe how to submit applications to the App Store.

Organization of This Document

The tool you use to create Mac OS X applications is Xcode—Apple's IDE (integrated development environment). "Creating the Project" (page 8) shows you how to create a new project.

Whereas Xcode is the application you use to create and manage the project, Cocoa is the name given to the collection of APIs that you use to develop programs for Mac OS X. You should typically use the highest level of abstraction available to accomplish whatever task you want to perform. For example, to create a desktop application, you should use the Objective-C-based frameworks that provide the infrastructure you need to implement graphical, event-driven applications.

To use the Objective-C frameworks effectively, you need to follow a number of conventions and design patterns. Following the Model-View-Controller (MVC) design pattern, the application uses a controller object to mediate between the user interface (view objects) and the underlying representation of a track (a model object).

"Understanding Fundamental Design Patterns" (page 15) provides an overview of the design patterns you'll use.

You design the user interface graphically, using Xcode, rather than by writing code. The interface objects are stored in an archive called a nib file.

> **Terminology** Although an interface document may have a `.xib` extension, historically the extension was `.nib` (an acronym for NextStep Interface Builder), so they are referred to colloquially as *nib files*.

The nib file doesn't just describe the layout of user interface elements, it also defines non-user-interface elements and connections between elements. The controller object is represented in the nib file. You make connections from the text field, slider, and button to send messages to update the track's volume, and from the controller to the text field and slider to keep the display in sync with the track's volume. At runtime, when a nib file is loaded the objects are unarchived and restored to the state they were in when you saved the file—including all the connections between them.

"Inspecting the Main Nib File" (page 17) provides an orientation to the application project and briefly explains how an application launches. More important, it introduces the application's nib file and the important elements in the file. Next, "Application Architecture" (page 20) describes the pieces of the application you're going to create, then "Configuring the User Interface" (page 22) shows you how to edit the project's main nib file to add and configure the button, text field, and slider.

"Adding a Track Object" (page 36) shows you how to add a new class to represent features of a track (in this case, just its volume), and how to create an instance of it in the application delegate.

"Completing the Application Delegate" (page 46) shows you how to finish the implementations of the methods associated with the text field, slider, and button. It also shows how to ensure that at application launch time the user interface is consistent with the values it is displaying.

"Application Polish" (page 50) shows you how to make two small changes that make your application behave more elegantly.

"Where to Next?" (page 56) offers suggestions as to what directions you should take next in learning about Mac OS X development.

See Also

To learn more about the technologies in Cocoa, read *Mac OS X Technology Overview*.

Creating, Building, and Testing the Project

The tool you use to create Mac OS X applications is Xcode—Apple's IDE (integrated development environment). You can also use it to create a variety of other project types, including command-line utilities and (if you also install the iOS SDK) iOS applications.

Create the Project

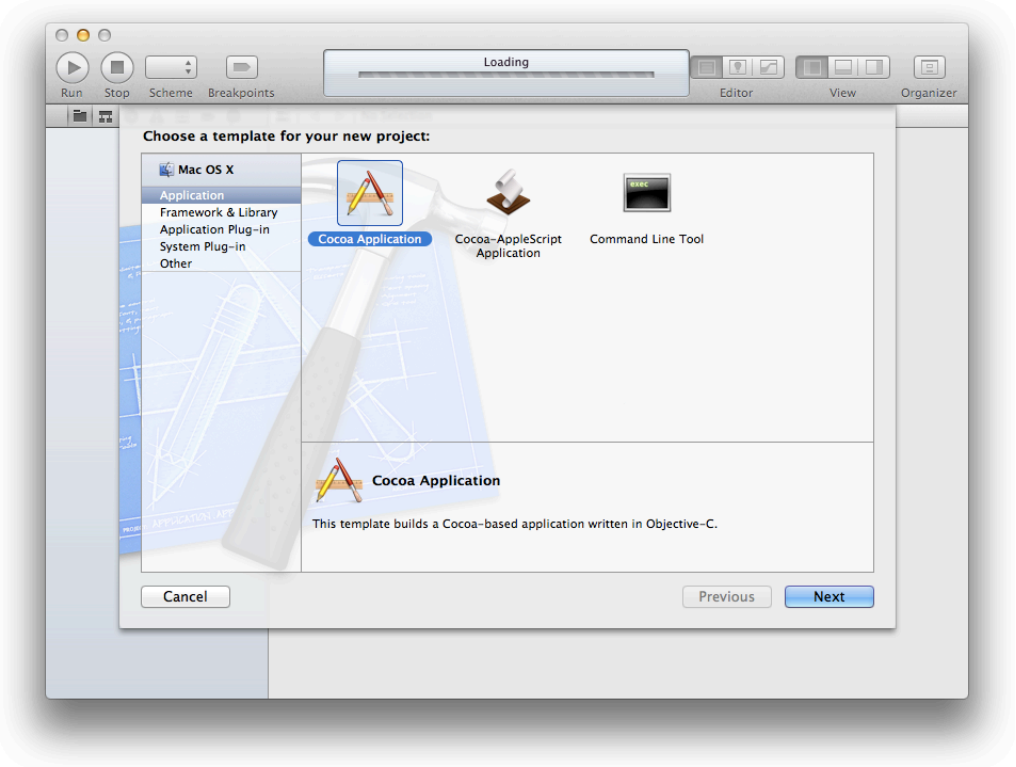
When you create a project, Xcode makes a new folder for the project at the path you provide. It makes several new files in that folder to give you a working template, which you will use to create an application.

Note In code listings, comments included in Xcode template files are not shown.

To create a new project . . .

1. Launch Xcode (by default it's in `/Developer/Applications`), then create a new project by choosing File > New > New Project.

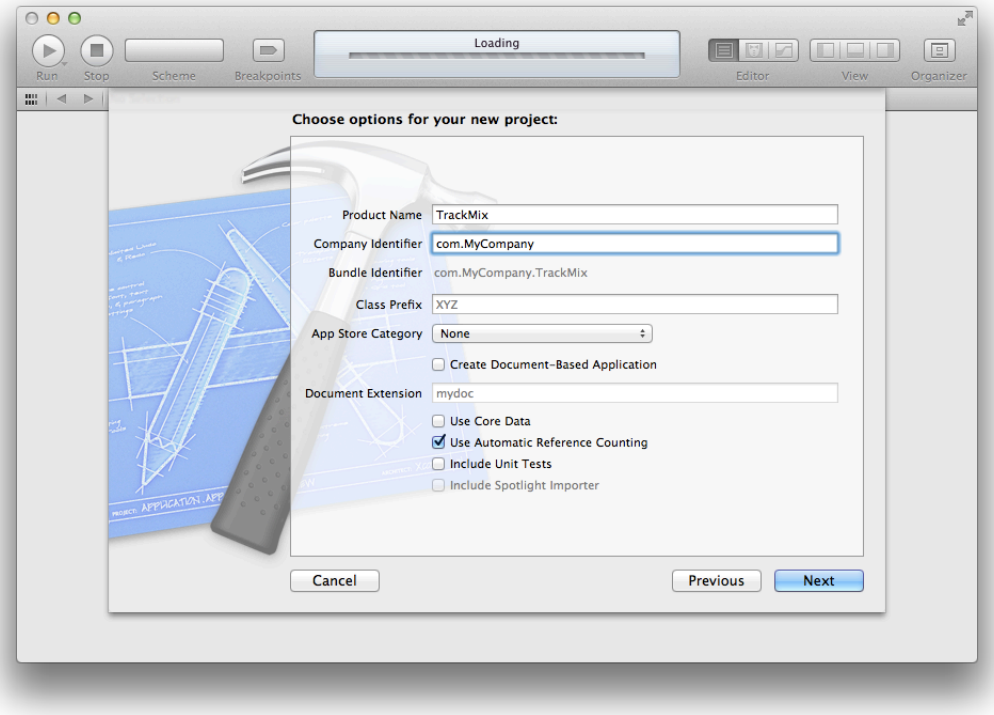
The new project window appears.



If you have also installed the iOS SDK, you will see two sections on the left hand side of the sheet.

2. In the Mac OS X section on the left, select Application.
3. In the project type section on the left, select Cocoa Application and click Next.

You should see the options that you can configure for the project.



4. Choose the following options:
    - In the Product Name field, type `TrackMix`.

      This is the name your application will have.

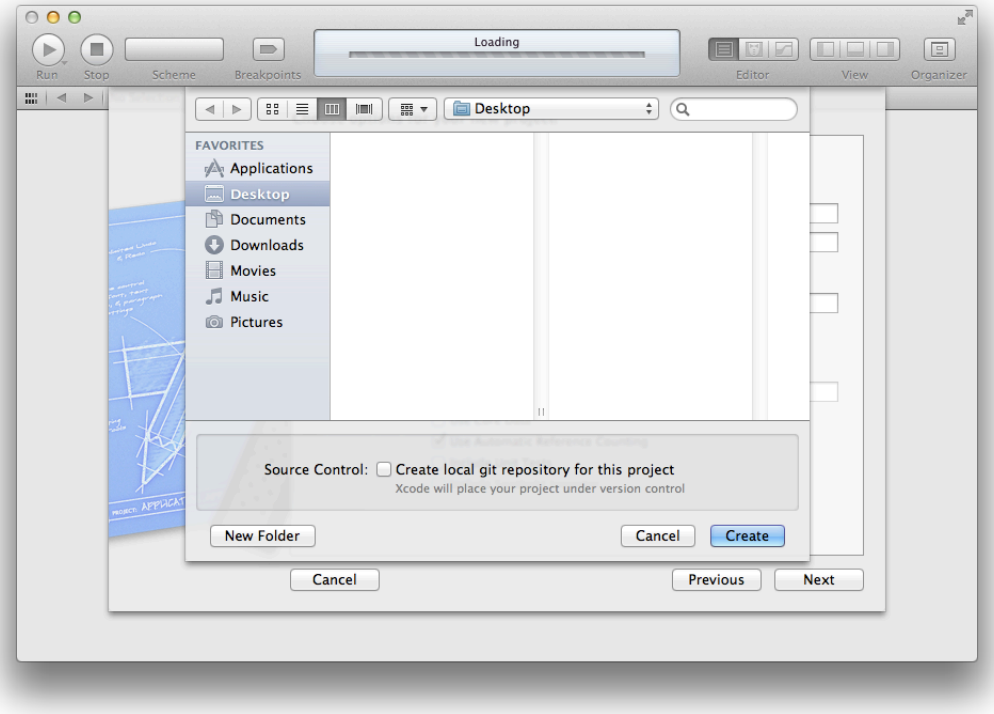
      > **Note** The remainder of the tutorial assumes that you named the project `TrackMix`, so the class that you will add code to in this tutorial is called `TrackMixDelegate`. If you name your project something else, then this class will be called *YourProjectName* `AppDelegate`.

    - In the Company Identifier field, type the identifier for your company, or `com.MyCompany`.
    - In the App Store Category pop-up, choose None.
    - Do not change any of the other options.

      This is not a document-based application, and it does not use Core Data. Although use of unit tests is considered to represent best practice for Cocoa application development, unit testing is not discussed in this introductory tutorial.

5. Click Next.

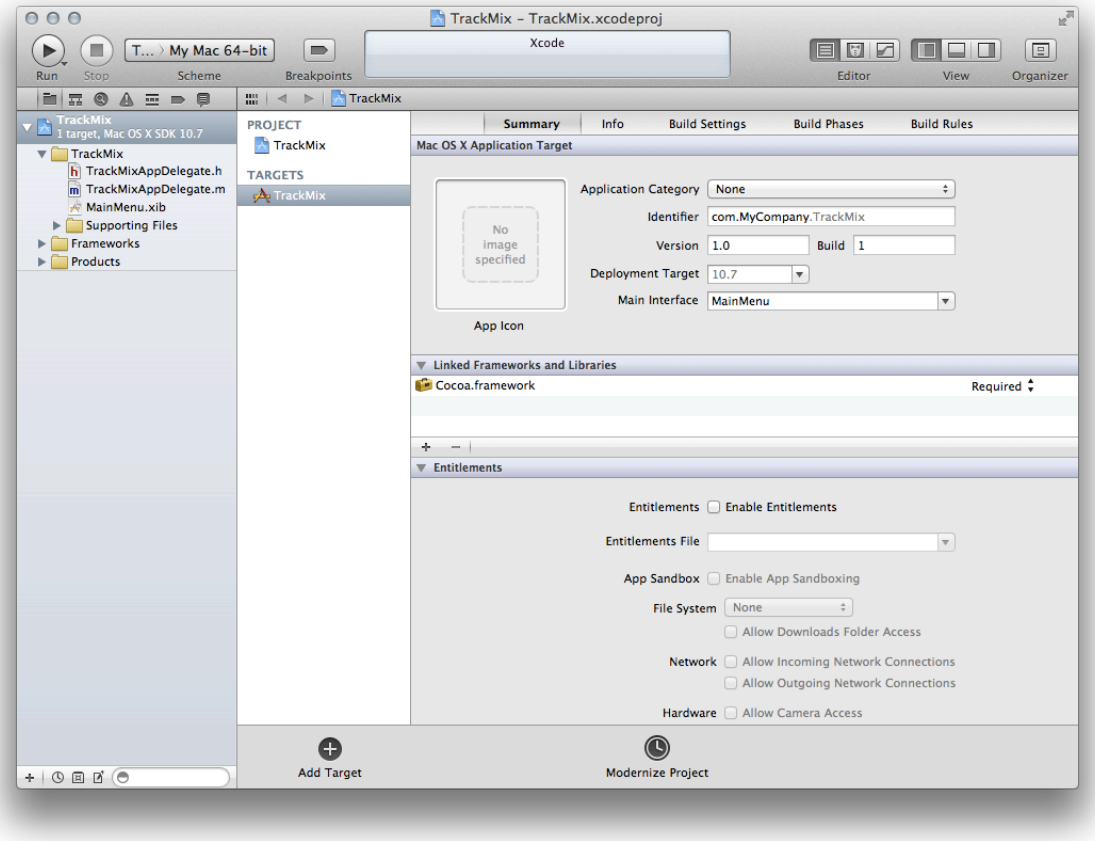
   A sheet appears to allow you to select where your project will be saved.



6. Select a suitable location for your project (such as the Desktop or a custom Projects directory) and click Save.
7. Deselect the option to create a local repository.

   Although it's a good idea to use version control when developing an application, there is no need to do so for this project unless you want to. The tutorial does not explain how to use source control systems.

After creating the project, you should see a window like this:



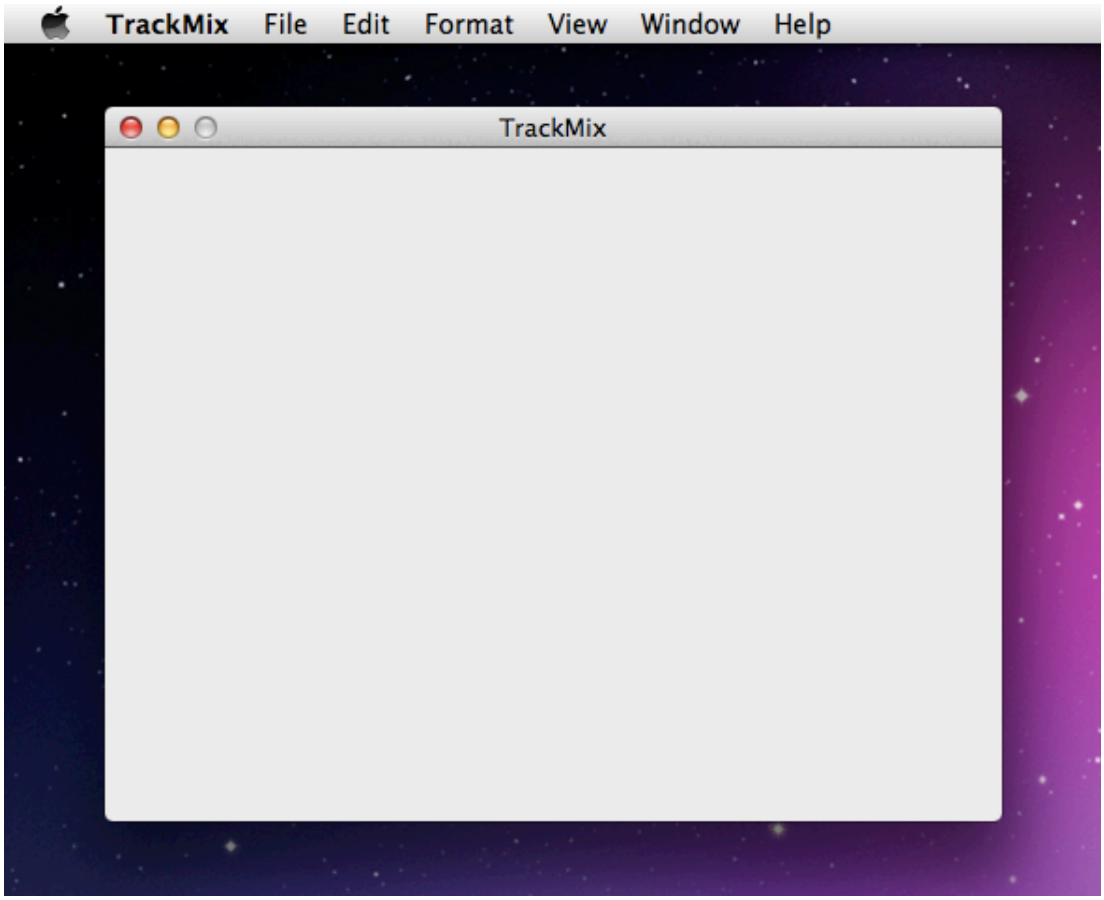
Build, Run, and Test

You can now build and run your application to see what the project provides for you out of the box.

To build and test the application . . .

1. Choose Product > Run or click the Run button in the toolbar.

Xcode should build your project and launch the application. When your application starts up, it should have a standard menu bar and display a single window.



2. Test the application.

   You can move and resize the window, however if you close the window, there is no way to get it back. You should also find that menus display when you click on them, and if you choose TrackMix > About TrackMix, an About window is displayed.

3. Quit the application.

If you haven't used Xcode before, take a moment to explore the application. You should read *Xcode 4 User Guide* to understand the organization of the project window and how to perform basic tasks like editing and saving files. You can also use the Help menu to get assistance with performing various tasks.

Recap

In this chapter, you created a new project using Xcode and built and ran the default application. You also looked at some of the basic pieces of a project and a nib file, and learned how the application starts up. In the next chapter, you'll learn about the design patterns you use when developing a Cocoa application.

Understanding Fundamental Design Patterns

In this chapter, you learn about the main design patterns you'll use in Cocoa applications. There's not much practical to do in this chapter, but the design patterns are fundamental for writing an application , so allow some time to read and think about them.

The main patterns you're going to use in this tutorial are:

- Model-View-Controller
- Delegation
- Target-Action

Here's a summary of these patterns and an indication of where they'll be used in the application.

Model-View-Controller

The Model-View-Controller (MVC) design pattern sets out three roles for objects in an application:

- **Model objects** represent data such as SpaceShips and Weapons in a game, ToDo items and Contacts in a productivity application, or shapes such as Circles and Squares in a drawing application.

  In this application, the model is very simple—a Track object that manages the volume of the track.

- **View objects** know how to display data (model objects) and may allow a user to edit the data.

  In this application, you need a window to contain the other views—a text field and slider to capture information from the user, and a button.

- **Controller objects** mediate between models and views.

  In this application, the application delegate is also the controller object. It takes the data from the text field and slider, updates the Track object, and updates the user interface appropriately. It also responds to the button being pressed to set the track's volume to zero and again update the user interface.

Delegation

Delegation is a design pattern in which one object sends messages to another object—specified as its delegate—to ask for input or to notify it that an event is occurring. Delegation is often used as an alternative to class inheritance to extend the functionality of reusable objects. For example, a window's delegate might ask if it's all right to resize to a particular size. The delegate might reply with a value equivalent to no if the window is too big or too small for its content; the delegate might also instead specify a different size so as to constrain the window content to a particular aspect ratio (see the `windowWillResize:toSize:` message.)

Delegate methods are typically grouped into a protocol. A protocol is basically just a list of methods. The delegate protocol specifies all the messages an object might send to its delegate. If a class conforms to (or *adopts*) a protocol, it guarantees that it implements the required methods of a protocol. (Protocols may also include optional methods).

In this application, the application object tells its delegate that the main startup routines have finished by sending it the `applicationDidFinishLaunching:` message. The delegate is then able to perform additional tasks if it wants.

Target-Action

Target-action is a design pattern that enables an object to respond to a user event by sending a message to another object. The other object interprets the message and handles it as an application-specific instruction. For example, in this application a button (the object) tells the controller to send a message to the track (the target), setting the volume to zero (the action).

Other Design Patterns

Cocoa makes pervasive use of a number of design patterns beyond those just described.The design patterns chapter in *Cocoa Fundamentals Guide* provides a comprehensive overview of all the design patterns you use in Cocoa application development—consider it to be essential reading. Understanding and using the patterns will make it easier for you to use the various technologies Cocoa provides, and to transfer skills you've learned in one area to another. Following the patterns also means that your code will work better with Cocoa.

Inspecting the Main Nib File

The template project you created already sets up the basic application environment. It creates an application object, connects to the window server, establishes the run loop, and so on. Most of the work is done by the `NSApplicationMain` function, called by the `main` function in `main.m`. It creates a singleton instance of `NSApplication`. It also scans the application's `Info.plist` file.

The **information** property list or `Info.plist` file is a property list that contains information about the application, such as its name and icon. The name of this file is typically **ProjectName**`-Info.plist`.

Select the Info.plist file from the Supporting Files group in the project window to display the file in the editor. Among other key-value pairs you should see:

Key	Value
`Main nib file base name`	MainMenu

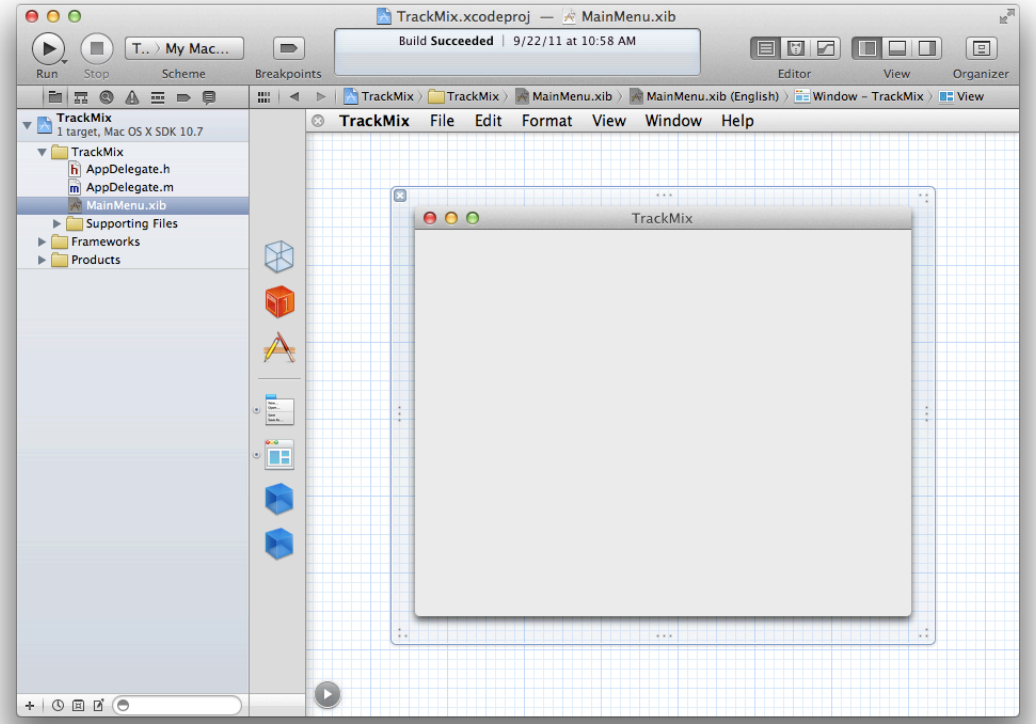
"Main nib file base name" specifies the name of the nib file that the application object should load when it launches. Nib files are an archive of user interface elements and other objects. The main nib file usually contains the parts of your user interface that are visible the entire time you application is running, such as the menu bar.

To look at the nib file, and the window in the nib file . . .

1. Click `MainMenu.xib` under the TrackMix group in the project navigator.

   The file has the extension `.xib` but by convention it is referred to as a *nib file*. Xcode displays the file in the editor area.

2. To display the application window, click the window icon in the sidebar.



The nib file contains several items split into two groups by a dividing line. Above the line are placeholders—objects that are not created as part of the nib file itself, but exist externally:

- The File's Owner proxy object (the pale blue cube).

  In this case, the File's Owner object will actually be the `NSApplication` instance.

- The First Responder proxy object (the opaque red cube).

  The First Responder object is not used in this tutorial but you can learn more about it by reading "Event Architecture" in *Cocoa Event-Handling Guide*.

- A representation of the singleton instance of `NSApplication` (the "A" icon). For various reasons, having a representation of the application instance is sometimes useful in a nib file. In this case it can be a little confusing since it means there are actually two icons that represent the application object—this and the File's Owner.

Below the line are objects created as part of the nib file:

- A menu

This object is the application's main menu displayed in the menu bar.

- A window with a plain gray background

  This object is the window you see when the application launches

- An instance of `TrackMixAppDelegate` (a dark blue cube), set to be the application's delegate.

  When the application object has completed its setup, it sends its delegate an `applicationDidFinishLaunching:` message. This message gives the delegate an opportunity to configure the user interface and perform other tasks before the application is displayed.

- An instance of `NSFontManager` (a dark blue cube).

  This object manages the application's font menu, but you don't use it in this example.

You'll edit the nib file in "Configuring the User Interface" (page 22); first, though, you need to know more about the application architecture—what is the goal in creating the application?

Application Architecture

The goal of the application in this tutorial is to manage the volume of a track. The track is a model object, represented by an instance of the `Track` class (this will be a custom class you implement later). There are three view objects: a slider, text field, and button. They allow the user to see and/or change the current volume setting of the track. Following the Model-View-Controller design pattern, mediating between these three view objects is a controller object. In this application, the controller is also the application object's delegate.

> **Terminology and roles** In this tutorial, the same object acts as both the controller and the application delegate, because it is a convenient place for this logic. It is referred to for the rest of this tutorial as the application delegate.
>
> In other applications, this is not necessarily the case. There may be other (or even multiple) controller objects, in addition to a separate application delegate. The application object may not even have a delegate.

You have to implement the application delegate and the track classes. You also have to define the user interface. To allow the various objects to communicate with each other, you need to establish connections between them. You define the user interface and the connections between most of the objects in the nib file.

A connection in a nib file from one object to another, such as from the application delegate to the slider, is called an outlet. Outlets are represented by declared properties, and are discussed in more detail in a later section ("Outlets" (page 26)).

- The application object has an outlet to its delegate. You don't need to set this outlet. It is established for you when the project is created.
- The application delegate will have outlets for the track and to the text field and slider. It doesn't need a reference to the button because the button's display doesn't need to be updated.
- The text field and slider will be connected to the application delegate. When the user changes their values, they need to send a message to the controller to tell it to update the track's volume appropriately.
- The button will be connected to the application delegate. When the user clicks the button, it needs to send a message to the application delegate to set the track's volume to zero.

There are a couple of other things to notice. First, the track doesn't have a reference to the application delegate, and none of the views has a reference to the track. Second, you don't create custom control objects (text field, button, and slider). You don't add code to any of these to update the track's volume—that code goes in the application delegate.

You'll configure the user interface and establish connections between the various objects in "Configuring the User Interface" (page 22).

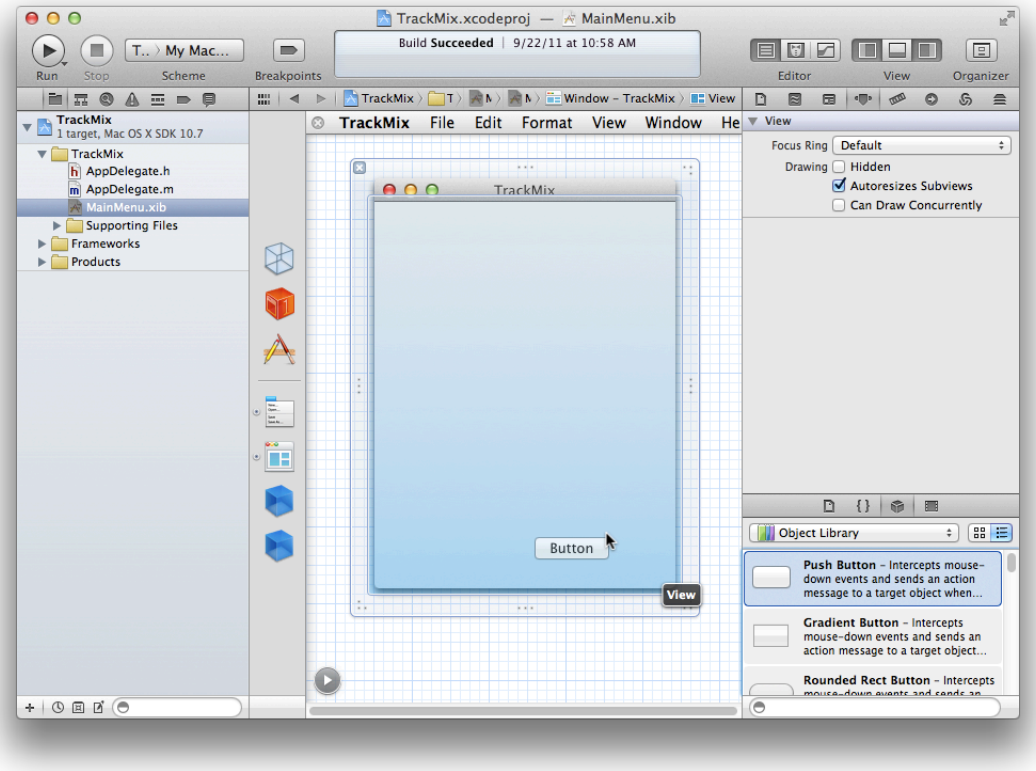
Configuring the User Interface

Xcode provides a library of objects that you can add to a nib file. Some of these are user interface elements, such as buttons and text fields. Others are controller objects such as view controllers. Your nib file already contains a window—now you need to add the button, slider, and text field, and then make connections to and from the application delegate. You add user interface elements by dragging them from the Object Library.

If you build and run your application now, when it launches you should see the user interface elements as you positioned them: If you click the button it should highlight, you can type new values in the text field, and you can move the slider. At the moment, though, changing the slider value doesn't update the text field and vice versa, and clicking the button doesn't set the text field and slider values to zero. To remedy this and add other functionality, you need to make appropriate connections to and from the application delegate.

Create and Configure the User Interface

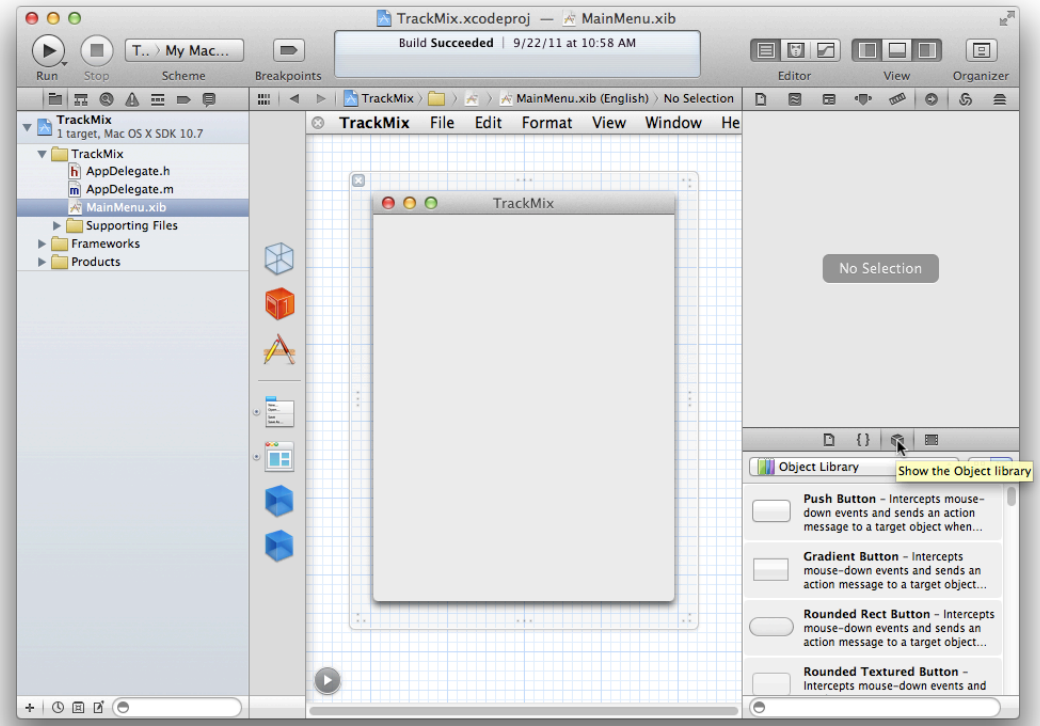
You can drag view items from the library and drop them onto the view, just as you might in a drawing application.



To create and configure the user interface by editing the nib file . . .

1. In the project navigator, under the Resources group, select the MainMenu nib file.
2. If necessary, display the window by clicking its icon.
3. Make the window narrower using the resize handles.
4. Display the utility area, by clicking the rightmost icon in the rightmost group of three icons in the toolbar. Selecting the icon displays the utility area on the right of the project window.

5. Select the Object Library.



6. Add a text field (`NSTextField`), a vertical slider (`NSSlider`), and a button (`NSButton`) to the window.

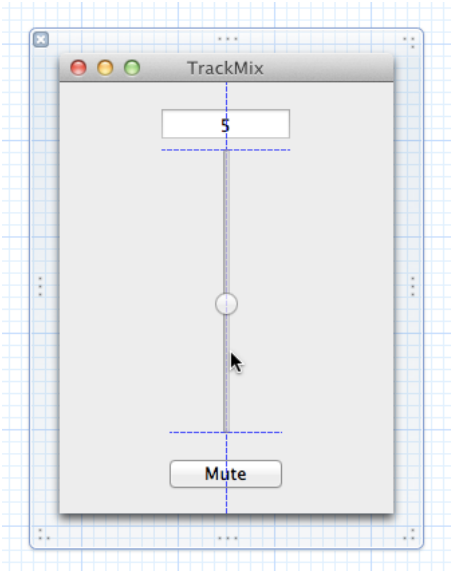
   Use the search field at the base of the utility area to find the objects you want easily.

You resize the items using resize handles, where appropriate, and reposition them by dragging. As you move items within the view, alignment guides are displayed as dashed blue lines.

To complete the layout of the user interface elements . . .

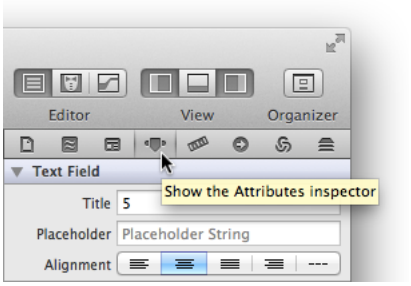
1. Move the elements so that they are centered vertically in the window and are the appropriate distance from each other and from the edges of the window

The window should look something like this:



2. Add a default string `5` to the text field by selecting the text field then changing its title in the Attributes inspector.

   There are a number of different inspector panes; select the Attributes inspector by clicking the icon that looks like a slider.



   Alternatively, double-click in the text field and enter the number directly.

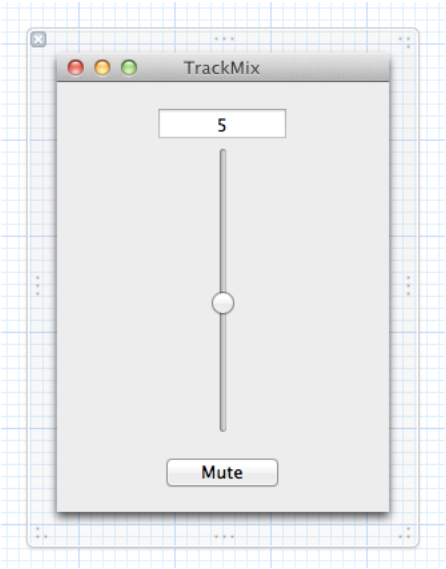
3. Use the Attributes inspector to set the text alignment for the text field to centered.
4. Select the slider and use the Attributes inspector to set its minimum value to zero (0), maximum value to 11, and current value to 5 (to match the text field).
5. Use the Attributes inspector to set the slider to be continuous.

   As a result, the slider will send messages while it's being moved rather than only when a new value has been set.

6. Set the title for the button by double-clicking inside the button and typing `Mute`.

7. Save the file.

The configured window should look something like this:



Make Connections

You need to make two sorts of connections: outlets, which provide a reference from one object to another, and target-action, which tells control objects what message to send to which object.

Outlets

Notice that the application delegate already has one outlet defined—for the window. The outlet is defined as a declared property with an additional keyword, `IBOutlet`.

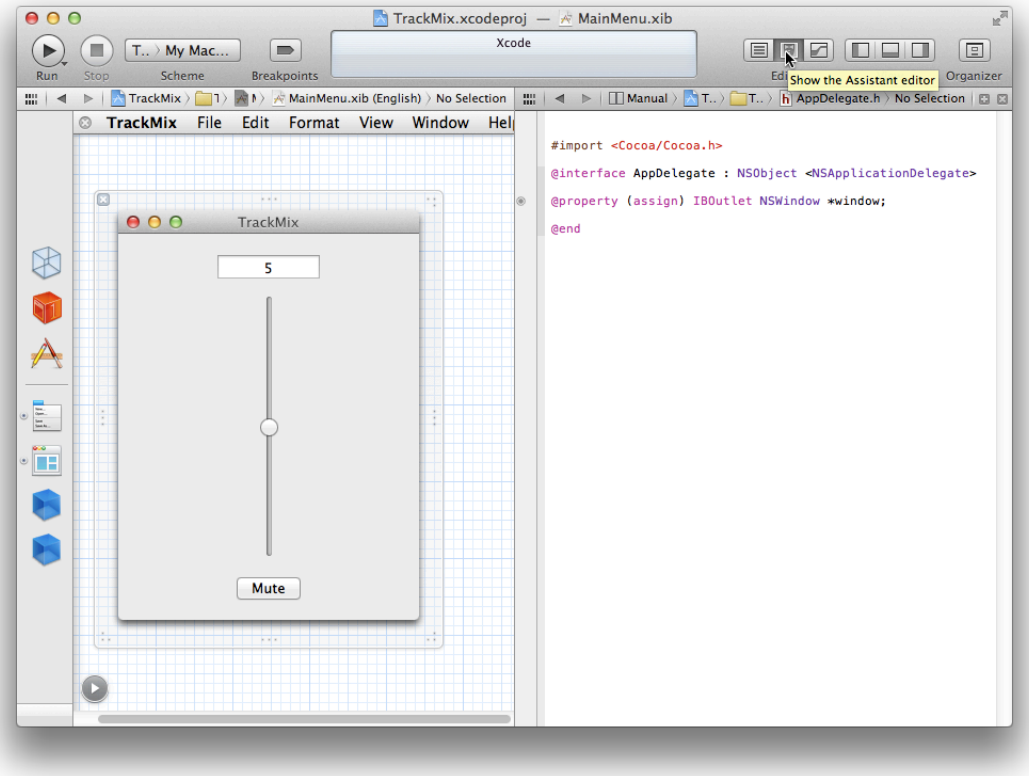
- Properties are described in the "Declared Properties" chapter in *The Objective-C Programming Language*.This declaration specifies that an instance of `TrackMixAppDelegate` has a property that you can access using the getter and setter methods `window` and `setWindow:`, respectively, and that the instance has a weak relationship to the property.
- `IBOutlet` is solely a hint to Xcode that this property should be exposed as an outlet in a nib file. It's actually a `#define` macro that evaluates to nothing.

You need to add to the header file outlet declarations for the slider and text field, and in the nib file establish connections from the application delegate to the slider and text field instances. You can connect an outlet in a nib file and write the required corresponding code at the same time using the assistant editor.

To add outlets . . .

1. With the nib file in the main editor, display the assistant editor.

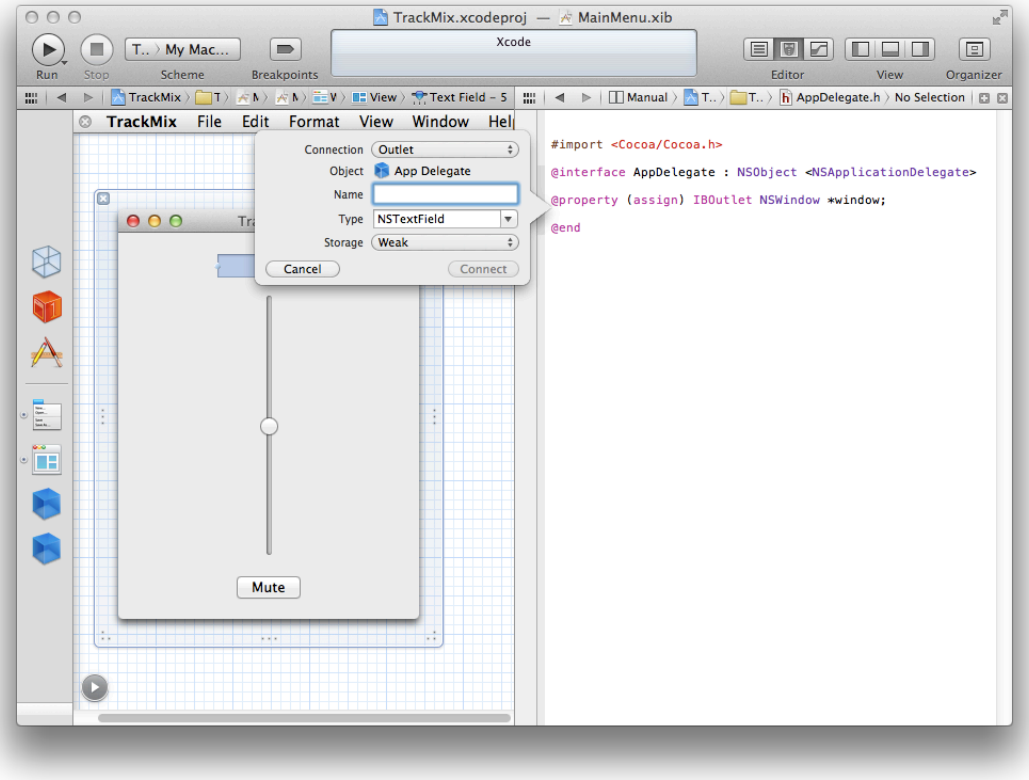
   The editor should show the header file for the application delegate (`TrackMixAppDelegate.h`)—if the header file doesn't, select it from the jump bar.



2. In the nib file, Control-drag from the text field to whitespace between the `@property` declaration and the `@end` symbol.

   When you release the mouse, Xcode presents a dialog so that you can configure the connection.

3. Using the connection dialog, configure an outlet connection called `textField` of type `NSTextField`, then click Connect.



Xcode adds an appropriate property declaration to the header file:

```
@property (weak) IBOutlet NSTextField *textField;
```

It also adds a corresponding `@synthesize` statement to the implementation file (`TrackMixAppDelegate.m`), which tells the compiler to generate the instance variable and accessor methods for this property:

```
@synthesize textField = _textField;
```

Tip There is no need to give the properties the same name as the type of object to which a connection is made—you could call the property for the text field outlet `aardvark` if you wanted—but it helps to call it something that helps to identify its purpose. Property names should, however, begin with a lower-case letter (an exception would be `URL`, for example).

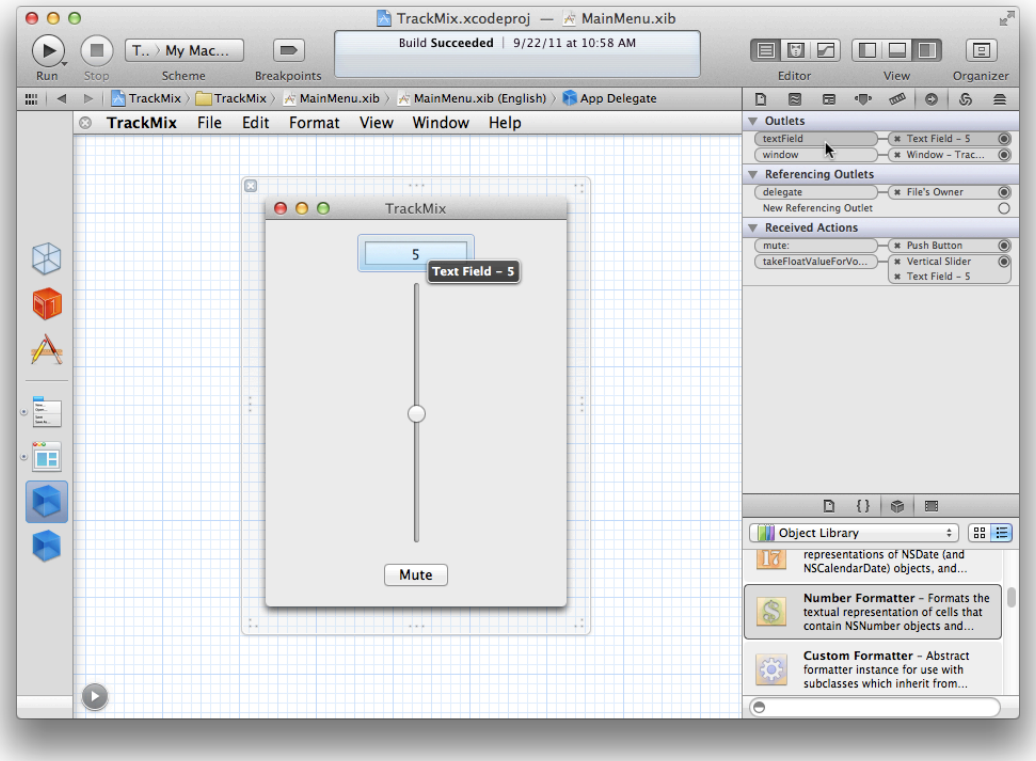
There is nothing in Cocoa that enforces this convention. Naming conventions, though, are a pervasive aspect of Cocoa, and are a vitally important feature. If you disregard this and other conventions, as you adopt other Cocoa technologies, you may find that your application behaves in unexpected ways.

Using the **connection inspector**, you can see the connection that was made.

To use the connection inspector to see the connection you made . . .

1. Select the blue cube icon for the application delegate object, then display the connection inspector.
2. Display the utility area, then select the connection inspector icon (a circle containing an arrow).

   The new connection is the `textField` outlet.



> **Note** Because you dragged from the text field to the source file, it may seem counterintuitive to select the application delegate object. It's important to realize, though, that the outlet was added to the application delegate and set to point to the text field. If you inspect the text field, you should notice that the Referencing Outlets section of the inspector shows `textField` coming from the application delegate, in the same way that the application delegate in the image above shows a referencing outlet called `delegate` coming from the File's Owner.

Follow similar steps to add an outlet for the slider.

To add an outlet for the slider . . .

1. In the nib file, Control-drag from the slider to whitespace between the last `@property` declaration and the `@end` symbol.
2. Using the connection dialog, configure an outlet connection called `slider` of type `NSSlider`, then click Connect.

Target-Action Connections

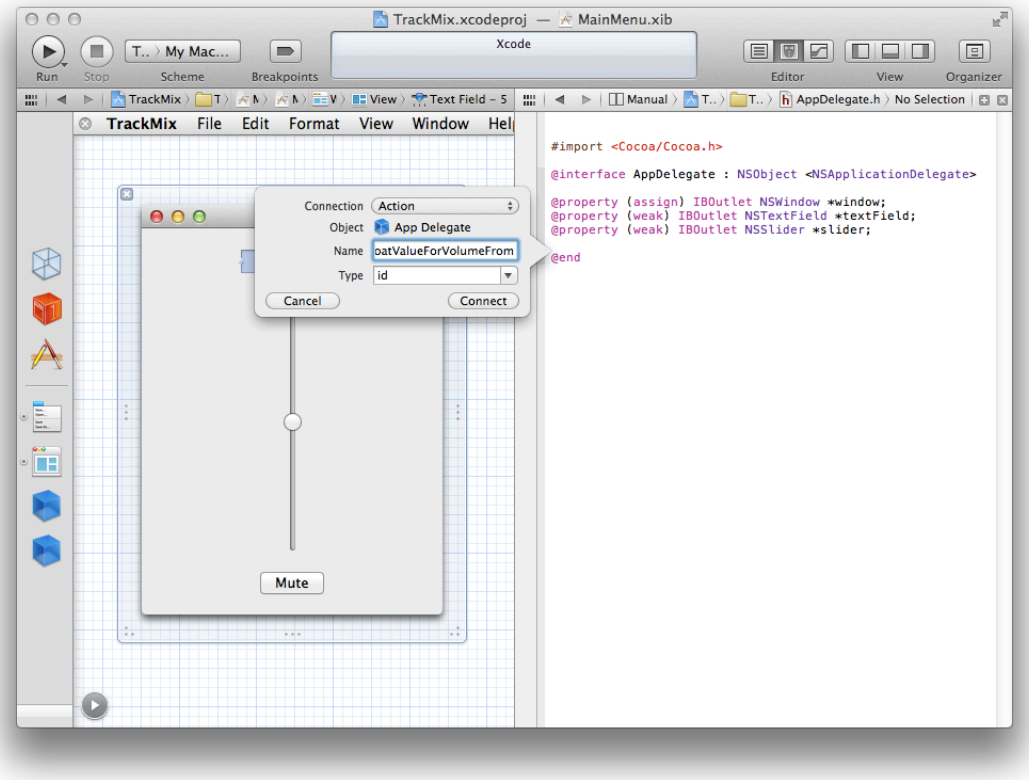
You can make target-action connections for controls (buttons, sliders, and so on) using the same technique as for outlets. When you make the connection, you specify two things: the object to which a control should send a message (the target), and the name of the message it should send (the action).

You need to establish three connections in total, all to the application delegate object: The text field and slider both invoke the same method, and the button invokes another method. Start with the text field.

To make the target action connections . . .

1. In the nib file, Control-drag from the text field to whitespace in the area in the header file between the last `@property` declaration and the `@end` symbol.
2. In the connection dialog, from the Connection pop-up menu select Action.

3. Configure an action connection called `takeFloatValueForVolumeFrom:` with parameter type `id`, then click Connect.



Make sure you set the value of the Type field to `id`. The type is the class of the object expected to send the message; `id` means "an object of any class." If you leave the type as `NSTextField`, you won't be able to connect the slider to the same method.

When you establish a target-action connection like this, Xcode:

- Adds an appropriate method declaration to the header file:

```
- (IBAction)takeFloatValueForVolumeFrom:(id)sender;
```

- Adds a corresponding stub implementation to the implementation file (`TrackMixAppDelegate.m`):

```
- (IBAction)takeFloatValueForVolumeFrom:(id)sender {
}
```

`IBAction` is a special keyword that is used only to tell Interface Builder to treat a method as an action for target-action connections. It's actually a `#define` macro that evaluates to `void`.

Tip In terms of the Objective-C language, there is nothing special about the method name `takeFloatValueForVolumeFrom:`. You could have called the method whatever you want provided it has the correct signature (that is, taking a single argument with a return type of `IBAction`), and any object could implement this method to do whatever it wants.

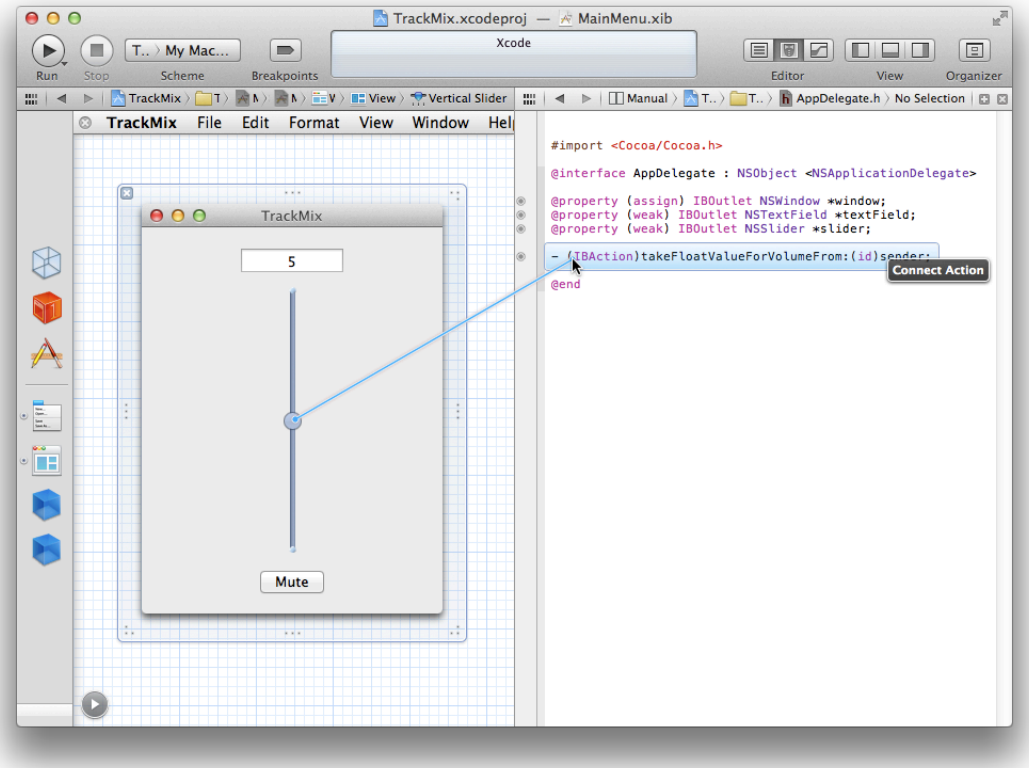
In practice, however, the naming convention here indicates what's expected of the target object. In its implementation of the method, the target should send a `floatValue` message to the sender to get a new value.

You can now connect the button and the slider. You need to need to create a new action (called `mute:`) for the button in the same way that you did for the text field. Because the slider invokes the same `takeFloatValueForVolumeFrom:` method as the text field, you simply connect it to the existing declaration.

To connect the slider . . .

- In the nib file, Control-drag from the slider to the existing declaration of the `takeFloatValueForVolumeFrom:` method in the header file.

Xcode highlights the method declaration and shows a Connect Action label. Release the mouse button to make the connection.



To connect the button . . .

1. In the nib file, Control-drag from the button to whitespace in the area in the header file between the `@property` declarations and the `@end` symbol.
2. In the connection dialog, configure an Action connection called `mute:` of type `id`, then click Connect.

Check your Progress

Your interface file should look similar to this:

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


@interface TrackMixAppDelegate : NSObject <NSApplicationDelegate>
```

```
@property (assign) IBOutlet NSWindow *window;
@property (weak) IBOutlet NSTextField *textField;
@property (weak) IBOutlet NSSlider *slider;

- (IBAction)takeFloatValueForVolumeFrom:(id)sender;
- (IBAction)mute:(id)sender;

@end
```

It doesn't matter in what order the properties and methods are declared; they must, though, be after the `@interface` line and before `@end`.

So that you can test the project at the end of this section, in the implementation file, `TrackMixAppDelegate.m`, add log statements to the action methods.

To add log statements to the action methods . . .

1. Display the implementation file in the text editor by selecting `TrackMixAppDelegate.m` in the navigator area in the left pane.
2. Add a single `NSLog` statement to the `mute:` method to indicate that the method has been invoked.

```
- (IBAction)mute:(id)sender {
    NSLog(@"received a mute: message");
}
```

3. Implement the `takeFloatValueForVolumeFrom:` method as follows:

```
- (IBAction)takeFloatValueForVolumeFrom:(id)sender {

    NSString *senderName = nil;
    if (sender == textField) {
        senderName = @"textField";
    }
    else {
        senderName = @"slider";
    }
```

```
    NSLog(@"%@ sent takeFloatValueForVolumeFrom: with value %1.2f",
senderName, [sender floatValue]);
}
```

The method checks which control was the sender and logs a message indicating both the sender and its current float value. The `NSLog` function logs a string using the format specified by the format string. (This is very similar to the C `printf` function.) The string `%@` indicates that a string object should be substituted. To learn more about strings, see *String Programming Guide*.

The method combines two fairly common patterns in Cocoa: determining the sender of a message and sending a message back to it. In this case, it's not necessary to know which object sent the message—all that's needed is its float value—but it serves to illustrate the point.

Test the Application

You can now test the application.

To test the application . . .

- Click the Run button in the toolbar to build and run the project.

  You should find that when you move the slider or press the button, appropriate messages are logged. To cause the text field to send a message, press Return when the focus is in the field.

A defect of this example is that the user interface is not synchronized (that is, the text field and slider may have different values). You could synchronize the user interface at this stage by sending suitable messages (`setFloatValue:`) to the controls in the current implementation. If you want a small challenge, try doing that now. The goal of the tutorial, however, is to keep the user interface synchronized with the value of a track object. So the next step, in the next chapter, is to implement the `Track` class.

Recap

You added instance variables and property declarations, and a declaration for the action method, to the view controller class interface.

Adding a Track Object

In this application, there are two custom classes. The Xcode application template provided an application delegate class, and an instance was created in the nib file. You now need to implement a `Track` class and create an instance of it in the application delegate.

Create Header and Implementation Files

The first task is to create header and implementation files for the new class. The class inherits from `NSObject`.

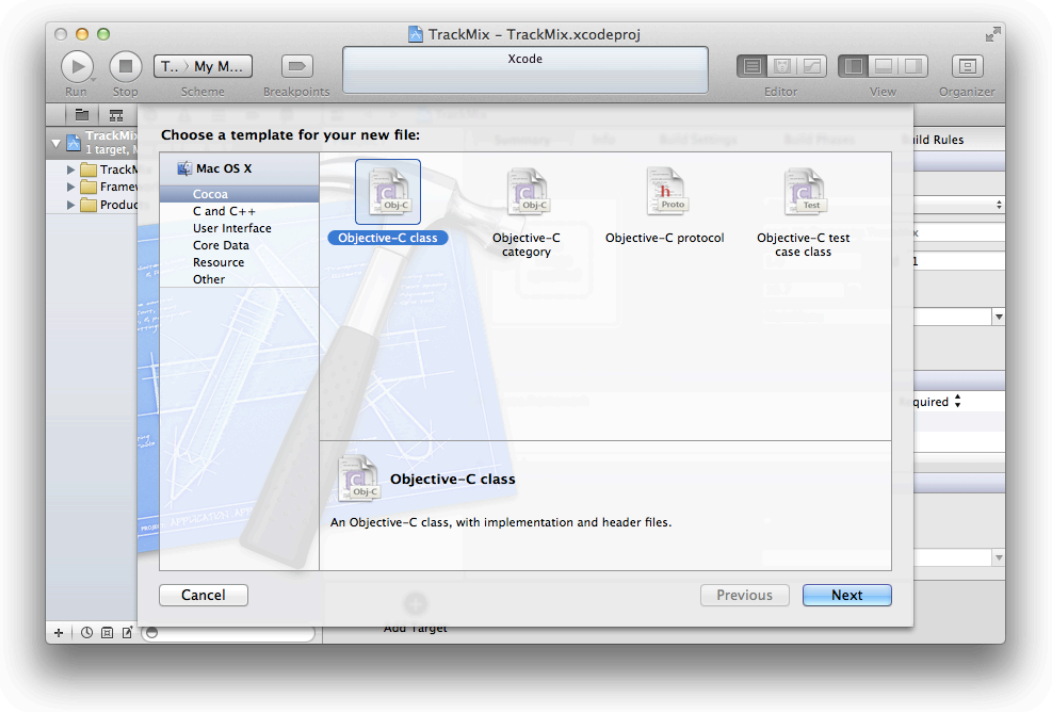
To create the files for the new class . . .

1. In Xcode, in the project organizer select the Classes group folder.

   When you add new files in the next steps, they are added to the project organizer under this selection.

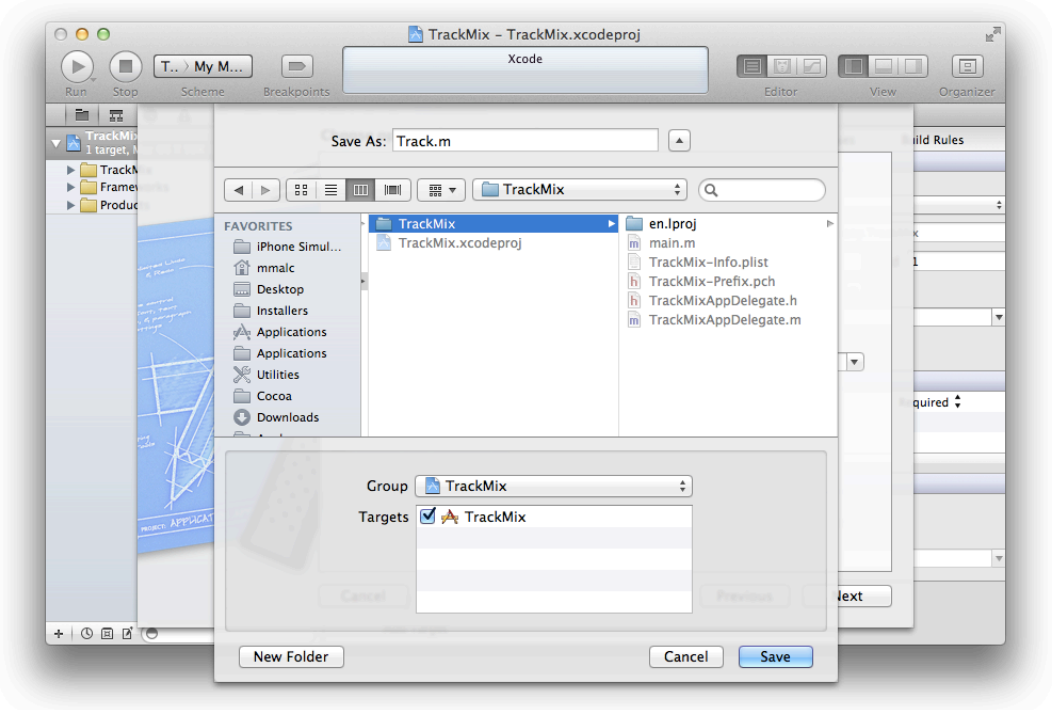
2. Select Objective-C class as follows:
   - Choose File > New> New File.
   - In the New File sheet select the Cocoa group.

- Select the Objective-C class template and click Next.



3. In the screen that appears, leave the new class as a subclass of the `NSObject` class and click Next.

4. In the Save sheet, name the file `Track.m` (by convention, class names begin with a capital letter) and click Save.



You should find the new source files in the project. If you look at the files, you'll see that stub implementations of various methods are already given to you. You need to add an attribute to represent the volume.

Attributes, relationships, and properties Cocoa uses three generic terms to refer to features of an object.

An **attribute** is a characteristic of the object itself. For example, hair color is an attribute of a person; likewise, the volume of the track object is an attribute of a `Track` object. In Cocoa, attributes are typically represented by value objects.

A **relationship** is a reference from one model object either to a single model object or to a collection of model objects. These are called to-one relationships and to-many relationships, respectively. For example, the relationship from a person to his or her spouse is a to-one relationship to another person, and the relationship from a person to his or her children is a to-many relationship to multiple people.

Collectively, attributes and relationships are known as **properties**. Properties in this generic sense are not the same as the declared property feature of Objective-C. Typically, however, attributes and to-one relationships are implemented using declared properties, and all may be referred to as *properties* . This overloading of terminology can be confusing, so take care in understanding the use of the word in any given context.

It is also important to appreciate that how a property is represented in an object is an implementation detail. Although a property is frequently represented by an instance variable, it does not have to be. In particular, **derived properties** are typically calculated from existing property values on demand rather than stored as an instance variable. The canonical example of a derived property is a person's full name, which is made up of the first and last name.

Implement the Track Class

The attribute to represent the track's volume is represented by a declared property.

To add a volume property . . .

1. In the Track header file (`Track.h`), add a property declaration for a `float` value called `volume`.

   The header file should look like this:

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


@interface Track : NSObject
@property (assign) float volume;
@end
```

2. In the Track implementation file (`Track.m`), synthesize the `volume` property.

   Add the following line of code after the `@implementation` declaration:

```
@synthesize volume;
```

That's it. With the property declaration and the `@synthesize` statement, the compiler generates the instance variable and accessor methods for the `volume` property.

> **Note** In this case, you asked the compiler to synthesize the instance variable rather than declaring it explicitly as you did in the application delegate's outlets. Either approach is acceptable for either case—there's nothing special about the outlet properties. Keep in mind, though that it's easier to ask the compiler to do the work when Xcode doesn't do it for you. In the next section, "Add a Track Property" (page 40), you'll omit the instance variable declaration again.

The value of the `volume` instance variable is automatically set to `0` when an instance is created, and because the property is a `float` there are no memory management issues to consider.

Add a Track Property

The application delegate object manages an instance of `Track`. The compiler will generate an error, though, if you declare a property—and corresponding instance variable—without telling it about the `Track` class. You could import the header file for the `Track` class (`Track.h`), but typically in Cocoa you instead provide a forward declaration using `@class`. A **forward declaration** is a promise to the compiler that `Track` will be defined somewhere else and that it needn't waste time checking for it now. (Doing this also avoids circularities if two classes need to refer to each other and would otherwise include each other's header files.) You then import the header file itself in the implementation file.

To add a track property to the application delegate . . .

1. In the application delegate's header file (`TrackMixAppDelegate.h`), add a forward declaration for the Track class before the interface declaration for `TrackMixAppDelegate`.

```
@class Track;
```

2. Add a declaration for the property between the last `@property` declaration and the `@end` symbol.

```
@property (strong) Track *track;
```

The property specifies the `strong` attribute rather than the `weak` attribute, indicating that the instance of the `Track` class should be retained at least as long as this reference to it exists. The difference between strong and weak references is discussed in more detail in *Transitioning to ARC Release Notes*.

Tip Like C, Objective-C is case-sensitive. `Track` refers to a class; `track` is a variable that here contains an instance of `Track`. A common programming error in Cocoa is to misspell a symbol name, frequently by using a letter of an inappropriate case (for example, "tableview" instead of "tableView"). This tutorial deliberately uses `Track` and `track` to encourage you to look at the name carefully. When writing code, make sure you use the appropriate symbol.

To make sure you're on track, confirm that your `TrackMixAppDelegate` class interface file (`TrackMixAppDelegate.h`) looks like this (comments are not shown, and the order of instance variable declarations and property and method declarations is not important):

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

@class Track;

@interface TrackMixAppDelegate : NSObject <NSApplicationDelegate>

@property (assign) IBOutlet NSWindow *window;
@property (weak) IBOutlet NSTextField *textField;
@property (weak) IBOutlet NSSlider *slider;

@property (strong) Track *track;

- (IBAction)takeFloatValueForVolumeFrom:(id)sender;
- (IBAction)mute:(id)sender;

@end
```

Next, create an instance of the track.

Create the Track Instance

Now that you've added the `track` property to the application delegate, you need to actually create an instance of `Track` and set it as the value for the property.

To create an instance of a track . . .

- In the implementation file for the application delegate class (`TrackMixAppDelegate.m`), create an instance of `Track` by adding the following code as the first statements in the implementation of the `applicationDidFinishLaunching:` method:

```
Track *aTrack = [[Track alloc] init];
[self setTrack:aTrack];
```

There's quite a lot in just these two lines. What they do is:

- Allocate memory for an instance of the `Track` class.
- Initialize the instance of the `Track` class to prepare it for use.
- Set the new track to be the `track` instance variable using an accessor method.

  Remember that you didn't separately declare `setTrack:`. It was implied as part of the property declaration—see "Adding a Track Property" (page 40).

You could replace `[self setTrack:aTrack];` with:

```
self.track = aTrack;
```

The dot notation invokes exactly the same accessor method (`setTrack:`) as in the original implementation. The dot notation provides a more compact syntax for invoking accessor methods—especially when you use nested expressions.

Tip Which syntax you choose is largely personal preference, although using the dot syntax has additional benefits when used in conjunction with properties—see "Declared Properties" in *The Objective-C Programming Language*. For more about the dot syntax, see "Dot Syntax" in *The Objective-C Programming Language* in "Objects, Classes, and Messaging" in *The Objective-C Programming Language*.

Housekeeping

There are a few unfinished tasks in the implementation file (`TrackMixAppDelegate.m`). You need to import the `Track` class's header file and synthesize the accessor method.

To complete the housekeeping tasks . . .

1. Import the header file for the `Track` class.

   At the top of the file, write an import statement like this:

   ```
   #import "Track.h"
   ```

2. In the `@implementation` block of the class, tell the compiler to synthesize the accessor methods for the view controller

   ```
   @synthesize track;
   ```

3. Save the source files.

Memory management You have created several objects, but you never explicitly deallocated any of them. Don't worry, you haven't been creating memory leaks; you've been using ARC.

Automatic Reference Counting (ARC) is a compiler feature that uses the Cocoa naming conventions for methods to automatically insert memory management code at compile time. ARC is enabled by default when you create a new project in Xcode.

As you learn about Mac OS X development, you are likely to encounter older code that does manual reference counting by calling the `retain`, `release`, and `autorelease` methods. In most cases, you can just ignore these methods.

For more information on ARC, see *Transitioning to ARC Release Notes*.

Check Your Progress

To check your progress, confirm that your `TrackMixAppDelegate` class implementation file (`TrackMixAppDelegate.m`) looks like this:

```
#import "TrackMixAppDelegate.h"
#import "Track.h"
```

```
@implementation TrackMixAppDelegate

@synthesize window = _window;
@synthesize textField = _textfield;
@synthesize slider = _slider;
@synthesize track = _track;


- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {

    Track *aTrack = [[Track alloc] init];
    [self setTrack:aTrack];
}


- (IBAction)takeFloatValueForVolumeFrom:(id)sender {

    NSString *senderName = nil;
    if (sender == textField) {
        senderName = @"textField";
    }
    else {
        senderName = @"slider";
    }
    NSLog(@"%@ sent takeFloatValueForVolumeFrom: with value %1.2f", senderName,
[sender floatValue]);
}


- (IBAction)mute:(id)sender {
    NSLog(@"received a mute: message");
}
```

```
@end
```

Test the Application

You can now test your application.

To test the application . . .

- Compile and run the project (choose Build > Build and Run, or click the Build and Run button in Xcode's toolbar).

Your application should compile without errors, and you should see the same user interface as before. Disappointingly, the user interface should also behave as it did before. Although you now have a `Track` object, the application delegate doesn't do anything with it beyond creating it. There is a little more work to do to complete the implementation.

Recap

In the application delegate, you declared an instance variable and accessor methods for a `Track` instance. You also synthesized the accessor methods and performed a few other housekeeping tasks.

Completing the Application Delegate

There are several parts to implementing the application delegate. You need to change the implementation of the `takeFloatValueForVolumeFrom:` method to update the track's volume, and the implementation of the `mute:` method to set the track's volume to zero. After both these operations, you need to ensure that the user interface is made consistent.

Implement the takeFloatValueForVolumeFrom: Method

The text field and slider both send a `takeFloatValueForVolumeFrom:` message to the application delegate. The method already retrieves the float value from the sender, but it doesn't update the track's volume.

To implement the takeFloatValueForVolumeFrom: method . . .

- In the `TrackMixAppDelegate.m` file, change the implementation of the `takeFloatValueForVolumeFrom:` method as follows:

```
- (IBAction)takeFloatValueForVolumeFrom:(id)sender {

    float newValue = [sender floatValue];
    [self.track setVolume:newValue];
    [self updateUserInterface];
}
```

There are several pieces to this method:

- `float newValue = [sender floatValue];`

  This line retrieves the sender's float value and stores it in a local variable.

- `[self.track setVolume:newValue];`

  This line changes the track's volume property to be the new value.

- `[self updateUserInterface];`

  This line invokes a new method to ensure that the user interface is consistent with the value of the track's volume.

The method doesn't exist yet. You could put the code here, but similar functionality will be required in the `mute:` method; placing the code in its own method means you need write it only once.

Notice that Xcode shows a warning for the `[self updateUserInterface];` statement. It does this because you haven't declared or implemented the `updateUserInterface` method yet. In a Cocoa project, you should typically treat warnings as errors.

Implement the updateUserInterface Method

The information shown in the user interface needs to be kept synchronized with the what is stored in the model. In this application, the volume is shown in two places. It might be tempting to let one of them update the other directly, but doing so tends to be difficult to maintain and debug. The best way to keep the user interface consistant is to always use the value in the model object as the true value, and synchronize everything in the user interface against it.

To synchronize the user interface . . .

1. In the `TrackMixAppDelegate.h` file, declare the `updateUserInterface` method as follows:

```
- (void)updateUserInterface;
```

2. In the `TrackMixAppDelegate.m` file, just before the `@end` statement, implement the `updateUserInterface` method as follows:

```
- (void)updateUserInterface {

    float volume = [self.track volume];
    [self.textField setFloatValue:volume];
    [self.slider setFloatValue:volume];
}
```

   The method first retrieves the track's volume, then sets that as the float value of both the text field and the slider.

3. Run the application.

   The application should behave as you expect. Moving the slider should update the value shown in the text field, and typing a new value in the text field (and clicking Return) should update the slider.

The next task is to implement the action method for the button.

Implement the mute: Method

The implementation of the `mute:` method follows the same pattern as the `takeFloatValueForVolumeFrom:` method, except that the track's volume is set to zero.

To implement and test the mute: method . . .

1. In the `TrackMixAppDelegate.m` file, change the implementation of the `mute:` method as follows:

```
- (IBAction)mute:(id)sender {
    [self.track setVolume:0.0];
    [self updateUserInterface];
}
```

2. Run the application.

   In addition to the text field and slider staying in sync, pressing the Mute button should now set both their values to zero.

Although the application appears to be working correctly now, there is one subtle bug. In setting up the nib file, you entered starting values for the text field and slider (both were set to 5). When the application starts, however, the track is created and its volume is set to zero. You need to ensure that the user interface and model values are properly synchronized at launch.

Make the User Interface Consistent at Launch

To ensure that the user interface is consistent with the model values at launch, you can invoke the `updateUserInterface` method after creating the track.

To ensure that the application is consistent at launch . . .

1. Update the `applicationDidFinishLaunching:` method to invoke `updateUserInterface`.

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
```

```
    Track *aTrack = [[Track alloc] init];
    [self setTrack:aTrack];

    [self updateUserInterface];
}
```

2. Run the application.

   The application should generally behave as it did before, except that when it launches it displays a value of 0 rather than 5.

Recap

You implemented the methods of the application delegate that provide the application's core functionality.

The next task is to look more closely at how the application behaves to see whether the user experience might be improved.

Application Polish

Presentation is a critical component of a successful Mac OS X application. You should read *Mac OS X Human Interface Guidelines* to learn about the user interface conventions expected in a Mac application. Although the application behaves "correctly" now, there are a couple more changes you can make to improve the overall polish.

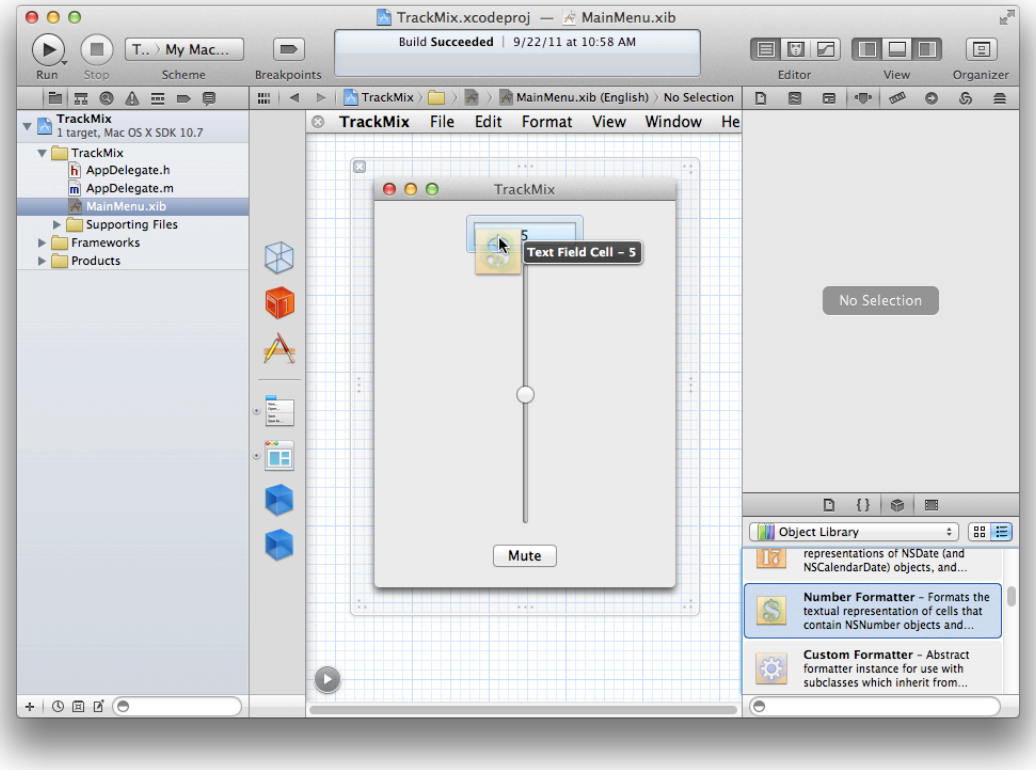
Add a Number Formatter

If you move the slider, the text field displays all the digits of the corresponding float value, for example 3.141592653589793. Typically, all these digits are unnecessary—3.14 would probably suffice. You can use a formatter object to display a rounded representation of the number. Using a formatter has two additional benefits:

- A formatter can impose maximum and minimum values on the number entered.
- A formatter presents the number appropriately for the user's locale. For example, if the local is French, users would see 3,14 instead of 3.14.

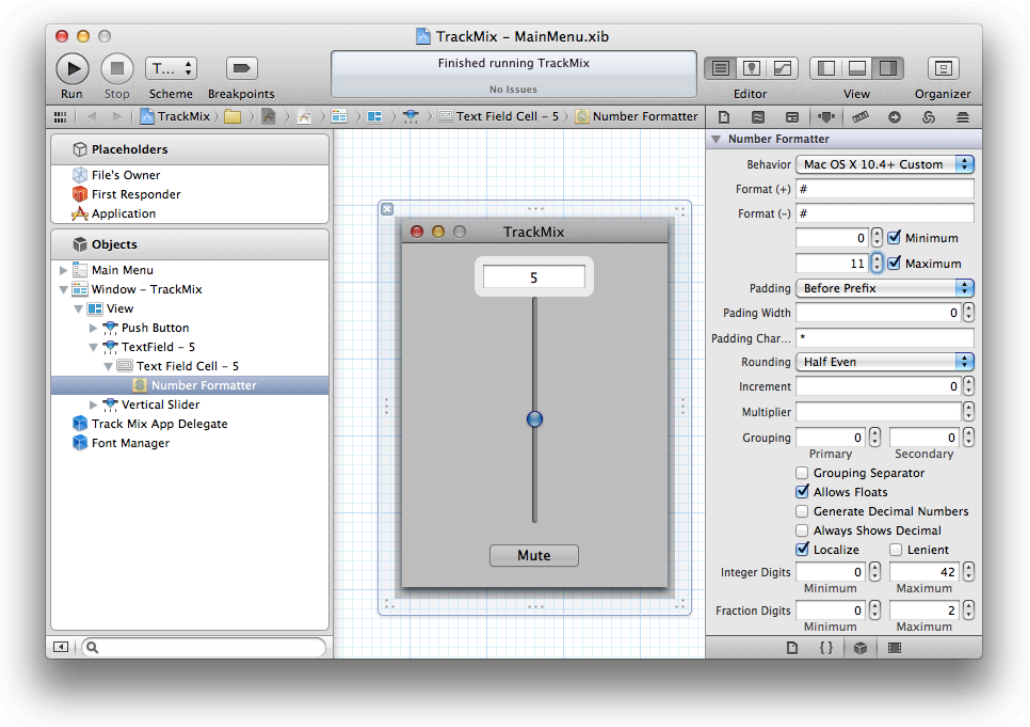
To add a number formatter . . .

1. Drag a number formatter from the Object Library into the text field.



2. To configure the formatter, use the disclosure button at the bottom left of the Interface Builder editor window to show the outline view of the nib file.
3. In the Objects section, use the disclosure triangles to drill down through the contents of the window to reveal the number formatter.

4. Select the number formatter, then use the Attributes inspector to configure its settings.



Normally, you should choose one of the standard formats—short, medium, or long—because these standard formats take the user's preferences into account. In this case, you want a custom formatter that shows just two decimal places.

5. Use the defaults, except for the following settings:
    - Behavior: Mac OS X 10.4+ Custom
    - Minimum: 0
    - Maximum: 11
    - Allows Floats: Yes
    - Localize: Yes
    - Fraction Digits: Minimum 0, Maximum 2

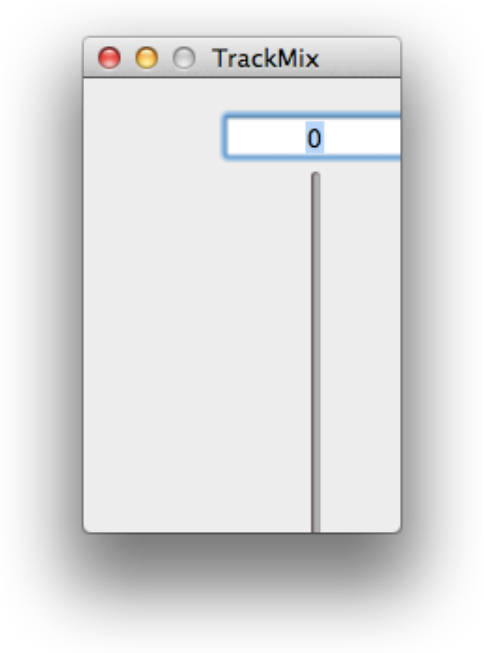
6. Save the file.

7. Build, run, and test the application.

    You should find that that the number displayed in the text field is correctly formatted and that the text field does not accept numbers outside of the given range.

Change the Window's Resize Behavior

Currently, if you resize the window, the user interface elements stay locked in position and remain constant size. This behavior leads to poor layout, as illustrated in this image.



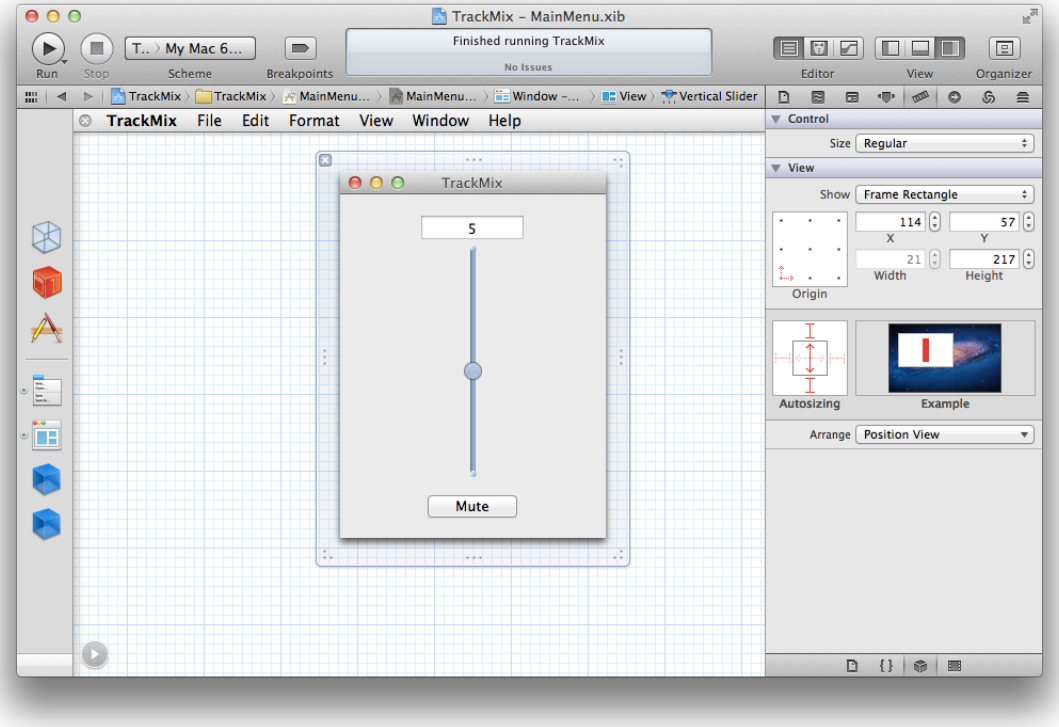
You can correct this problem by changing the window's resize behavior. One option is to prevent the window from being resized, and for a recording console this might be the most appropriate (to keep the sliders a constant size). For the sake of illustration, however, you might want to ensure that the text field and button remain in their fixed relative positions and allow the slider to grow and shrink in height with as the window resizes.

> **Note** The following sections describe how to use the "springs and struts" model for managing resizing behavior. In Mac OS X v10.7 and later, you can instead use the Cocoa Auto Layout feature. For more details, see *Cocoa Auto Layout Guide*.

To change the window's resize behavior . . .

1. Open the MainMenu nib file and select the slider.
2. In the Utility area, show the Size inspector.

3. Use the Autosizing view to change the resizing behavior of the slider.



In the Autosizing view, the inner square represents the selected object (in this case, the slider), and the outer square its enclosing view (in this case, the window—or more accurately, the window's content view).

- Clicking an arrowed line in the inner square toggles between flexible size (solid red line) and fixed size (light, dashed line) for the given dimension.

  In the slider example, the height is flexible and the width is fixed.

- Clicking the outer bars toggles between fixed distance shown by the (solid red line) and flexible distance shown by the (light, dashed line) for the given dimension.

  In the slider example, the top and bottom of the slider are both held at a fixed distance from the top and bottom of the window, respectively, and the left and right distances are flexible.

Together, these settings mean that, as the window resizes, the slider is kept centered horizontally, and it expands and contracts vertically to ensure that its top and bottom remain the same distance from the top and bottom of the window, respectively. The behavior is animated in the Example view, at the right of the Autosizing view.

To change the resizing behavior of the text field and button . . .

1. Use the Autosizing view to change the resizing behavior of the text field: It should be a fixed width, centered horizontally, a fixed distance from the top of the window, and a flexible distance from the bottom of the window.
2. Use the Autosizing view to change the resizing behavior of the button: It should be a fixed width, centered horizontally, a flexible distance from the top of the window, and a fixed distance from the bottom of the window.

Test the Application

Rather than building and running the application, you can test to see the changes you've made directly. Using Cocoa Simulator, you can test whether the user interface elements behave as you expect. Note, though, that only the user interface elements are recreated. Cocoa Simulator doesn't create an application delegate for you or simulate any of your application logic. So although you can test the resizing behavior and click the button, for example, the slider and text field do not stay in sync.

To test the user interface using Cocoa Simulator . . .

1. Choose Editor > Simulate Document.

   Xcode launches Cocoa Simulator. This loads the nib file and allows you to interact with it directly.

2. To quit the simulator, choose Cocoa Simulator > Quit Cocoa Simulator.

If the interface didn't behave as you expected, review the Autosizing settings you made and try again. Otherwise test the actual application.

To test the application . . .

1. Save the nib file.
2. Build and run the application.

   Make sure that the user interface resizing behavior is the same as it was in Cocoa Simulator.

Where to Next?

This chapter suggests further directions you might take to learn about developing applications for Mac OS X.

Refactor the Application Delegate

In this tutorial, you used the application delegate as the controller object, largely as a matter of convenience. Typically, you use separate controller objects. You can factor the controller logic out of the application delegate as follows:

1. Create a new controller class, and move the code for managing the track and the user interface from the application delegate to this new class.
2. Create an instance of the controller class in the nib file, and make appropriate connections to and from it, rather than to and from the application delegate.
3. Add a connection from the application delegate to the new controller object. When the application has finished launching, the application delegate should send a message to the controller to display the window.

Cocoa provides a special method, `awakeFromNib`, that is sent to all objects in a nib file that implement it, after the nib file has been completely loaded. You typically use this method to configure the user interface. Currently, however, the `Track` object is created in `applicationDidFinishLaunching:`—which will no longer be invoked because the controller instance is not the application delegate. The easiest remedy is to rename the `applicationDidFinishLaunching:` method to `awakeFromNib`.

```
- (void)awakeFromNib {

    Track *aTrack = [[Track alloc] init];
    [self setTrack:aTrack];

    [self updateUserInterface];
}
```

Use a Second Nib File

Many applications load parts of the user interface dynamically on demand. In addition to allowing easy replication of user interface elements, dynamic loading can also reduce that an amount of memory the application uses on launch.

After you factor the controller code into a separate class, you add a new nib file to the project. Cut the window from the main menu nib file and paste it into the new nib file. Set the class of the File's Owner of the new nib file to be `Controller` (or whatever you named the new controller class). Reestablish the connections to and from the File's Owner just as you did to the application delegate in the main menu nib file.

You need to implement a new method in the `Controller` class to display the window. This method should check whether the window already exists, and if it doesn't, load the new nib file. To load the nib file, you use the `NSBundle` class method, `loadNibNamed:owner:`. The method might look something like this:

```
- (void)showWindow {

    if (!self.window) {
        [NSBundle loadNibNamed:<#NewNibFileName#> owner:self];
    }

    [self.window makeKeyAndOrderFront:self];
}
```

Hide Private Methods in a Category

In this tutorial, you put the declaration of the `updateUserInterface` method in the `TrackMixAppDelegate.h` file. Although this approach is correct as far as the Objective-C language goes, from one perspective it advertises the `updateUserInterface` method as being available to an object with a reference to the application delegate. Really, though, it's a private method that only the application delegate itself should use. For this situation, it's common to use a category.

To use a category to hide a private method . . .

1. In the `TrackMixAppDelegate.h` file, remove the declaration of the `updateUserInterface` method.
2. At the top of the `TrackMixAppDelegate.m` file—after the `#import` statements but before the `@interface` declaration, add the declaration of a category of `TrackMixAppDelegate` that includes the `updateUserInterface` method.

```
@interface TrackMixAppDelegate (PrivateMethods)
- (void)updateUserInterface;
@end
```

It doesn't matter what name you give the category. By convention, this technique "hides" the method declaration from other users of the `TrackMixAppDelegate` class but exposes it to the compiler so that the compiler won't generate any warnings if it encounters use of the method before its implementation.

Experiment with New Directions

Next, try expanding on the functionality. There are many directions in which you can go, for example:

- You can often increase the size of your potential marketplace by localizing your application. Internationalization is the process of making your application localizable. To learn more about internationalization, read *Internationalization Programming Topics*.
- Your application currently creates a new track each time it launches, which sets the volume back to zero.To find out how to store the track in an external file and read it in when the application starts up, look at the *Archives and Serializations Programming Guide*.

Most important, out new ideas and experiment—there are many code samples you can look at for inspiration, and Apple's documentation will help you to understand concepts and programming interfaces.

Consider Best Practices for Application Development

As you progress, in addition to extending the functionality of the application itself, you should learn about other best practices for application development.

- Enable sandboxing.

  By default, a running application has all the rights and privileges of the user, and can do *anything* the user can do. This ability represents a security problem: if the application becomes compromised through a buffer overflow or other security hole, an attacker also gains the ability to do anything that the user can do. You can mitigate the potential problems by enabling application sandboxing. Sandboxing is optional, but strongly recommended. The additional security provided by sandboxing significantly outweighs the effort required to use it.

  For more about sandboxing, see *Code Signing Guide*.

- Use performance analysis tools.

  Performance is critical to good user experience on Mac OS X. You should learn to use the various performance tools provided with Mac OS X—in particular Instruments—to tune your application so that it minimizes its resource requirements.

  To learn more about how you can improve your application's performance, see *Cocoa Performance Guidelines*. To learn more about Instruments, see *Instruments User Guide*.

- Add unit testing.

  When you created the project, you didn't select the option to add support for unit tests. Testing, however, is an important aspect of best practices for Cocoa development. In some respects, you can understand unit testing as the counterpart of the design principle of encapsulation—it ensures that if the implementation of a method changes, it still works as advertised.

  To add support for unit testing, do one of two things: Either create a new version of the tutorial that sets up unit testing. Or, using the current project in Xcode, choose File > New > New Target then select Other > Cocoa Unit Testing Bundle. Look to see what Xcode adds to the project.

  To learn about unit testing, see *Xcode Unit Testing Guide*.

Code Listings

This appendix provides listings for the two classes you define. For ease of reading, these listings purposely omit comments and other method implementations that are contained in the file templates.

TrackMixAppDelegate

The header file, TrackMixAppDelegate.h

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

@class Track;

@interface TrackMixAppDelegate : NSObject <NSApplicationDelegate>

@property (assign) IBOutlet NSWindow *window;
@property (weak) IBOutlet NSTextField *textField;
@property (weak) IBOutlet NSSlider *slider;

@property (strong) Track *track;

- (IBAction)takeFloatValueForVolumeFrom:(id)sender;
- (IBAction)mute:(id)sender;

- (void)updateUserInterface;

@end
```

The implementation file, TrackMixAppDelegate.m

```
#import "TrackMixAppDelegate.h"
#import "Track.h"

@implementation TrackMixAppDelegate

@synthesize window = _window;
@synthesize textField = _textField;
@synthesize slider = _slider;
@synthesize track = _track;

- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {

    Track *aTrack = [[Track alloc] init];
    [self setTrack:aTrack];

    [self updateUserInterface];
}


- (IBAction)takeFloatValueForVolumeFrom:(id)sender {

    float newValue = [sender floatValue];
    [self.track setVolume:newValue];
    [self updateUserInterface];
}


- (IBAction)mute:(id)sender {
    [self.track setVolume:0.0];
    [self updateUserInterface];
}


- (void)updateUserInterface {
```

```
    float volume = [self.track volume];
    [self.textField setFloatValue:volume];
    [self.slider setFloatValue:volume];
}

@end
```

Track

The header file, Track.h

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

@interface Track : NSObject

@property (assign) float volume;

@end
```

The implementation file, Track.m

```
#import "Track.h"

@implementation Track

@synthesize volume = _volume;

- (id)init {
    self = [super init];
    if (self) {
    }
    return self;
}
```

```
@end
```

Document Revision History

This table describes the changes to *Your First Mac App* .

Date	Notes
2012-01-09	Minor link corrections.
2011-09-27	Updated code listings to use ARC. Minor editorial changes throughout.
2011-07-06	A new tutorial that introduces application development for Mac OS X.

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