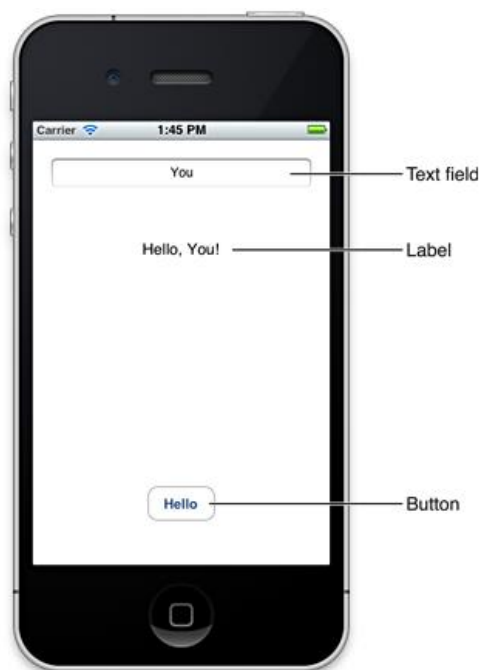


Creating Your First iOS App

Your First iOS App introduces you to the Three Ts of iOS app development:

- **Tools.** How to use Xcode to create and manage a project.
- **Technologies.** How to create an app that responds to user input.
- **Techniques.** How to take advantage of some of the fundamental design patterns that underlie all iOS app development.

After you complete all the steps in this tutorial, you'll have an app that looks something like this:



As you can see above, there are three main user interface elements in the app that you create:

- A text field (in which the user enters information)
- A label (in which the app displays information)
- A button (which causes the app to display information in the label)

When you run the finished app, you click inside the text field to reveal the system-provided keyboard. After you use the keyboard to type your name, you dismiss it (by clicking the Done key) and then you click the Hello button to see the string “Hello, *Your Name!*“ in the label between the text field and the button.

Later in *Start Developing iOS Apps Today* you will work through another tutorial, *Internationalize Your App*, and in the process add a Chinese localization to the app you will create in this tutorial.

To benefit from this tutorial, it helps to have some familiarity with the basics of computer programming in general and with object-oriented programming and the Objective-C language in particular. If you haven't used Objective-C before, don't worry if the code in this tutorial is hard to understand. When you finish *Start Developing iOS Apps Today*, you'll understand the code much better.

Getting Started

To create the iOS app in this tutorial, you need Xcode 4.3 or later. Xcode is Apple's integrated development environment (or IDE) for both iOS and Mac OS X development. When you install Xcode on your Mac, you also get the iOS SDK, which includes the programming interfaces of the iOS platform.

Create and Test a New Project

To get started developing your app, you create a new Xcode project.

1. Open Xcode (by default it's in `/Applications`).

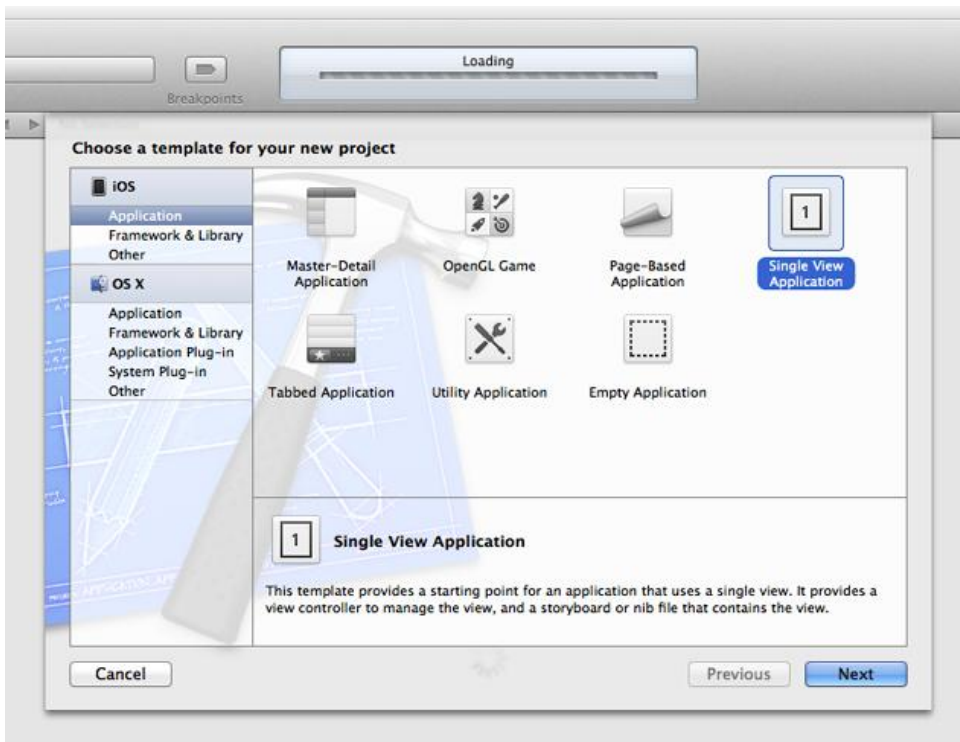
If you've never created or opened a project in Xcode before, you should see a Welcome to Xcode window similar to this:



If you've created or opened a project in Xcode before, you might see a project window instead of the Welcome to Xcode window.

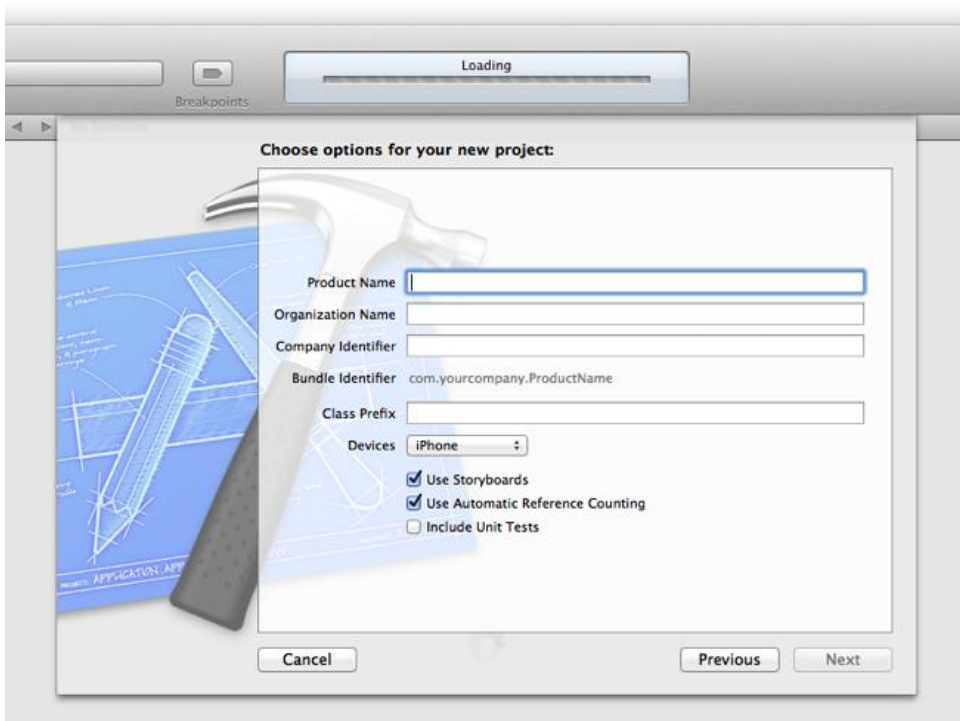
2. In the Welcome to Xcode window, click "Create a new Xcode project" (or choose `File > New > New project`).

Xcode opens a new window and displays a dialog in which you can choose a template. Xcode includes several built-in app templates that you can use to develop common styles of iOS apps. For example, the Tabbed template creates an app that is similar to iTunes and the Master-Detail template creates an app that is similar to Mail.



3. In the iOS section at the left side of the dialog, select Application.
4. In the main area of the dialog, select Single View Application and then click Next.

A new dialog appears that prompts you to name your app and choose additional options for your project.



5. Fill in the Product Name, Company Identifier, and Class Prefix fields.

You can use the following values:

- Product Name: HelloWorld
- Company Identifier: Your company identifier, if you have one.
- Class Prefix: HelloWorld

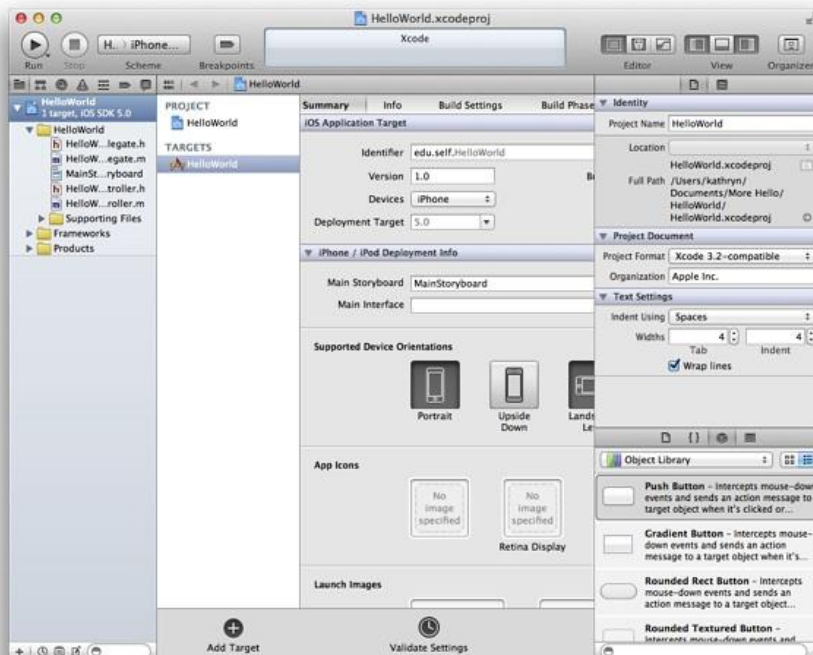
Note: Xcode uses the product name you entered to name your project and the app. Xcode uses the class prefix name to name the classes it creates for you. For example, Xcode automatically creates an app delegate class and names it `HelloWorldAppDelegate`. If you enter a different value for the class prefix, then the app delegate class is named *YourClassPrefixNameAppDelegate*. To keep things simple, this tutorial assumes that you named your product `HelloWorld` and that you used `HelloWorld` for the class prefix value.

6. In the Device Family pop-up menu, make sure that iPhone is chosen.
7. Make sure that the Use Storyboard and Use Automatic Reference Counting options are selected and that the Include Unit Tests option is unselected.
8. Click Next.

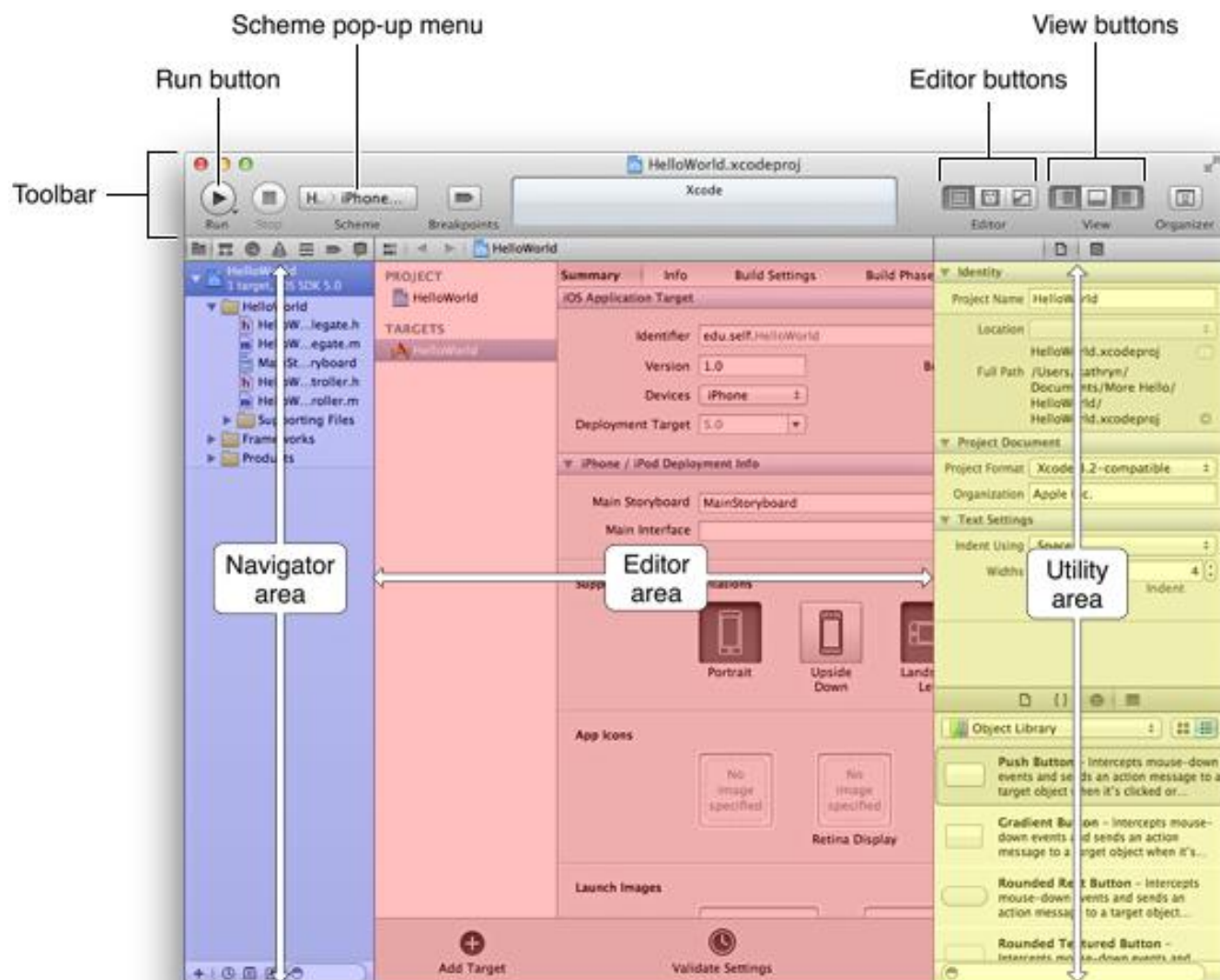
Another dialog appears that allows you to specify where to save your project.

9. Specify a location for your project (leave the Source Control option unselected) and then click Create.

Xcode opens your new project in a window (called the *workspace window*), which should look similar to this:



Take a few moments to familiarize yourself with the workspace window that Xcode opens for you. You'll use the buttons and areas identified in the window below throughout the rest of this tutorial.



If the utilities area in your workspace window is already open (as it is in the window shown above), you can close it for now because you won't need it until later in the tutorial. The rightmost View button controls the utilities area. When the utilities area is visible, the button looks like this:



If necessary, click the rightmost View button to close the utilities area.

Even though you haven't yet written any code, you can build your app and run it in the Simulator app that is included in Xcode. As its name implies, Simulator allows you to get an idea of how your app would look and behave if it were running on an iOS-based device.

1. Make sure that the Scheme pop-up menu in the Xcode toolbar has HelloWorld > iPhone 6.0 Simulator chosen.

If the pop-up menu does not display that choice, open it and choose iPhone 6.0 Simulator from the menu.

2. Click the Run button in the Xcode toolbar (or choose Product > Run).

Xcode updates you on the build process.

After Xcode finishes building your project, Simulator should start automatically. Because you specified an iPhone product (rather than an iPad product), Simulator displays a window that looks like an iPhone. On the simulated iPhone screen, Simulator opens your app, which should look like this:



Right now, your app is not very interesting: it simply displays a blank white screen. To understand where the white screen comes from, you need to learn about the objects in your code and how they work together to start the app. For now, quit Simulator (choose iOS Simulator > Quit iOS Simulator; make sure that you don't quit Xcode).

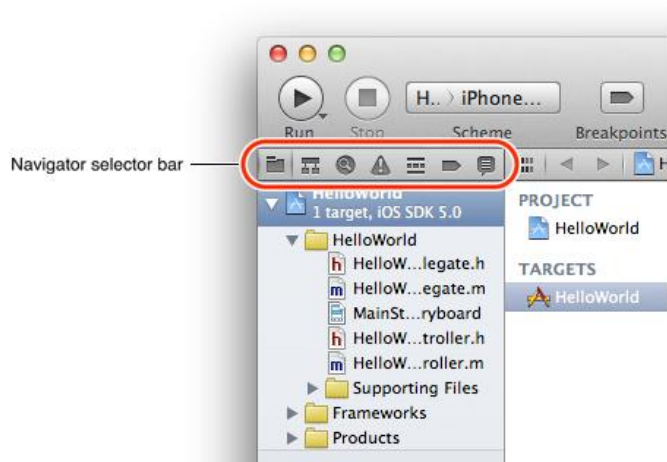
Find Out How an App Starts Up

Because you based your project on an Xcode template, much of the basic app environment is automatically set up when you run the app. For example, Xcode creates an application object which, among a few other things, establishes the run loop (a run loop registers input sources and enables the delivery of input events to your app). Most of this work is done by the `UIApplicationMain` function, which is supplied for you by the UIKit framework and is automatically called in your project's `main.m` source file.

Note: The UIKit framework provides all the classes that an app needs to construct and manage its user interface. The UIKit framework is just one of many object-oriented frameworks provided by Cocoa Touch, which is the app environment for all iOS apps.

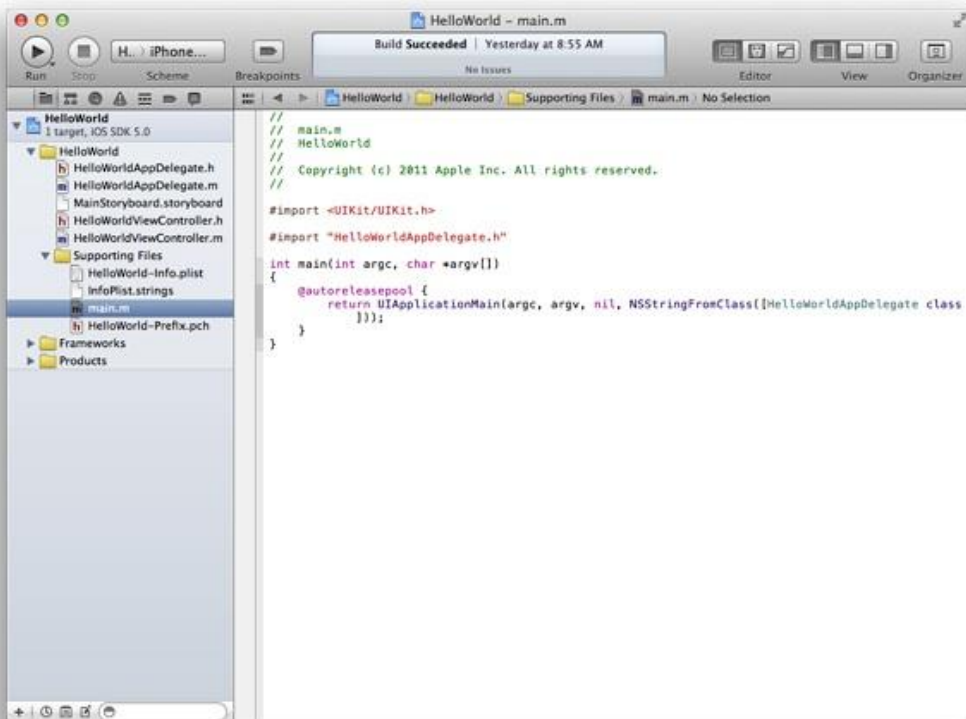
1. Make sure the project navigator is open in the navigator area.

The project navigator displays all the files in your project. If the project navigator is not open, click the leftmost button in the navigator selector bar:



2. Open the Supporting Files folder in the project navigator by clicking the disclosure triangle next to it.
3. Select `main.m`.

Xcode opens the source file in the main editor area of the window, which should look similar to this:



The main function in `main.m` calls the `UIApplicationMain` function within an autorelease pool:

```
@autoreleasepool {
    return UIApplicationMain(argc, argv, nil, NSStringFromClass([HelloWorldAppDelegate
class]));
}
```

The `@autoreleasepool` statement supports the Automatic Reference Counting (ARC) system. ARC provides automatic object-lifetime management for your app, ensuring that objects remain in existence for as long as they're needed and no longer.

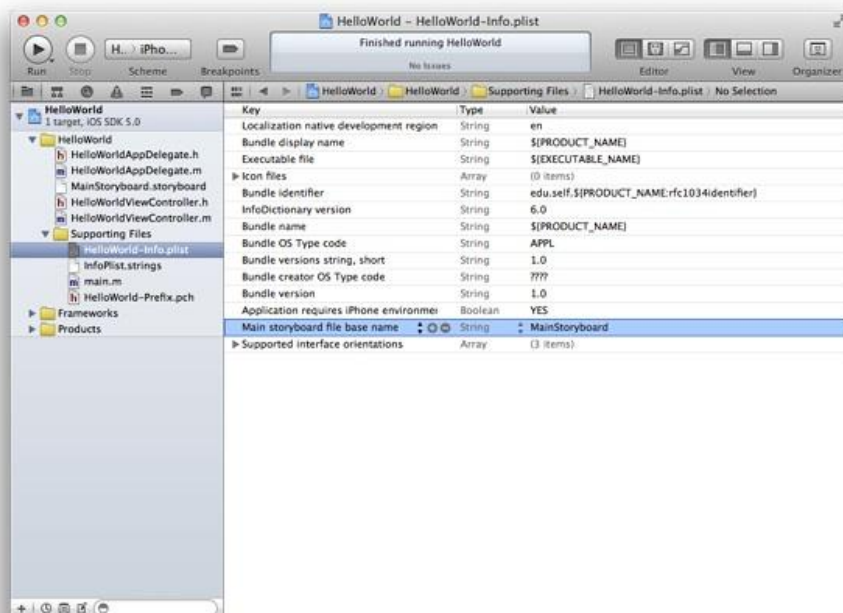
The call to `UIApplicationMain` creates an instance of the `UIApplication` class and an instance of the app delegate (in this tutorial, the app delegate is `HelloWorldAppDelegate`, which is provided for you by the Single View template). The main job of the **app delegate** is to provide the window into which your app's content is drawn. The app delegate can also perform some app configuration tasks before the app is displayed. (**Delegation** is a design pattern in which one object acts on behalf of, or in coordination with, another object.)

In an iOS app, a **window** object provides a container for the app's visible content, helps deliver events to app objects, and helps the app respond to changes in the device's orientation. The window itself is invisible.

The call to `UIApplicationMain` also scans the app's `Info.plist` file. The `Info.plist` file is an information property list—that is, a structured list of key-value pairs that contains information about the app such as its name and icon.

- In the Supporting Files folder in the project navigator, select `HelloWorld-Info.plist`.

Xcode opens the `Info.plist` file in the editor area of the window, which should look similar to this:



In this tutorial, you won't need to look at any other files in the Supporting Files folder, so you can minimize distractions by closing the folder in the project navigator. Again click the disclosure triangle next to the folder icon to close the Supporting Files folder.

Because you chose to use a storyboard in this project, the `Info.plist` file also contains the name of the storyboard file that the application object should load. A **storyboard** contains an archive of the objects, transitions, and connections that define an app's user interface.

In the HelloWorld app, the storyboard file is named `MainStoryboard.storyboard` (note that the `Info.plist` file shows only the first part of this name). When the app starts, `MainStoryboard.storyboard` is loaded and the initial view controller is instantiated from it. A **view controller** is an object that manages an area of content; the **initial view controller** is simply the first view controller that gets loaded when an app starts.

The HelloWorld app contains only one view controller (specifically, `HelloWorldViewController`). Right now, `HelloWorldViewController` manages an area of content that is provided by a single view. A **view** is an object that draws content in a rectangular area of the screen and handles events caused by the user's touches. A view can also contain other views, which are called **subviews**. When you add a subview to a view, the containing view is called the **parent view** and its subview is called a **child view**. The parent view, its child views (and their child views, if any) form a **view hierarchy**. A view controller manages a single view hierarchy.

Note: The views and the view controller in the HelloWorld app represent two of the three roles for app objects that are defined by the design pattern called Model-View-Controller (MVC). The third role is the model object. In MVC, model objects represent data (such as a to-do item in a calendar app or a shape in a drawing app), view objects know how to display the data represented by model objects, and controller objects mediate between models and views. In the HelloWorld app, the model object is the string that holds the name that the user enters. You don't need to know more about MVC right now, but it's a good idea to begin thinking about how the objects in your app play these different roles.

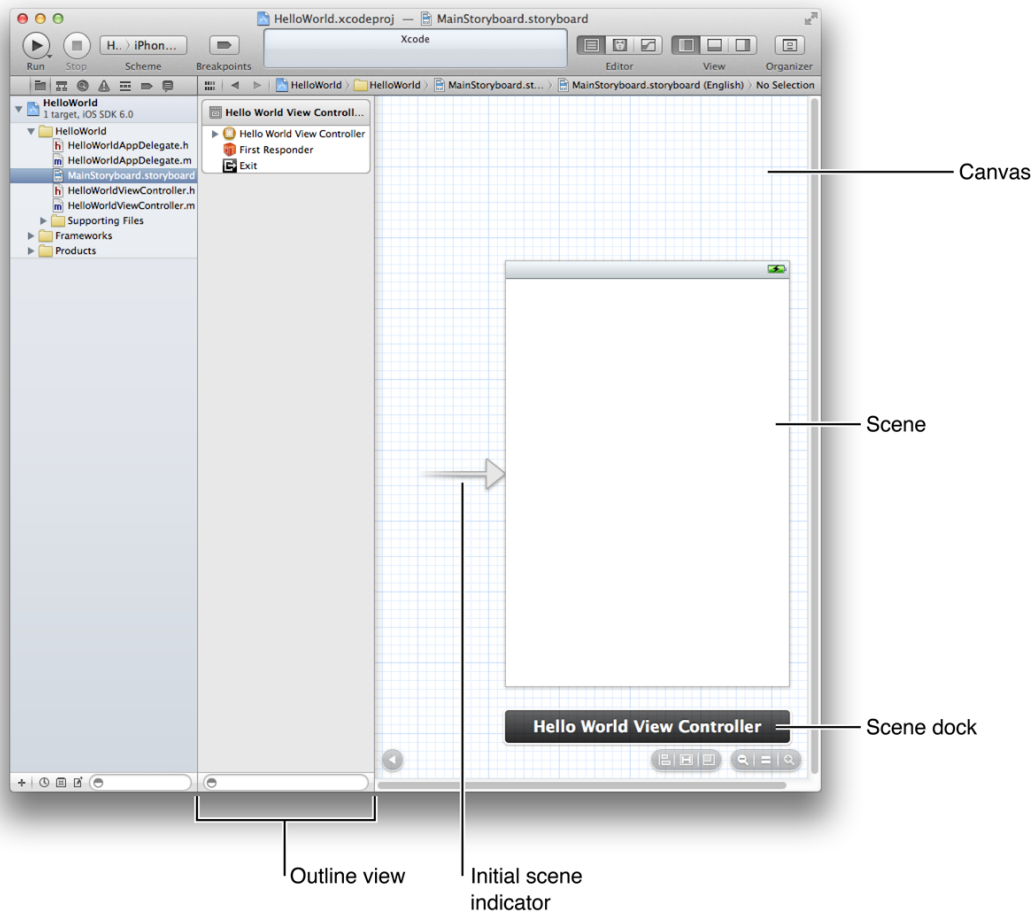
In a later step, you'll create a view hierarchy by adding three subviews to the view that's managed by `HelloWorldViewController`; these three subviews represent the text field, the label, and the button.

You can see visual representations of the view controller and its view in the storyboard.

- Select `MainStoryboard.storyboard` in the project navigator.

Xcode opens the storyboard in the editor area. (The area behind the storyboard objects—that is, the area that looks like graph paper—is called the **canvas**.)

When you open the default storyboard, your workspace window should look similar to this:



A storyboard contains scenes and segues. A **scene** represents a view controller, and a **segue** represents a transition between two scenes.

Because the Single View template provides one view controller, the storyboard in your app contains one scene and no segues. The arrow that points to the left side of the scene on the canvas is the **initial scene indicator**, which identifies the scene that should be loaded first when the app starts (typically, the initial scene is the same as the initial view controller).

The scene that you see on the canvas is named Hello World View Controller because it is managed by the `HelloWorldViewController` object. The Hello World View Controller scene consists of a few items that are displayed in the Xcode **outline view** (which is the pane that appears between the canvas and the project navigator). Right now, the view controller consists of the following items:

- A first responder placeholder object (represented by an orange cube).

The **first responder** is a dynamic placeholder that represents the object that should be the first to receive various events while the app is running. These events include editing-focus events (such as tapping a text field to bring up the keyboard), motion events (such as shaking the device), and action messages (such as the message a button sends when the user taps it), among others. You won't be doing anything with the first responder in this tutorial.

- A placeholder object named **Exit** for unwinding seques.

By default, when a user dismisses a child scene, the view controller for that scene *unwinds* (or returns) to the parent scene—that is the scene that originally transitioned to the child scene. However, the Exit object enables a view controller to unwind to an arbitrary scene.

- The `HelloWorldViewController` object (represented by a pale rectangle inside a yellow sphere).

When a storyboard loads a scene, it creates an instance of the view controller class that manages the scene.

- A view, which is listed below the view controller (to reveal this view in the outline view, you might have to open the disclosure triangle next to Hello World View Controller).

The white background of this view is what you saw when you ran the app in Simulator.

Note: An app’s window object is not represented in the storyboard.

The area below the scene on the canvas is called the **scene dock**. Right now, the scene dock displays the view controller’s name (that is, Hello World View Controller). At other times, the scene dock can contain the icons that represent the first responder, the Exit placeholder object, and the view controller object.

Configuring the View

Xcode provides a library of objects that you can add to a storyboard file. Some of these are user interface elements that belong in a view, such as buttons and text fields. Others are higher-level objects, such as view controllers and gesture recognizers.

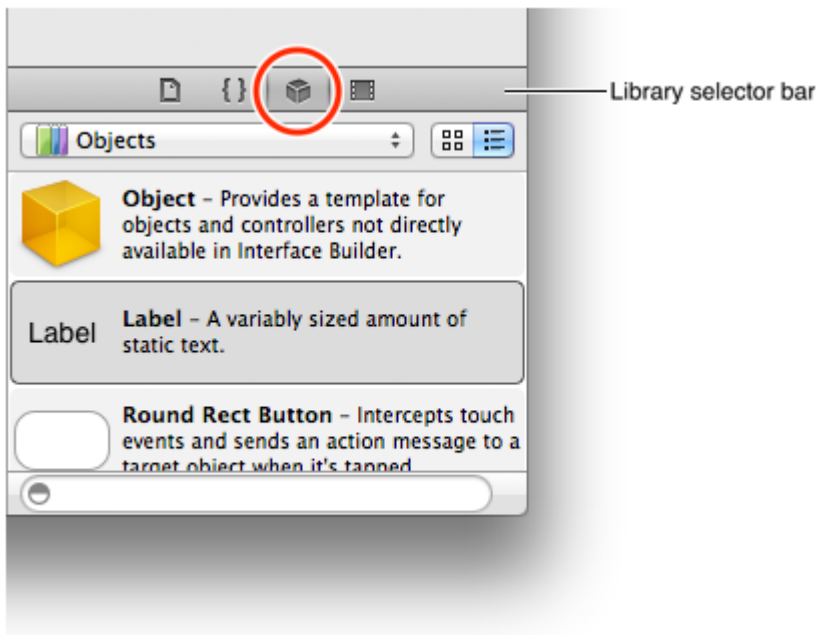
The Hello World View Controller scene already contains a view. Now you need to add a button, a label, and a text field. Then, you make connections between these elements and the view controller class so that the elements provide the behavior you want.

Add the User Interface Elements

You add user interface (UI) elements by dragging them from the object library to a view on the canvas. After the UI elements are in a view, you can move and resize them as appropriate.

1. If necessary, select `MainStoryboard.storyboard` in the project navigator to display the Hello World View Controller scene on the canvas.
2. If necessary, open the object library.

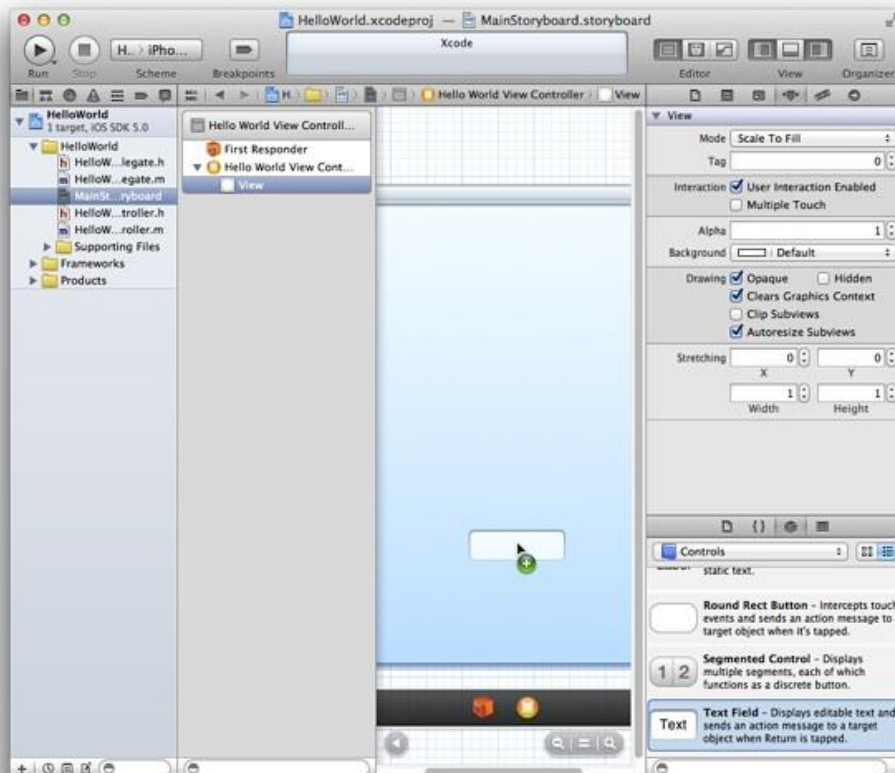
The object library appears at the bottom of the utilities area. If you don’t see the object library, you can click its button, which is the third button from the left in the library selector bar:



3. In the object library, choose Controls from the Objects pop-up menu.

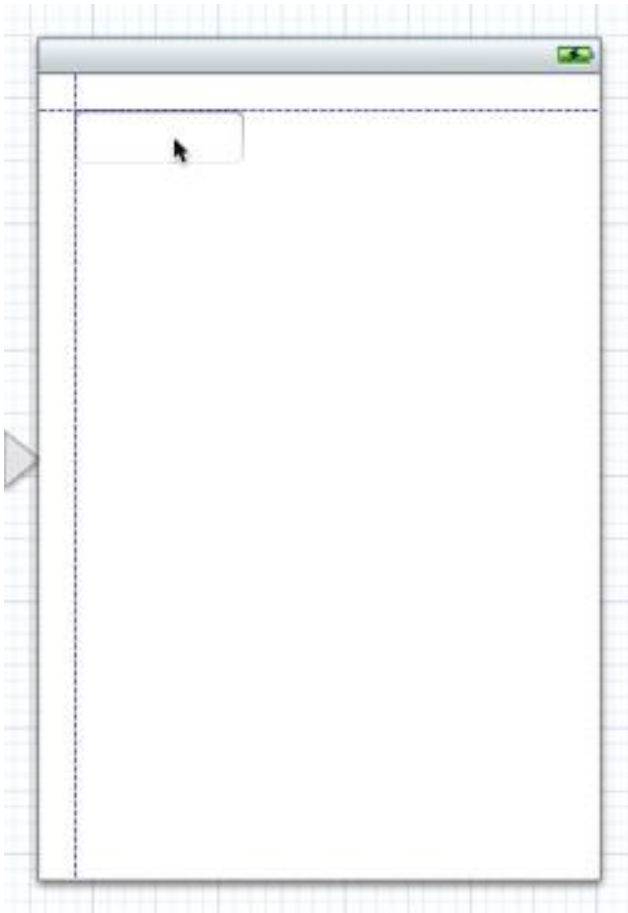
Xcode displays a list of controls below the pop-up menu. The list displays each control's name and appearance, and a short description of its function.

4. One at a time, drag a text field, a rounded rectangle (Round Rect) button, and a label from the list, and drop each of them onto the view.



5. In the view, drag the text field so that it's near the upper-left corner of the view.

As you move the text field (or any UI element), dashed blue lines—called *alignment guides*—appear that help you align the item with the center and edges of the view. Stop dragging the text field when you can see the view's left and upper alignment guides, as shown here:



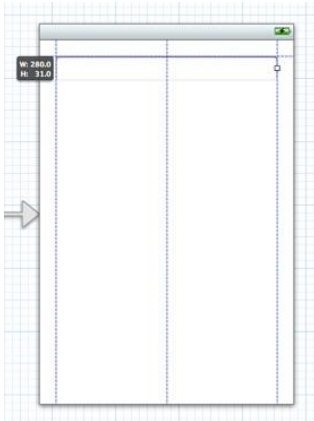
6. In the view, prepare to resize the text field.

You resize a UI element by dragging its *resize handles*, which are small white squares that can appear on the element's borders. In general, you reveal an element's resize handles by selecting it on the canvas or in the outline view. In this case, the text field should already be selected because you just stopped dragging it. If your text field looks like the one below, you're ready to resize it; if it doesn't, select it on the canvas or in the outline view.



7. Drag the text field's right resize handle until the view's rightmost alignment guide appears.

Stop resizing the text field when you see something like this:

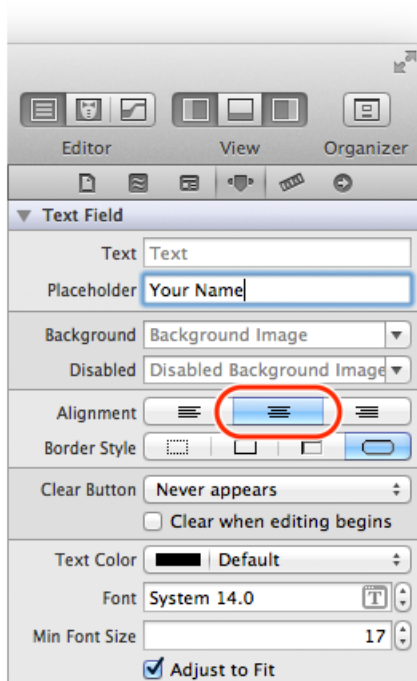


8. With the text field still selected, open the Attributes inspector (if necessary).
9. In the Placeholder field near the top of the Text Field Attributes inspector, type the phrase `Your Name`.

As its name suggests, the Placeholder field provides the light gray text that helps users understand the type of information they can enter in the text field. In the running app, the placeholder text disappears as soon as the user taps inside the text field.

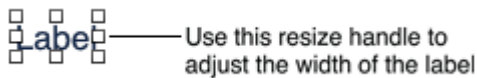
10. Still in the Text Field Attributes inspector, click the middle Alignment button to center the text field's text.

After you enter the placeholder text and change the alignment setting, the Text Field Attributes inspector should look something like this:



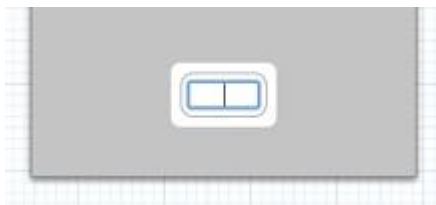
11. In the view, drag the label so that it's below the text field and its left side is aligned with the left side of the text field.
12. Drag the label's right resize handle until the label is the same width as the text field.

A label has more resize handles than a text field. This is because you can adjust both the height and the width of a label (you can adjust only the width of a text field). You don't want to change the height of the label, so be sure to avoid dragging one of the resize handles in the label's corners. Instead, drag the resize handle that's in the middle of the label's right side.

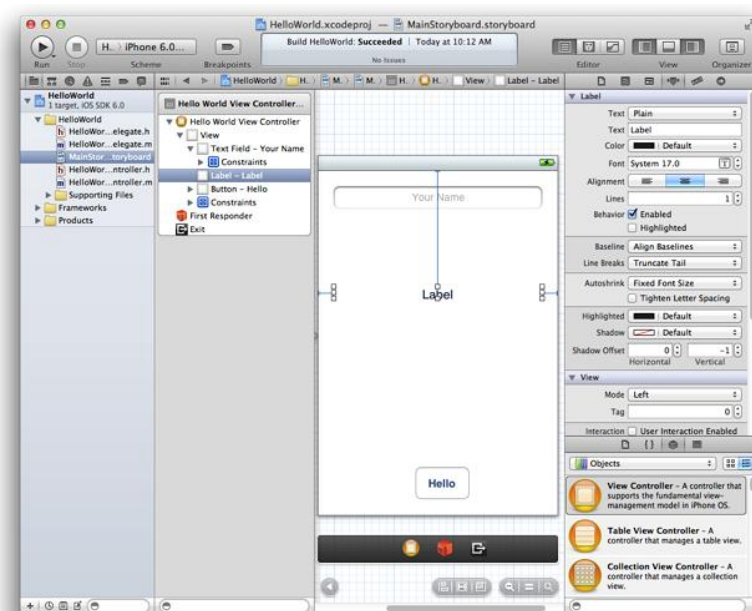


13. In the Label Attributes inspector, click the middle Alignment button (to center the text you'll display in the label).
14. Drag the button so that it's near the bottom of the view and centered horizontally.
15. On the canvas, double-click the button and enter the text Hello.

When you double-click the button in the view (and before you enter the text), you should see something like this:



After you add the text field, label, and button UI elements and make the recommended layout changes, your project should look similar to this:



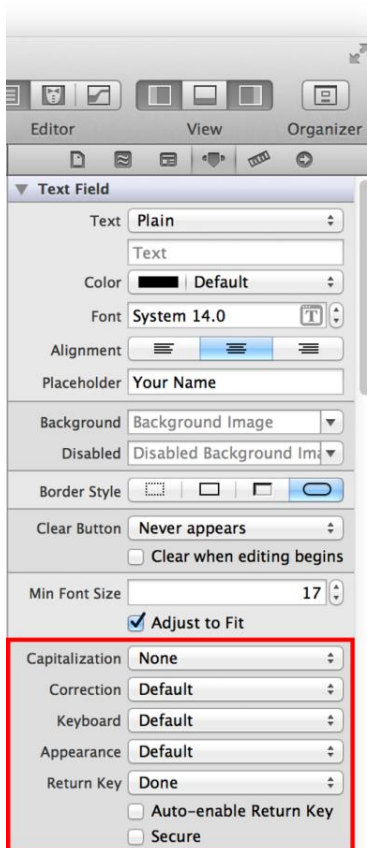
You probably noticed that when you added the text field, label, and button to the background view, Xcode inserted items in the outline view named **Constraints**. Cocoa Touch features an Auto Layout system that lets you define layout constraints for user-interface elements. Constraints represent relationships between user interface elements that affect how they alter their position and geometry when other views are resized or when there is an orientation change. You do not change the default constraints for the views you added to the user interface.

There are a few other changes you can make to the text field so that it behaves as users expect. First, because users will be entering their names, you can ensure that iOS suggests capitalization for each word they type. Second, you can make sure that the keyboard associated with the text field is configured for entering names (rather than numbers, for example) and that the keyboard displays a Done button.

This is the principle behind these changes: Because you know at design time what type of information a text field will contain, you can configure it so that its runtime appearance and behavior are well suited to the user's task. You make all of these configuration changes in the Attributes inspector.

1. In the view, select the text field.
2. In the Text Field Attributes inspector, make the following choices:
 - In the Capitalization pop-up menu, choose Words.
 - Ensure that the Keyboard pop-up menu is set to Default.
 - In the Return Key pop-up menu, choose Done.

After you make these choices, the Text Field Attributes inspector should look like this:



Run your app in Simulator to make sure that the UI elements you added look the way you expect them to. If you click the Hello button, it should become highlighted, and if you click inside the text field, the keyboard should appear. At the moment, though, the button doesn't do anything, the label remains empty, and there's no way to dismiss the keyboard after it appears. To add this functionality, you need to make the appropriate connections between the UI elements and the view controller. These connections are described next.

Note: Because you're running the app in Simulator, and not on a device, you activate controls by clicking them instead of tapping them.

Create an Action for the Button

When the user activates a UI element, the element can send an action message to an object that knows how to perform the corresponding action method (such as “add this contact to the user's list of contacts”). This interaction is part of the *target-action* mechanism, which is another Cocoa Touch design pattern.

In this tutorial, when the user taps the Hello button, you want it to send a “change the greeting” message (the *action*) to the view controller (the *target*). The view controller responds to this message by changing the string (that is, the model object) that it manages. Then, the view controller updates the text that's displayed in the label to reflect the change in the model object's value.

Using Xcode, you can add an action to a UI element and set up its corresponding action method by Control-dragging from the element on the canvas to the appropriate part of a source file (typically, a class extension in a view controller's implementation file). The storyboard archives the connections that you create in this way. Later, when the app loads the storyboard, the connections are restored.

1. If necessary, select `MainStoryboard.storyboard` in the project navigator to display the scene on the canvas.
2. In the Xcode toolbar, click the Utilities button to hide the utilities area and click the Assistant Editor button to display the assistant editor pane.

The Assistant Editor button is the middle Editor button and it looks like this: .

3. Make sure that the Assistant displays the view controller's implementation file (that is, `HelloWorldViewController.m`).

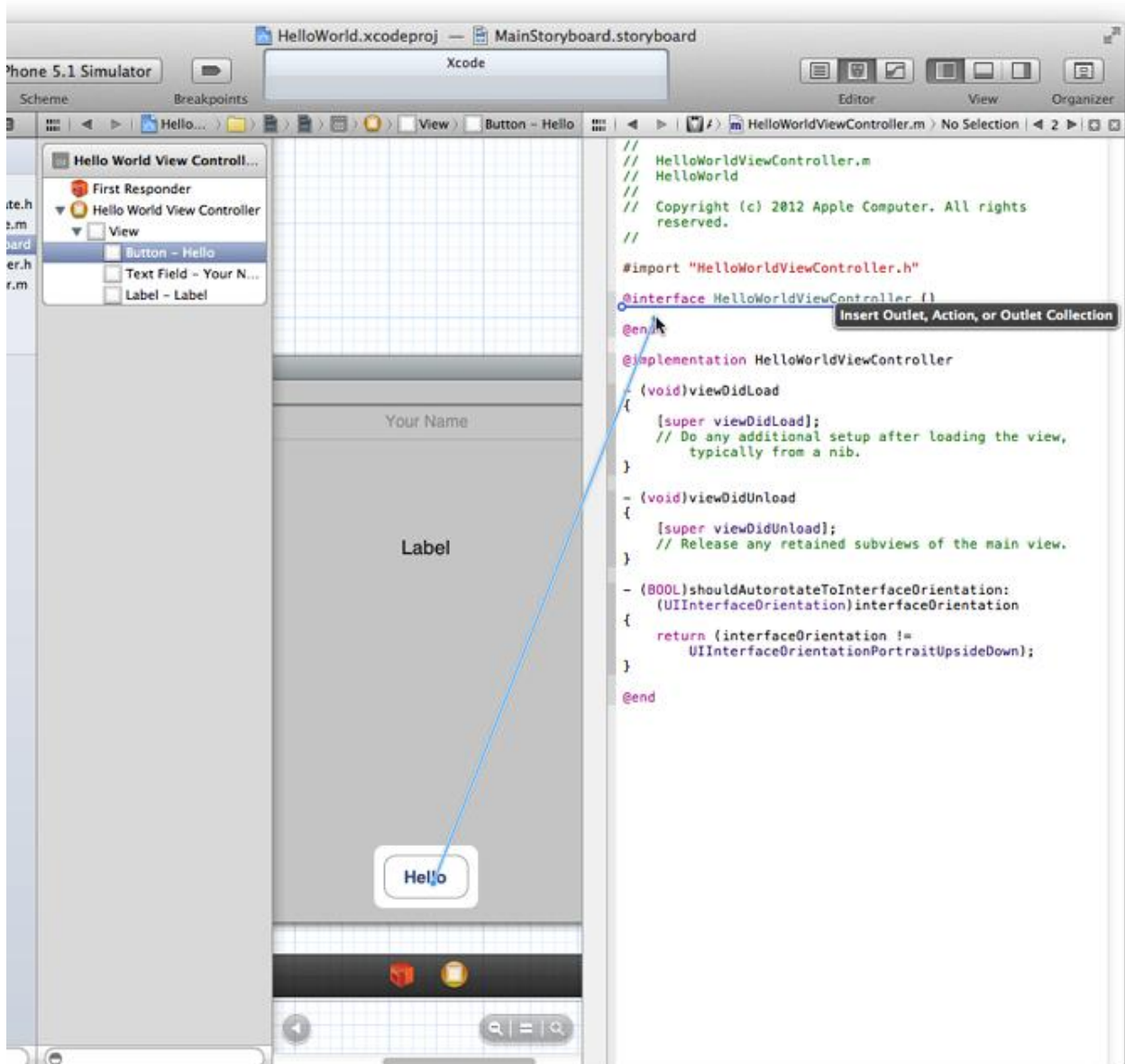
If it shows `HelloWorldViewController.h` instead, select `HelloWorldViewController.m` in the project navigator.

4. On the canvas, Control-drag from the Hello button to the class extension in `HelloWorldViewController.m`.

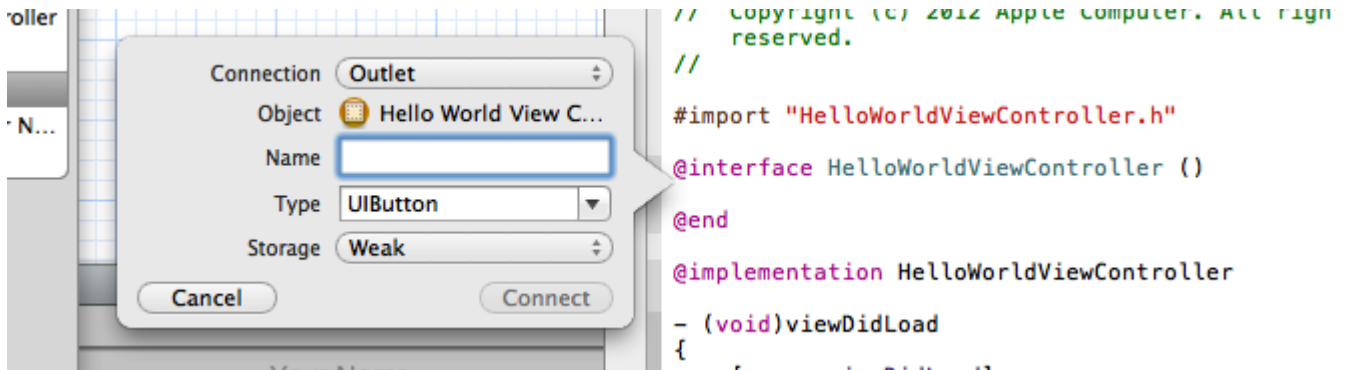
A class extension in an implementation file is a place for declaring properties and methods that are private to a class. (You will learn more about class extensions in *Write Objective-C Code*.) Outlets and actions should be private. The Xcode template for a view controller includes a class extension in the implementation file; in the HelloWorld project, the class extension looks like this:

```
@interface HelloWorldViewController()  
  
@end
```

To Control-drag, press and hold the Control key while you drag from the button to the implementation file in the assistant editor pane. As you Control-drag, you should see something like this:



When you release the Control-drag, Xcode displays a popover in which you can configure the action connection you just made:



Note: If you release the Control-drag somewhere other than in the class extension area of `HelloWorldViewController.m`, you might see a different type of popover or nothing at all. If this happens, click inside the view on the canvas to close the popover (if necessary) and try Control-dragging again.

5. In the popover, configure the button's action connection:
 - o In the Connection pop-up menu, choose Action.
 - o In the Name field, enter `changeGreeting:` (be sure to include the colon).

In a later step, you'll implement the `changeGreeting:` method so that it takes the text that the user enters into the text field and displays it in the label.

- o Make sure that the Type field contains `id`.

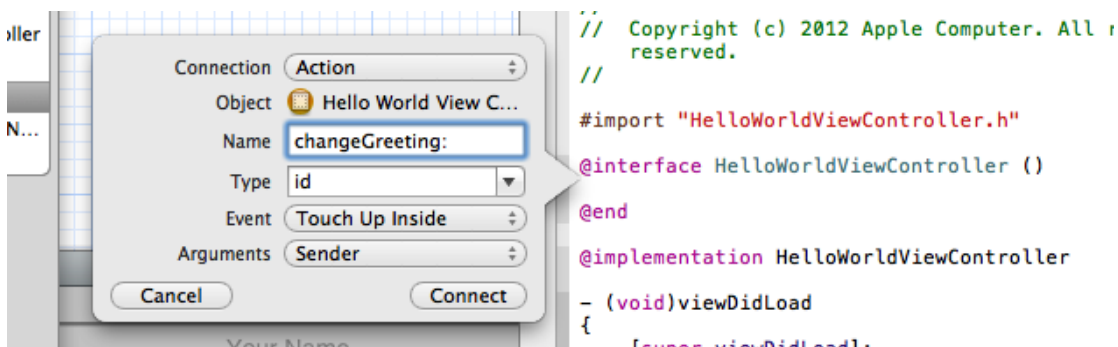
The `id` data type can represent any Cocoa object. You want to use `id` here because it doesn't matter what type of object sends the message.

- o Make sure that the Event pop-up menu contains Touch Up Inside.

You specify the Touch Up Inside event because you want the message to be sent when the user lifts the finger inside the button.

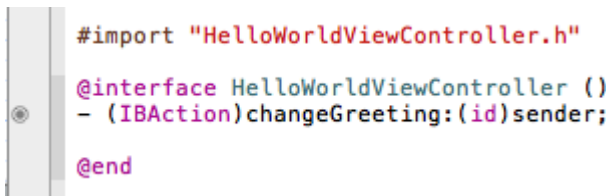
- o Make sure that the Arguments pop-up menu contains Sender.

After you configure the action connection, the popover should look like this:



6. In the popover, click Connect.

Xcode adds a stub implementation of the new `changeGreeting:` method and indicates that the connection has been made by displaying a filled-in circle to the left of the method:



```
#import "HelloWorldViewController.h"

@interface HelloWorldViewController ()
- (IBAction)changeGreeting:(id)sender;

@end
```

When you Control-dragged from the Hello button to the class extension in `HelloWorldViewController.m` file and configured the resulting action, you accomplished two things: You added, through Xcode, the appropriate code to the view controller class (in `HelloWorldViewController.m`), and you created a connection between the button and the view controller. Specifically, Xcode did the following things:

- To `HelloWorldViewController.m`, it added the following action method declaration to the class extension:

```
- (IBAction)changeGreeting:(id)sender;
```
- and added the following stub method to the implementation area:

```
- (IBAction)changeGreeting:(id)sender {
}
```
- **Note:** `IBAction` is a special keyword that is used to tell Xcode to treat a method as an action for target-action connections. `IBAction` is defined to `void`.
- The `sender` parameter in the action method refers to the object that is sending the action message (in this tutorial, the sender is the button).
- It created a connection between the button and the view controller.

Next, you create connections between the view controller and the two remaining UI elements (that is, the label and the text field).

Create Outlets for the Text Field and the Label

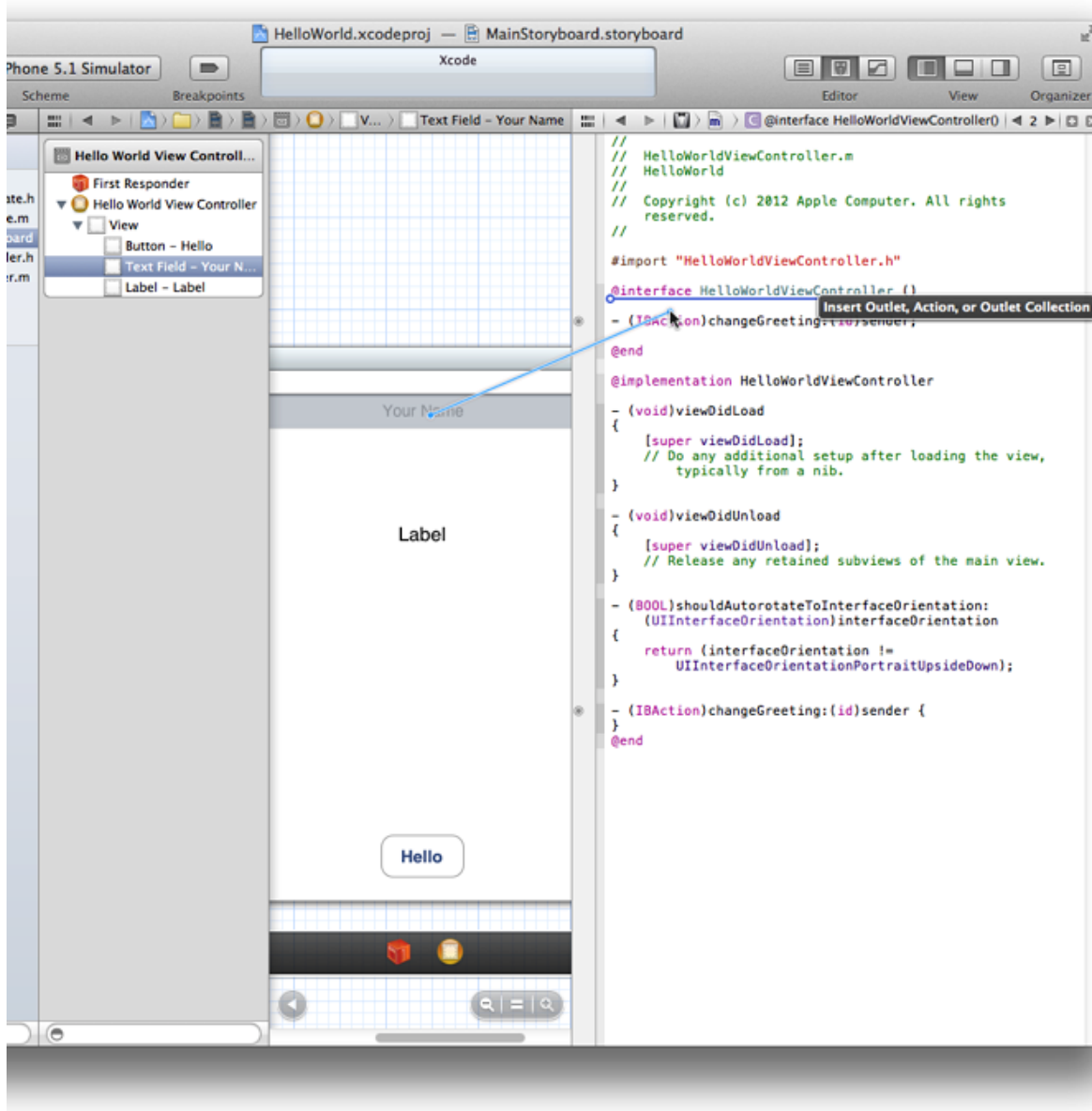
An *outlet* describes a connection between two objects. When you want an object (such as the view controller) to communicate with an object that it contains (such as the text field), you designate the contained object as an outlet. When the app runs, the outlet you create in Xcode is restored, allowing the objects to communicate with each other at runtime.

In this tutorial, you want the view controller to get the user's text from the text field and then display the text in the label. To ensure that the view controller can communicate with these objects, you create outlet connections between them.

The steps you take to add outlets for the text field and label are very similar to the steps you took when you added the button's action. Before you start, make sure that the main storyboard file is still visible on the canvas and that `HelloWorldViewController.m` is still open in the assistant editor.

1. Control-drag from the text field in the view to the class extension in the implementation file.

As you Control-drag, you should see something like this:



It does not matter where you release the Control-drag as long as it's inside the class extension. In this tutorial, the outlet declarations for the text field and the label are shown above the method declaration for the Hello button.

2. In the popover that appears when you release the Control-drag, configure the text field's connection:
 - Make sure that the Connection pop-up menu contains Outlet.
 - In the Name field, type "textField".

You can call the outlet whatever you want, but your code is more understandable when an outlet name bears some relationship to the item it represents.

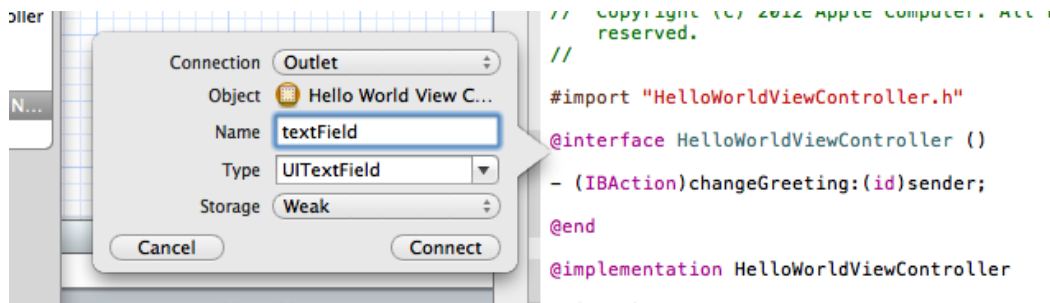
- Make sure that the Type field contains "UITextField".

Setting the Type field to "UITextField" ensures that Xcode connects the outlet only to a text field.

- Make sure that the Storage pop-up menu contains Weak, which is the default value.

You will learn more about strong and weak storage later, in *Acquire Foundational Programming Skills*.

3. After you make these settings, the popover should look like this:



4. In the popover, click Connect.

You accomplished two things by adding an outlet for the text field. Through this procedure:

- Xcode added appropriate code to the implementation file (`HelloWorldViewController.m`) of the view controller class.

Specifically, it added the following declaration to the class extension:

```
@property (weak, nonatomic) IBOutlet UITextField *textField;
```

Note: `IBOutlet` is a special keyword that is used only to tell Xcode to treat the object as an outlet. It's actually defined as nothing so it has no effect at compile time.

- Xcode established a connection from the view controller to the text field.

By establishing the connection between the view controller and the text field, the text that the user enters can be passed to the view controller. As Xcode did with the `changeGreeting:` method declaration, it indicates that the connection has been made by displaying a filled-in circle to the left of the text field declaration.

Note: Earlier versions of Xcode add an `@synthesize` directive in the implementation block for each property you declare using the Control-drag approach. Because the compiler automatically synthesizes accessor methods, these directives are unnecessary. You may safely delete them.

Now add an outlet for the label and configure the connection. Establishing a connection between the view controller and the label allows the view controller to update the label with a string that contains the user's text. The steps you follow for this task are the same as the ones you followed to add the outlet for the text field, but with appropriate changes to the configuration. (Make sure that `HelloWorldViewController.m` is still visible in the assistant editor.)

1. Control-drag from the label in the view to the class extension in `HelloWorldViewController.m` in the assistant editor.
2. In the popover that appears when you release the Control-drag, configure the label's connection:
 - o Make sure that the Connection pop-up menu contains Outlet.
 - o In the Name field, type "label".
 - o Make sure that the Type field contains "UILabel".
 - o Make sure that the Storage pop-up menu contains Weak.
3. In the popover, click Connect.

At this point in the tutorial, you've created a total of three connections to your view controller:


- An action connection for the button
- An outlet connection for the text field
- An outlet connection for the label

You can verify these connections in the Connections inspector.

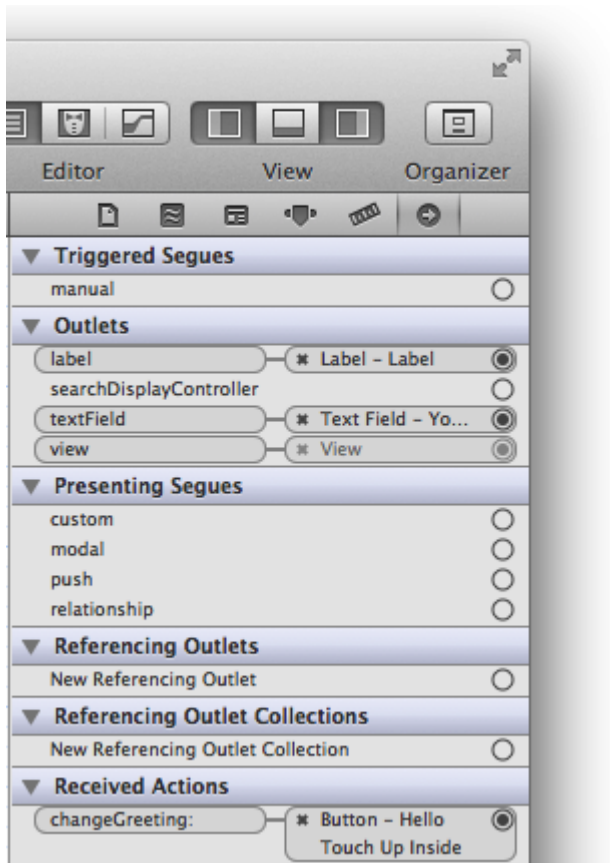
1. Click the Standard editor button to close the assistant editor and switch to the standard editor view.

The Standard editor button is the leftmost Editor button and it looks like this: 

2. Click the Utilities view button to open the utilities area.
3. Select Hello World View Controller in the outline view.
4. Show the Connections inspector in the utilities area.

The Connections inspector button is the rightmost button in the inspector selector bar, and it looks like this: 

In the Connections inspector, Xcode displays the connections for the selected object (in this case, the view controller). In your workspace window, you should see something like this:



You'll see a connection between the view controller and its view, in addition to the three connections you created. Xcode provides this default connection between the view controller and its view; you do not have to access it in any way.

Make the Text Field's Delegate Connection

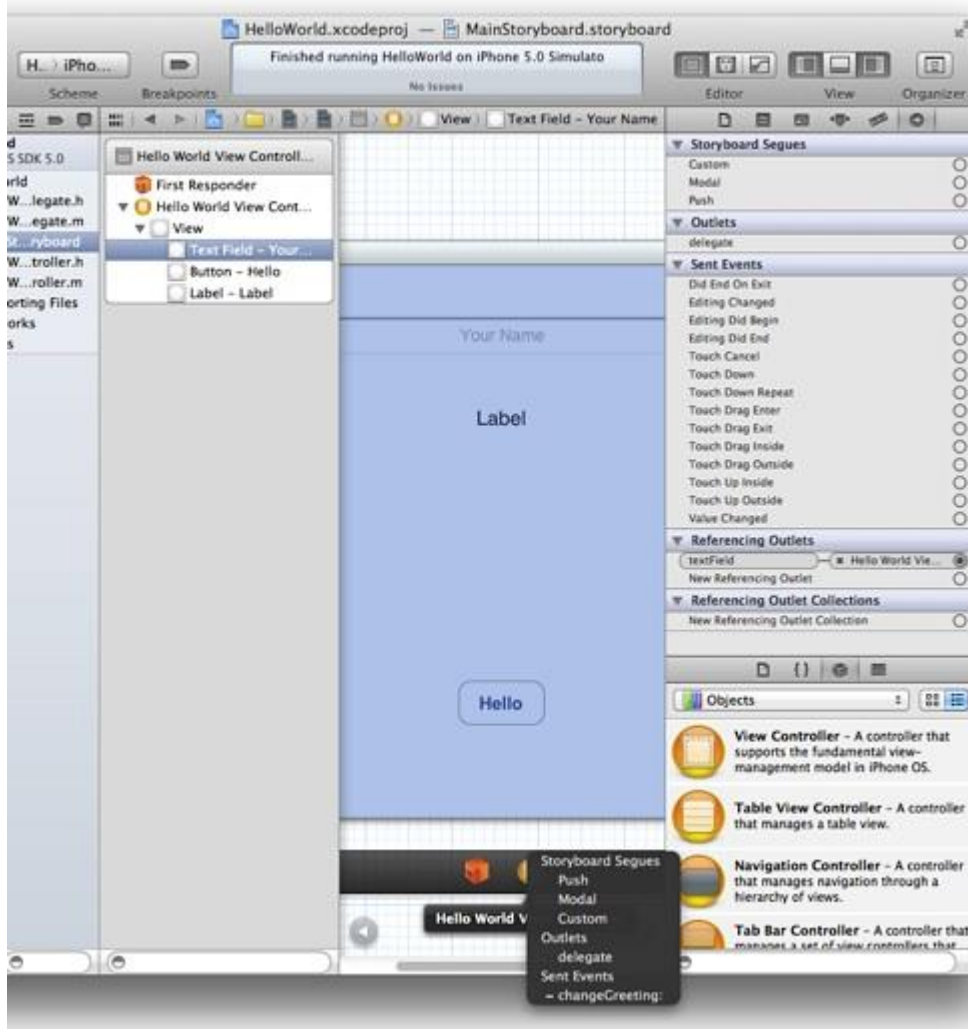
You have one more connection to make in your app: You need to connect the text field to an object that you specify as its delegate. In this tutorial, you use the view controller for the text field's delegate.

You need to specify a delegate object for the text field. This is because the text field sends a message to its delegate when the user taps the Done button in the keyboard (recall that a delegate is an object that acts on the behalf of another object). In a later step, you'll use the method associated with this message to dismiss the keyboard.

Make sure that the storyboard file is open on the canvas. If it's not, select `MainStoryboard.storyboard` in the project navigator.

1. In the view, Control-drag from the text field to the yellow sphere in the scene dock (the yellow sphere represents the view controller object).

When you release the Control-drag, you should see something like this:



2. Select `delegate` in the Outlets section of the translucent panel that appears.

Prepare the App for Accessibility

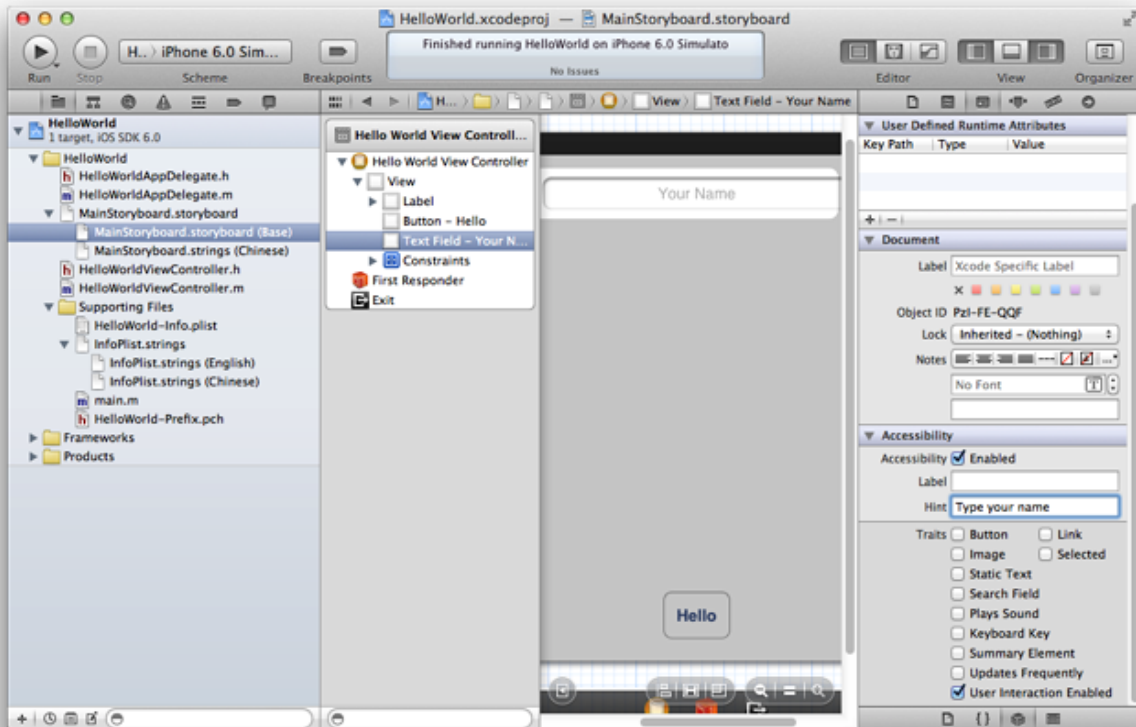
The iOS operating system provides a host of features that help make apps accessible to all users, including those with visual, auditory, and physical disabilities. By making your app accessible, you open it up to millions of people who would otherwise not be able to use it.

A major accessibility feature is VoiceOver, Apple's innovative screen-reading technology. With VoiceOver, users can navigate and control the parts of an app without having to see the screen. By touching a control or other object in the user interface, users can learn where they are, what they can do, and what will happen if they do something.

You can add several accessibility attributes to any view in your user interface. These attributes include the current value of the view (such as the text in a text field), its label, a hint, and a number of traits. For the HelloWorld app, you are going to add a hint to the text field.

Note: Any accessibility text that you add should be localized. To learn how to do this, see *Internationalize Your App* in App Design.

1. Select the storyboard file (base internationalization) in the project navigator.
2. Select the text field.
3. In the Accessibility section of the Identity inspector, type “Type your name” in the Hint field.



Test the App

Click Run to test your app.

You should find that the Hello button becomes highlighted when you click it. You should also find that if you click in the text field, the keyboard appears and you can enter text. However, there's still no way to dismiss the keyboard. To do that, you have to implement the relevant delegate method. You'll do that in the next chapter. For now, quit Simulator.

Implementing the View Controller

There are several parts to implementing the view controller: You need to add a property for the user's name, implement the `changeGreeting:` method, and ensure that the keyboard is dismissed when the user taps Done.

Add a Property for the User's Name

You need to add a property declaration for the string that holds the user's name, so that your code always has a reference to it. Because this property should be public—that is, visible to clients and subclasses—you add this declaration to the view controller's header file, `HelloWorldViewController.h`. Public properties indicate how you intend objects of your class to be used.

A *property declaration* is a directive that tells the compiler how to generate the accessor methods for a variable, such as the variable used to hold the user's name. (You'll learn about accessor methods after you add the property declaration.)

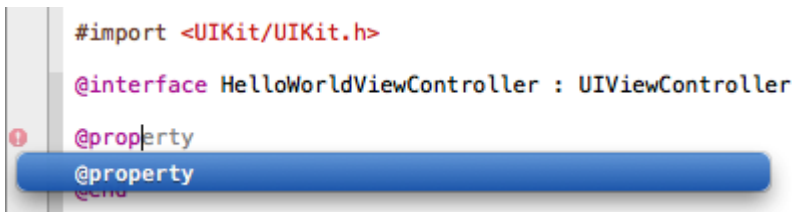
At this point in the tutorial, you don't need to make any further changes to the storyboard file. To give yourself more room in which to add the code described in the following steps, hide the utilities area by clicking the Utilities View button again (or by choosing `View > Utilities > Hide Utilities`).

1. In the project navigator, select `HelloWorldViewController.h`.
2. Before the `@end` statement, write an `@property` statement for the string.

The property declaration should look like this:

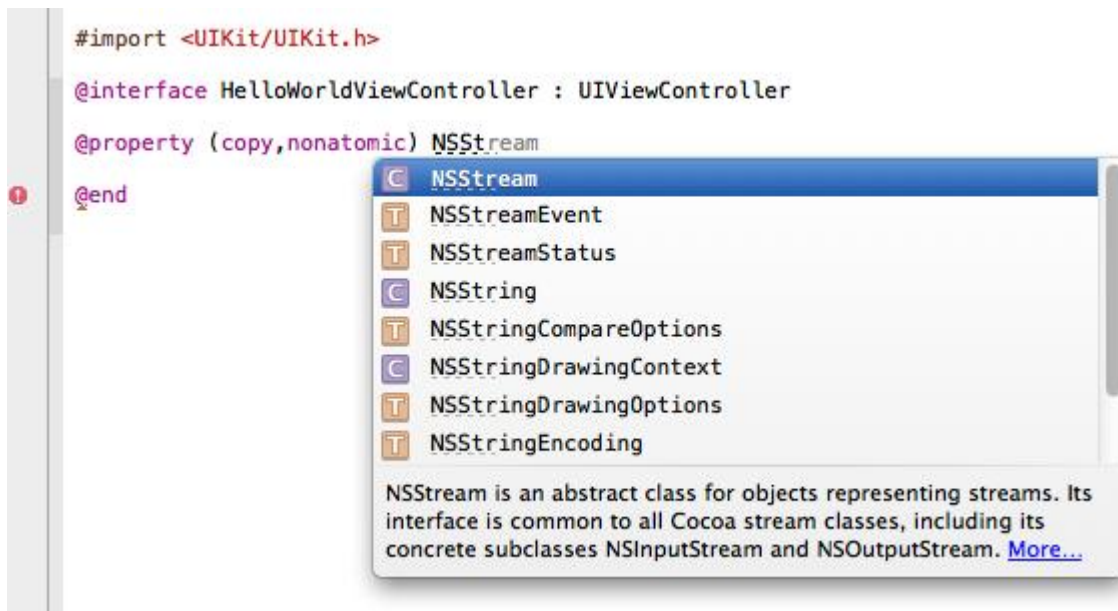
```
@property (copy, nonatomic) NSString *userName;
```

You can copy and paste the code above or you can type it into the editor pane. If you decide to type the code, notice that Xcode suggests completions to what you're typing. For example, as you begin to type `@prop...` Xcode guesses that you want to enter `@property`, so it displays this symbol in an inline suggestion panel that looks similar to this:



If the suggestion is appropriate (as it is in the example shown above), press `Return` to accept it.

As you continue to type, Xcode might offer a list of suggestions from which you can choose. For example, Xcode might display the following list of completions as you type `NSStr...`:



When Xcode displays a completion list, press Return to accept the highlighted suggestion. If the highlighted suggestion isn't correct (as is the case in the list shown above), use the arrow keys to select the appropriate item in the list.

The compiler automatically synthesizes accessor methods for any property you declare. An *accessor method* is a method that gets or sets the value of an object's property (sometimes, accessor methods are also called "getters" and "setters"). For example, the compiler generates declarations of the following getter and setter for the `userName` property you just declared, along with their implementations:

- `-(NSString *)userName;`
- `-(void)setUserName:(NSString *)newUserName;`

The compiler also automatically declares private instance variables to back each declared property. For example, it declares an instance variable named `_userName` that backs the `userName` property.

Note: The compiler adds the accessor methods that it generates to the compiled code; it does not add them to your source code.

Implement the `changeGreeting:` Method

In the previous chapter, "Configuring the View," you configured the Hello button so that when the user taps it, it sends a `changeGreeting:` message to the view controller. In response, you want the view controller to display in the label the text that the user entered in the text field. Specifically, the `changeGreeting:` method should:

- Retrieve the string from the text field and set the view controller's `userName` property to this string.
- Create a new string that is based on the `userName` property and display it in the label.

1. If necessary, select `HelloWorldViewController.m` in the project navigator.

You might have to scroll to the end of the file to see the `changeGreeting:` stub implementation that Xcode added for you.

2. Complete the stub implementation of the `changeGreeting:` method by adding the following code:

```
- (IBAction)changeGreeting:(id)sender {  
  
    self.userName = self.textField.text;  
  
    NSString *nameString = self.userName;  
    if ([nameString length] == 0) {  
        nameString = @"World";  
    }  
    NSString *greeting = [[NSString alloc] initWithFormat:@"Hello, %@!",  
nameString];  
    self.label.text = greeting;  
}
```

There are several interesting things to note in the `changeGreeting:` method:

- `self.userName = self.textField.text;` retrieves the text from the text field and sets the view controller's `userName` property to the result.

In this tutorial, you don't actually use the string that holds the user's name anywhere else, but it's important to remember its role: It's the very simple model object that the view controller is managing. In general, the controller should maintain information about app data in its own model objects—app data shouldn't be stored in user interface elements such as the text field of the HelloWorld app.

- `NSString *nameString = self.userName;` creates a new variable (of type `NSString`) and sets it to the view controller's `userName` property.
- `@"World"` is a string constant represented by an instance of `NSString`. If the user runs your app but does not enter any text (that is, `[nameString length] == 0`), `nameString` will contain the string "World".
- The `initWithFormat:` method is supplied for you by the Foundation framework. It creates a new string that follows the format specified by the format string you supply (much like the `printf` function of the ANSI C library).

In the format string, `%@` acts as a placeholder for a string object. All other characters within the double quotation marks of this format string will be displayed onscreen exactly as they appear.

Configure the View Controller as the Text Field's Delegate

If you build and run the app, you should find that when you click the button, the label shows “Hello, World!” If you select the text field and start typing on the keyboard, though, you should find that you still have no way to dismiss the keyboard when you’re finished entering text.

In an iOS app, the keyboard is shown automatically when an element that allows text entry becomes the first responder; it is dismissed automatically when the element loses first responder status. (Recall that the first responder is the object that first receives notice of various events, such as tapping a text field to bring up the keyboard.) Although there’s no way to directly send a message to the keyboard from your app, you can make it appear or disappear as a side effect of toggling the first responder status of a text-entry UI element.

The `UITextFieldDelegate` protocol is defined by the `UIKit` framework, and it includes the `textFieldShouldReturn:` method that the text field calls when the user taps the Return button (regardless of the actual title of this button). Because you set the view controller as the text field’s delegate (in [“To set the text field’s delegate”](#)), you can implement this method to force the text field to lose first responder status by sending it the `resignFirstResponder` message—which has the side effect of dismissing the keyboard.

Note: A *protocol* is basically just a list of methods. If a class conforms to (or *adopts*) a protocol, it guarantees that it implements the required methods of a protocol. (Protocols can also include optional methods.) A delegate protocol specifies all the messages an object might send to its delegate.

1. If necessary, select `HelloWorldViewController.m` in the project navigator.
2. Implement the `textFieldShouldReturn:` method in the `HelloWorldViewController.m` file.

The method should tell the text field to resign first responder status. The implementation should look something like this:

```
- (BOOL)textFieldShouldReturn:(UITextField *)theTextField {
    if (theTextField == self.textField) {
        [theTextField resignFirstResponder];
    }
    return YES;
}
```

In this app, it’s not really necessary to test the `theTextField == self.textField` expression because there’s only one text field. This is a good pattern to use, though, because there may be occasions when your object is the delegate of more than one object of the same type and you might need to differentiate between them.

3. Select `HelloWorldViewController.h` in the project navigator.

4. To the end of the `@interface` line, add `<UITextFieldDelegate>`.

Your interface declaration should look like this:

```
@interface HelloWorldViewController : UIViewController <UITextFieldDelegate>
...

```

This declaration specifies that your `HelloWorldViewController` class adopts the `UITextFieldDelegate` protocol.

Test the App

Build and run the app. This time, everything should behave as you expect. In Simulator, click Done to dismiss the keyboard after you have entered your name, and then click the Hello button to display “Hello, *Your Name!*” in the label.

Troubleshooting and Reviewing the Code

If you are having trouble getting your app to work correctly, try the problem-solving approaches described in this chapter. If your app still isn’t working as it should, compare your code with the listings shown at the end of this chapter.

Code and Compiler Warnings

Your code should compile without any warnings. If you do receive warnings, it’s recommended that you treat them as very likely to be errors. Because Objective-C is a very flexible language, sometimes the most you get from the compiler is a warning.

Check the Storyboard File

As a developer, if things don’t work correctly, your natural instinct is probably to check your source code for bugs. But when you use Cocoa Touch, another dimension is added. Much of your app’s configuration may be “encoded” in the storyboard. For example, if you haven’t made the correct connections, your app won’t behave as you expect.

- If the text doesn’t update when you click the button, it might be that you didn’t connect the button’s action to the view controller, or that you didn’t connect the view controller’s outlets to the text field or label.
- If the keyboard does not disappear when you click Done, you might not have connected the text field’s delegate or connected the view controller’s `textField` outlet to the text field. Be sure to check the text field’s connections on the storyboard: Control-click the text field to reveal the translucent connections panel. You should see filled-in circles next to the `delegate` outlet and the `textField` referencing outlet.

If you have connected the delegate, there might be a more subtle problem (see the next section, “Delegate Method Names”).

Delegate Method Names

A common mistake with delegates is to misspell the delegate method name. Even if you've set the delegate object correctly, if the delegate doesn't use the right name in its method implementation, the correct method won't be invoked. It's usually best to copy and paste delegate method declarations, such as `textFieldShouldReturn:`, from the documentation.

Code Listings

This section provides listings for the interface and implementation files of the `HelloWorldViewController` class. Note that the listings don't show comments and other method implementations that are provided by the Xcode template.

The Interface file: `HelloWorldViewController.h`

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

@interface HelloWorldViewController : UIViewController <UITextFieldDelegate>

@property (copy, nonatomic) NSString *userName;

@end
```

The Implementation File: `HelloWorldViewController.m`

```
#import "HelloWorldViewController.h"

@interface HelloWorldViewController ()

@property (weak, nonatomic) IBOutlet UITextField *textField;
@property (weak, nonatomic) IBOutlet UILabel *label;

- (IBAction)changeGreeting:(id)sender;

@end

@implementation HelloWorldViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
}

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
{
    return (interfaceOrientation != UIInterfaceOrientationPortraitUpsideDown);
}
```

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

    self.userName = self.textField.text;

    NSString *nameString = self.userName;
    if ([nameString length] == 0) {
        nameString = @"World";
    }
    NSString *greeting = [[NSString alloc] initWithFormat:@"Hello, %@!", nameString];
    self.label.text = greeting;
}

- (BOOL)textFieldShouldReturn:(UITextField *)textField {

    if (textField == self.textField) {
        [textField resignFirstResponder];
    }
    return YES;
}

@end
```