Start Developing iOS Apps Today

Developer

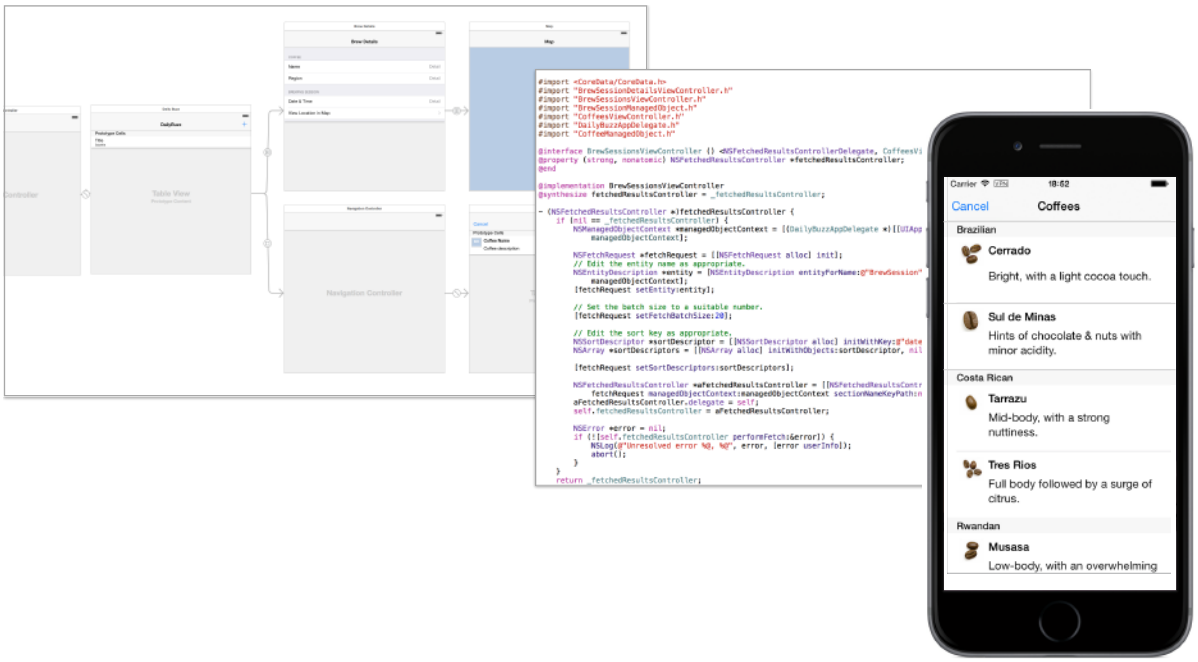
Contents

Objective-CSwift

Introduction

Setup

Start Developing iOS Apps Today is the perfect starting point for creating apps that run on iPad, iPhone, and iPod touch. View this guide's four short modules as a gentle introduction to building your first app—including the tools you need and the major concepts and best practices that will ease your path.



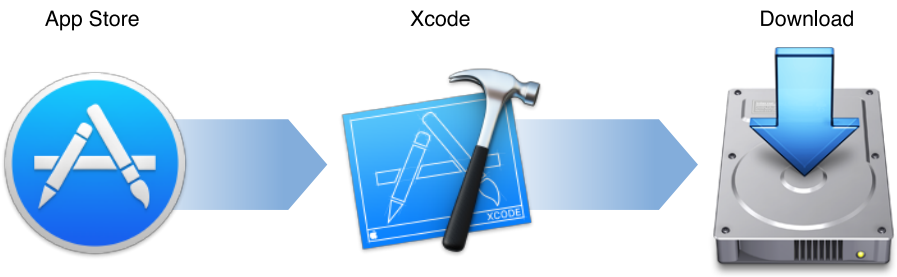
The first three modules end with a tutorial, where you'll implement what you've learned. At the end of the last tutorial, you'll have created a simple to-do list app.

After you build your first app and before you start your next endeavor, read the fourth module. It explores the technologies and frameworks you might want to adopt.

Even though it takes you through every step of building a simple app, to benefit most from this guide, it helps to be acquainted with computer programming in general and with object-oriented programming in particular.

Get the Tools

Before you start developing great apps, set up a development environment to work in and make sure you have the right tools.



To develop iOS apps, you need:

- A Mac computer running OS X 10.9.4 or later
- Xcode (latest version)
- iOS SDK

Xcode is Apple's integrated development environment (IDE). Xcode includes a source editor, a graphical user interface editor, and many other features. The iOS SDK extends Xcode to include the tools, compilers, and frameworks you need specifically for iOS development.

Download the latest version of Xcode on your Mac for free from the App Store. The iOS SDK is included with Xcode.

To download the latest version of Xcode

1. Open the App Store app on your Mac (by default it's in the Dock).
2. In the search field in the top-right corner, type `Xcode` and press the Return key.
3. Click Free FREE .

   Xcode is downloaded into your `/Applications` directory.

Review a Few Objective-C Concepts

As you write code in the tutorials, you'll be working with the *Objective-C programming language*. Objective-C is built on top of the C programming language and provides object-oriented capabilities and a dynamic runtime. You get all of the familiar elements, such as primitive types (`int`, `float`, and so on), structures, functions, pointers, and control flow constructs (`while`, `if...else`, and `for` statements). You also have access to the standard C library routines, such as those declared in `stdlib.h` and `stdio.h`.

Objects Are Building Blocks for Apps

When you build an iOS app, most of your time is spent working with objects.

Objects package data with related behavior. An app is a large ecosystem of interconnected objects that communicate with each other to perform specific tasks, such as displaying a visual interface, responding to user input, and storing information. You use many different types of objects to build your app, ranging from interface elements, such as buttons and labels, to data objects, such as strings and arrays.

Classes Are Blueprints for Objects

A *class* describes the behavior and properties common to any particular type of object.

In the same way that multiple buildings constructed from the same blueprint are identical in structure, every instance of a class shares the same properties and behavior as all other instances of that class. You can write your own classes or use framework classes that have been defined for you.

You make an object by creating an *instance* of a particular class. You do this by allocating the object and initializing it with acceptable default values. When you *allocate* an object, you set aside enough memory for the object and set all instance variables to zero. *Initialization* sets an object's initial state—that is, its instance variables and properties—to reasonable values and then returns the object. The purpose of initialization is to return a usable object. You need to both allocate and initialize an object to be able to use it.

A fundamental concept in Objective-C programming is *class inheritance*, the idea that a class inherits behaviors from a parent class. When one class inherits from another, the child—or *subclass*—inherits all the behavior and properties defined by the parent. The subclass can define its own additional behavior and properties or override the behavior of the parent. Thus you can extend the behaviors of a class without duplicating its existing behavior.

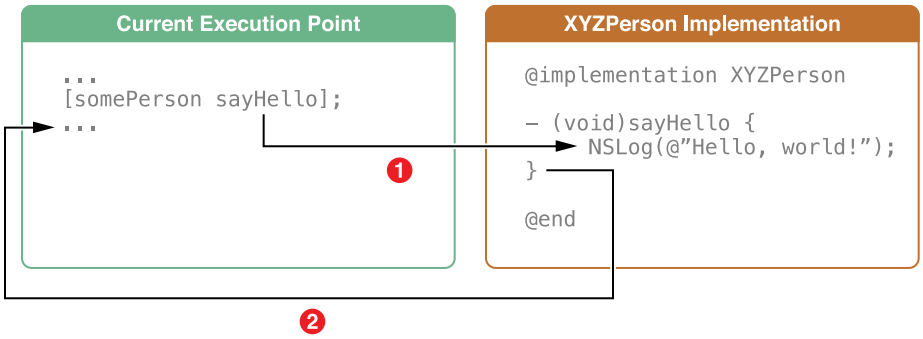
Objects Communicate Through Messages

Objects interact by sending each other messages at runtime. In Objective-C terms, one object **sends a message** to another object by **calling a method** on that object.

Although there are several ways to send messages between objects in Objective-C, by far the most common is the basic syntax that uses square brackets. If you have an object `somePerson` of class `XYZPerson`, you can send it the `sayHello` message like this:

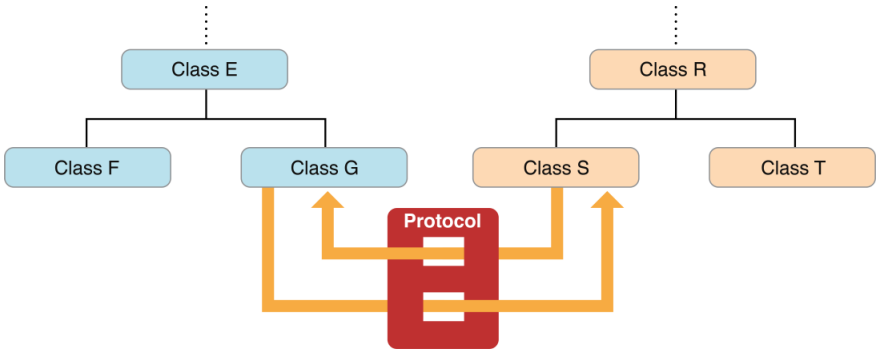
```
[somePerson sayHello];
```

The reference on the left, `somePerson`, is the receiver of the message. The message on the right, `sayHello`, is the name of the method to call on that receiver. In other words, when the above line of code is executed, `somePerson` will be sent the `sayHello` message.



Protocols Define Messaging Contracts

A *protocol* defines a set of behaviors that are expected of an object in a given situation. A protocol comes in the form of a programmatic interface, one that any class may implement. Using protocols, two classes distantly related by inheritance can communicate with each other to accomplish a certain goal, such as parsing XML code or copying an object.



Any class that can provide useful behavior to other classes can declare a programmatic interface for vending that behavior anonymously. Any other class can choose to adopt the protocol and implement one or more of the protocol's methods, making use of the behavior.

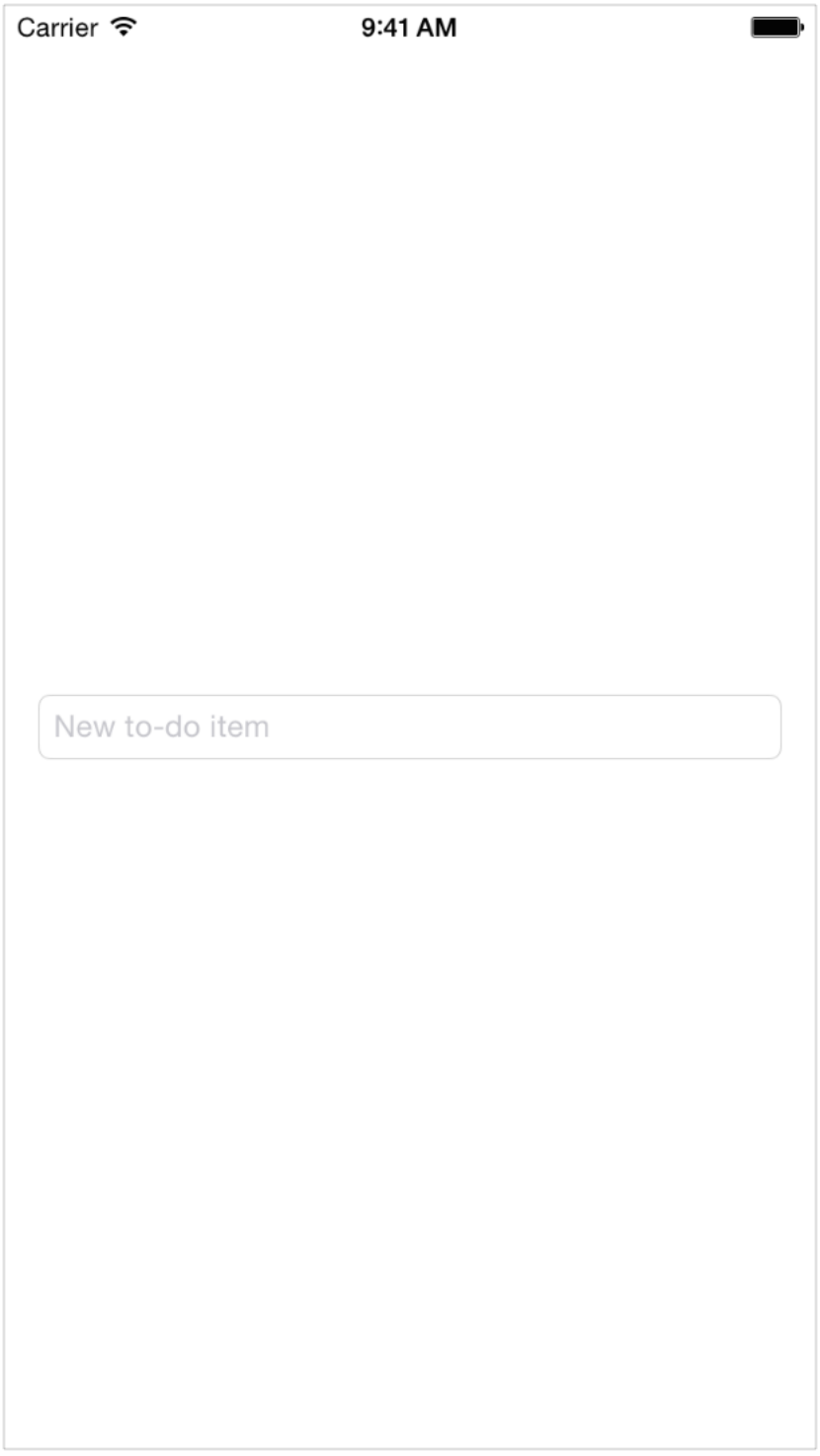
Tutorial: Basics

This tutorial takes you through the process of creating a simple *user interface* and adding the custom behavior that transforms the interface into a working app. The finished app will run on iPhone and iPad.

This tutorial teaches you how to:

- Use Xcode to create and manage a project
- Identify the key pieces of an Xcode project
- Run your app in iOS Simulator
- Create a basic user interface in a storyboard
- Preview your user interface
- Adopt Auto Layout to add flexibility to your user interface

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



These tutorials use Xcode 6.1.1 and iOS SDK 8.1, so if you're following along using other software versions, you might notice some differences.

Create a New Project

Xcode includes several built-in app **templates** for developing common types of iOS apps, such as games, apps with tab-based navigation, and table-view-based apps. Most of these templates have preconfigured interface and source code files. For this tutorial, you'll start with the most basic template: Single View Application.

Working with the Single View Application template helps you understand the basic structure of an iOS app and how content gets onscreen. After you learn how everything works, you can use one of the other templates for your own app.

To create a new project

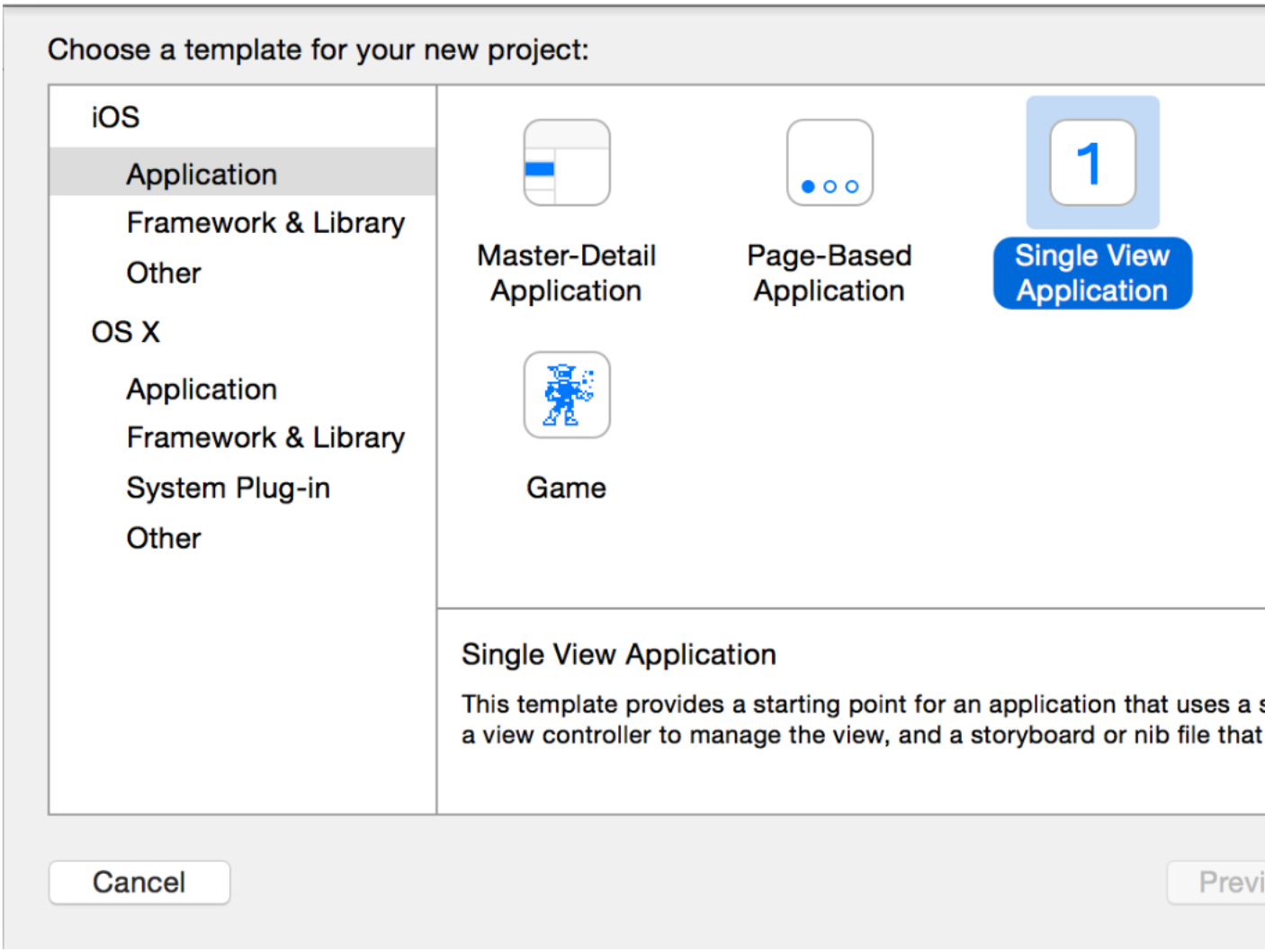
1. Open Xcode from the `/Applications` directory.

The Xcode welcome window appears.

If a project window appears instead of the welcome window, don't worry—you probably created or opened a project in Xcode previously. Just use the menu item in the next step to create the project.

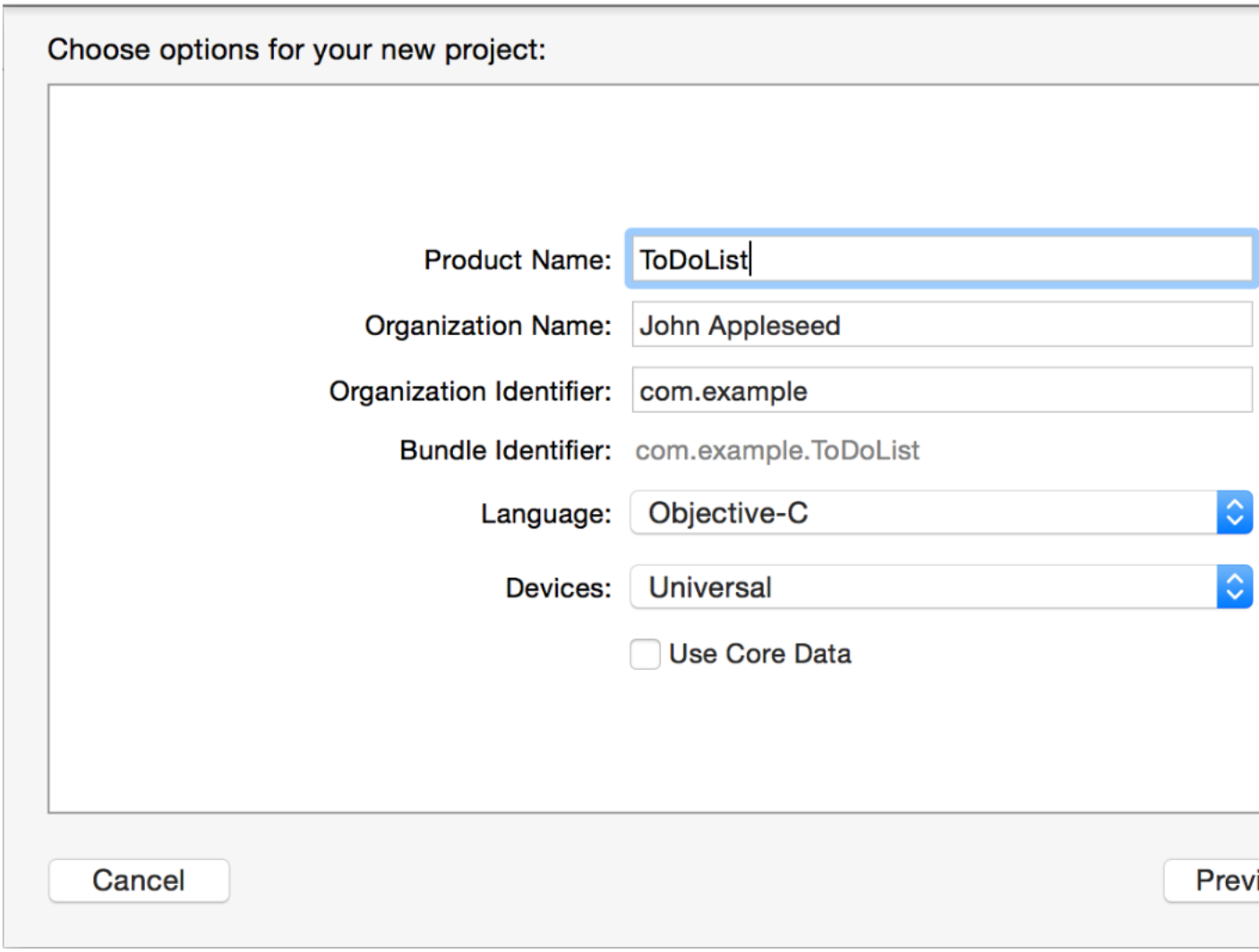
2. In the welcome window, click "Create a new Xcode project" (or choose File > New > Project).

Xcode opens a new window and displays a dialog in which you choose a template.



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

5. In the dialog that appears, name your app and choose additional options for your project.



Use the following values:

- Product Name: `ToDoList`

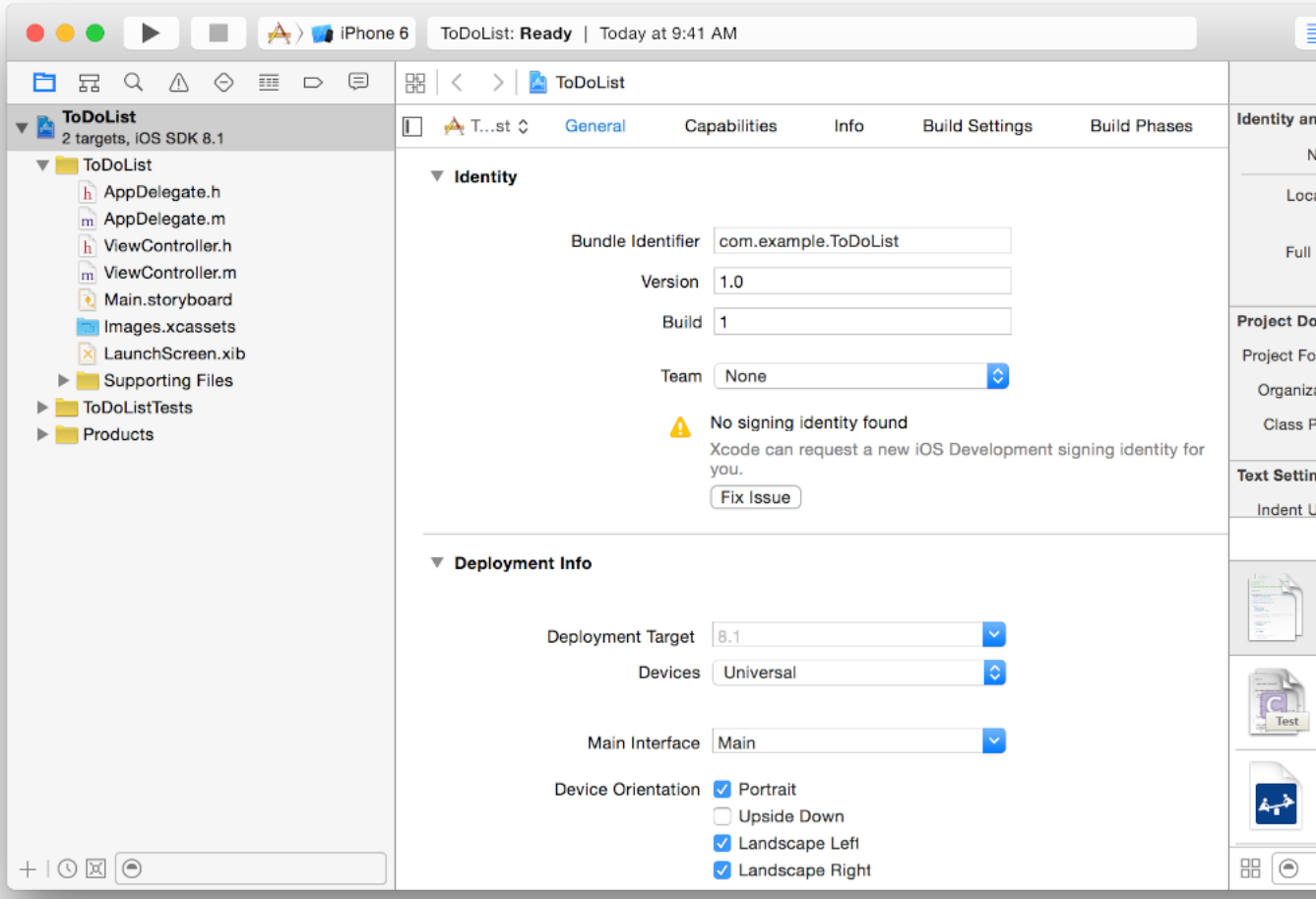
  Xcode uses the product name you entered to name your project and the app.
- Organization Name: The name of your organization or your own name. You can leave this blank.
- Organization Identifier: Your organization identifier, if you have one. If you don't, use `com.example`.
- Bundle Identifier: This value is automatically generated based on your product name and organization identifier.
- Language: Objective-C
- Devices: Universal

  A Universal app is one that runs on both iPhone and iPad.

- Use Core Data: Leave unselected.

6. Click Next.
7. In the dialog that appears, choose a location to save your project and click Create.

   Xcode opens your new project in a window (called the *workspace window*):

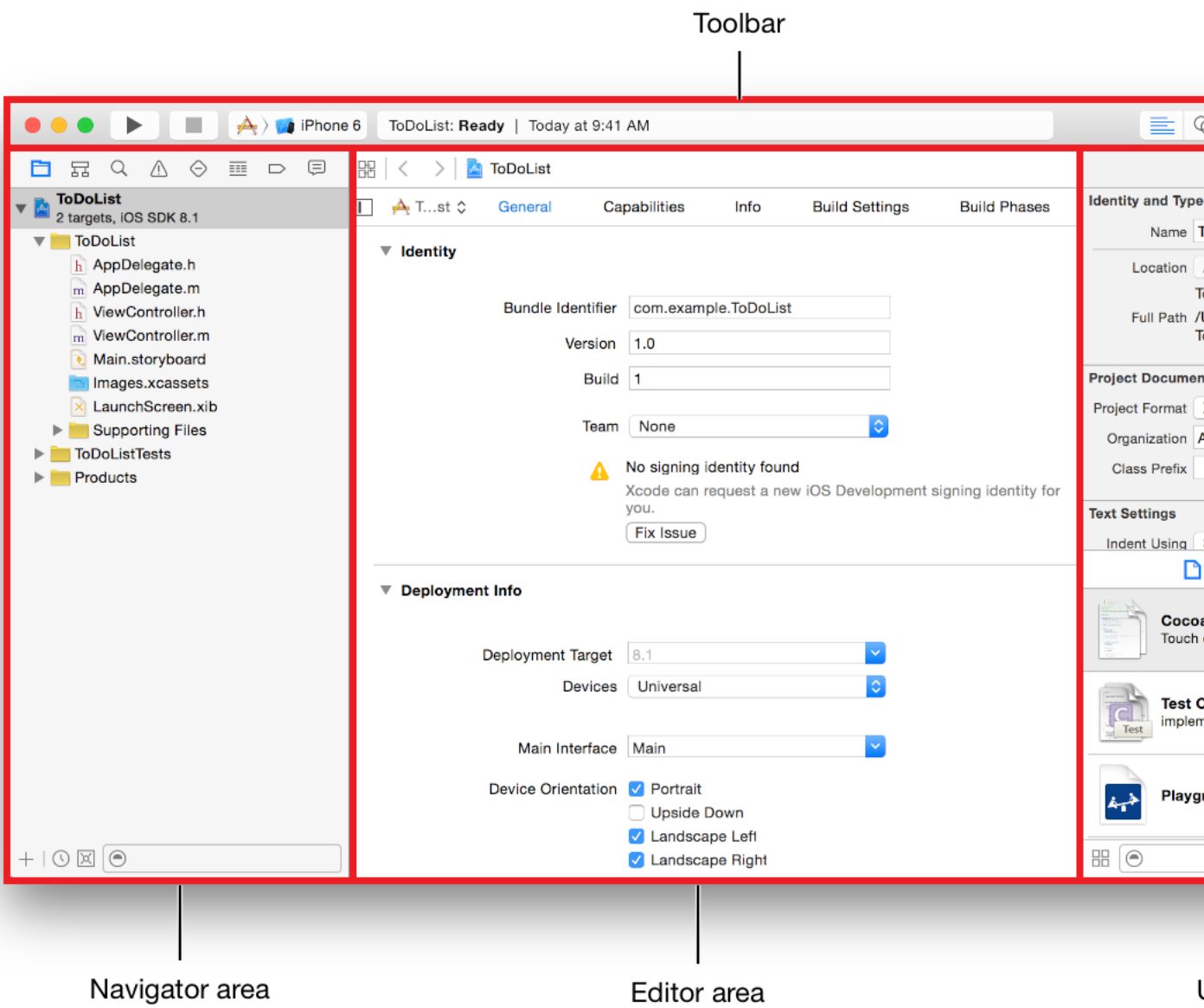


You might notice a warning message that says “No signing identity found.” This warning means you haven’t set up Xcode for iOS development yet, but you can complete the tutorial without doing it. At the end of this document, there’s a link to a guide that helps you set up Xcode to continue iOS development.

Get Familiar with Xcode

Xcode includes everything you need to create an app. It not only organizes the files that go into creating an app, it provides editors for code and interface elements, allows you to build and run your app, and includes a powerful integrated debugger.

Take a few moments to familiarize yourself with the Xcode workspace. You'll use the controls identified in the window below throughout this tutorial. Click different buttons to get a feel for how they work. If you want more information on part of the interface, read the help articles for it—to find them, Control-click an area of Xcode and choose the article from the shortcut menu that appears.



Run iOS Simulator

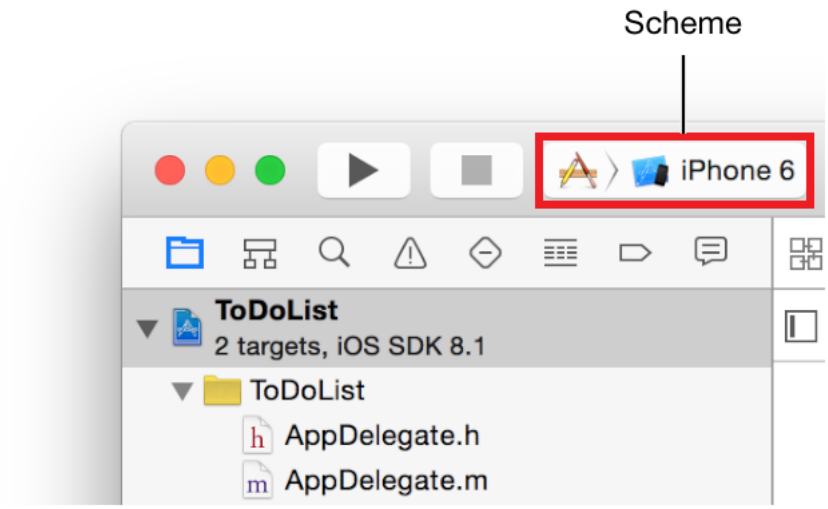
Because you based your project on an Xcode template, the basic app environment is automatically set up for you. Even though you haven't written any code, you can build and run the Single View Application template without any additional configuration.

To build and run your app, use the **iOS Simulator** app that's included in Xcode. As its name implies, iOS Simulator gives you an idea of how your app would look and behave if it were running on an iOS device.

iOS Simulator can model a number of different types of hardware—iPad, iPhone with different screen sizes, and so on—so you can simulate your app on every device you're developing for. In this tutorial, use the iPhone 6 option.

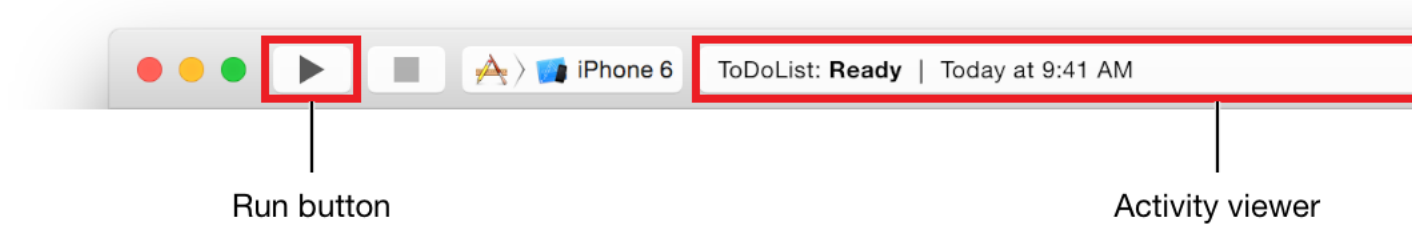
To run your app in iOS Simulator

1. Make sure iPhone 6 is selected in the Scheme pop-up menu in the Xcode toolbar.



   Go ahead and look through the menu to see what other hardware options are available in iOS Simulator.

2. Click the Run button, located in the top-left corner of the Xcode toolbar.



   Alternatively, choose Product > Run (or press Command-R).

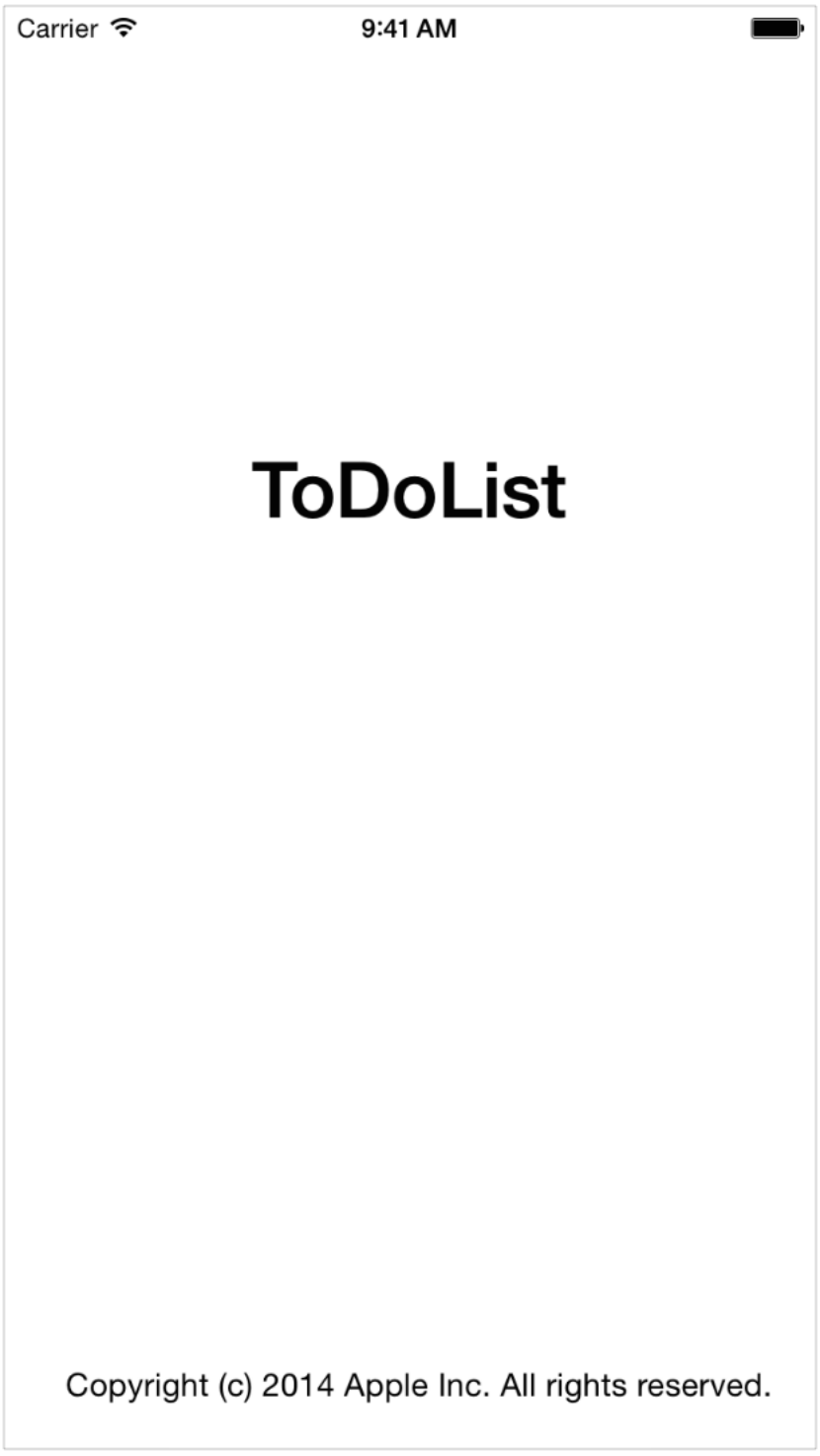
If you're running an app for the first time, Xcode asks whether you'd like to enable developer mode on your Mac. Developer mode allows Xcode access to certain debugging features without requiring you to enter your password each time. Decide whether you'd like to enable developer mode and follow the prompts. If you choose not to enable it, you may be asked for your password later on. The tutorial assumes developer mode is enabled.

3. Watch the Xcode toolbar as the build process completes.

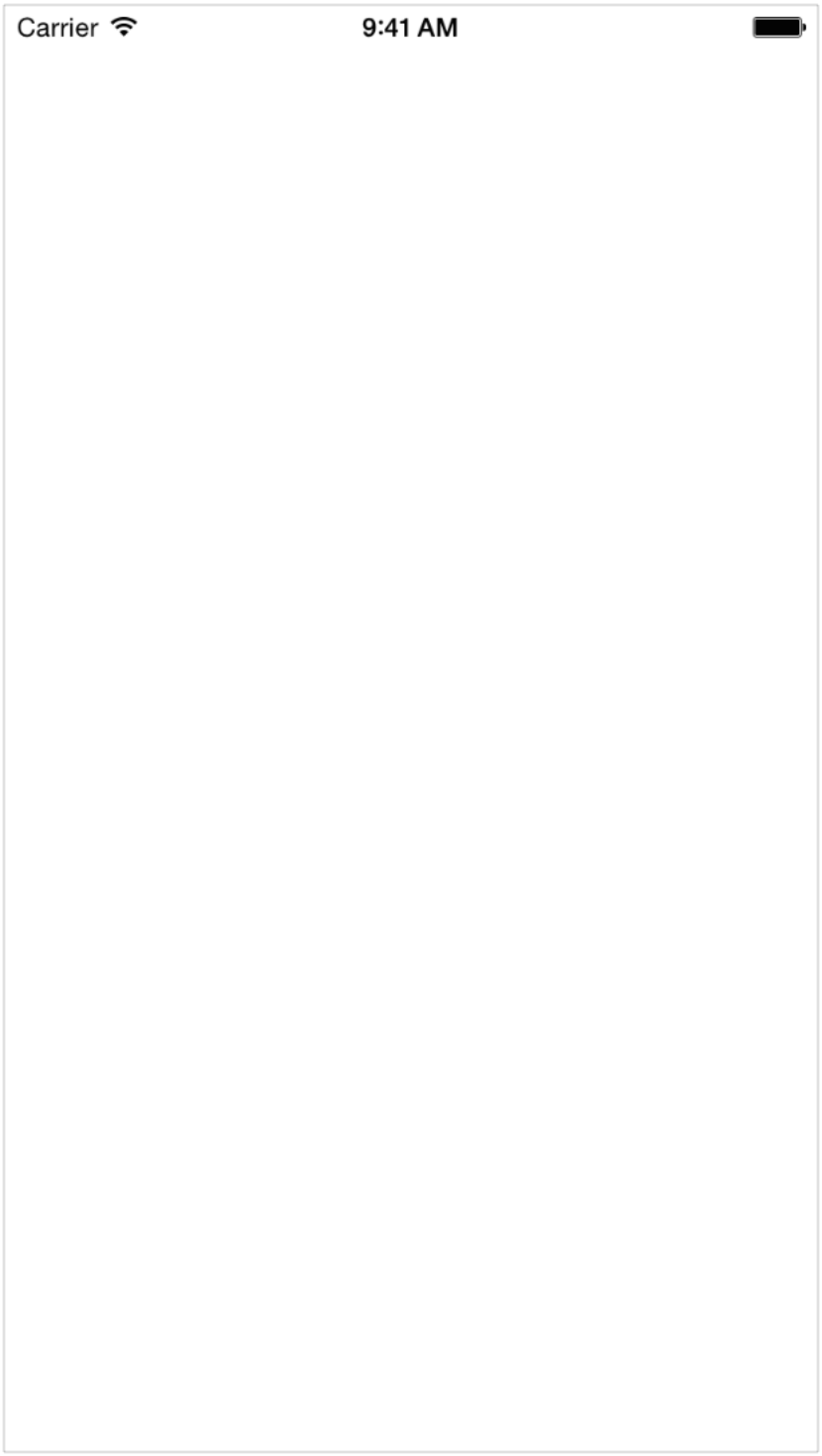
   Xcode displays messages about the build process in the *activity viewer*, which is in the middle of the toolbar.

After Xcode finishes building your project, iOS Simulator starts automatically. It may take a few moments to start up the first time.

iOS Simulator opens in iPhone mode, just as you specified. On the simulated iPhone screen, iOS Simulator launches your app. Before the app finishes launching, you'll see a launch screen with your app's name, ToDoList.



Then, you should see something like this:



Right now, the Single View Application template doesn't do much—it just displays a white screen. Other templates have more complex behavior. It's important to understand a template's uses before you extend it to make your own app. Running your app in iOS Simulator with no modifications is a good way to start developing that understanding.

Quit iOS Simulator by choosing iOS Simulator > Quit iOS Simulator (or pressing Command-Q).

Review the Source Code

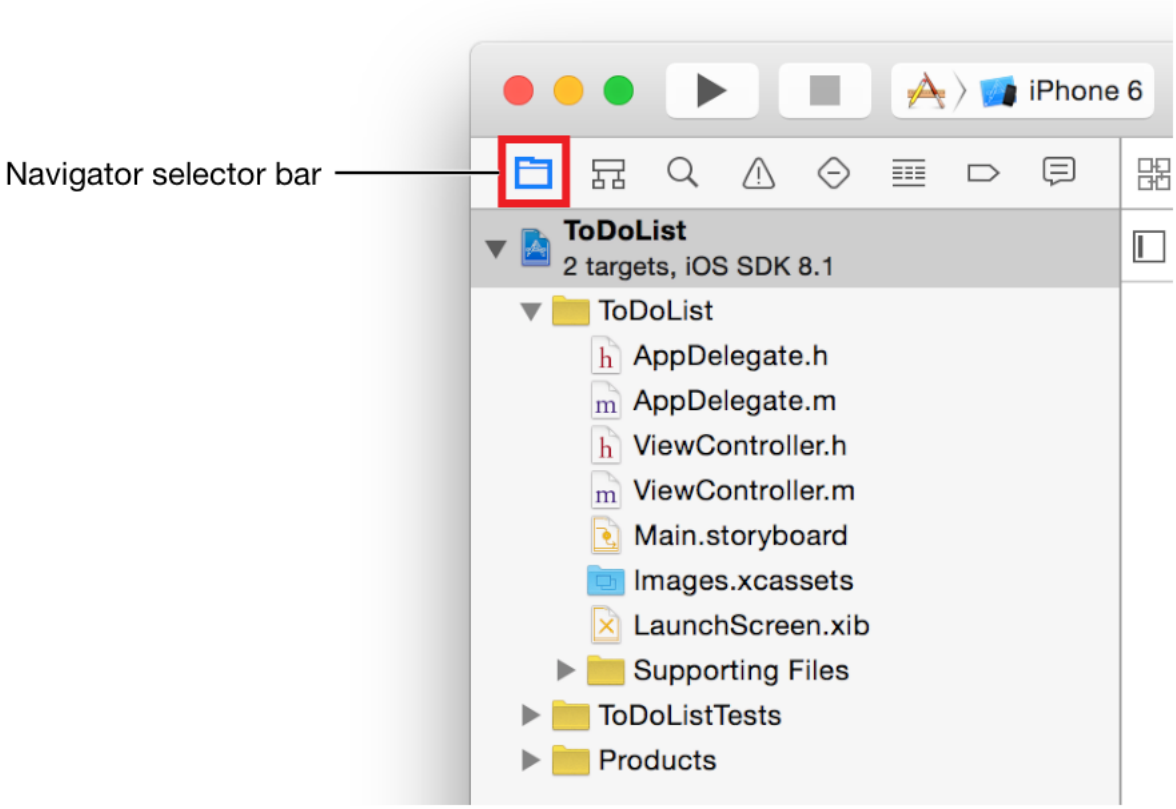
The Single View Application template comes with a few source code files that set up the app environment. Most of the work is done by the `UIApplicationMain` function, which is automatically called in your project's `main.m` source file. The `UIApplicationMain` function creates an application object that sets up the infrastructure for your app to work with the iOS system. This includes creating a *run loop* that delivers input events to your app.

You won't be dealing with the `main.m` source file directly, but it's interesting to understand how it works.

To look at the main.m source file

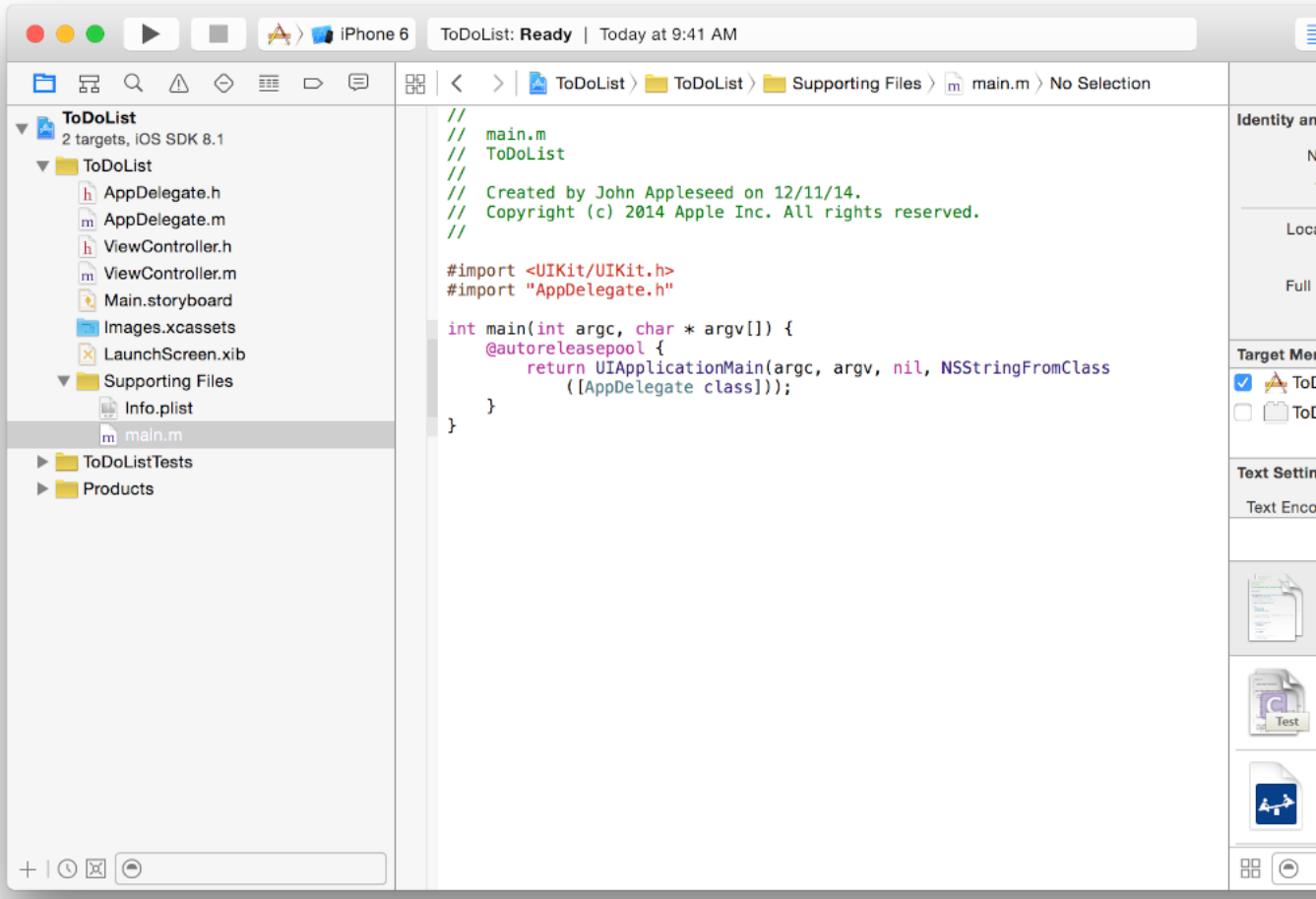
1. Make sure the project navigator is open in the navigator area.

The *project navigator* displays all the files in your project. If the project navigator isn't open, click the leftmost button in the navigator selector bar. (Alternatively, display it by choosing View > Navigators > Show Project Navigator.)



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.



Alternatively, double-click the `main.m` file to open it in a separate window.

The main.m File and the UIApplicationMain Function

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

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

The `@autoreleasepool` statement supports memory management for your app. Automatic Reference Counting (ARC) makes memory management straightforward by getting the compiler to keep track of who owns an object; `@autoreleasepool` is part of the memory management infrastructure.

The call to `UIApplicationMain` creates two important initial components of your app:

- An instance of the `UIApplication` class, called the *application object*.

  The application object manages the app event loop and coordinates other high-level app behaviors. The `UIApplication` class, defined in the UIKit framework, doesn't require you to write any additional code to get it to do its job.

- An instance of the `AppDelegate` class, called the **app delegate**.

  Xcode created this class for you as part of setting up the Single View Application template. The app delegate creates the window where your app's content is drawn and provides a place to respond to state transitions within the app. The app delegate is where you write your custom app-level code. Like all classes, the `AppDelegate` class is defined in two source code files in your app: in the interface file, `AppDelegate.h`, and in the implementation file, `AppDelegate.m`.

As your app starts up, the application object calls predefined methods on the app delegate to give your custom code a chance to do its job—that's when the interesting behavior for an app is executed.

The App Delegate Source Files

To understand the role of the app delegate in more depth, view your app delegate source files, `AppDelegate.h` (the interface file) and `AppDelegate.m` (the implementation file). To view the app delegate interface file, select `AppDelegate.h` in the project navigator. The app delegate interface contains a single property: `window`. With this property the app delegate keeps track of the window in which all of your app content is drawn.

Next, view the app delegate implementation file by selecting `AppDelegate.m` in the project navigator. The app delegate implementation contains "skeletons" of important methods. These predefined methods allow the application object to talk to the app delegate. During a significant runtime event—for example, app launch, low-memory warnings, and app termination—the application object calls the corresponding method in the app delegate, giving it an opportunity to respond appropriately. You don't need to do anything special to make sure these methods get called at the correct time—the application object handles that part of the job for you.

Each of these automatically implemented methods has a default behavior. If you leave the skeleton implementation empty or delete it from your `AppDelegate.m` file, you get the default behavior whenever that method is called. Use these skeletons to put additional custom code that you want to be executed when the methods are called. In this tutorial, you won't be using any custom app delegate code, so you don't have to make any changes to the `AppDelegate.m` file.

Open Your Storyboard

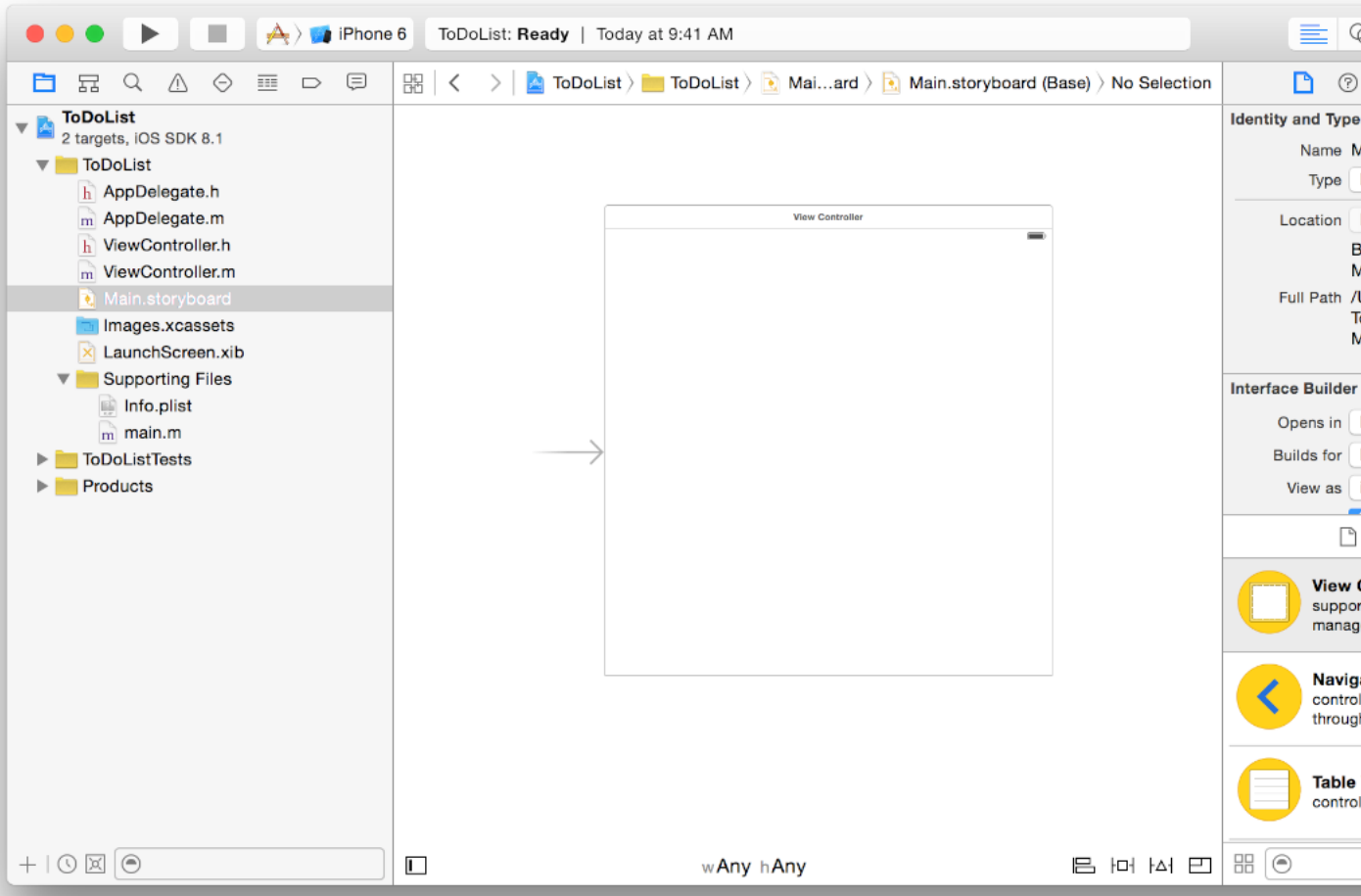
You're ready to start working on a storyboard for your app. A *storyboard* is a visual representation of the app's user interface, showing screens of content and the transitions between them. You use storyboards to lay out the flow—or story—that drives your app.

To open your storyboard

- In the project navigator, select `Main.storyboard`.

  Xcode opens the storyboard in *Interface Builder*—its visual interface editor—in the editor area. The background of the storyboard is the *canvas*. You use the canvas to add and arrange user interface elements.

Your storyboard should look similar to this:



At this point, the storyboard in your app contains one *scene*, which represents a screen of content in your app. The arrow that points to the left side of the scene on the canvas is the *storyboard entry point*, which means that this scene is loaded first when the app starts. Right now, the scene that you see on the canvas contains a single view that's managed by a view controller. You'll learn more about the roles of views and view controllers after this tutorial.

When you ran your app in the iPhone 6 iOS Simulator, the view in this scene is what you saw on the device screen. But when you look at the scene on the canvas, you'll notice that it doesn't have the exact dimensions of the iPhone 6 screen. This is because the scene on the canvas is a generalized representation of your interface that can apply to any device in any orientation. You use this representation to create an *adaptive* interface, which is an interface that automatically adjusts so that it looks good in the context of the current device and orientation. You'll learn how to make your interface adaptive in a little while.

Build the Basic Interface

It's time to build the basic interface for the scene that lets you add a new item to the to-do list.

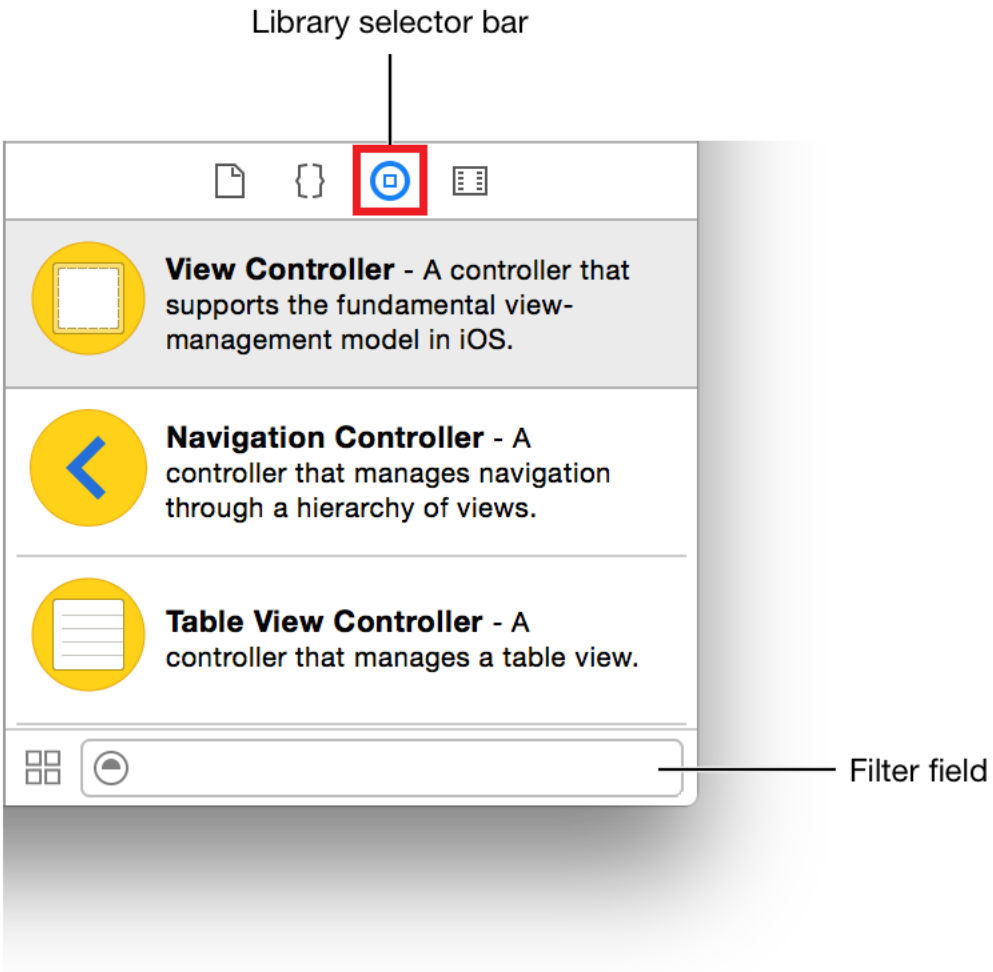
Xcode provides a library of objects that you can add to a storyboard file. Some of these are user interface elements that belong in a view, such as buttons and text fields. Others, such as view controllers and gesture recognizers, define the behavior of your app but don't appear onscreen.

To be able to add an item to the to-do list, you need a text field, the interface element that lets a user input a single line of text.

To add a text field to your scene

1. Open the Object library.

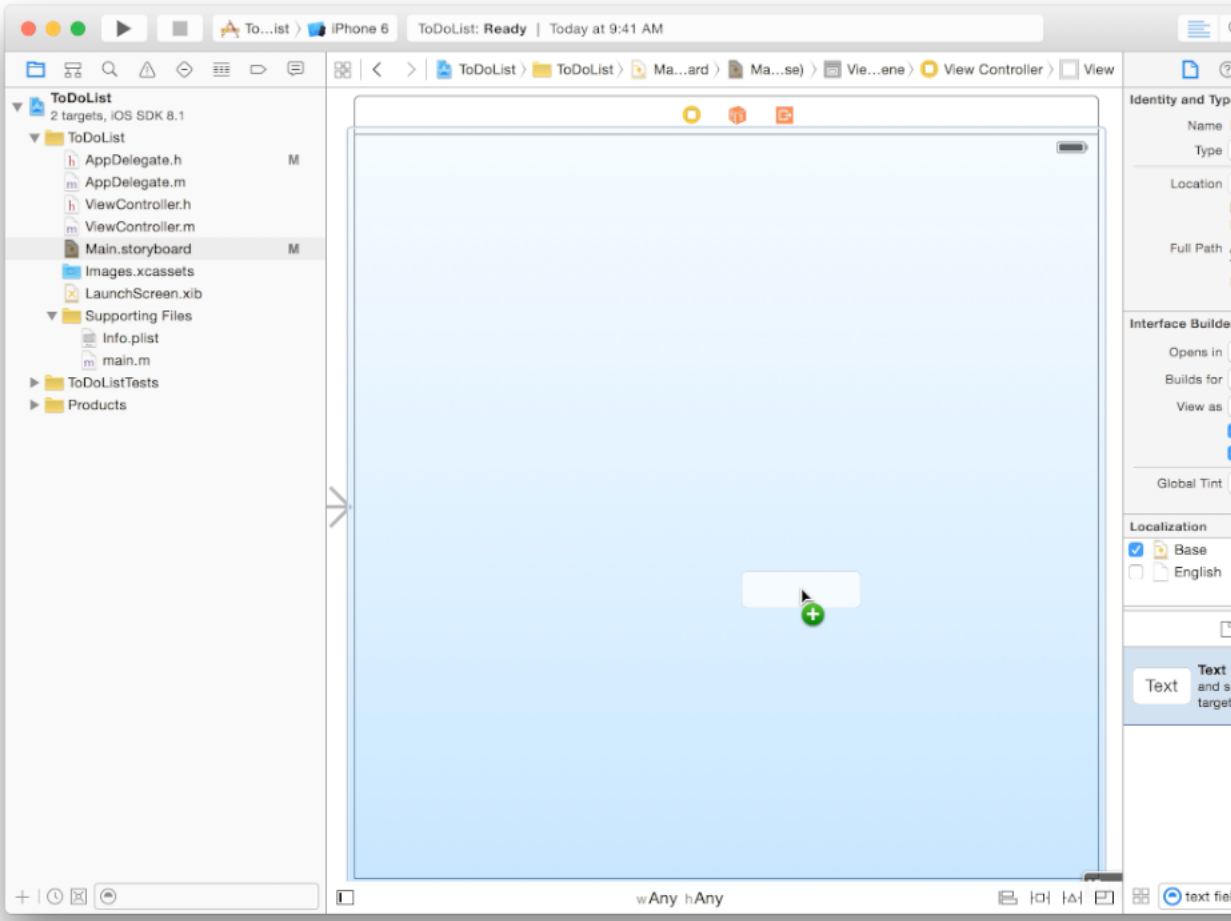
The *Object library* appears at the bottom of the utility area. If you don't see the Object library, click its button, which is the third button from the left in the library selector bar. (Alternatively, display it by choosing View > Utilities > Show Object Library.)



A list appears showing each object's name, description, and visual representation.

2. In the Object library, type `text field` in the filter field to find the Text Field object quickly.

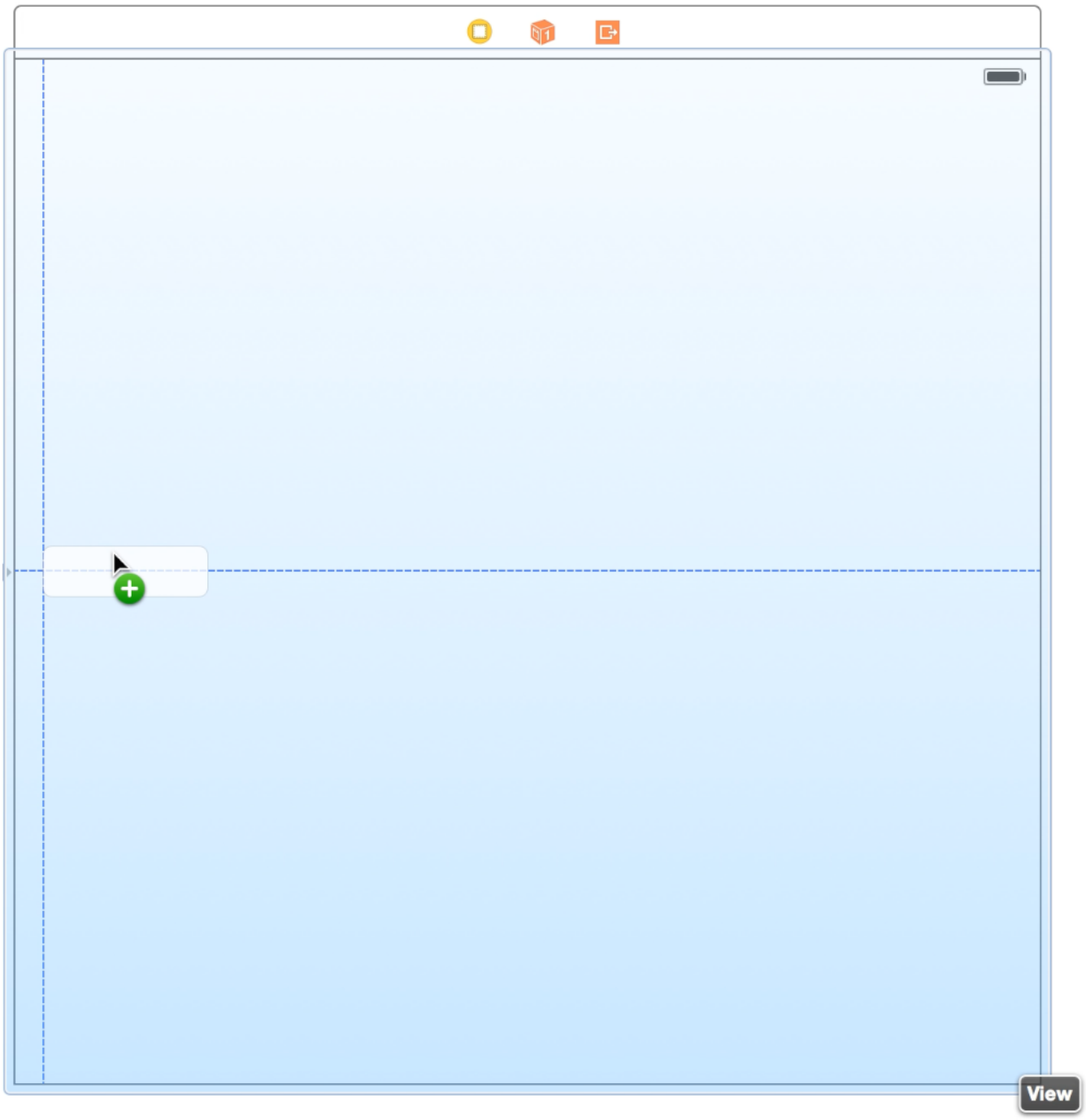
3. Drag a Text Field object from the Object library to your scene.



If you need to, you can zoom in using Editor > Canvas > Zoom.

4. Drag the text field so that it's centered vertically and aligned with the left margin in the scene.

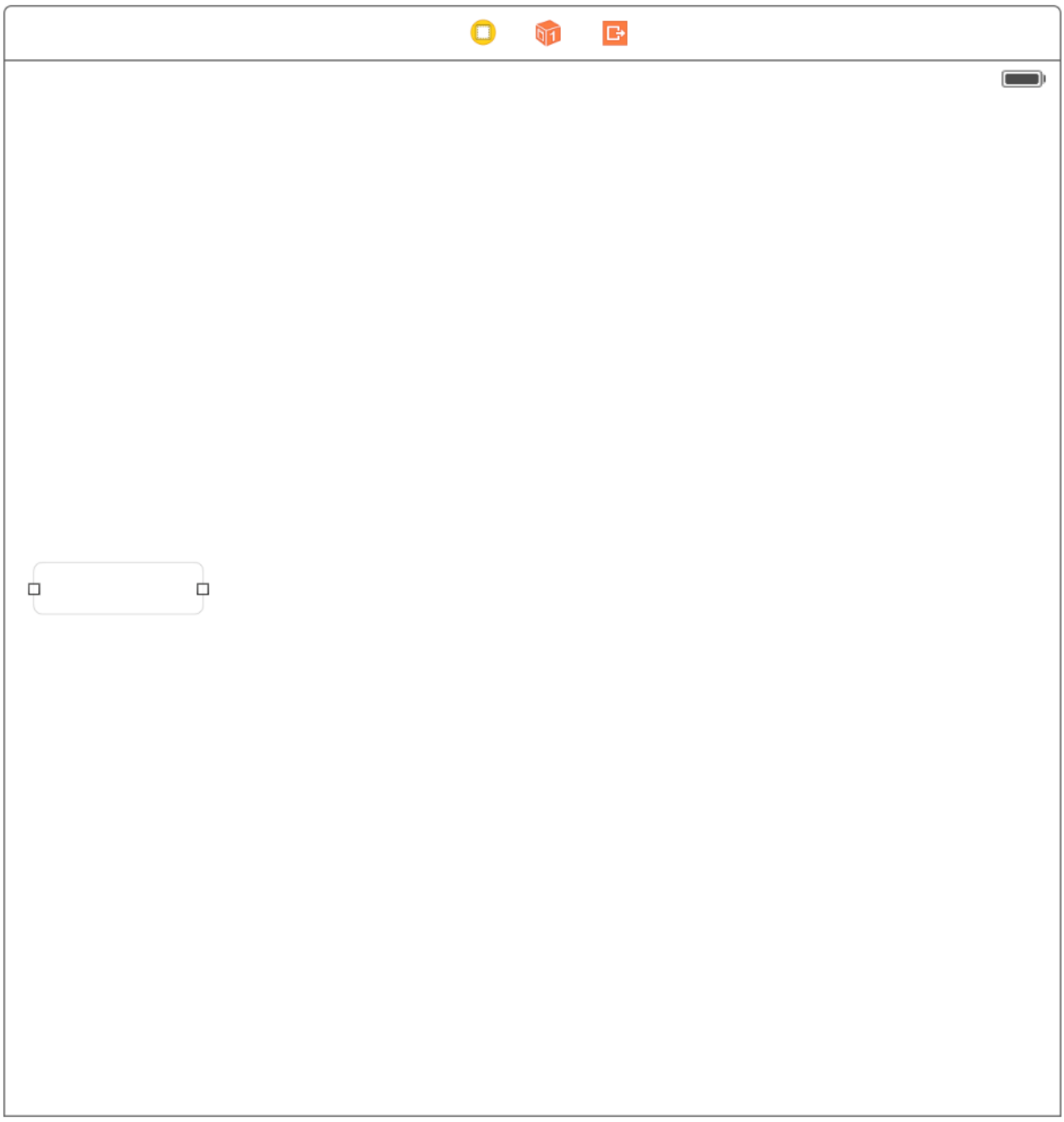
Stop dragging the text field when you see something like this:



The blue layout guides help you place the text field. Layout guides are visible only when you drag or resize objects next to them; they disappear when you let go of the text field.

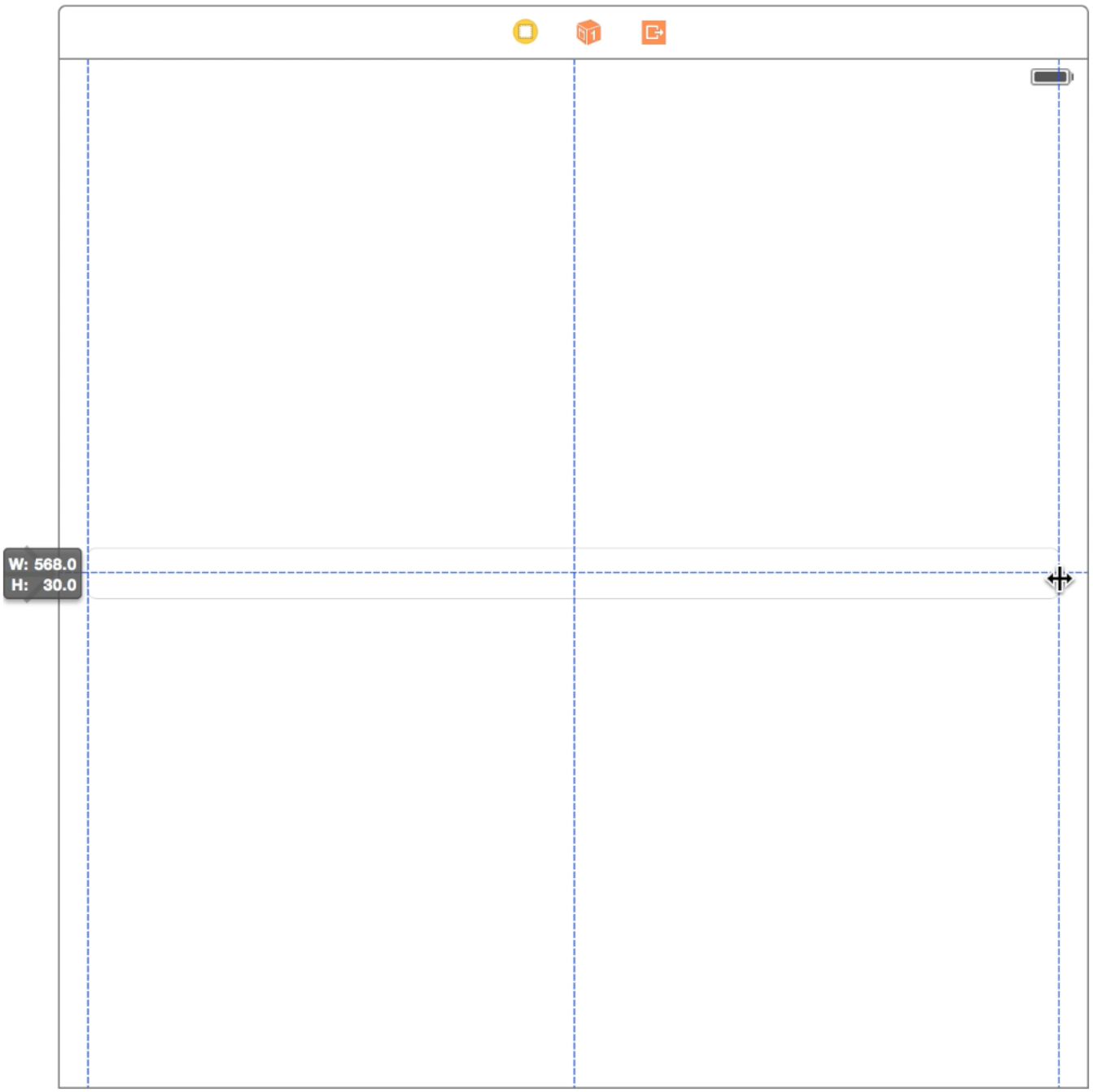
5. If necessary, click the text field to reveal the resize handles.

You resize a UI element by dragging its *resize handles*, which are small white squares that appear on the element's borders. You reveal an element's resize handles by selecting it. 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.



6. Resize the left and right edges of the text field until you see three vertical layout guides appear.

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

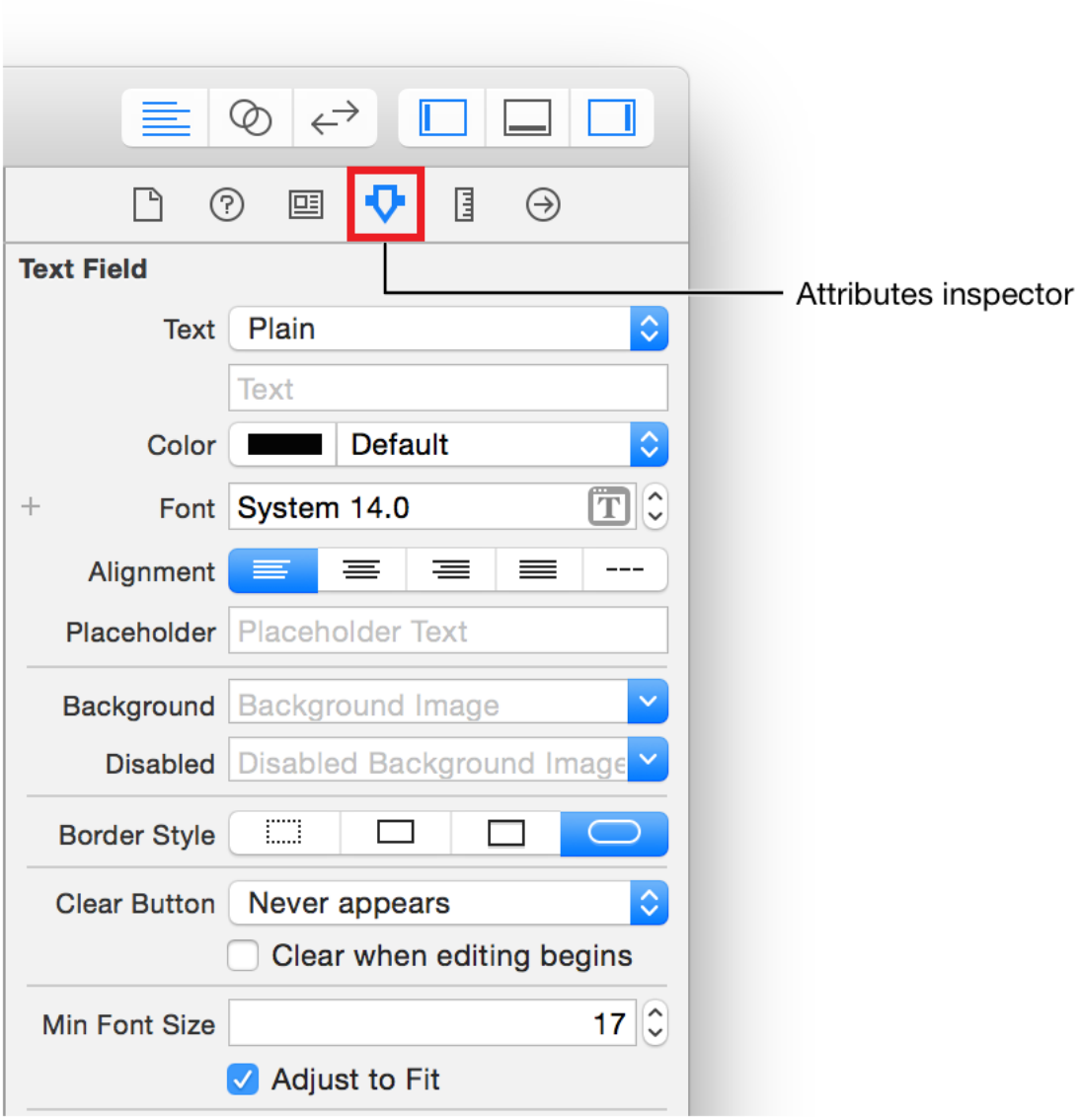


Although you have the text field in your scene, there's no instruction to the user about what to enter in the field. Use the text field's placeholder text to prompt the user to enter the name of a new to-do item.

To configure the text field's placeholder text

1. With the text field selected, open the Attributes inspector in the utility area.

The *Attributes inspector* appears when you select the fourth button from the left in the inspector selector bar. It lets you edit the properties of an object in your storyboard.



2. In the Attributes inspector, find the field labeled Placeholder and type `New to-do item`.
3. Press Return to display the new placeholder text in the text field.

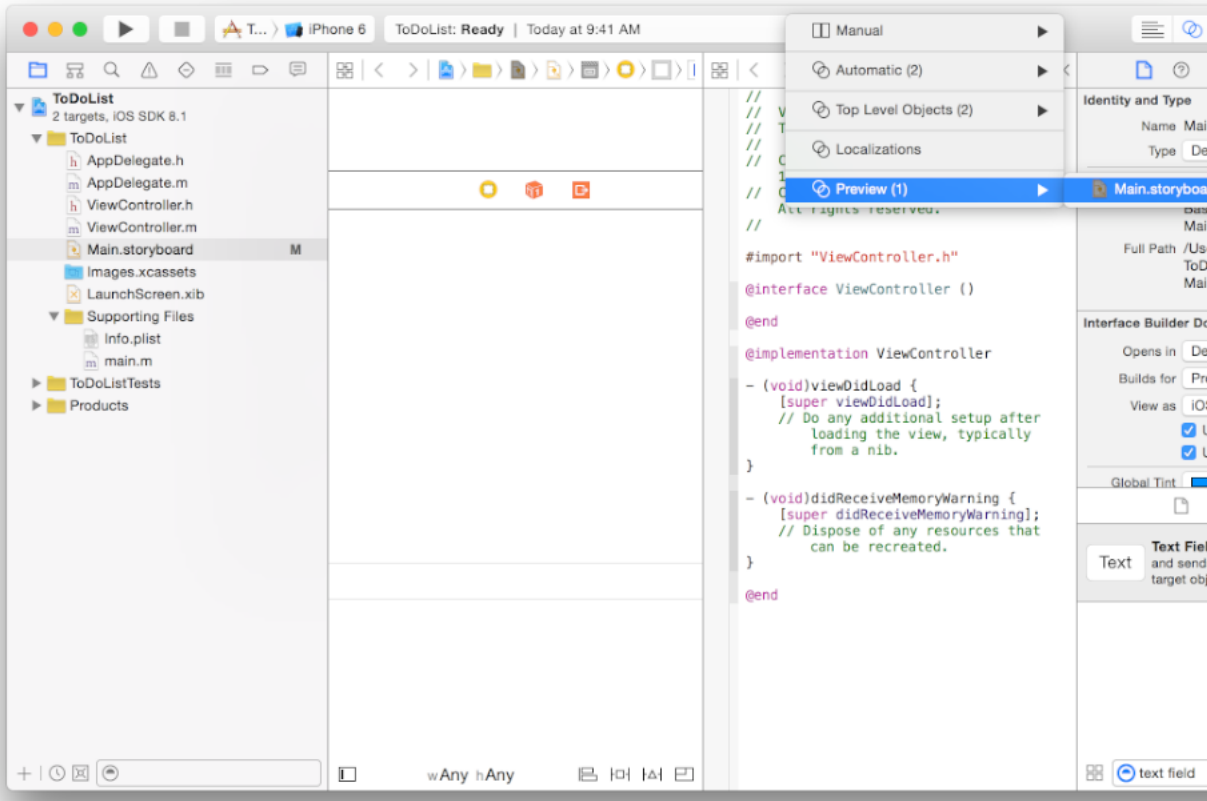
Preview Your Interface

Preview your app periodically to check that everything is looking the way you expect. You can preview your app interface using the *assistant editor*, which displays a secondary editor side-by-side with your main one.

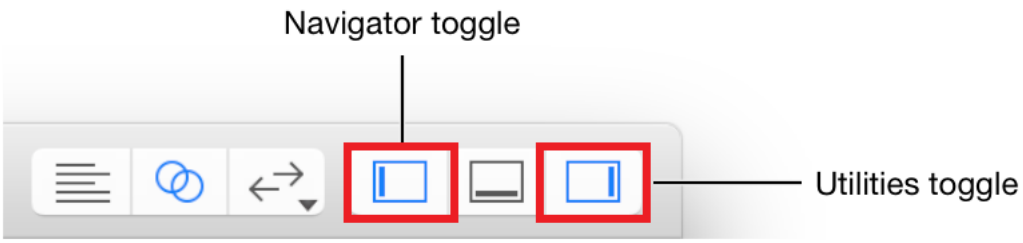
To preview your interface

1. Click the Assistant button in the Xcode toolbar to open the assistant editor.

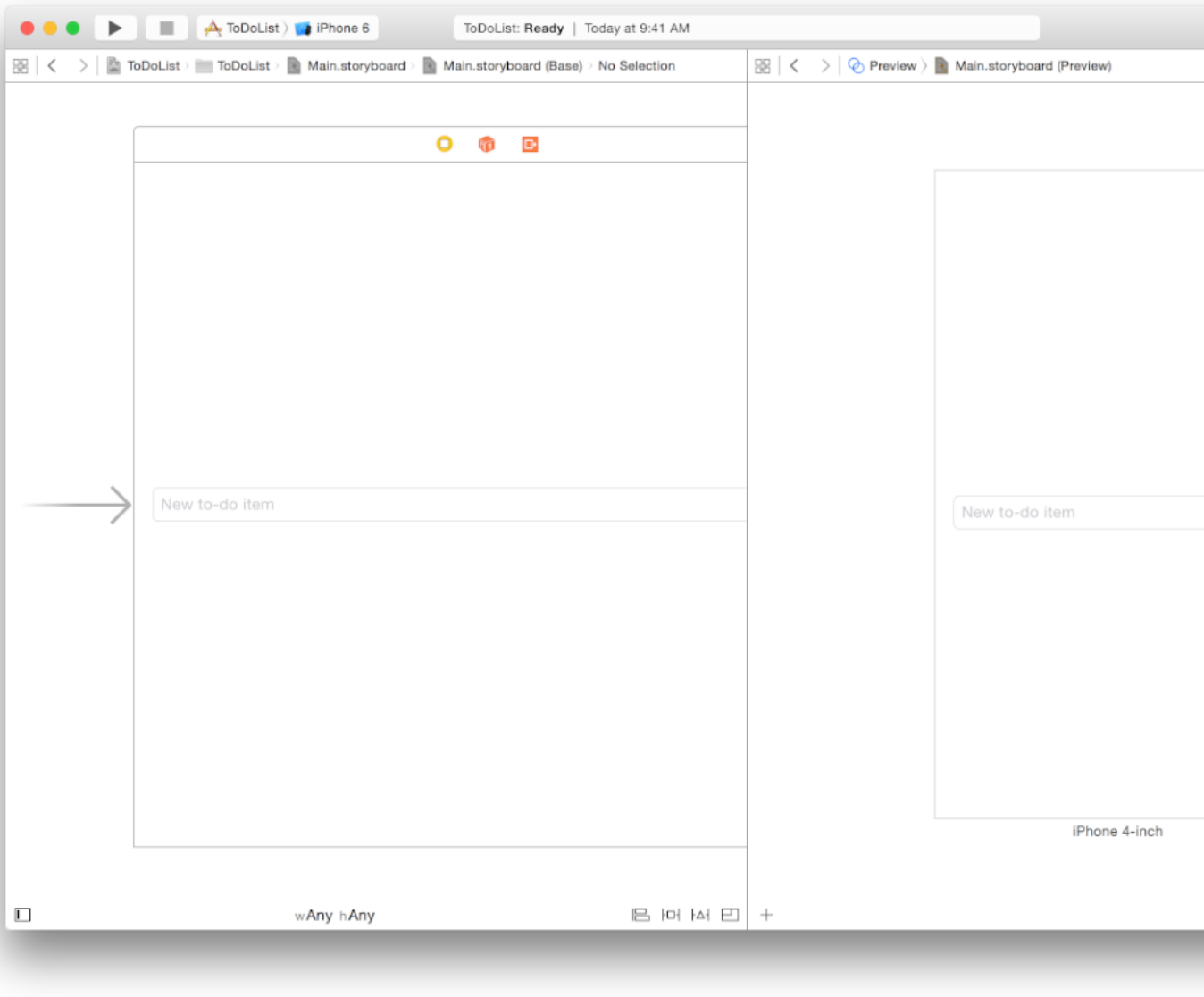
2. In the editor selector bar, switch the assistant editor from Automatic to Preview.



3. If you want more space to work, collapse the project navigator and utility area by clicking the Navigator and Utilities buttons in the Xcode toolbar.



Your Xcode window should look something like this:



As you see, the text field doesn't look quite right. It extends past the edge of the screen. The interface you specified in your storyboard looks correct, so why is this happening in the iPhone preview?

As you learned earlier, you're actually building an adaptive interface that scales for different sizes of iPhone and iPad. The scene you see by default in your storyboard shows a generalized version of your interface. Here, you'll need to specify how the interface should adjust for specific devices. For example, when the interface shrinks down to an iPhone size, the text field should shrink. When the interface grows to an iPad size, the text field should grow. You can specify these kinds of interface rules easily using Auto Layout.

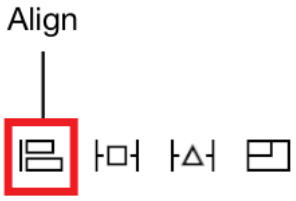
Adopt Auto Layout

Auto Layout is a powerful layout engine that helps you design adaptive layouts with ease. You describe your intent for the positioning of elements in a scene and then let the layout engine determine how best to implement that intent. You describe your intent using *constraints*—rules that explain where one element should be located relative to another, what size it should be, or which of two elements should shrink first when something reduces the space available for each of them.

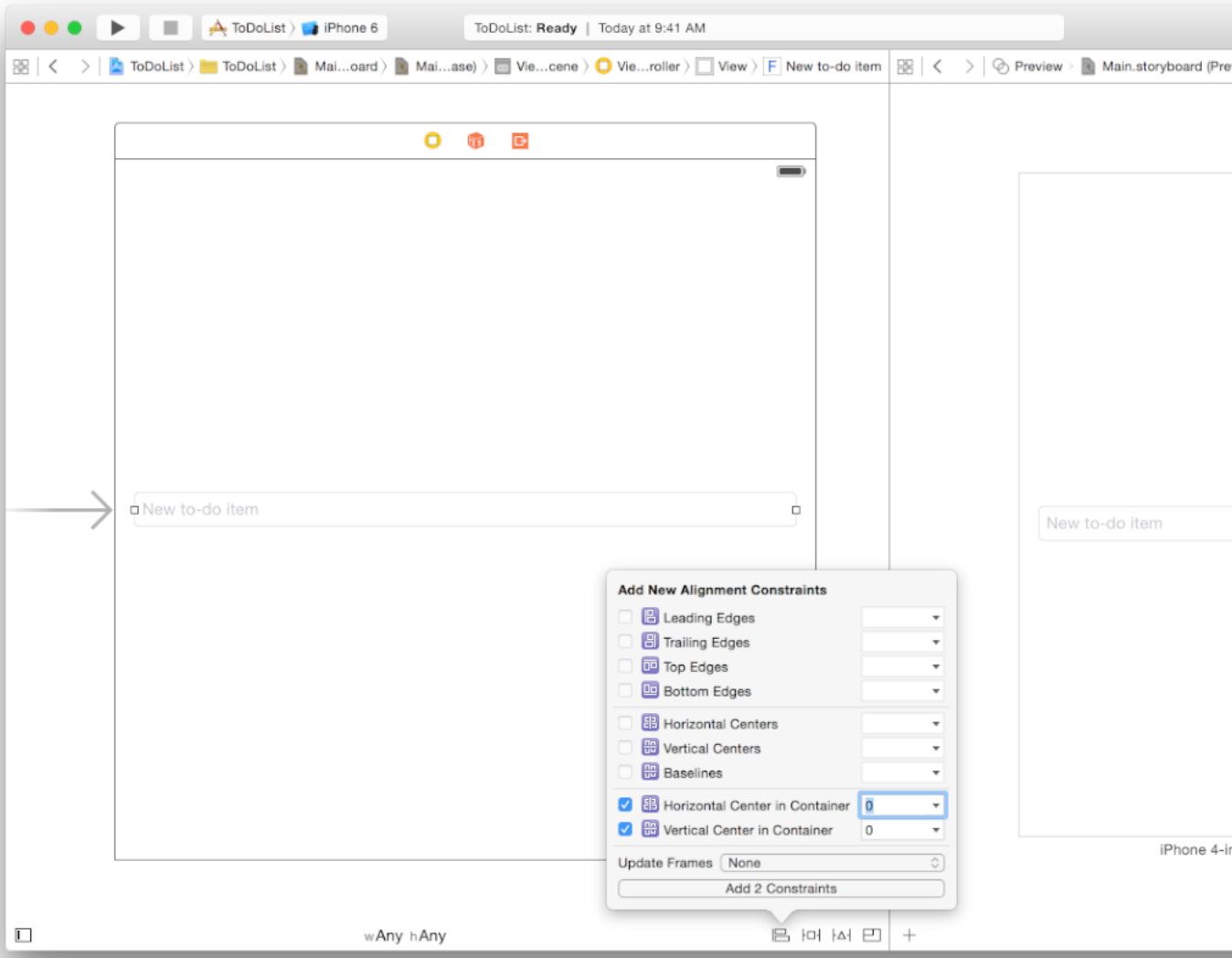
The text field in this interface should appear a fixed amount of space from the top of the device screen and stretch to the horizontal margins, regardless of device.

To position the text field using Auto Layout

1. In your storyboard, select the text field.
2. On the canvas, click the Auto Layout Align icon.

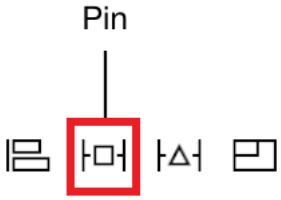


3. Select the checkboxes next to Horizontal Center in Container and Vertical Center in Container.

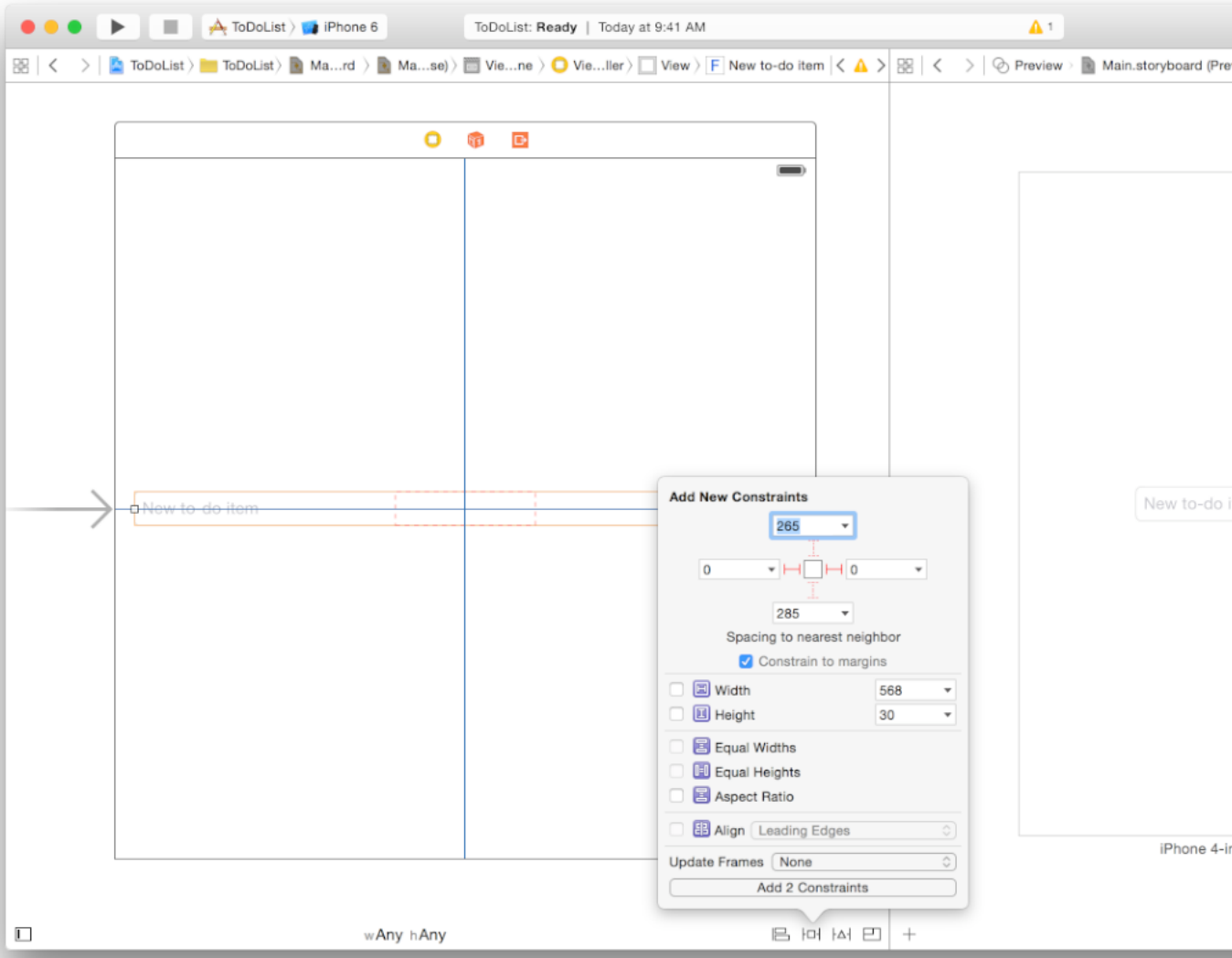


4. Click the Add 2 Constraints button.

5. On the canvas, click the Auto Layout Pin icon.



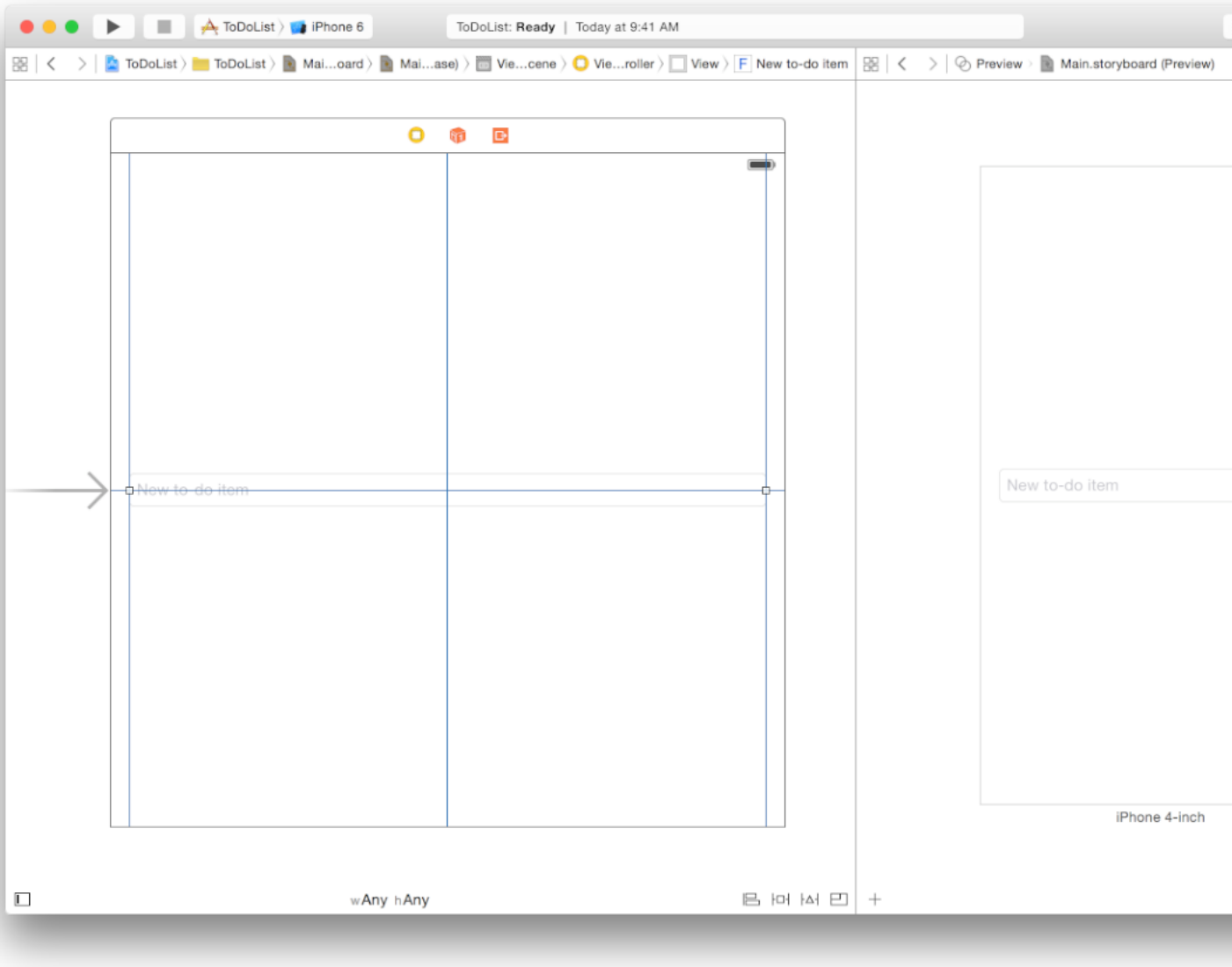
6. Above "Spacing to nearest neighbor," select the two horizontal red constraints by clicking on them.



These constraints indicate spacing to the nearest leading and trailing neighbors. Because the "Constrain to margins" checkbox is selected, the text field in this case is constrained to the margins of the view.

7. Click the Add 2 Constraints button.

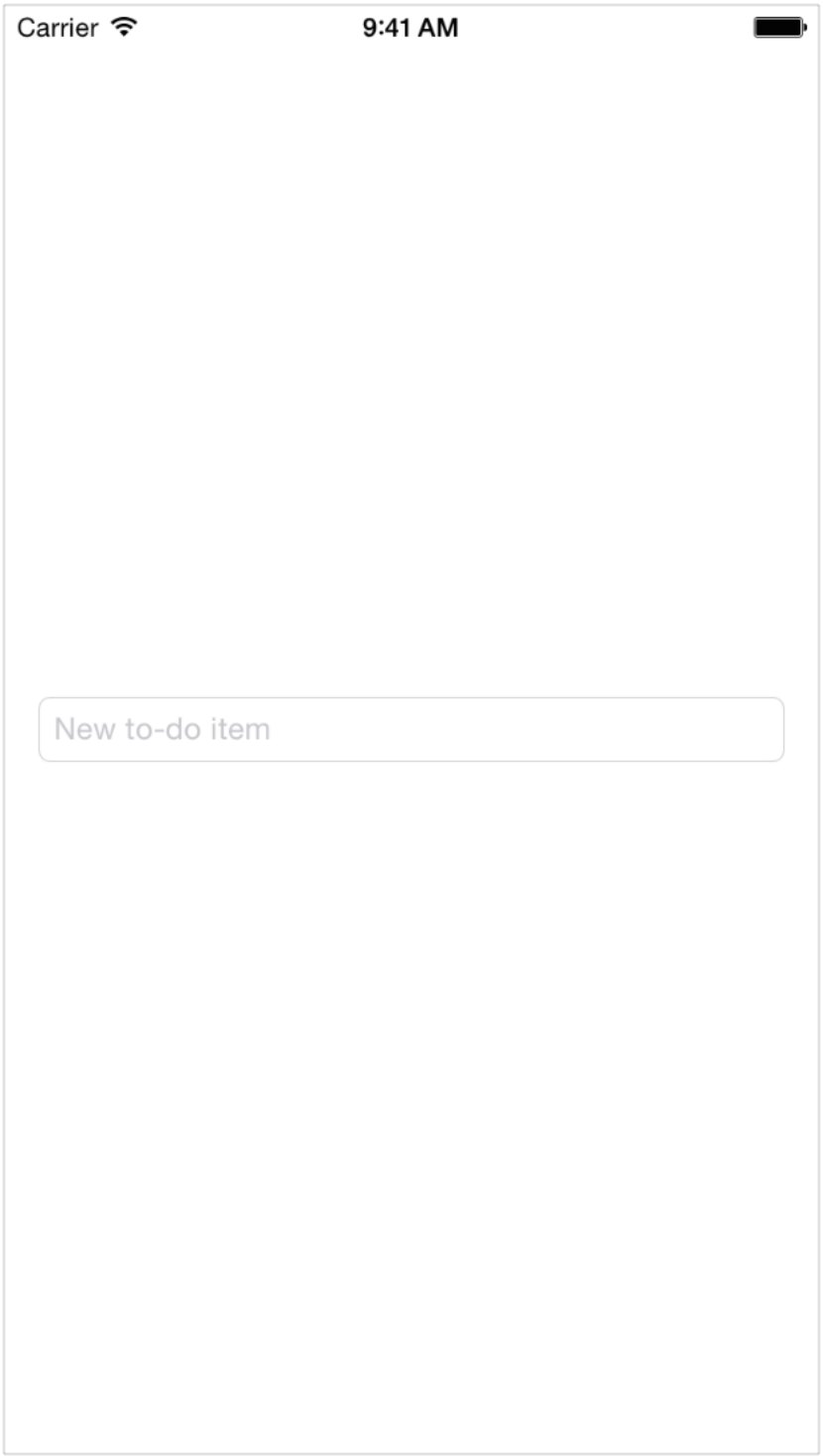
The app interface preview updates to reflect the new constraints:



If you don't get the behavior you expect, use the Xcode Auto Layout debugging features to help you. With the text field selected, click the Resolve Auto Layout Issues icon and choose Reset to Suggested Constraints to have Xcode update your interface with a valid set of constraints. Or click the Resolve Auto Layout Issues icon and choose Clear Constraints to remove all constraints on the text field, and then try following the steps above again.

Checkpoint: Run your app in iOS Simulator to make sure that the scene you created looks the way you expect it to. You should be able to click inside the text field and enter text using the keyboard (if you'd like, toggle the software keyboard by pressing Command-K). If you rotate the device (Command-Left Arrow or

Command-Right Arrow) or run the app on a different device, the text field grows or shrinks to the appropriate size depending on the device's orientation and screen size. Note that on some devices, the text field might be obscured by the keyboard in landscape orientation.



Although this scene doesn't do much yet, the basic user interface is there and functional. Considering layout from the start ensures that you have a solid foundation to build upon.

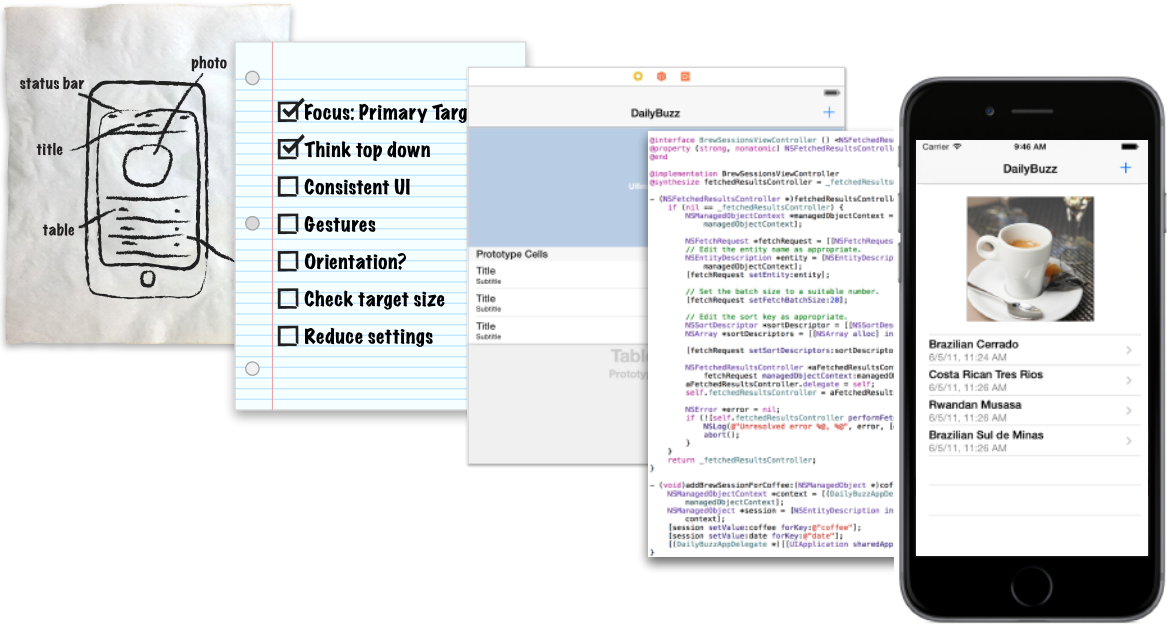
Recap

You're now well on your way to creating a basic interface with storyboards. In the remaining tutorials, you'll learn more about adding interaction to your interface and writing code to create custom behavior. The chapters between the tutorials guide you through the concepts that you'll put into practice while working on your app.

Structuring an App

Defining the Concept

Every great app starts with a concept. You don't need a completely polished and finished concept to start developing your app. You do need an idea of where you're going and what you need to do to get there.



Ask yourself these questions as your define your concept:

Who is your audience? Your app content and experience will differ depending on whether you're writing a children's game, a to-do list app, or simply a test app for your own learning.

What is the purpose of your app? An app needs a clearly defined purpose. Part of defining the purpose is understanding what one thing will motivate users to use your app.

What problem is your app trying to solve? A great app solves a single, well-defined problem instead of attempting solutions to multiple distinct problems. For example, the Settings app allows users to adjust all of the settings on their device. It provides a single interface for users to accomplish a related set of tasks. If your app is trying to solve unrelated problems, consider writing multiple apps.

What content will your app incorporate? Consider what type of content your app will present to users and how they'll interact with it. Design the user interface to complement the type of content that's presented in the app.

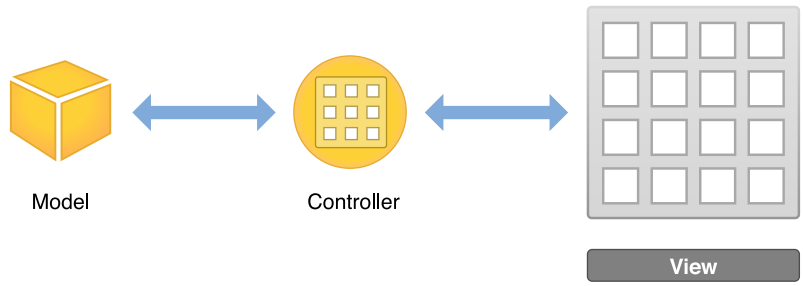
Designing a User Interface

A user needs to interact with an app interface in the simplest way possible. Design the interface with the user in mind, and make it efficient, clear, and straightforward.

Storyboards let you design and implement your interface in a graphical environment. You see exactly what you're building while you're building it, get immediate feedback about what's working and what's not, and make instantly visible changes to your interface.

When you build an interface in a storyboard, as you did in Tutorial: Basics (page 11), you're working with views. **Views** display content to the user. They are the building blocks for constructing your user interface and presenting your content in a clear, elegant, and useful way. As you develop more complex apps, you'll create interfaces with more scenes and more views.

.



The View Hierarchy

Views not only display themselves onscreen and react to user input, they can serve as containers for other views. As a result, views in an app are arranged in a hierarchical structure called the view hierarchy. The *view hierarchy* defines the layout of views relative to other views. Within that hierarchy, views enclosed within a view are called *subviews*, and the parent view that encloses a view is referred to as its *superview*. Even though a view can have multiple subviews, it can have only one superview.

At the top of the view hierarchy is the *window* object. Represented by an instance of the `UIWindow` class, a window object is the basic container into which you add your view objects for display onscreen. By itself, a window doesn't display any content. To display content, you add a *content view* object (with its hierarchy of subviews) to the window.

For a content view and its subviews to be visible to the user, the content view must be inserted into a window's view hierarchy. When you use a storyboard, this placement is configured automatically for you. When an app launches, the application object loads the storyboard, creates instances of the relevant view controller classes, unarchives the content view hierarchies for each view controller, and then adds the content view of the initial view controller into the window. You'll learn about managing view controllers in the next chapter; for now, you'll focus on creating a hierarchy within a single view controller in your storyboard.

Types of Views

When you design your app, it's important to know what kind of view to use for what purpose. For example, the kind of view you use to gather input text from a user, such as a text field, is different from what you might use to display static text, such as a label. Apps that use UIKit views for drawing are easy to create because you can assemble a basic interface quickly. A UIKit view object is an instance of the `UIView` class or one of its subclasses. The UIKit framework provides many types of views to help present and organize data.

Although each view has its own specific function, UIKit views can be grouped into these general categories.

View category	Purpose	Examples of views
 Content	Display a particular type of content, such as an image or text.	Image view, label
 Collections	Display collections or groups of views.	Collection view, table view
 Controls	Perform actions or display information.	Button, slider, switch
 Bars	Navigate, or perform actions.	Toolbar, navigation bar, tab bar
 Input	Receive user input text.	Search bar, text view

View category	Purpose	Examples of views
 Containers	Serve as containers for other views.	View, scroll view
Modal	Interrupt the regular flow of the app to allow a user to perform an action.	Action sheet, alert view

You can assemble views graphically using Interface Builder. **Interface Builder** provides a library of the standard views, controls, and other objects that you need to build your interface. After dragging these objects from the library, you drop them onto the canvas and arrange them in any way you want. Next, use inspectors to configure those objects before saving them in a storyboard. You see the results immediately, without the need to write code, build, and run your app.

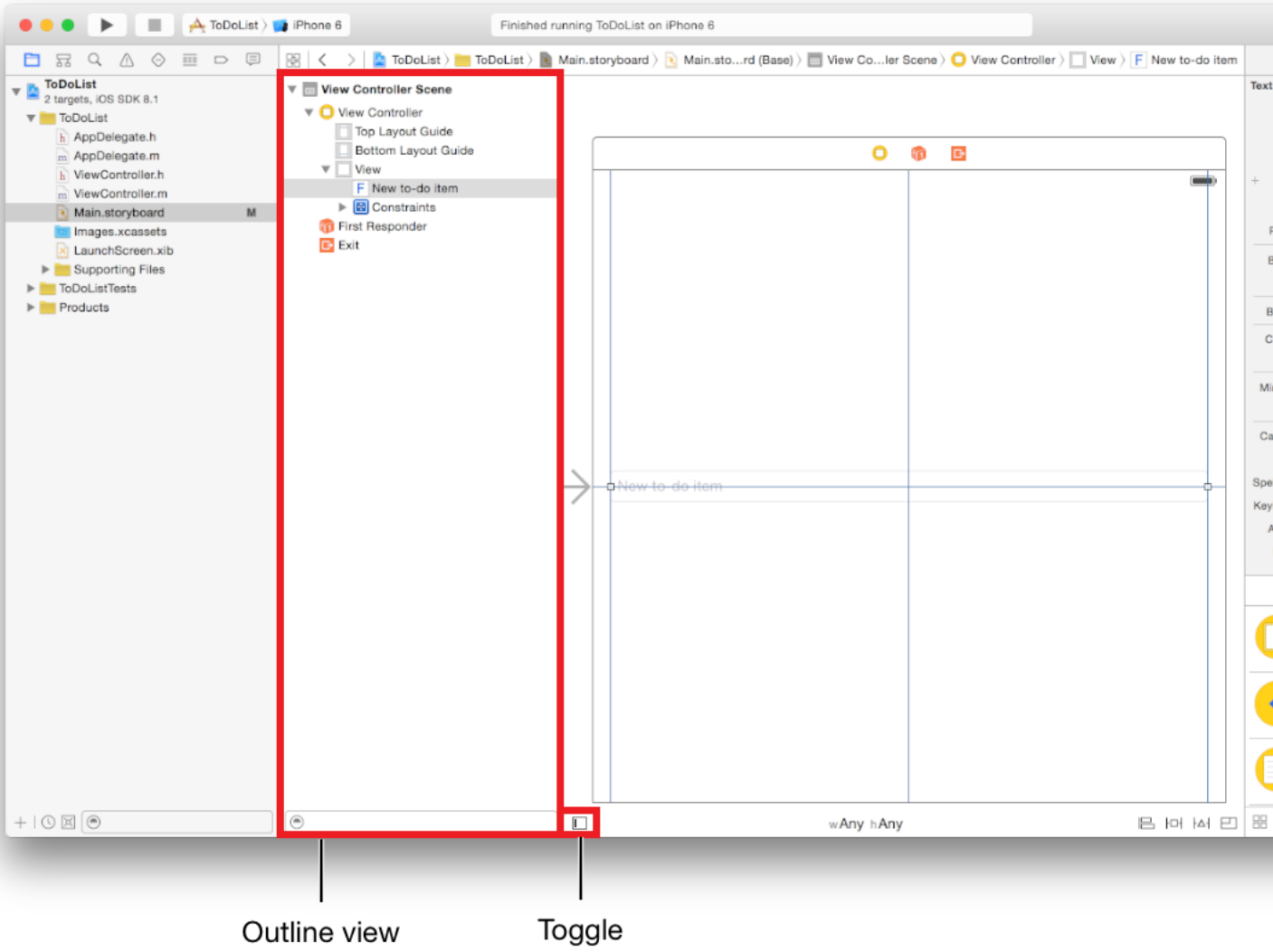
The UIKit framework provides standard views for presenting many types of content, but you can also define your own custom views by subclassing `UIView` (or its descendants). A custom view is a subclass of `UIView` in which you handle all of the drawing and event-handling tasks yourself. You won't be using custom views in these tutorials, but you can learn more about implementing a custom view in Defining a Custom View.

Use Storyboards to Lay Out Views

You use a storyboard to lay out your hierarchy of views in a graphical environment. Storyboards provide a direct, visual way to work with views and build your interface.

As you saw in Tutorial: Basics (page 11), storyboards are composed of scenes, and each scene has an associated view hierarchy. You drag a view out of the object library and place it in a storyboard scene to add it automatically to that scene's view hierarchy. The view's location within that hierarchy is determined by where you place it. After you add a view to your scene, you can resize, manipulate, configure, and move it on the canvas.

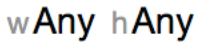
The canvas also shows an outline view of the objects in your interface. The *outline view*—which appears on the left side of the canvas—lets you see a hierarchical representation of the objects in your storyboard.



The view hierarchy that you create graphically in a storyboard scene is effectively a set of archived Objective-C objects. At runtime, these objects are unarchived. The result is a hierarchy of instances of the relevant classes configured with the properties you've set visually using the various inspectors in the utility area.

As you learned in Tutorial: Basics (page 11), the default interface configuration you work with in a storyboard applies to any version of your interface. When you need to adjust your interface for specific device sizes or orientations, you make the changes to specific size classes. A *size class* is a high-level way to describe the horizontal or vertical space that's available in a display environment, such as iPhone in portrait or iPad in landscape. There are two types of size classes: regular and compact. A display environment is characterized

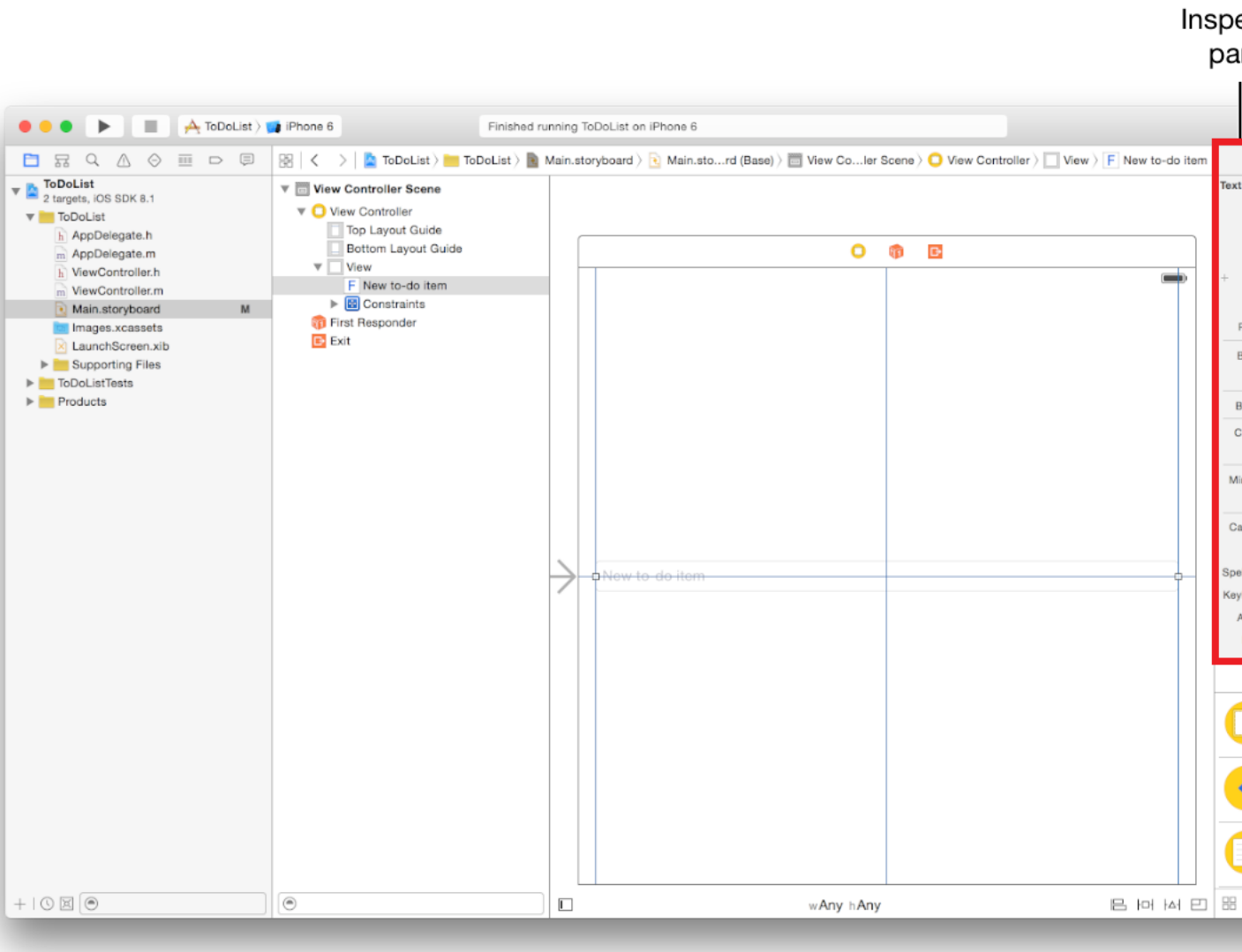
by a pair of size classes, one that describes the horizontal space and one that describes the vertical space. You can view and edit your interface for different combinations of regular and compact size classes using the size class control on the canvas:



You won't be working with specific size classes in this tutorial, but if you're curious about size classes, learn more about them in Size Classes Design Help.

Use Inspectors to Configure Views

The *inspector pane* appears in the utility area above the Object library.



Each inspector provides important configuration options for elements in your interface. When you select an object, such as a view, in your storyboard, you can use inspectors to customize different properties of that object.

- **File.** Specify general information about the storyboard.
- **Quick Help.** Get useful documentation about an object.
- **Identity.** Specify a custom class for your object and define its accessibility attributes.

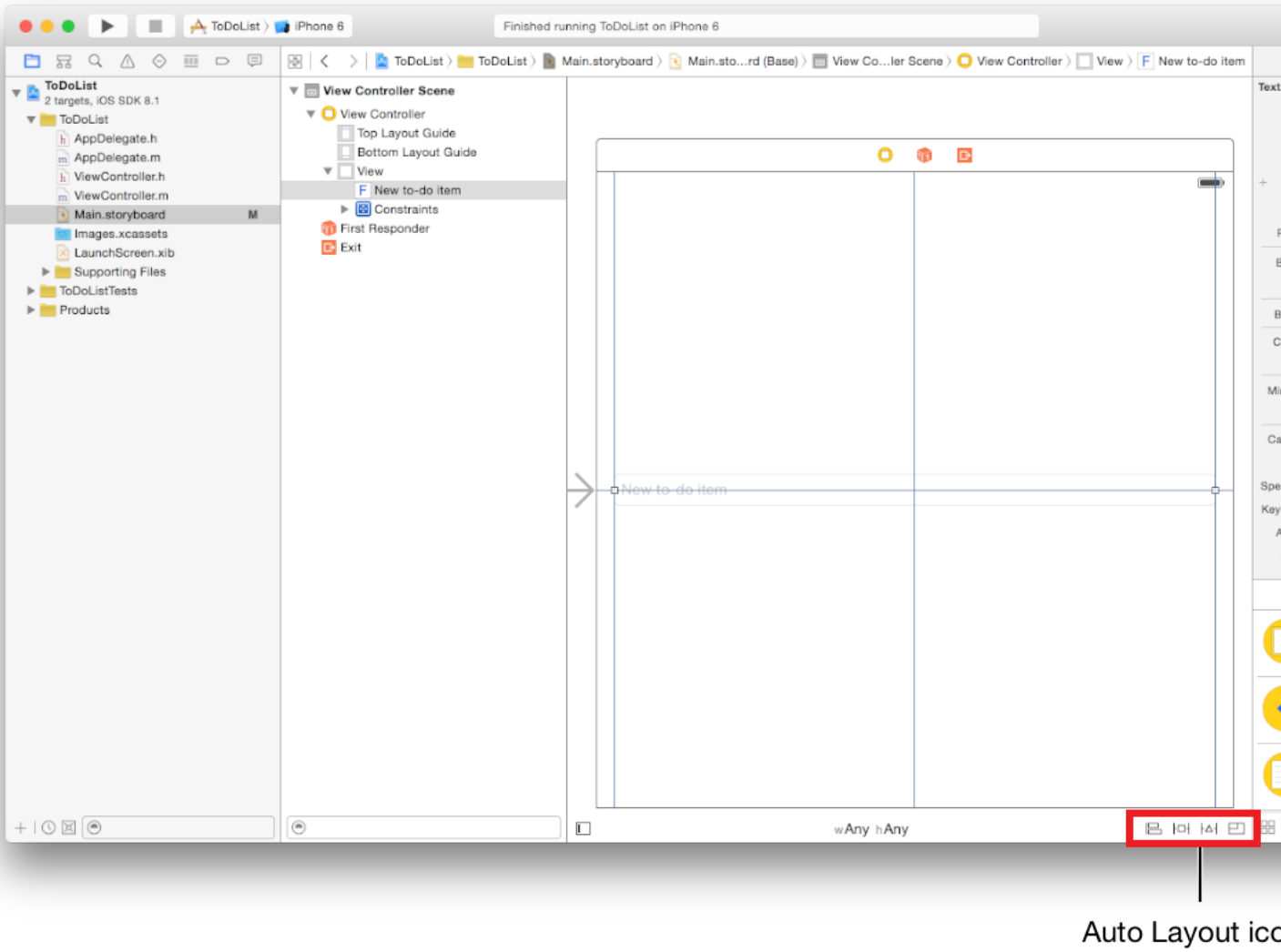
- **Attributes.** Customize visual attributes of an object.
- **Size.** Specify an object's size and Auto Layout attributes.
- **Connections.** Create connections between your interface and source code.

You began working with the Attributes inspector in the first tutorial. You'll continue using these inspectors throughout the rest of the tutorials to configure views and other objects in your storyboard. In particular, you'll use the Attributes inspector to configure your views, the Identity inspector to configure your view controllers, and the Connections inspector to create connections between your views and view controllers.

Use Auto Layout to Position Views

When you start positioning views in your storyboard, you need to consider a variety of situations. iOS apps run on a number of different devices, with various screen sizes, orientations, and languages. You need a dynamic interface that responds seamlessly to changes in screen size, device orientation, localization, and metrics.

As you already saw in Tutorial: Basics (page 11), Xcode offers a tool called Auto Layout to help create a versatile, adaptive interface. *Auto Layout* is a system for expressing relationships between views. It lets you define these relationships in terms of constraints on individual views or between sets of views.



Use the Auto Layout icons in the bottom-right area of your canvas to add various types of constraints to views on your canvas, resolve layout issues, and determine constraint resizing behavior.

- **Align.** Create alignment constraints, such as centering a view in its container, or aligning the left edges of two views.
- **Pin.** Create spacing constraints, such as defining the height of a view, or specifying its horizontal distance from another view.
- **Resolve Auto Layout Issues.** Resolve layout issues by adding or resetting constraints based on suggestions.

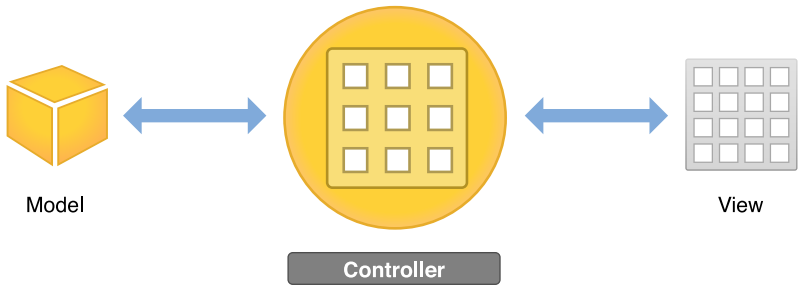
- **Resizing Behavior.** Specify how resizing affects constraints.

For the simple interface you're creating in the ToDoList app, you don't need to learn everything about Auto Layout at this point. As an app's interface layout gets more complex, you'll add a variety of constraints to specify exactly how the interface should lay out on different devices and orientations. At the end of this document, there's a link to the Auto Layout guide that helps you use constraints to define more intricate adaptive interfaces.

Defining the Interaction

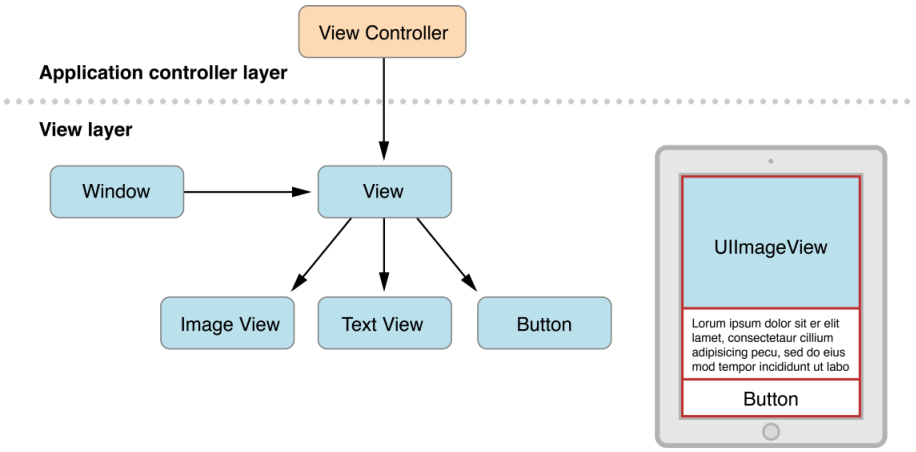
iOS apps are based on *event-driven programming*. That is, the flow of the app is determined by events: system events and user actions. The user performs actions on the interface, which trigger events in the app. These events result in the execution of the app's logic and manipulation of its data. The app's response to user action is then reflected back in the interface. Because the user, rather than the developer, is in control of when certain pieces of the app code get executed, you want to identify exactly which actions a user can perform and what happens in response to those actions.

You define much of your event-handling logic in view controllers. *Controllers* are objects that support your views by responding to user actions and populating the views with content. Controller objects are a conduit through which views interact with the data model. Views are notified of changes in model data through the app's controllers, and controllers communicate user-initiated changes—for example, text entered in a text field—to model objects. Whether they're responding to user actions or defining navigation, controllers implement your app's behavior.



View Controllers

After you've built a basic view hierarchy, your next step is to control the visual elements and respond to user input. In an iOS app, you use a *view controller* (`UIViewController`) to manage a content view with its hierarchy of subviews.



A view controller isn't part of the view hierarchy, and it's not an element in your interface. Instead, it manages the view objects in the hierarchy and provides them with behavior. Each content view hierarchy that you build in your storyboard needs a corresponding view controller, responsible for managing the interface elements and performing tasks in response to user interaction. This usually means writing a custom `UIViewController` subclass for each content view hierarchy. If your app has multiple content views, you use a different custom view controller class for each content view.

View controllers play many roles. They coordinate the flow of information between the app's data model and the views that display that data, manage the life cycle of their content views, and handle orientation changes when the device is rotated. But perhaps their most obvious role is to respond to user input.

You also use view controllers to implement transitions from one type of content to another. Because iOS apps have a limited amount of space in which to display content, view controllers provide the infrastructure needed to remove the views of one view controller and replace them with the views of another.

To define interaction in your app, you make your view controller files communicate with the views in your storyboard. You do this by defining connections between the storyboard and source code files through actions and outlets.

Actions

An *action* is a piece of code that's linked to an event that can occur in your app. When that event takes place, the code gets executed. You can define an action to accomplish anything from manipulating a piece of data to updating the user interface. You use actions to drive the flow of your app in response to user or system events.

You define an action by creating and implementing a method with an `IBAction` return type and a `sender` parameter.

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

The `sender` parameter points to the object that was responsible for triggering the action. The `IBAction` return type is a special keyword; it's like the `void` keyword, but it indicates that the method is an action that you can connect to from your storyboard in Interface Builder (which is why the keyword has the `IB` prefix). You'll learn more about how to link an `IBAction` action to an element in your storyboard in Tutorial: Storyboards (page 60).

Outlets

Outlets provide a way to reference interface objects—the objects you added to your storyboard—from source code files. To create an outlet, Control-drag from a particular object in your storyboard to a view controller file. This operation creates a property for the object in your view controller file, which lets you access and manipulate that object from code at runtime. For example, in the second tutorial, you'll create an outlet for the text field in your ToDoList app to be able to access the text field's contents in code.

Outlets are defined as `IBOutlet` properties.

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

The `IBOutlet` keyword tells Xcode that you can connect to this property from Interface Builder. You'll learn more about how to connect an outlet from a storyboard to source code in Tutorial: Storyboards (page 60).

Controls

A *control* is a user interface object such as a button, slider, or switch that users manipulate to interact with content, provide input, navigate within an app, and perform other actions that you define. Controls enable your code to receive messages from the user interface.

When a user interacts with a control, a control event is created. A *control event* represents various physical gestures that users can make on controls, such as lifting a finger from a control, dragging a finger onto a control, and touching down within a text field.

There are three general categories of control events:

- **Touch and drag events** occur when a user interacts with a control with a touch or drag. When a user initially touches a finger on a button, for example, the Touch Down Inside event is triggered; if the user drags out of the button, the respective drag events are triggered. Touch Up Inside is sent when the user lifts a finger off the button while still within the bounds of the button's edges. If the user has dragged a finger outside the button before lifting the finger, effectively canceling the touch, the Touch Up Outside event is triggered.
- **Editing events** occur when a user edits a text field.
- **Value-changed events** occur when a user manipulates a control, causing it to emit a series of different values.

As you define the interactions, know the action that's associated with every control in your app and then make that control's purpose obvious to users in your interface.

Navigation Controllers

If your app has more than one content view hierarchy, you need to be able to transition between them. For this, you'll use a specialized type of view controller: a navigation controller (`UINavigationController`). A *navigation controller* manages transitions backward and forward through a series of view controllers, such as when a user navigates through email accounts, inbox messages, and individual emails in the iOS Mail app.

The set of view controllers managed by a particular navigation controller is called its navigation stack. The *navigation stack* is a last-in, first-out collection of custom view controller objects. The first item added to the stack becomes the *root view controller* and is never popped off the stack. Other view controllers can be pushed on or popped off the navigation stack.

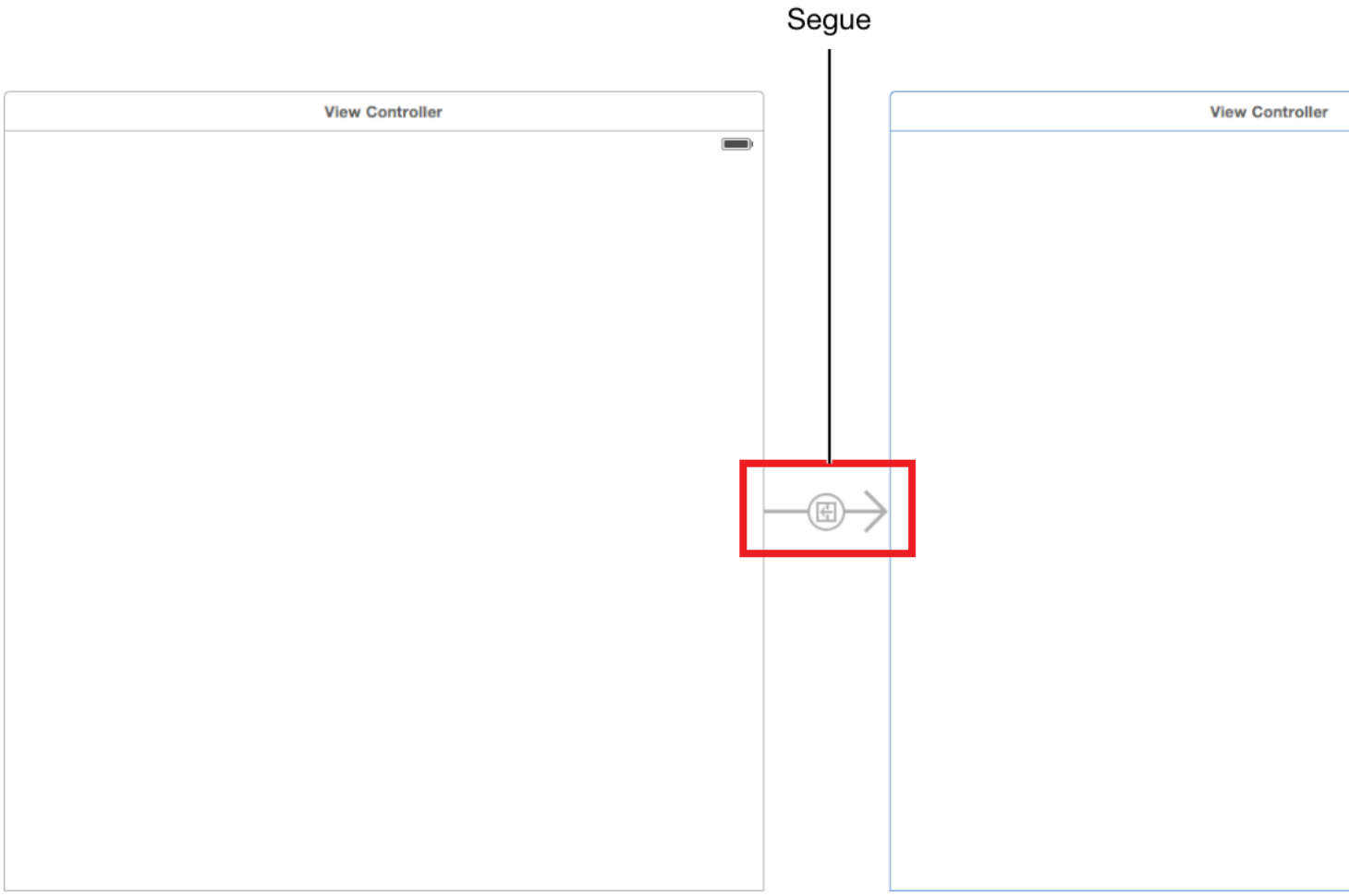
Although a navigation controller's primary job is to manage the presentation of your content view controllers, it's also responsible for presenting custom views of its own. Specifically, it presents a navigation bar—the view at the top of the screen that provides context about the user's place in the navigation hierarchy—which contains a back button and other buttons you can customize. Every view controller that's added to the navigation stack presents this navigation bar. You are responsible for configuring the navigation bar.

You generally don't have to do any work to pop a view controller off of the navigation stack; the back button provided by the navigation controller handles this for you. However, you do have to manually push a view controller onto the stack. You can do this using storyboards.

Use Storyboards to Define Navigation

So far, you've learned about using storyboards to create a single screen of content in your app. Now, you'll learn about using them to define the flow between multiple scenes in your app.

In the first tutorial, the storyboard you worked with had one scene. In most apps, a storyboard is composed of a sequence of scenes, each of which represents a view controller and its view hierarchy. Scenes are connected by **segues**, which represent a transition between two view controllers: the source and the destination.



There are several types of segues you can create in a storyboard:

- **Show.** A show segue pushes new content on top of the current view controller stack. Where the content shows up depends on the layout of view controllers in the scene.
- **Show detail.** A show detail segue either pushes new content on top of the current view controller stack or replaces the content that's shown, depending on the layout of view controllers in the scene.

- **Present modally.** A modal segue is one view controller presenting another controller modally, requiring a user to perform an operation on the presented controller before returning to the main flow of the app. A modal view controller isn't added to a navigation stack; instead, it's generally considered to be a child of the presenting view controller. The presenting view controller is responsible for dismissing the modal view controller it created and presented.
- **Popover presentation.** The presented view controller is shown as a popover anchored to an existing view.
- **Custom.** You can define your own custom transition by subclassing `UIStoryboardSegue`.
- **Unwind.** An unwind segue moves backward through one or more segues to return the user to an existing instance of a view controller. You use unwind segues to implement reverse navigation.

In addition to segues, scenes may be connected by a *relationship*. For example, there's a relationship between the navigation controller and its root view controller. In this case, the relationship represents the containment of the root view controller by the navigation controller.

Now that you've learned the basics of working with views and view controllers in storyboards, it's time to incorporate this knowledge into your ToDoList app in the next tutorial.

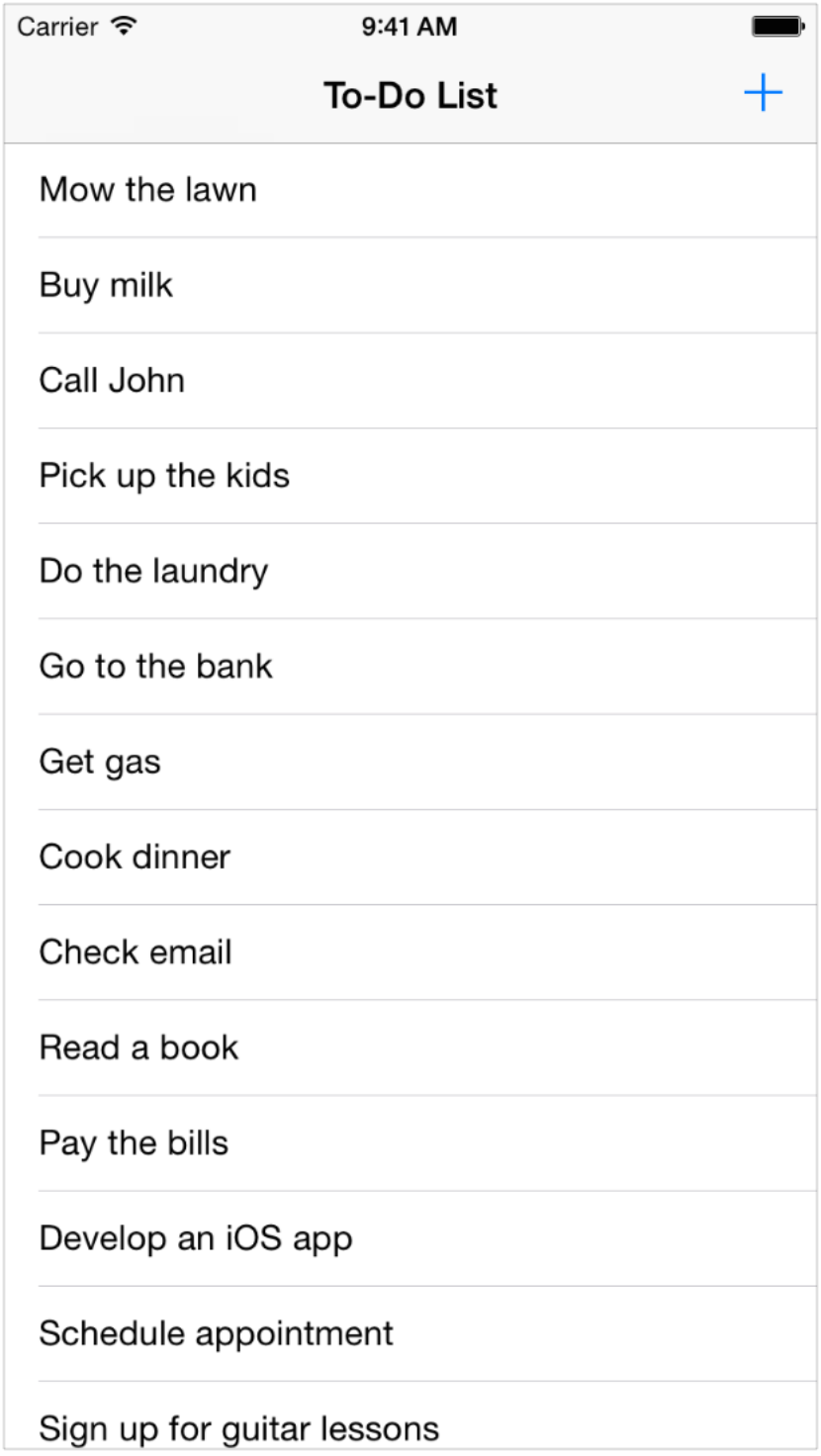
Tutorial: Storyboards

This tutorial builds on the project you created in the first tutorial (Tutorial: Basics (page 11)). You'll put to use what you learned about views, view controllers, actions, and navigation. You'll also create some of the key user interface flows for your ToDoList app and add behavior to the scene you've already created.

This tutorial teaches you how to:

- Use storyboards to define app content and flow
- Manage multiple view controllers
- Add actions to elements in your user interface

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



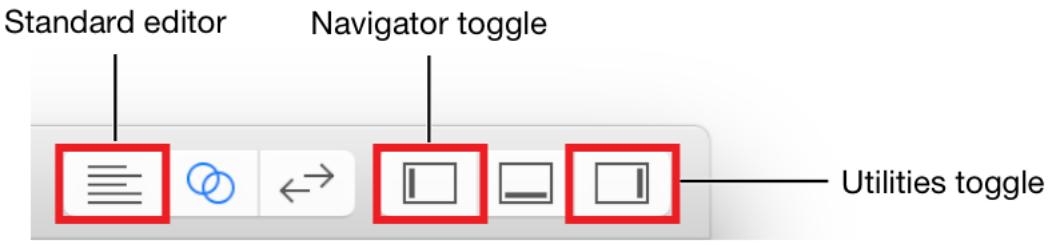
Create a Second Scene

So far, you've worked with a single scene managed by a view controller that represents a page where you can add an item to your to-do list: the add-to-do-item scene. Now it's time to create the scene that shows the entire to-do list. Fortunately, iOS comes with a powerful built-in class called a *table view* (`UITableView`) designed specifically to display a scrolling list of items.

To add a scene with a table view to your storyboard

1. In the project navigator, select `Main.storyboard`.

   If the project navigator is still collapsed, toggle it by clicking the Navigator button in the Xcode toolbar.



   If the assistant editor is still open, return to the standard editor by clicking the Standard button.

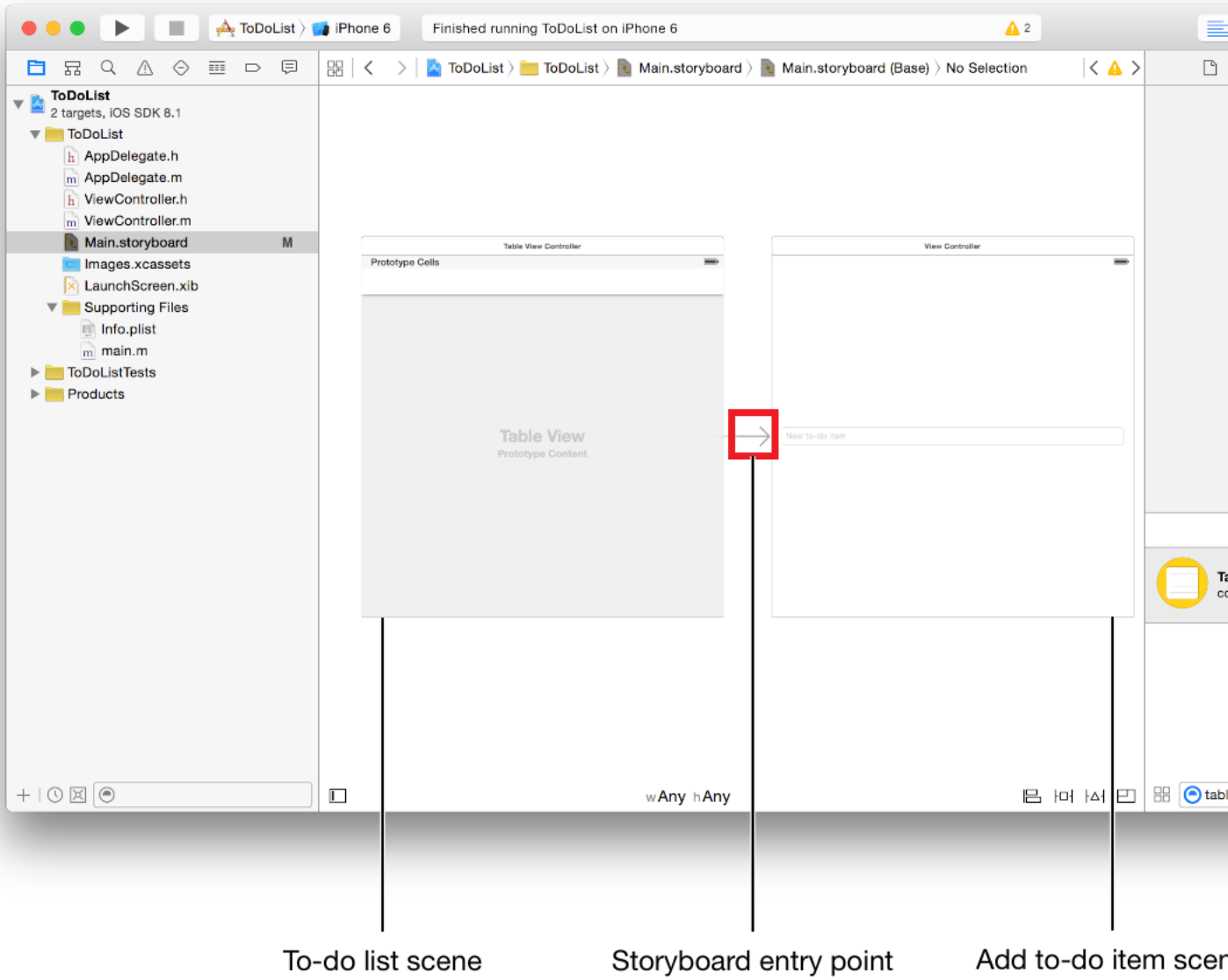
2. Open the Object library in the utility area. (To open the library with a menu command, choose View > Utilities > Show Object Library.)

3. In the Object library, type `table view controller` in the filter field to find a Table View Controller object quickly.

4. Drag a Table View Controller object from the list and drop it on the canvas to the left of the add-to-do-item scene. If you need to, you can zoom out using Editor > Canvas > Zoom.

   If you see a table view with content and nothing happens when you try to drag it to the canvas, you're probably dragging a table view rather than a table view controller. A table view is one of the things managed by a table view controller, but you want the whole package, so find the table view controller and drag it to the canvas.

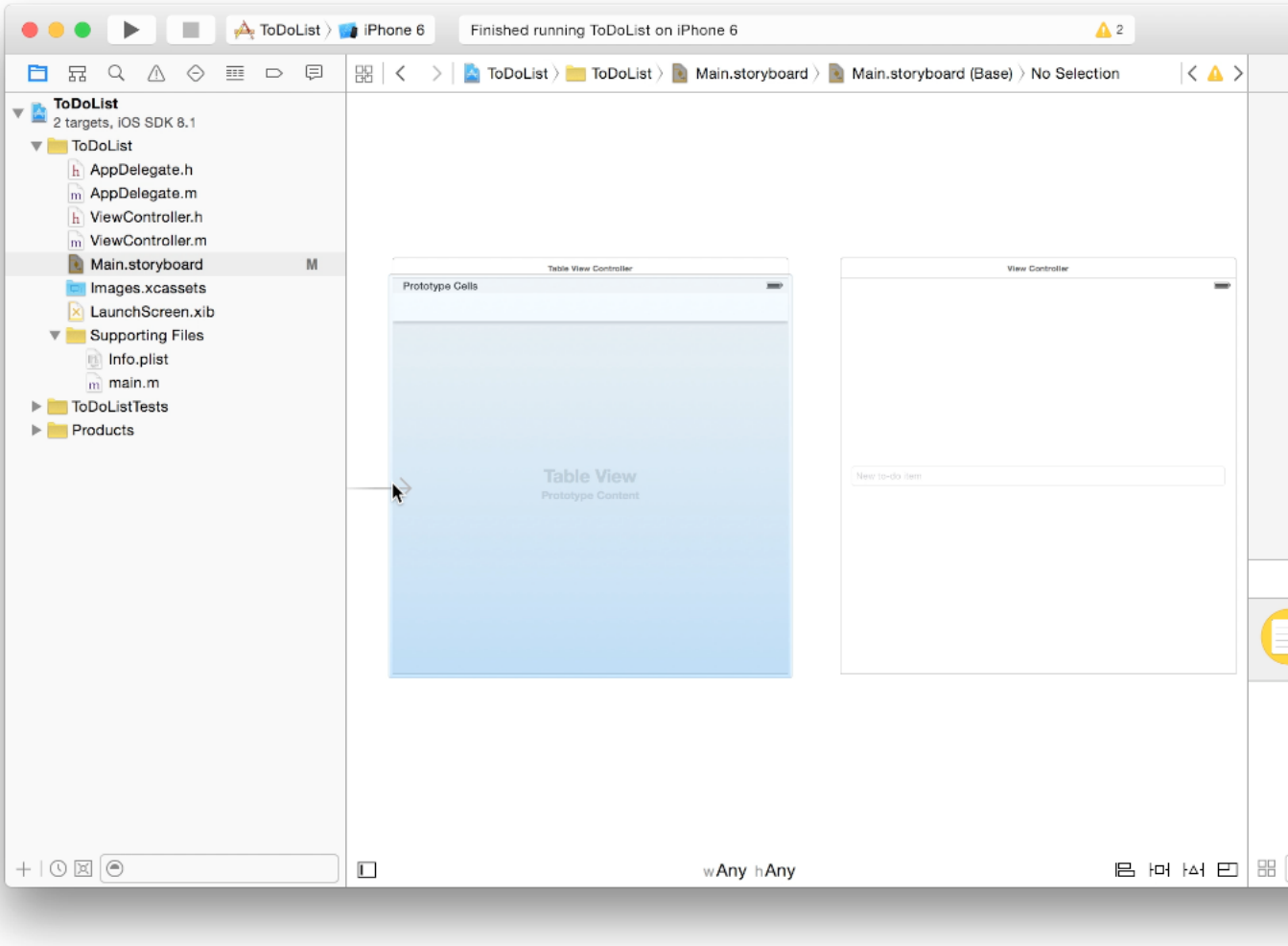
You now have two scenes, one for displaying the to-do list and one for adding to-do items.



It makes sense to have the to-do list be the first thing users see when they launch your app, so tell Xcode that's your intent by setting the table view controller as the first scene.

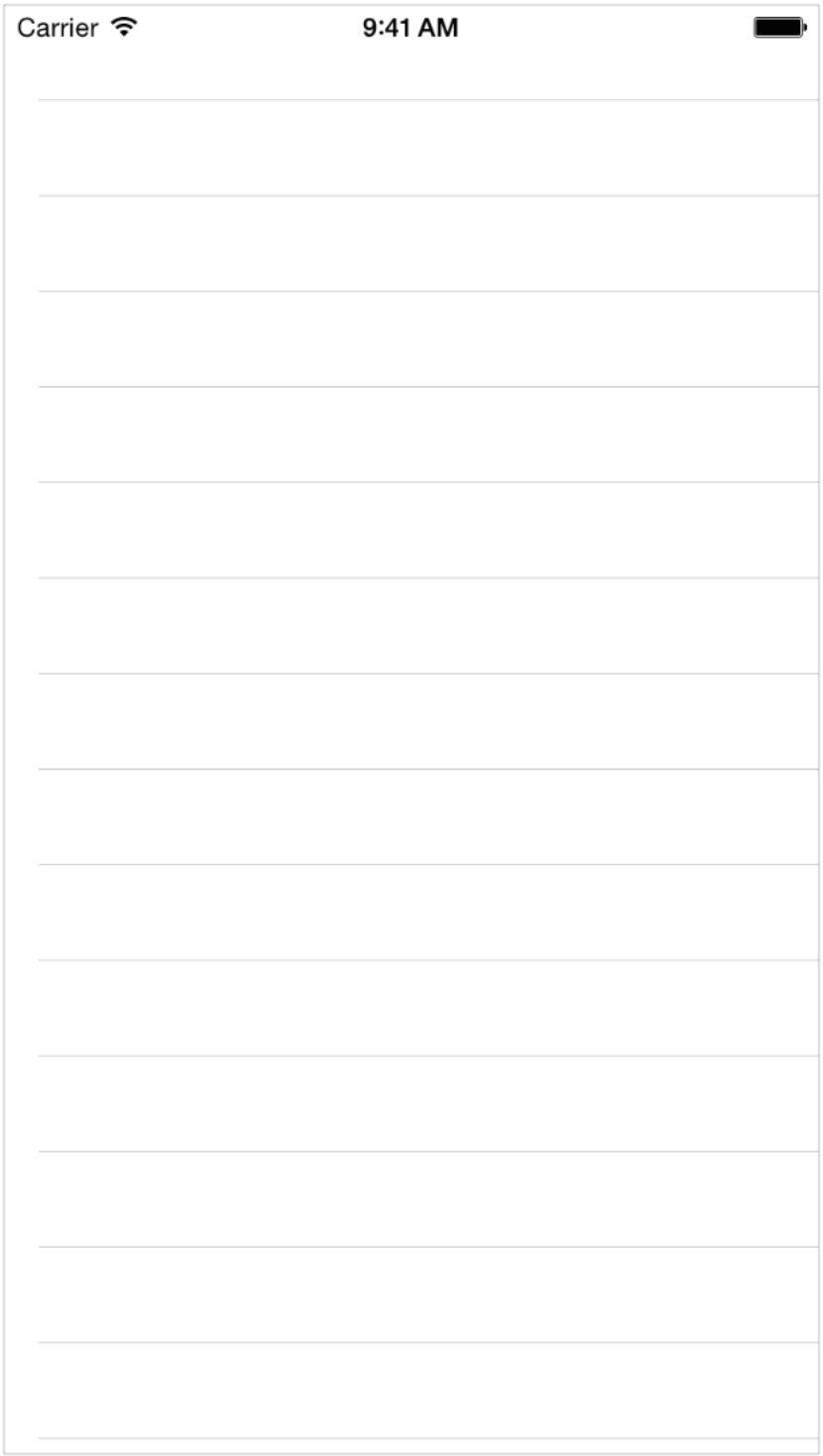
To set the table view controller as the initial scene

- Drag the storyboard entry point from the add-to-do-item scene to the table view controller.



The table view controller is set as the initial view controller in your storyboard, making it the first scene that loads on app launch.

Checkpoint: Run your app. Instead of the add-to-do-item scene with its text field, you should now see an empty table view—a screen with a number of horizontal dividers to separate it into rows, but with no content in each row.



Display Static Content in a Table View

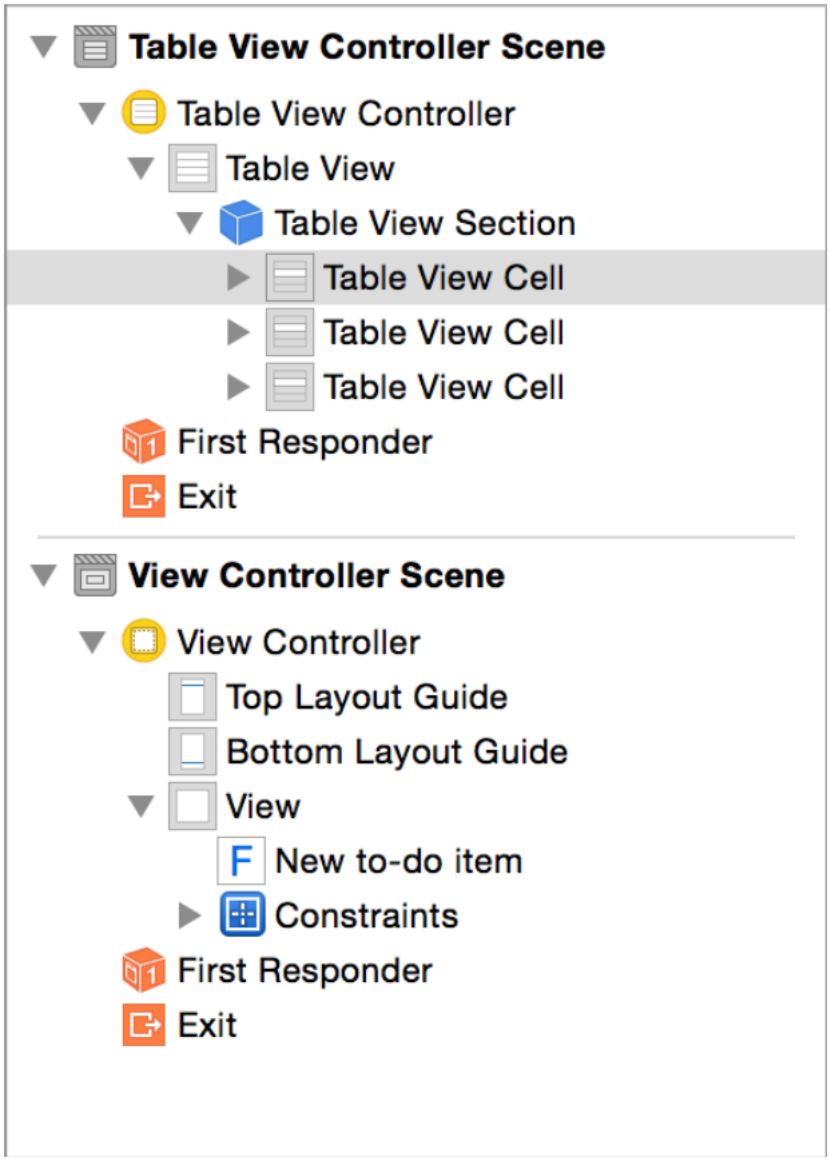
Because you haven't learned about storing data yet, it's too early to create and store to-do items and display them in the table view. But you don't need real data to prototype your user interface. Xcode allows you to create static content in a table view in Interface Builder. This makes it a lot easier to see how your user interface will behave, and it's a valuable way to try out different ideas.

To create a static cell in your table view

1. If necessary, open the outline view  .
2. In the outline view, select Table View under Table View Controller.
3. With the table view selected, open the Attributes inspector  in the utility area.
4. In the Attributes inspector, choose Static Cells from the pop-up menu next to the Content option.

   Three empty table view cells appear in your table view.

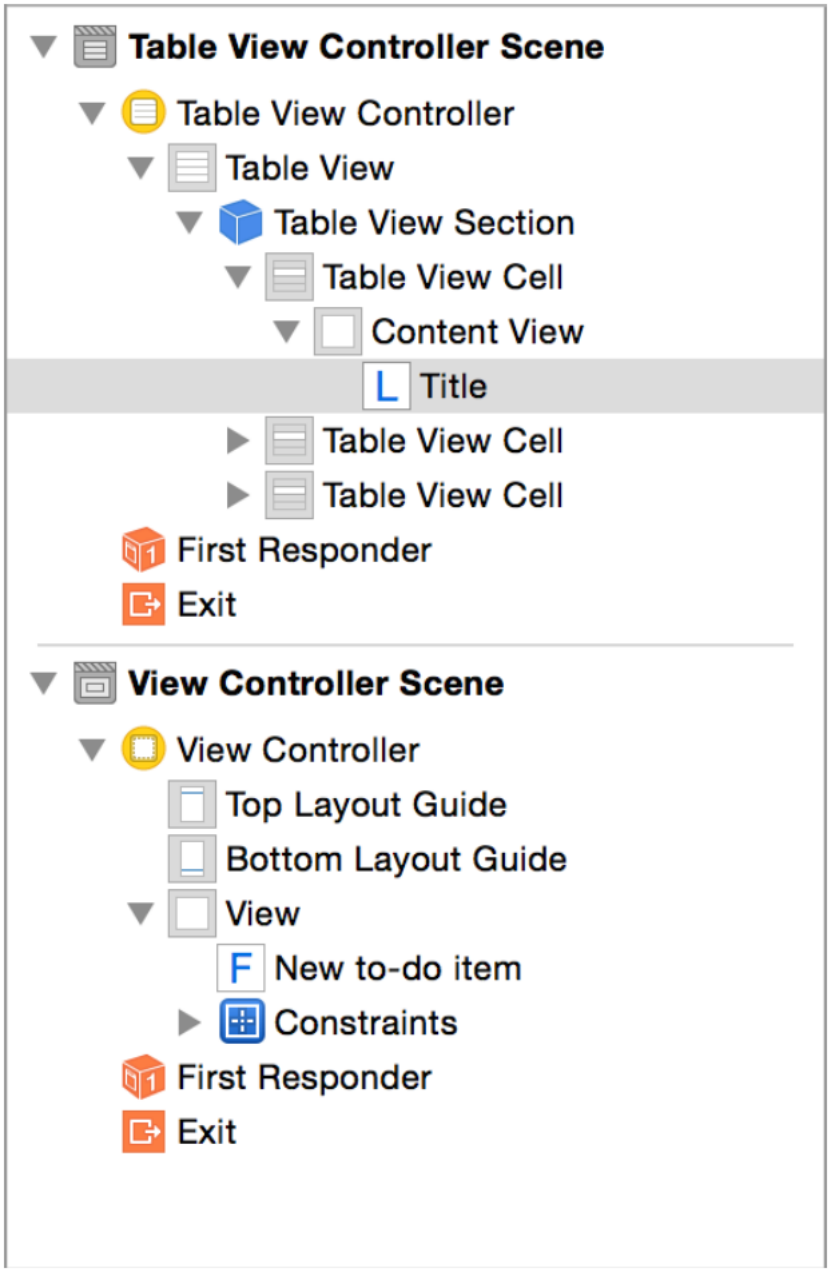
5. In the outline view, select the top cell.



6. In the Attributes inspector, choose Basic from the pop-up menu next to the Style option.

   The Basic style includes a label, so Xcode creates a label with the text “Title” in the table cell.

7. In the outline view, select the label.



8. In the Attributes inspector, change the text of the label from "Title" to "Mow the lawn." For the change to take effect, press Return or click outside the utility area.

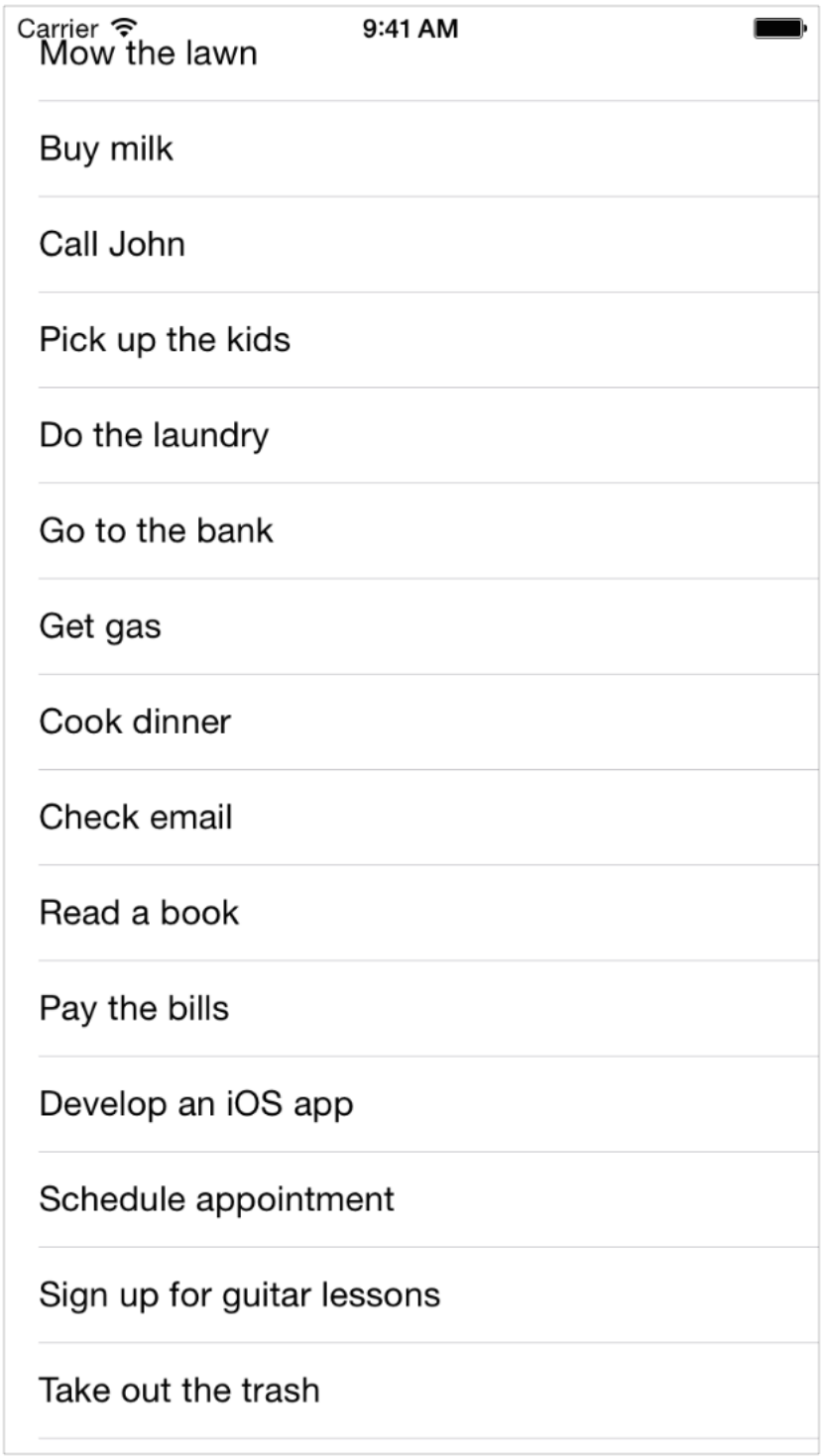
   Alternatively, you can edit a label by double-clicking it and editing the text directly.

9. Repeat steps 4–7 for the other cells, giving them text for other likely to-do items.

10. Create enough cells so that the items more than fill the screen. You can create new cells by copying and pasting them or by holding down the Option key when dragging a cell.

Checkpoint: Run your app. You should now see a table view with the preconfigured cells you added in Interface Builder. See how the new table view feels when you scroll it. Try rotating the simulated device—notice how the table view is already configured to lay out its contents properly. You get a lot of behavior for free by using a table view.

(You might notice that the top cell is overlapped by the device's status bar—the area that shows the time, battery level, and other information. Don't worry, you'll fix that next.)



Add a Segue to Navigate Forward

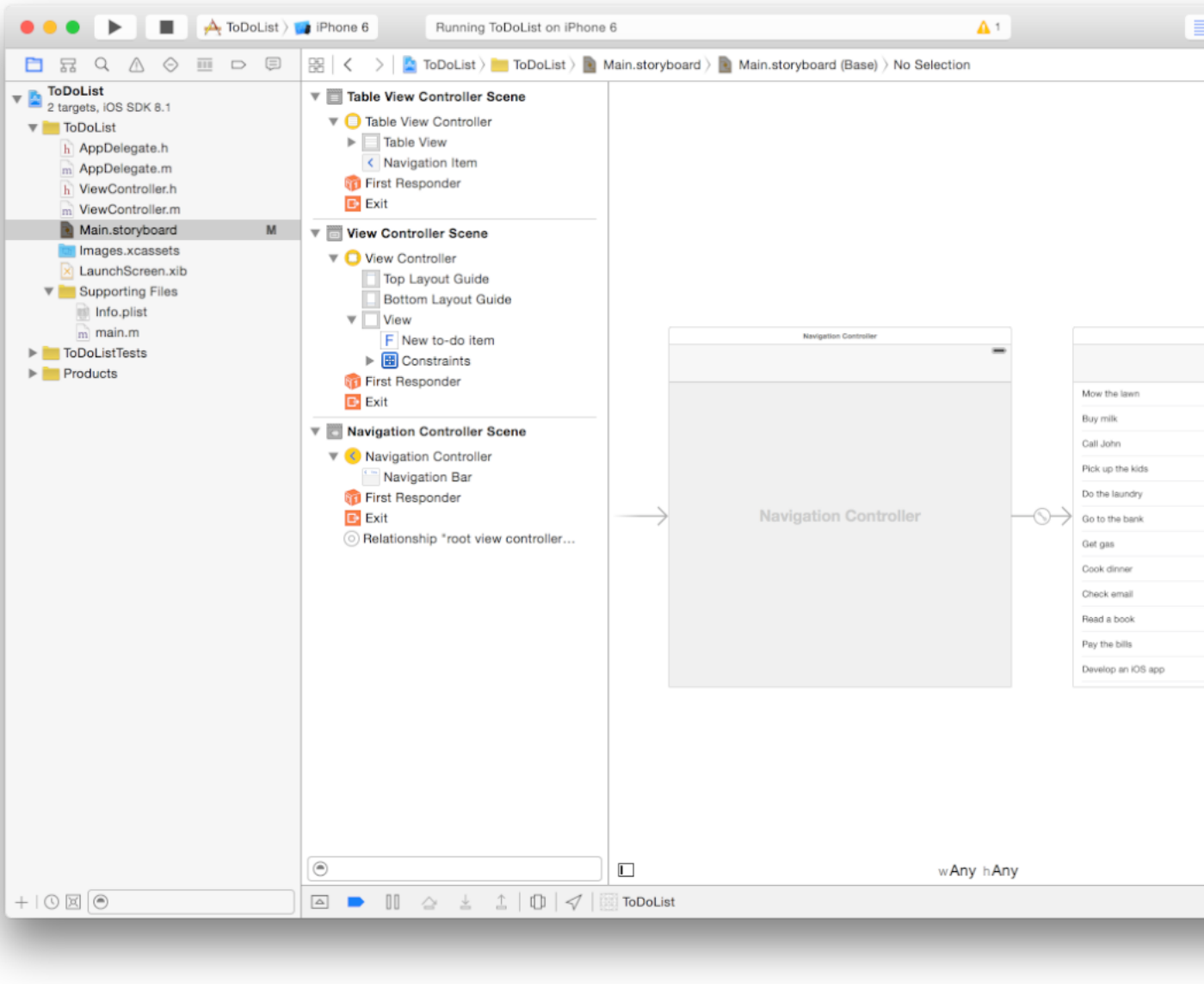
It's time to provide a way to navigate from this table view, with its list of to-do items, to the first scene you created, where a user can create a new to-do item. As your learned in Defining the Interaction (page 54), transitions between scenes are called *segues*.

Before creating a segue, you need to configure your scenes. First, you'll wrap your to-do list table view controller in a navigation controller. Recall from Defining the Interaction (page 54) that navigation controllers provide a navigation bar and keep track of the navigation stack. You'll add a button to this navigation bar to transition to the add-to-do-item scene.

To add a navigation controller to your table view controller

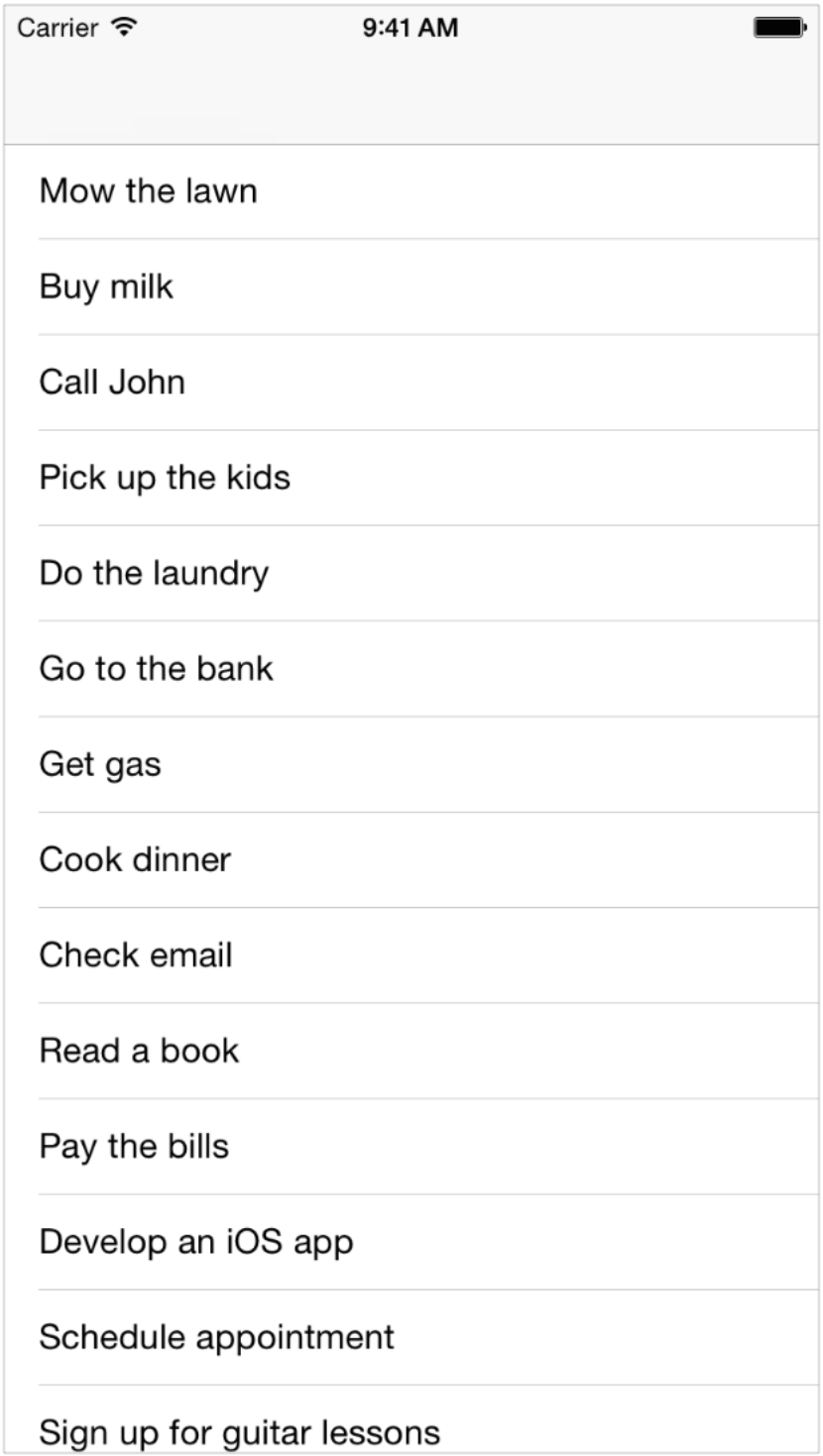
1. In the outline view, select Table View Controller.
2. With the table view controller selected, choose Editor > Embed In > Navigation Controller.

Xcode adds a new navigation controller to your storyboard, sets the storyboard entry point to it, and creates a relationship between the new navigation controller and your existing table view controller.



On the canvas, if you select the icon connecting the two scenes, you'll see that it's the root view controller relationship. This means that the view for the content displayed below the navigation bar will be your table view. The storyboard entry point is set to the navigation controller because the navigation controller holds all of the content that you'll display in your app—it's the container for both the to-do list and the add-to-do-item scenes.

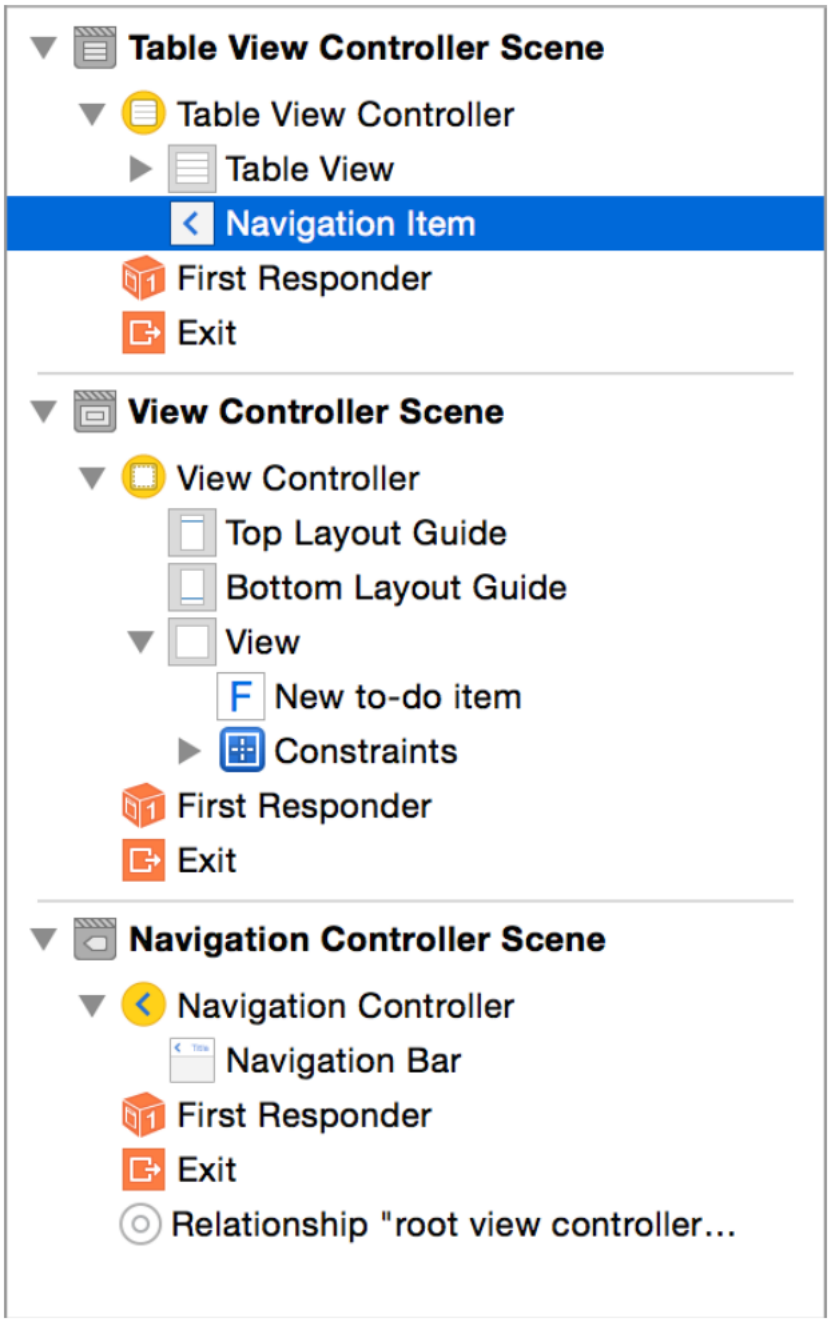
Checkpoint: Run your app. Above your table view you should now see extra space. This is the navigation bar provided by the navigation controller. The navigation bar extends its background to the top of the status bar, so the status bar doesn't overlap with your content anymore.



Now, you'll add a title (to the to-do list) and a button (to add additional to-do items) to the navigation bar. Navigation bars get their title from the view controller that the navigation controller currently displays—they don't themselves have a title. You set the title using the navigation item of your to-do list (the table view controller) rather than setting it directly on the navigation bar.

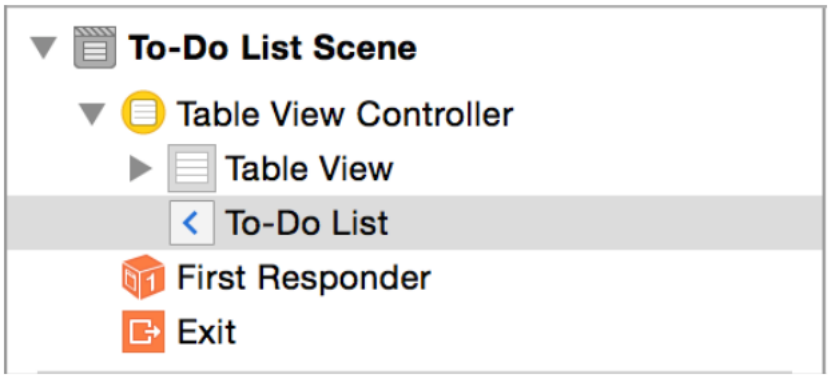
To configure the navigation bar in the table view controller

1. In the outline view or on the canvas, select Navigation Item under Table View Controller.



2. In the Attributes inspector, type `To–Do List` in the Title field. Press Return to save.

   Xcode changes the description of the scene from "Table View Controller Scene" to "To-Do List Scene" to make it easier for you to identify the scene. The description appears in the outline view.

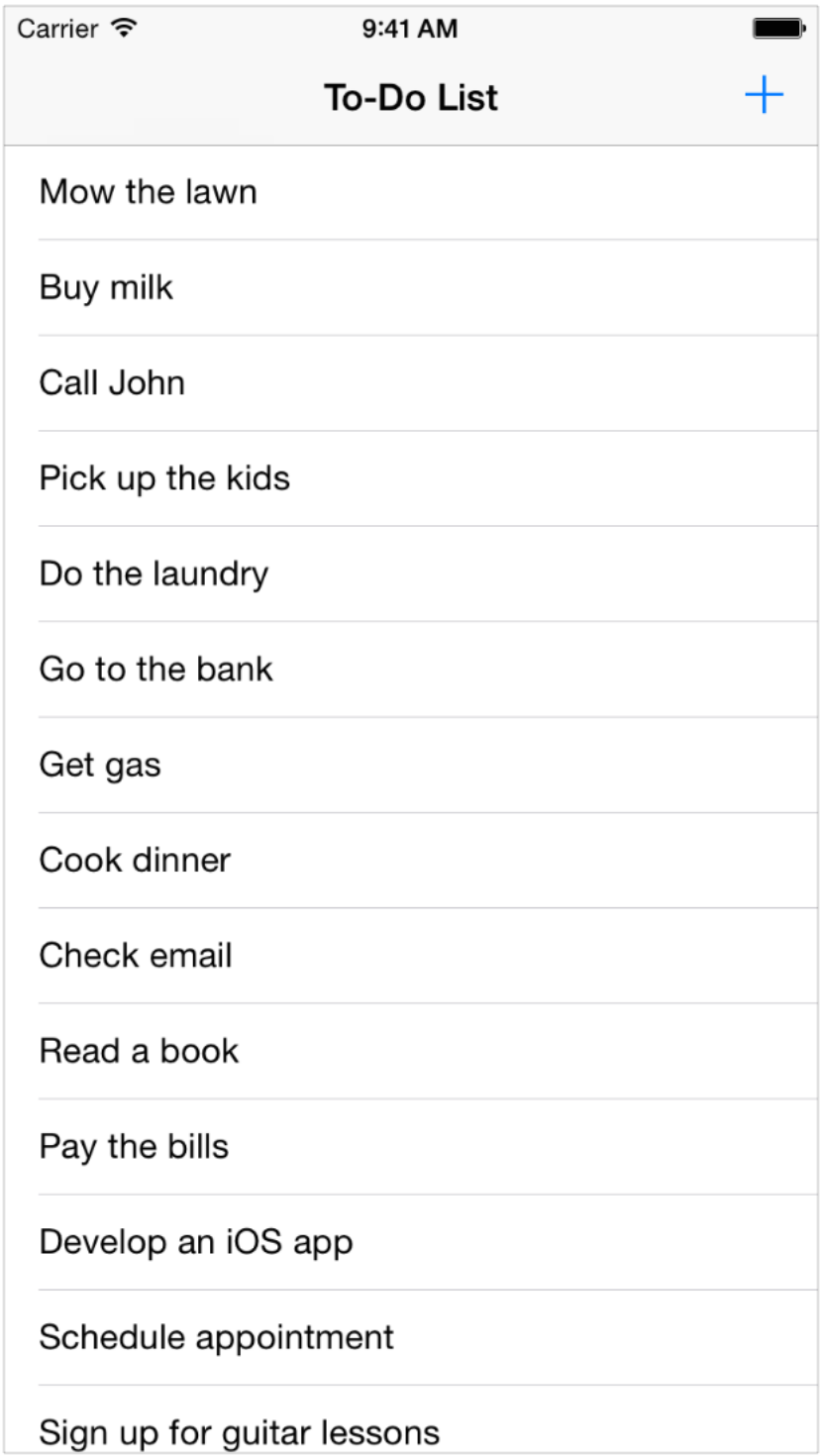


3. In the Object library, find a Bar Button Item object.
4. Drag a Bar Button Item object from the list to the far right of the navigation bar in the table view controller.

   A button containing the text "Item" appears where you dragged the bar button item.
5. In the outline view or on the canvas, select the bar button item.
6. In the Attributes inspector, find the Identifier option in the Bar Button Item section. Choose Add from the Identifier pop-up menu.

   The button changes to an Add button (+).

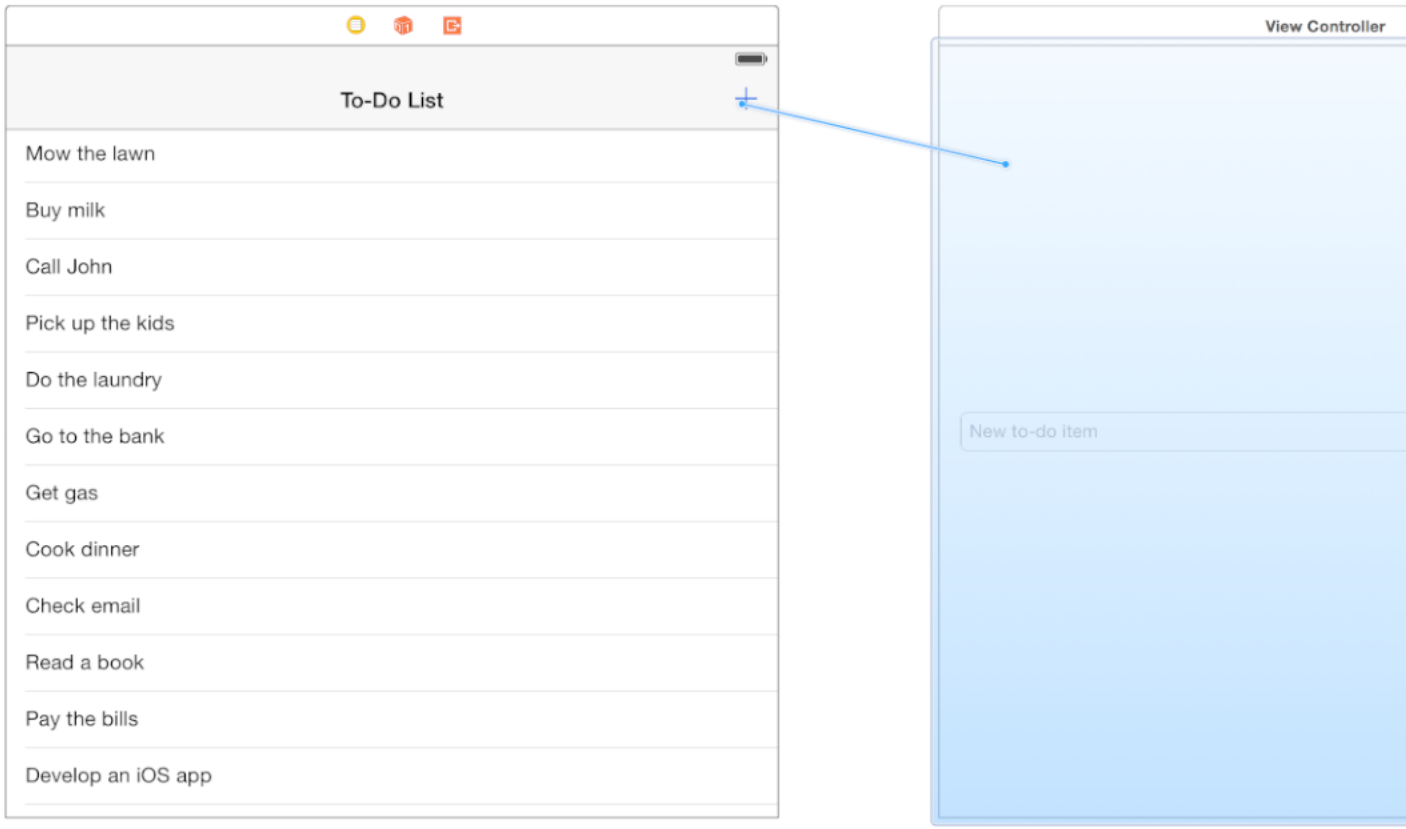
Checkpoint: Run your app. The navigation bar should now have a title and display an Add button. The button doesn't do anything yet. You'll fix that next.



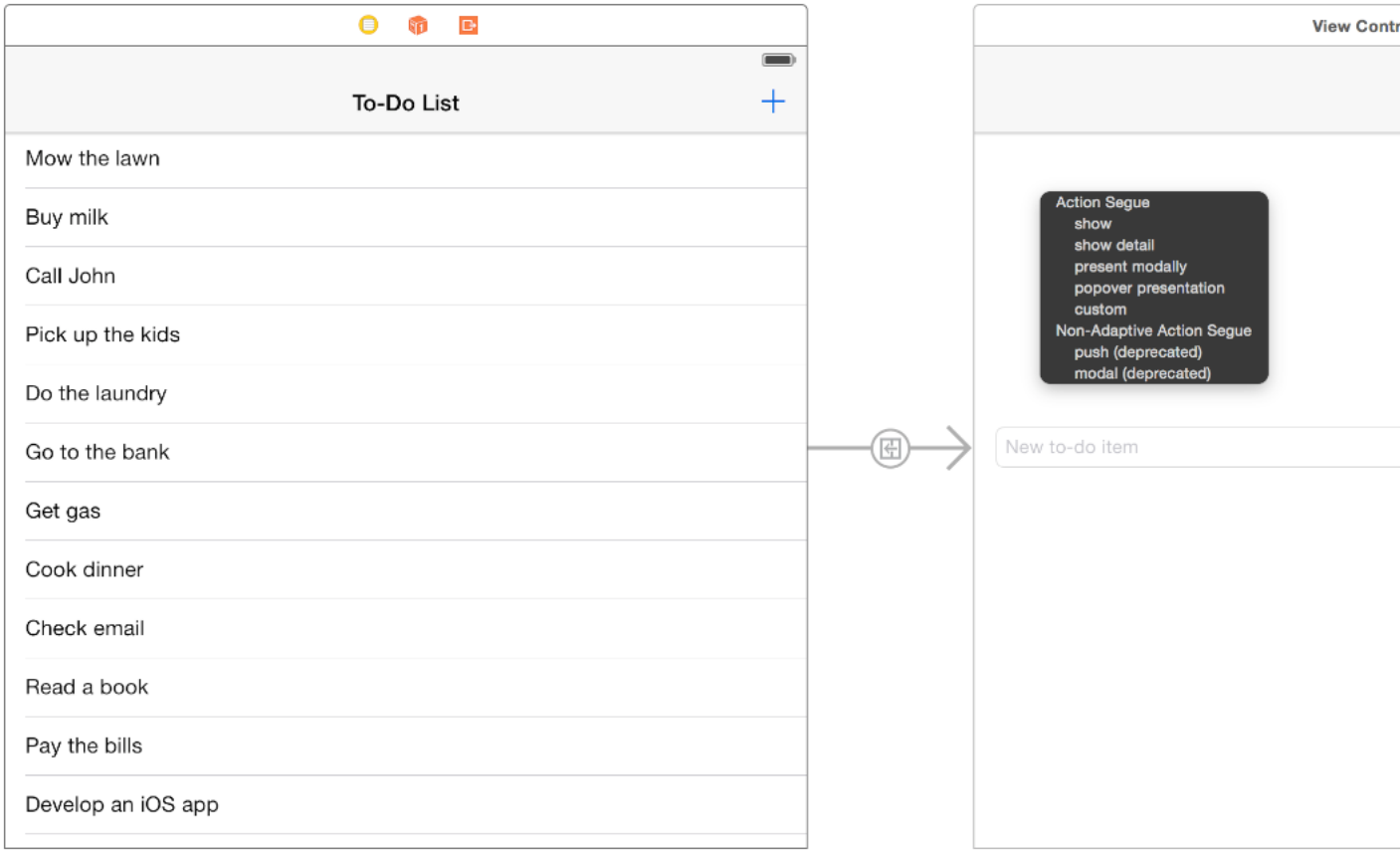
You want the Add button to bring up the add-to-do-item scene. The scene is already configured—it was the first scene you created—but it's not connected to the other scenes. You need to configure the Add button to bring up another scene when a user taps the button.

To configure the Add button

1. On the canvas, select the Add button.
2. Control-drag from the button to the add-to-do-item view controller.



A shortcut menu titled Action Segue appears in the location where the drag ended.

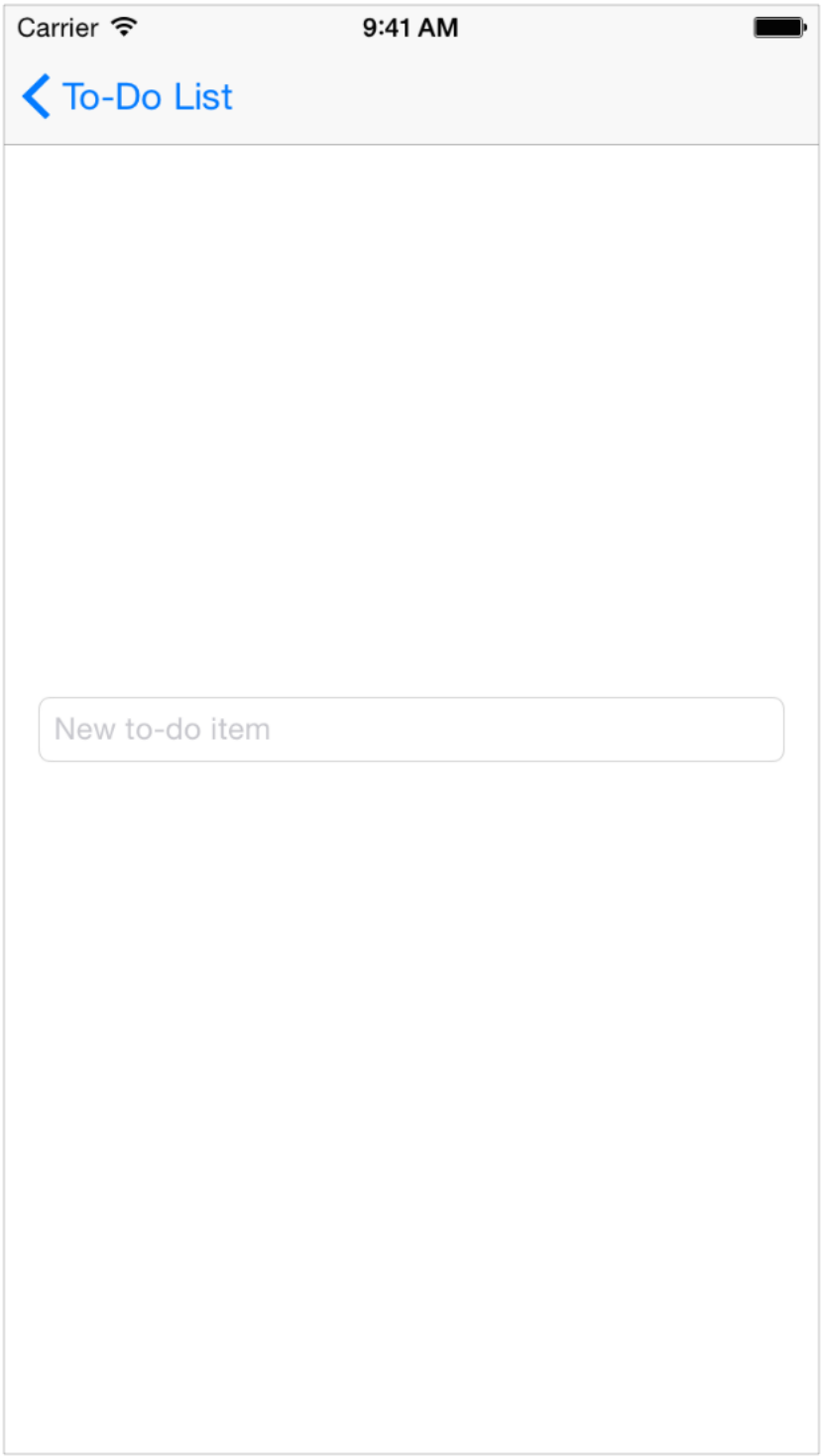


The Action Segue menu allows you to choose what type of segue to use to transition from the to-do list to the add-to-do-item view controller when the user taps the Add button.

3. Choose show from the Action Segue menu.

Xcode sets up the segue and configures the add-to-do-item view controller to be displayed in a navigation controller—you'll see the navigation bar in Interface Builder.

Checkpoint: Run your app. You can click the Add button and navigate to the add-to-do-item view controller from the table view. Because you're using a navigation controller with a show segue, the backward navigation is handled for you. This means you can click the back button to get back to the table view.

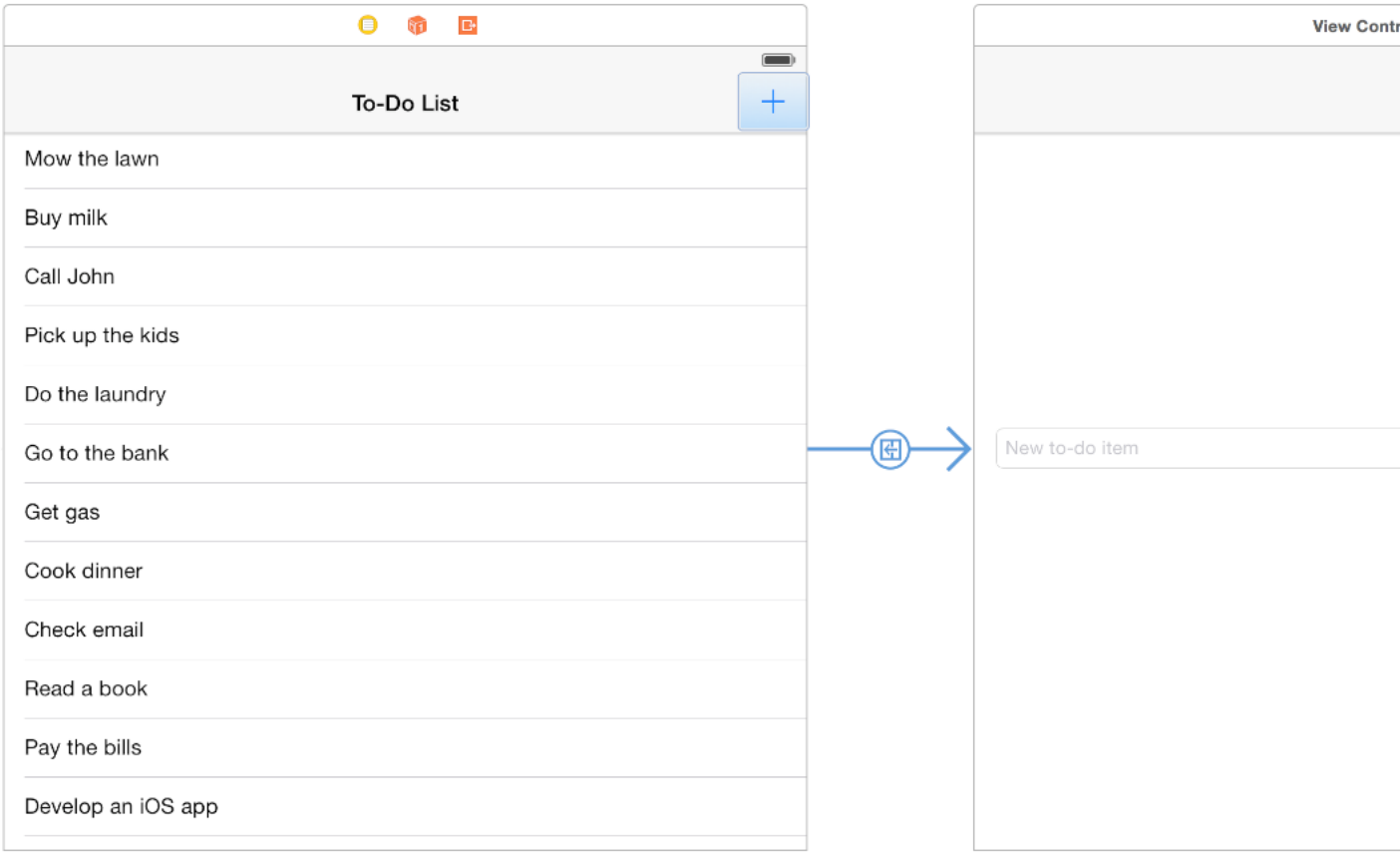


The push-style navigation you get by using the show segue is working just as it's supposed to—but it's not quite what you want when adding items. Push navigation is designed for a drill-down interface, where you're providing more information about whatever the user selected. Adding an item, on the other hand, is a modal operation—the user performs an action that's complete and self-contained, and then returns from that scene to the main navigation. The appropriate method of presentation for this type of scene is a modal segue.

Instead of deleting the existing segue and creating a new one, simply change the segue's style in Attributes inspector. Like most selectable elements in a storyboard, you can edit a segue's attributes using Attributes inspector.

To change the segue style

1. In the outline view or on the canvas, select the segue from the table view controller to the add-to-do-item view controller.



2. In the Attributes inspector, choose Present Modally from the pop-up menu next to the Segue option.

Because a modal view controller doesn't get added to the navigation stack, it doesn't get a navigation bar from the table view controller's navigation controller. However, you want to keep the navigation bar to provide the user with visual continuity. To give the add-to-do-item view controller a navigation bar when presented modally, embed it in its own navigation controller.

To add a navigation controller to the add-to-do-item view controller

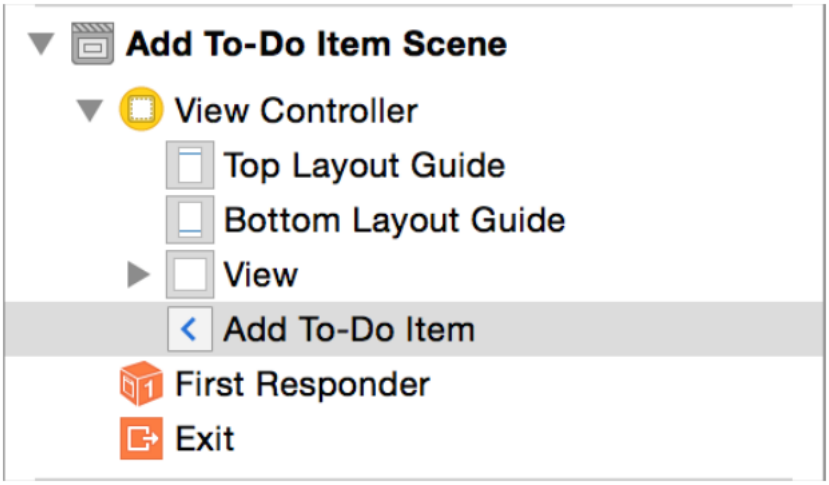
1. In the outline view, select View Controller.
2. With the view controller selected, choose Editor > Embed In > Navigation Controller.

As before, Xcode adds a navigation controller and shows the navigation bar at the top of the view controller. Next, configure this bar to add a title to this scene as well as two buttons, Cancel and Save. Later, you'll link these buttons to actions.

To configure the navigation bar in the add-to-do-item view controller

1. In the outline view or on the canvas, select Navigation Item under View Controller.
2. In the Attributes inspector, type `Add To-Do Item` in the Title field.

   Xcode changes the description of the scene from "View Controller Scene" to "Add To-Do Item Scene" to make it easier for you to identify the scene. The description appears in the outline view.

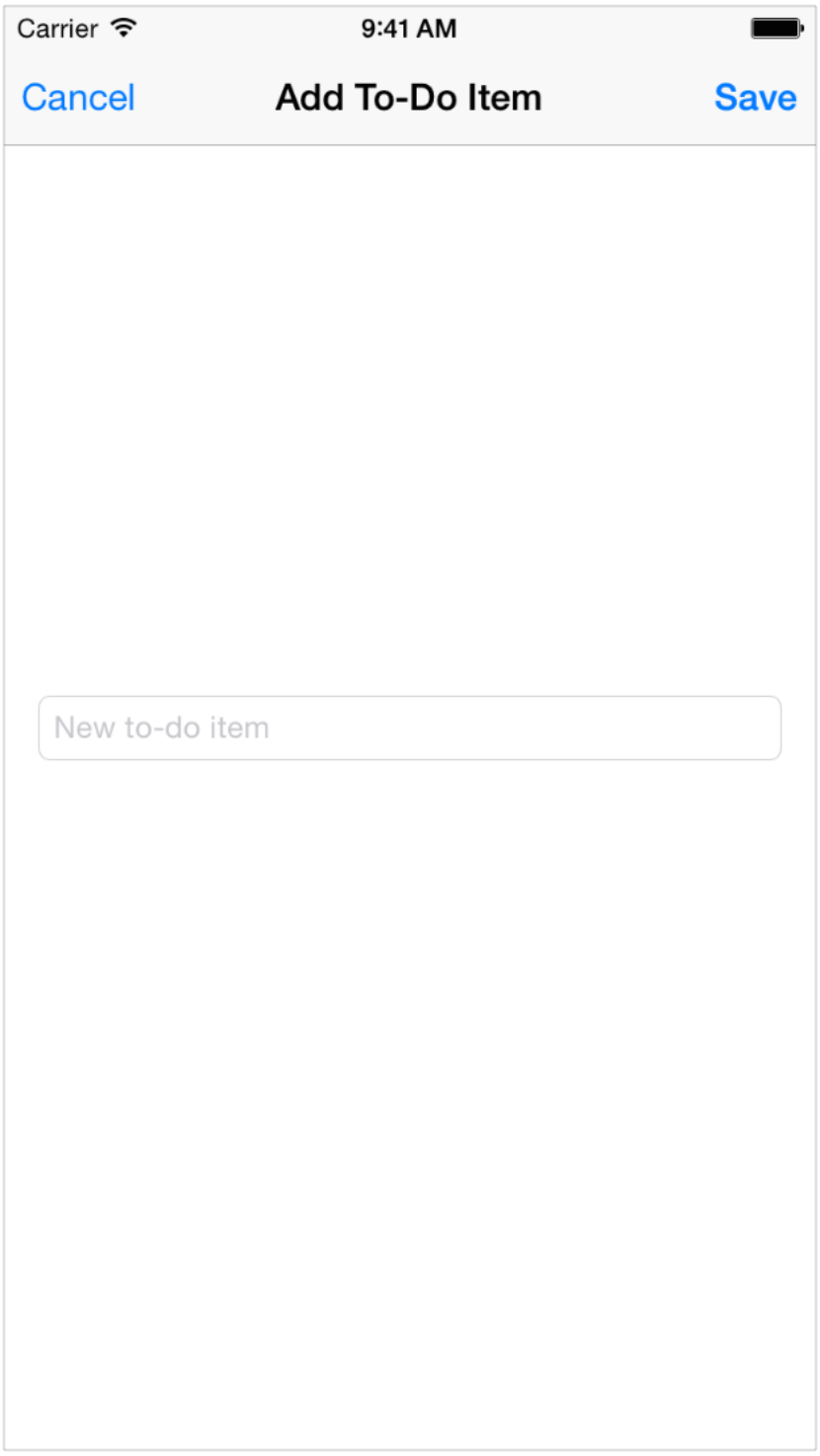


3. Drag a Bar Button Item object from the Object library to the far right of the navigation bar in the add-to-do-item view controller.
4. In the Attributes inspector, choose Save from the pop-up menu next to the Identifier option.

   The button text changes to "Save."
5. Drag another Bar Button Item object from the Object library to the far left of the navigation bar in the add-to-do-item view controller.
6. In the Attributes inspector, choose Cancel from the pop-up menu next to the Identifier option.

The button text changes to “Cancel.”

Checkpoint: Run your app. Click the Add button. You still see the add-to-do-item scene, but there's no longer a button to navigate back to the to-do list—instead, you see the two buttons you added, Save and Cancel. Those buttons aren't linked to any actions yet, so you can click them, but they don't do anything. Configuring the buttons to complete or cancel editing of the new to-do item—and bring the user back to the to-do list—is the next task.



Create Custom View Controllers

You've accomplished all of this configuration without writing any code. Configuring the completion of the add-to-do-item view controller requires some code, though, and you need a place to put it.

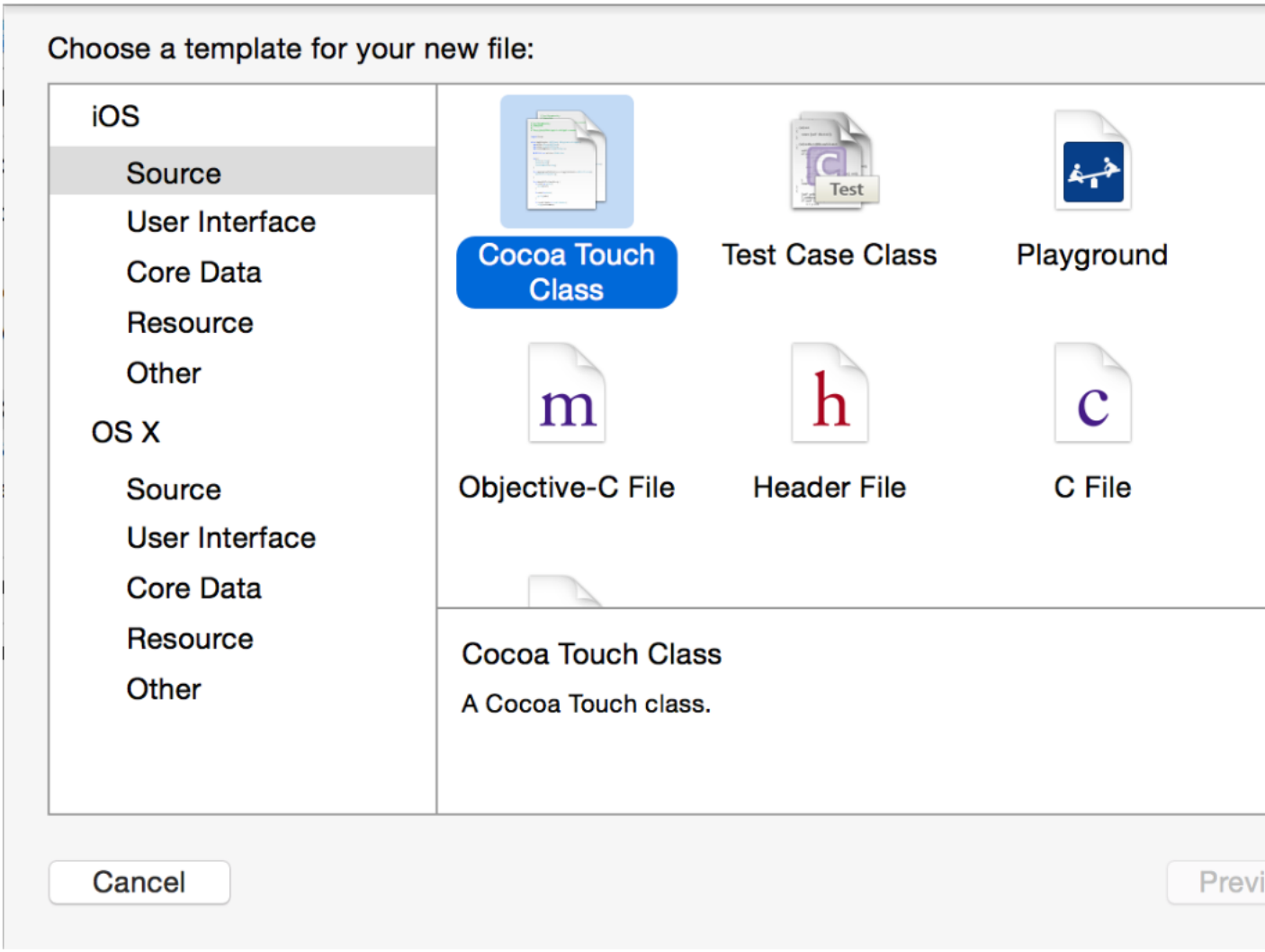
Right now Xcode has configured the add-to-do-item view controller to correspond to the code in `ViewController.h` and `ViewController.m` in your project navigator. These files represent a custom view controller subclass that got created for in you as part of the Single View Application template. However, it's important to learn how to create and configure custom classes on your own, because this is a common task in app development. So now, you'll create a custom class called `AddToDoItemViewController` to correspond with the add-to-do-item scene in your storyboard. `AddToDoItemViewController` will subclass `UIViewController` so that it gets all of the basic view controller behavior.

(Go ahead and delete `ViewController.h` and `ViewController.m` if you'd like, because you won't be using them. To delete a file from your project, select it in the project navigator, press the Delete key, and click Move to Trash in the message that appears.)

To create a subclass of UIViewController

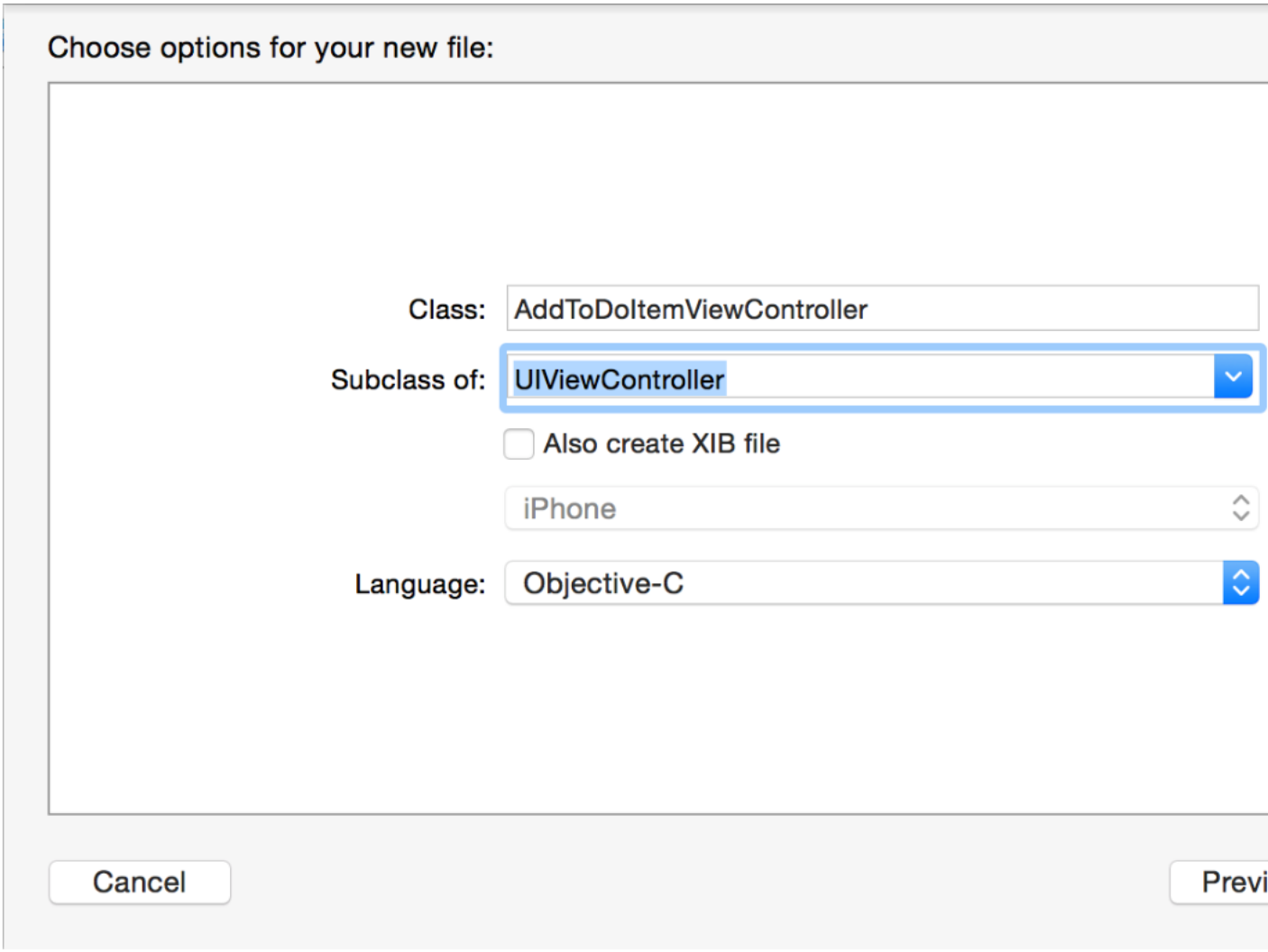
1. Choose File > New > File (or press Command-N).

2. On the left of the dialog that appears, select Source under iOS.



3. Select Cocoa Touch Class, and click Next.
4. In the Class field, type `AddToDoItem`.
5. Choose `UIViewController` in the "Subclass of" pop-up menu.

The class title changes to AddToDoItemViewController. Xcode makes it clear from the naming that you're creating a custom view controller, so leave the new name as is.



6. Make sure the "Also create XIB file" option is unselected.
7. Make sure the Language option is set to Objective-C.
8. Click Next.

   The save location defaults to your project directory.

   The Group option defaults to your app name, ToDoList.

   In the Targets section, your app is selected and the tests for your app are unselected.
9. Leave these defaults as they are, and click Create.

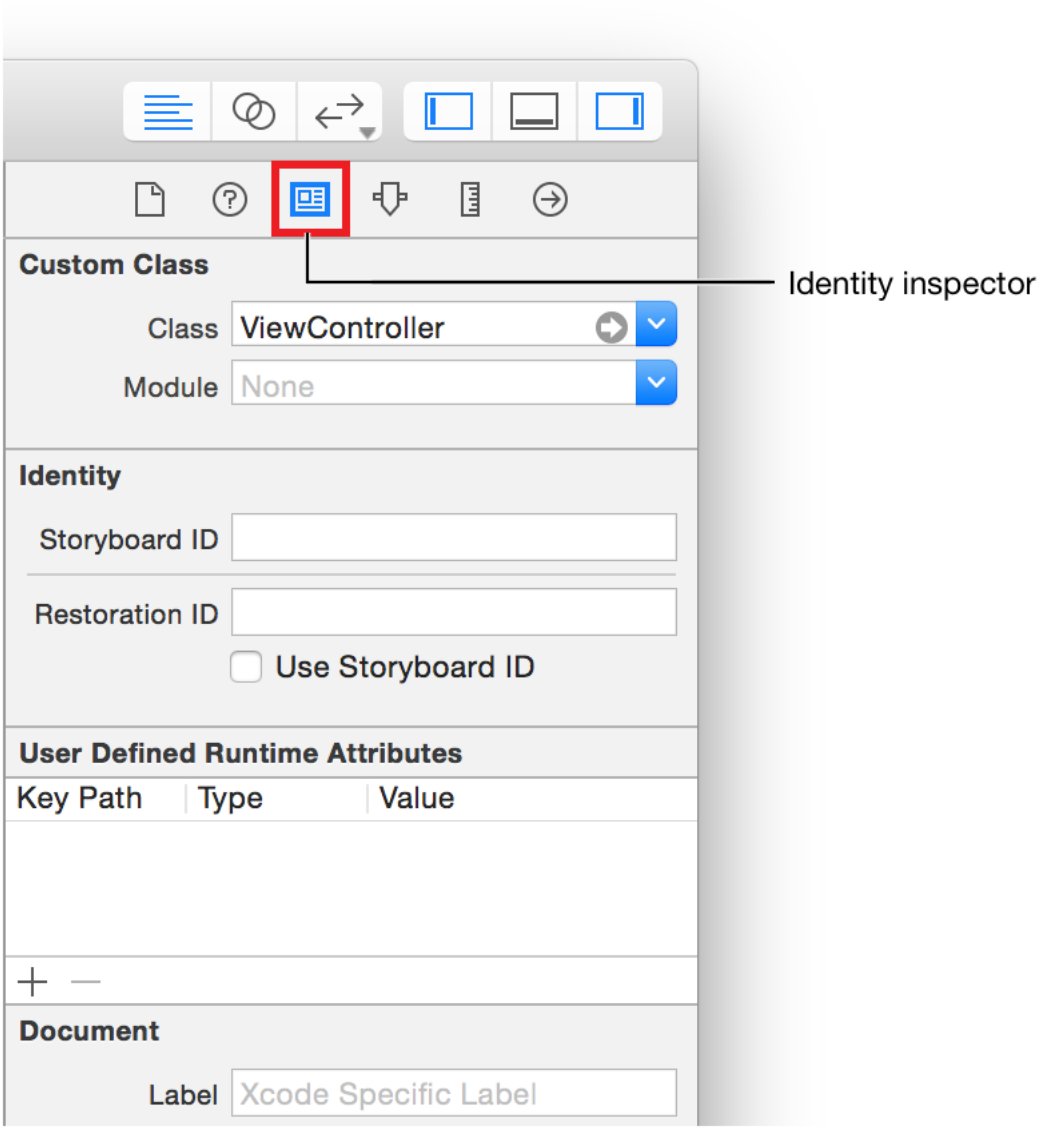
   Xcode creates a pair of files that define the `AddToDoItemViewController` class: the interface file (`AddToDoItemViewController.h`), and the implementation file (`AddToDoItemViewController.m`).

Now that you've created a custom view controller subclass, you need to tell your storyboard to use it: you'll switch the add-to-do-item scene from using the old `ViewController` class to the new `AddToDoItemViewController` class.

To identify AddToDoItemViewController as the class for the add-to-do-item scene

1. In the project navigator, select `Main.storyboard`.
2. If necessary, open the outline view  .
3. In the outline view, select View Controller under Add To-Do Item Scene.
4. With the view controller selected, click the third icon from the left in the utility area to open the Identity inspector  .

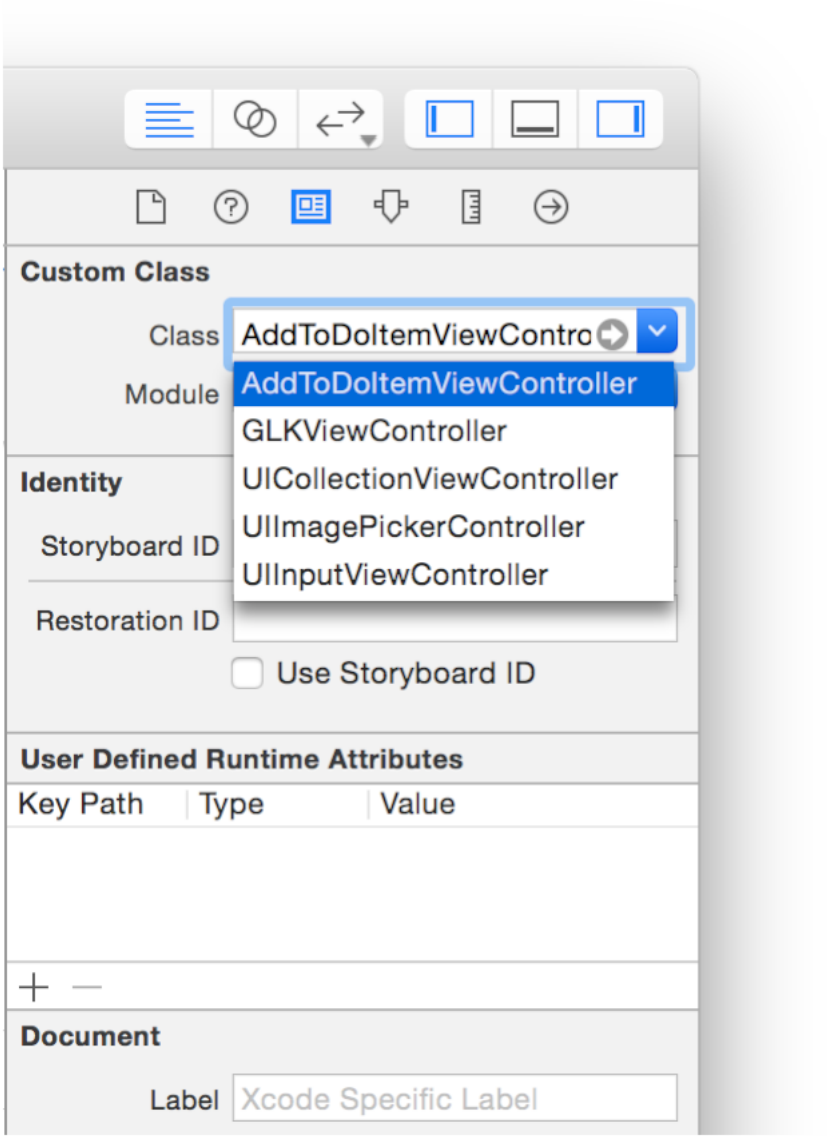
The *Identity inspector* lets you edit properties of an object in your storyboard related to that object's identity, such as what class the object belongs to.



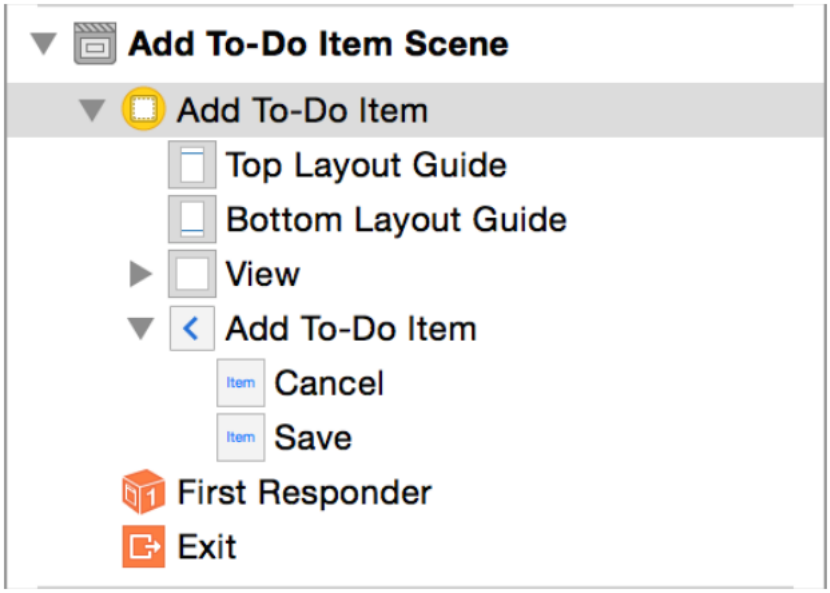
5. In the Identity inspector, open the pop-up menu next to the Class option.

   You'll see a list of all the view controller classes Xcode knows about. The first one in the list should be your custom view controller, `AddToDoItemViewController`.

6. Choose `AddToDoItemViewController` to use it for this scene.



Notice that Xcode changed the description of your add-to-do-item view controller from "View Controller" to "Add To-Do Item" in the outline view. Xcode interprets the name of the custom class to make it easier to understand what's going on in the storyboard.



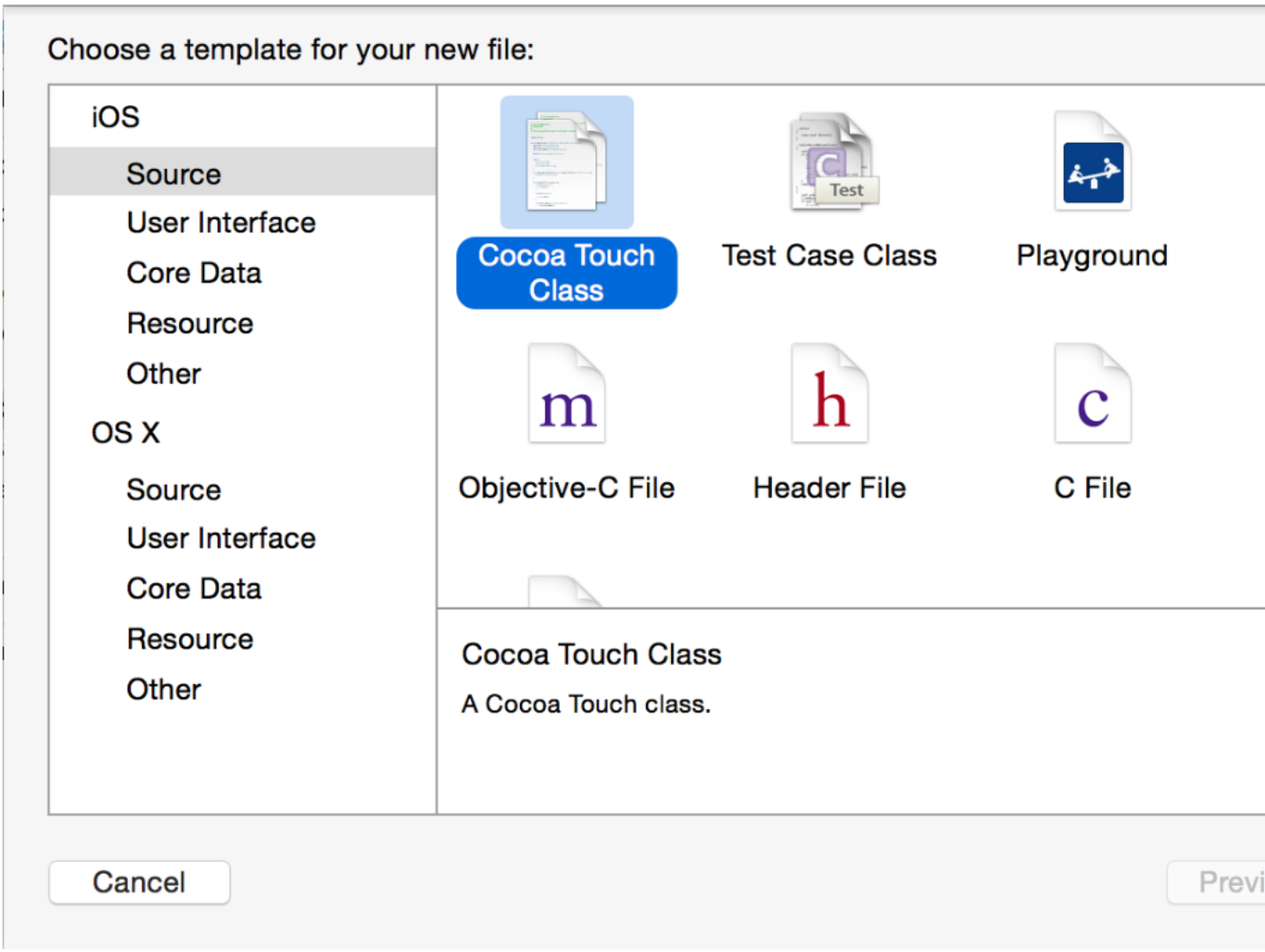
At runtime your storyboard will create an instance of `AddToDoItemViewController`, your custom view controller subclass. The add-to-do-item screen in the app will get the interface defined in your storyboard and the behavior defined in `AddToDoItemViewController.m`.

Now, create a custom class to correspond with the to-do list scene in your storyboard. Because the to-do list scene is a table view controller, this class should subclass `UITableViewController`. `UITableViewController` offers the same basic view controller behavior as `UIViewController` with a few specialized pieces just for table views.

To create a subclass of UITableViewController

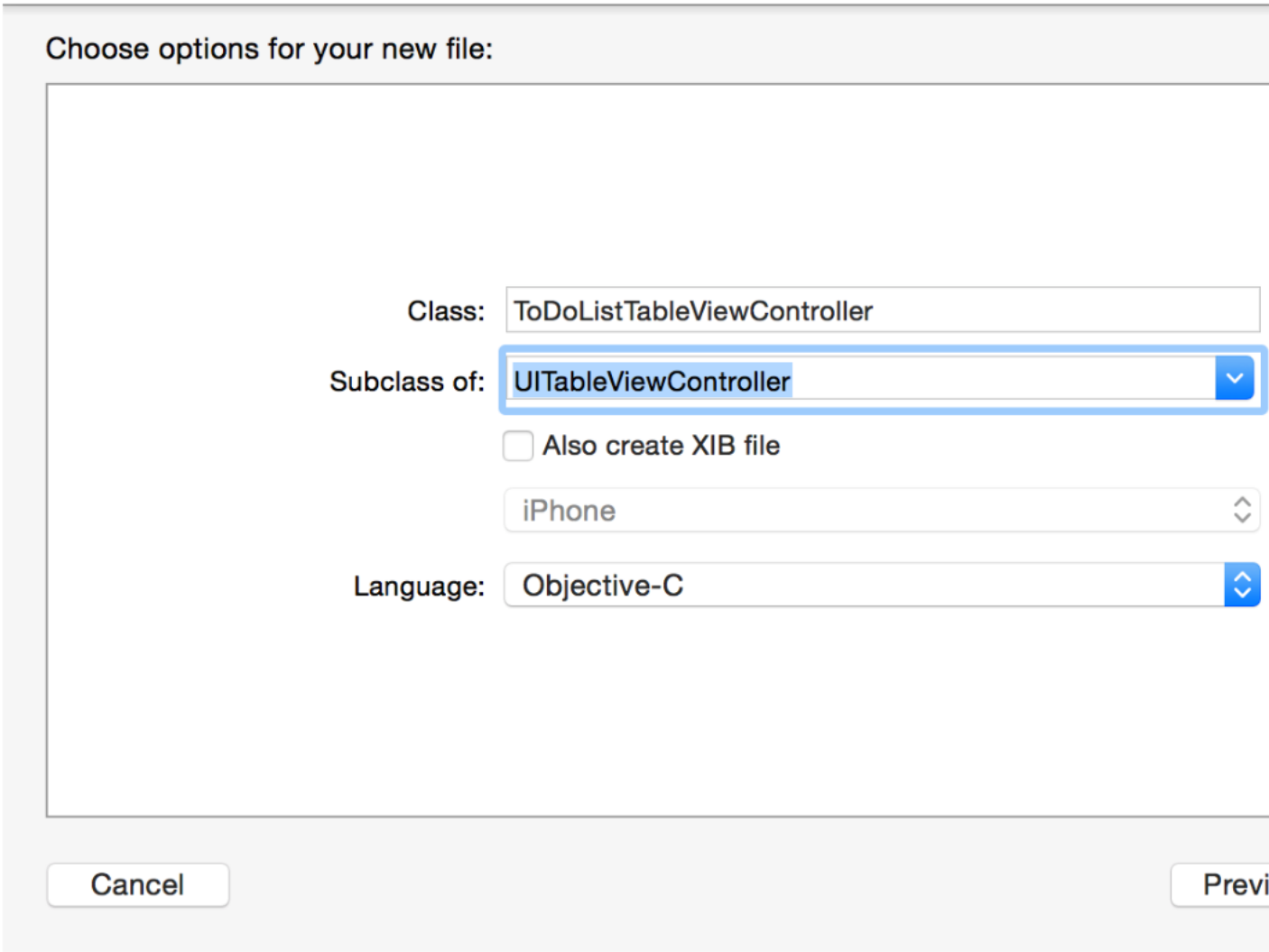
1. Choose File > New > File (or press Command-N).

2. On the left, select Source under iOS, then select Cocoa Touch Class. If you haven't created any classes since the last steps in the tutorial, it's probably already selected for you.



3. Click Next.
4. In the Class field, type `ToDoList` before `ViewController`.
5. Choose `UITableViewController` in the "Subclass of" pop-up menu.

The class title changes to ToDoListTableViewController. Leave that as is.



6. Make sure the "Also create XIB file" option is unselected.
7. Make sure the Language option is set to Objective-C.
8. Click Next.

   The save location defaults to your project directory.

   The Group option defaults to your app name, ToDoList.

   In the Targets section, your app is selected and the tests for your app are unselected.
9. Leave these defaults as they are, and click Create.

   Xcode creates a pair of files that define the `ToDoListTableViewController` class: the interface file (`ToDoListTableViewController.h`), and the implementation file (`ToDoListTableViewController.m`).

Once again, you need to make sure to configure your custom table view controller, `ToDoListTableViewController`, in your storyboard.

To identify ToDoListTableViewController as the class for the to-do list scene

1. In the project navigator, select `Main.storyboard`.
2. In the outline view, select Table View Controller under To-Do List Scene and open the Identity inspector  in the utility area.
3. In the Identity inspector, choose `ToDoListTableViewController` from the pop-up menu next to the Class option.

Now, you're ready to add custom code to your view controllers.

Create an Unwind Segue to Navigate Back

In addition to show and modal segues, Xcode provides an **unwind segue**. This segue allows users to go from a given scene back to a previous scene, and it provides a place for you to add your own code that gets executed when users navigate to the previous scene. You can use an unwind segue to navigate back from `AddToDoItemViewController` to `ToDoListTableViewController`.

To create an unwind segue, you first add an action method to the destination view controller (the view controller for the scene you want to unwind to). This method must return an action (`IBAction`) and take a segue (`UIStoryboardSegue`) as a parameter. Because you want to unwind back to the to-do list scene, you need to add an action method with this format to the `ToDoListTableViewController` interface and implementation.

To add an action method to ToDoListTableViewController

1. In the project navigator, open `ToDoListTableViewController.h`.
2. Add the following code on the line below `@interface`:

```
- (IBAction)unwindToList:(UIStoryboardSegue *)segue;
```

3. In the project navigator, open `ToDoListTableViewController.m`.
4. Add the following code on the line below `@implementation`:

```
- (IBAction)unwindToList:(UIStoryboardSegue *)segue {
```

```
}
```

You can name the unwind action anything you want. Calling it `unwindToList:` makes it clear where the unwind will take you. For future projects, adopt a similar naming convention, one where the name of the action makes it clear where the unwind will go.

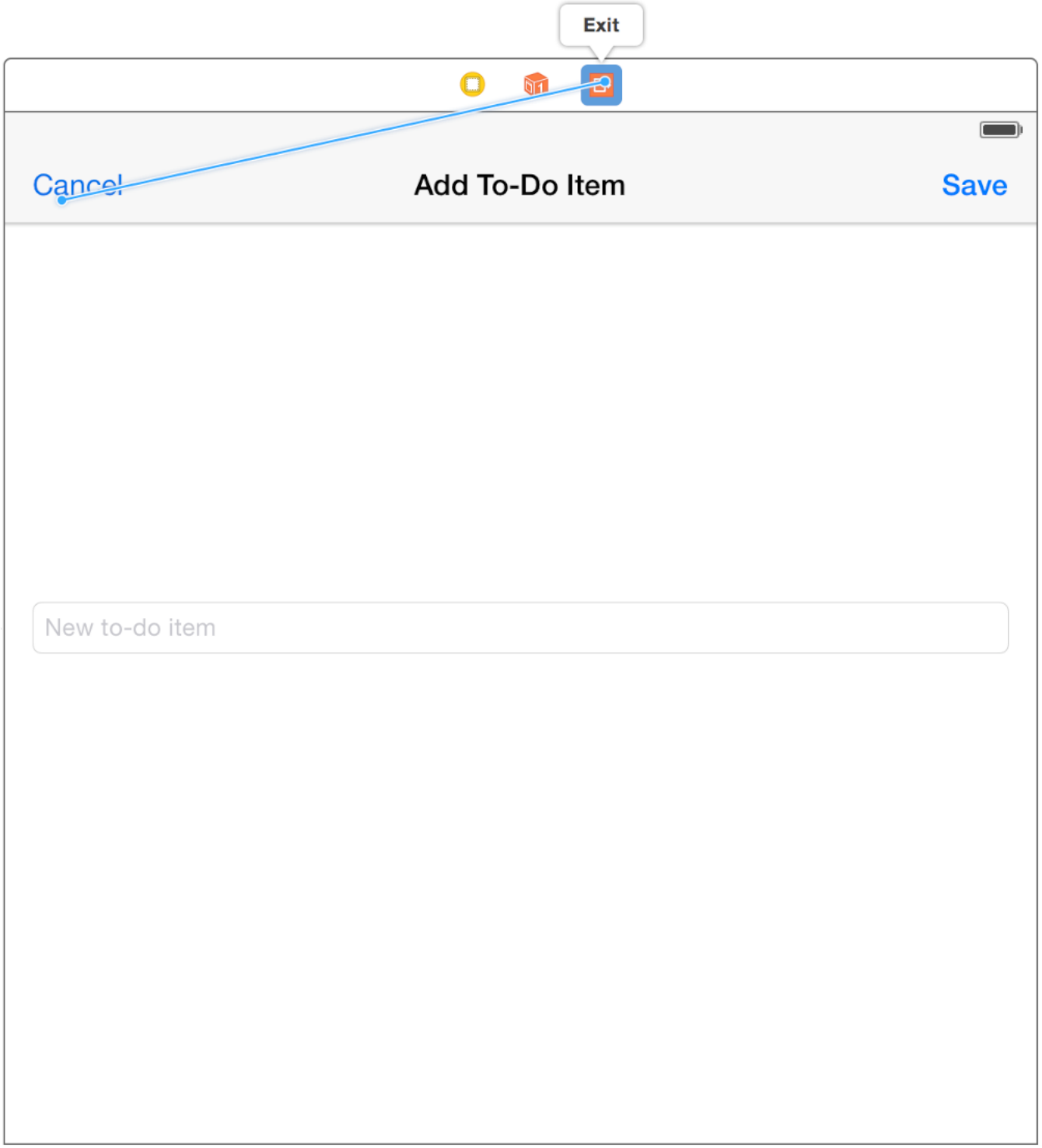
For now, leave this method implementation empty. Later, you'll use this method to retrieve data from the `AddToDoItemViewController` to add an item to your to-do list.

To create the unwind segue, link the Cancel and Save buttons to the `unwindToList:` action. In your storyboard, you do this using the Exit icon in the dock of the source view controller, `AddToDoItemViewController`.

To link buttons to the unwindToList: action method

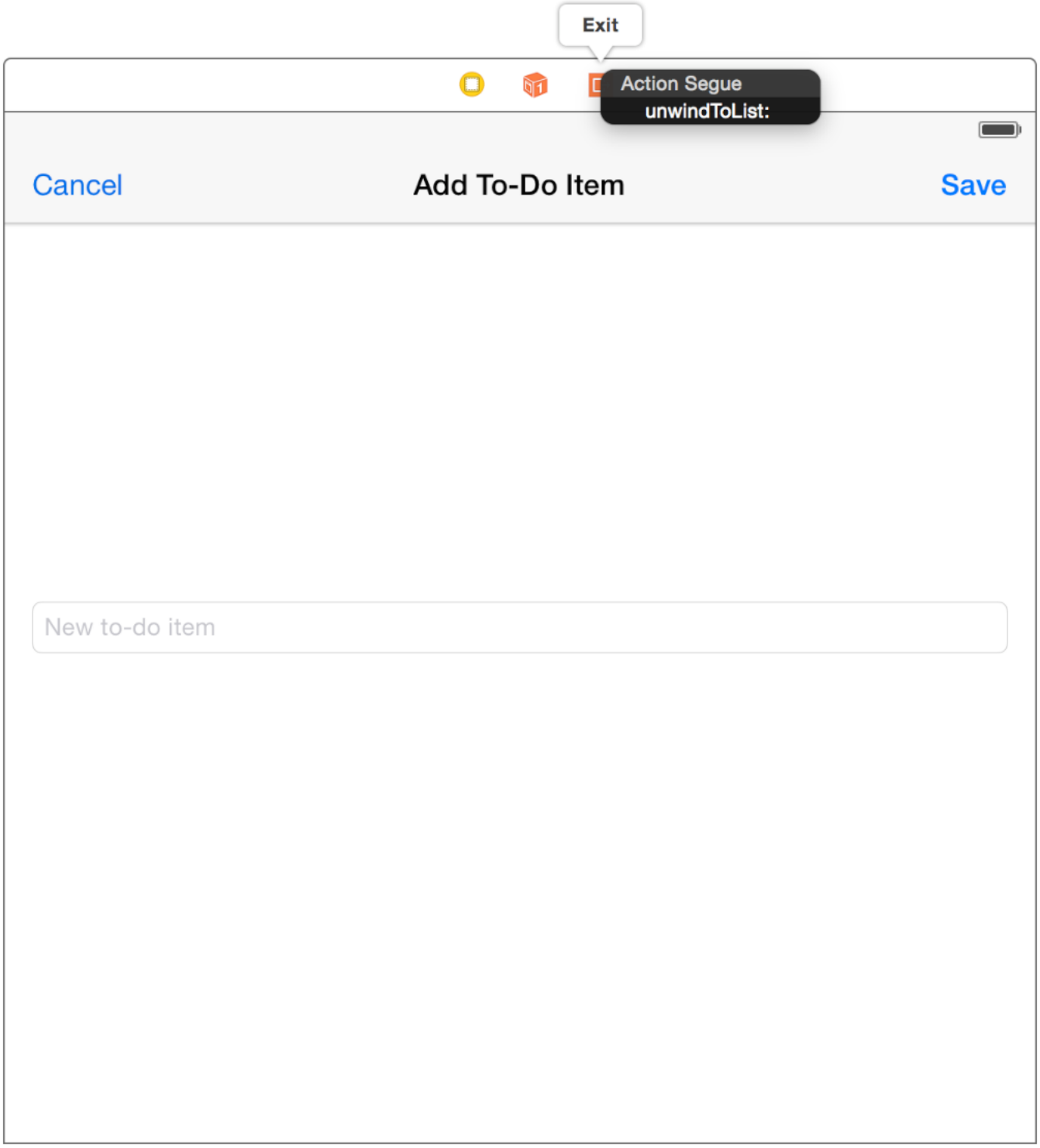
1. In the project navigator, select `Main.storyboard`.

2. On the canvas, Control-drag from the Cancel button to the Exit item at the top of the add-to-do-item scene.



If you don't see the Exit item but instead see the description of the scene, zoom in using Editor > Canvas > Zoom until you see it.

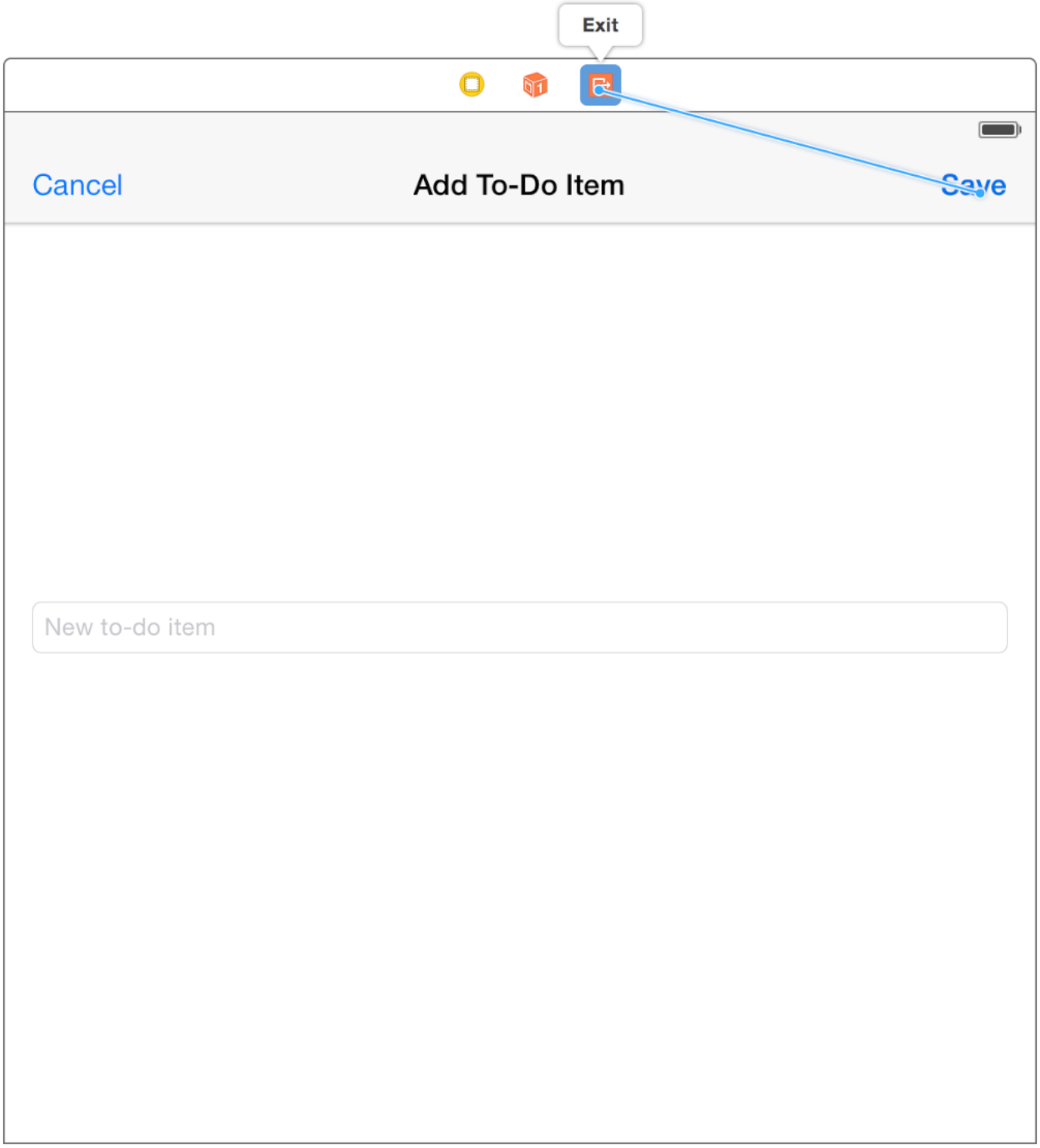
A menu appears in the location where the drag ended.



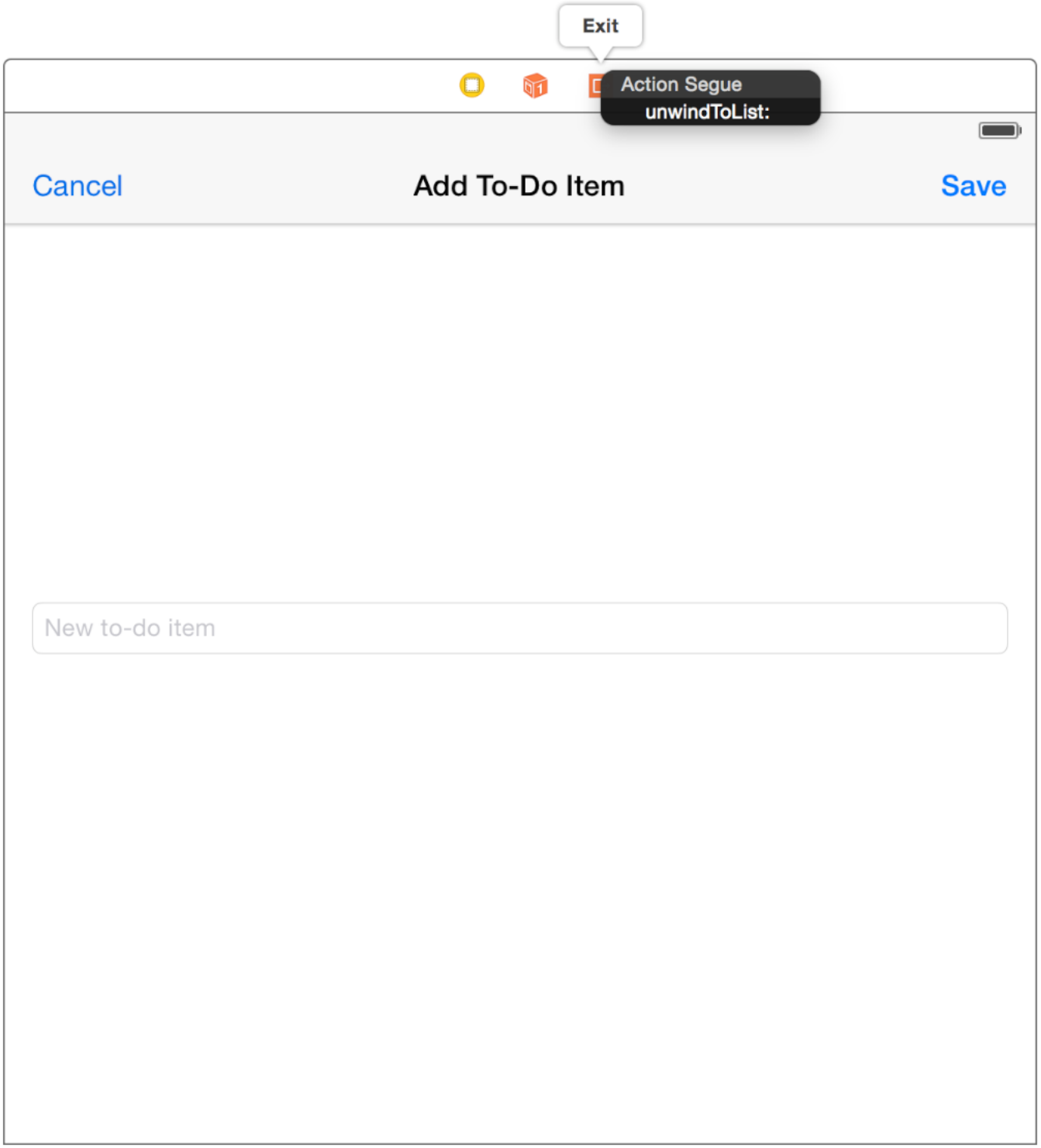
3. Choose `unwindToList:` from the shortcut menu.

This is the action you just added to the `ToDoListTableViewController.m` file. Now, when users tap the Cancel button, they will navigate back to the to-do list scene. At this point, `unwindToList:` will be called.

4. On the canvas, Control-drag from the Save button to the Exit item at the top of the add-to-do-item scene.



A menu appears in the location where the drag ended.



5. Choose `unwindToList:` from the shortcut menu.

Now, when users tap the Save button, they will navigate back to the to-do list scene. At this point, `unwindToList:` will be called.

Notice that you used the same action for both the Cancel and the Save buttons. In the next tutorial, you'll distinguish between the two different cases when you write the code to handle the unwind segue.

Checkpoint: Now, run your app. At launch, you see a table view—but there's no data in it. You can click the Add button and navigate to the add-to-do-item scene from the to-do list scene. You can click the Cancel and Save buttons to navigate back to the to-do list scene.

So why doesn't your data show up? Table views have two ways of getting data—statically or dynamically. When a table view's controller implements the required `UITableViewDataSource` methods, the table view asks its view controller for data to display, regardless of whether static data has been configured in Interface Builder. If you look at `ToDoListTableViewController.m`, you'll notice that it implements three methods—`numberOfSectionsInTableView:`, `tableView:numberOfRowsInSection:`, and `tableView:cellForRowAtIndexPath:`. You'll be working with these methods in the next tutorial to display dynamic data.

Recap

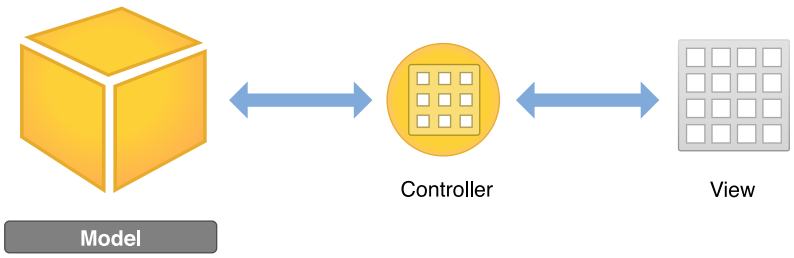
At this point, you've finished developing the interface for your app. You have two scenes—one for adding items to your to-do list and one for viewing the list—and you can navigate between them. In the next module, you'll implement the ability for users to add a new to-do item and have it appear in the list.

Implementing an App

Incorporating the Data

After you implement your app's behavior, you create a *data model* to support your app's interface. An app's data model defines the way you maintain data in your app. Data models can range from a basic dictionary of objects to complex databases. A good data model makes it easier to build a scalable app, improve functionality, and make changes to your features.

Your app's data model is composed of your data structures and (optionally) custom business logic needed to keep that data in a consistent state. You never want to design your data model in total isolation from your app's user interface. You do, however, want to implement your data model objects separately, without relying on the presence of specific views or view controllers. When you keep your data separate from your user interface, you'll find it easier to implement a universal app—one that can run on both iPad and iPhone—and easier to reuse portions of your code later.



Designing Your Model

If you simply need to store a small amount of data, Foundation framework classes may be your best option. Research existing Foundation classes to see what behaviors are available for you to use instead of attempting to implement the same thing on your own. For example, if your app only needs to keep track of a list of strings, you can rely on `NSArray` and `NSString` to do the job for you. You'll learn more about these and other Foundation classes in Working with Foundation (page 108).

If your data model requires custom business logic in addition to just storing data, you can write a custom class. Consider how you can incorporate existing framework classes into the implementation of your own classes. It's beneficial to use existing framework classes within your custom classes instead of trying to reinvent them. For example, a custom class might use `NSMutableArray` to store information—but define its own features for working with that information.

When you design your data model, ask yourself these questions:

How will the user use your app, and what types of data do you need to store? Make sure your model reflects the app's content and purpose. Even though the user doesn't interact with the data directly, there should be a clear correlation between the interface and the data. Whether you're storing text, documents, large images, or another type of information, design your data model to handle that particular type of content appropriately.

What data structures can you use? Determine where you can use framework classes and where you need to define classes with custom functionality.

How will you supply data to the user interface? Your model shouldn't communicate directly with your interface. To handle the interaction between the model and the interface, you'll need to add logic to your controllers.

Implementing Your Model

In Tutorial: Storyboards (page 60), you started working with the Objective-C programming language. Although this guide teaches you how to build a simple app, you'll want to become familiar with the language before writing your own fully functional app.

Some people learn the concepts by reading *Programming with Objective-C* and then writing a number of small test apps to solidify their understanding of the language and to practice writing good code. Others jump right into programming and look for more information when they don't know how to accomplish something. If you prefer this approach, keep *Programming with Objective-C* as a reference and make it an exercise to learn concepts and apply them to your app as you develop it.

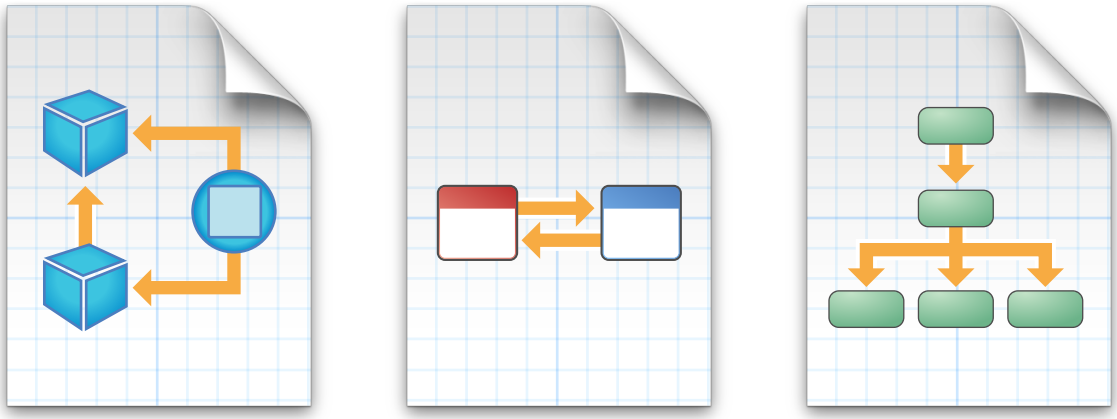
The most important goal in developing your first data model is to get something that works. Think carefully about the structure of your data model, but don't worry about making it perfect. Don't be afraid to iterate and refine your model after you begin implementing it.

Incorporating Real Data

When you first test your data model, you may want to use static or fake data until you know the model is assembled and connected properly. After you've defined a working data model, you can pull real data into your app.

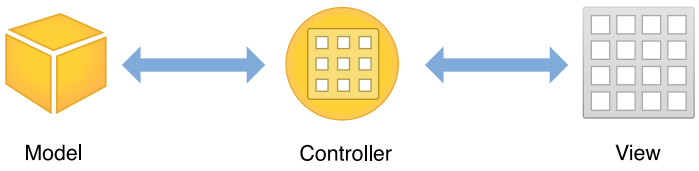
Using Design Patterns

A *design pattern* solves a common software engineering problem. Patterns are abstract designs, not code. You use them to help you define the structure of your data model and its interaction with the rest of your app. When you adopt a design, you adapt its general pattern to your specific needs. No matter what type of app you're creating, it's good to know the fundamental design patterns used in the frameworks. Understanding design patterns helps you use frameworks more effectively and allows you to write apps that are more reusable, more extensible, and easier to change.



MVC

Model-View-Controller (MVC) is central to a good design for any iOS app. MVC assigns the objects in an app to one of three roles: model, view, or controller. In this pattern, models keep track of your app's data, views display your user interface and make up the content of an app, and controllers manage your views. By responding to user actions and populating views with content from the data model, controllers serve as a gateway for communication between the model and views.

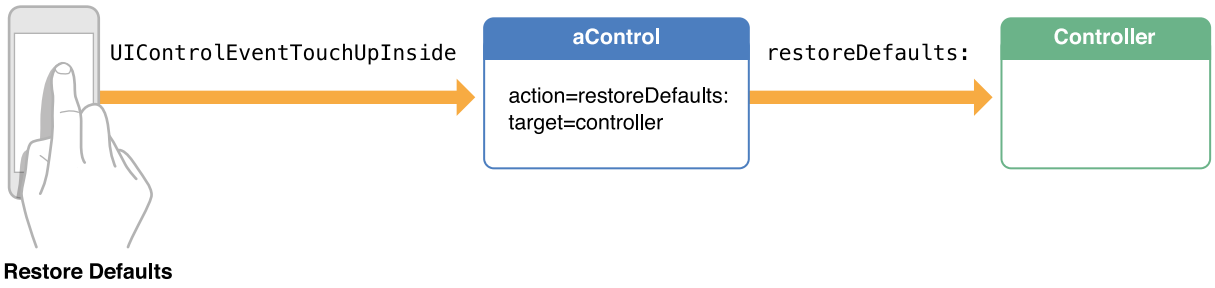


As you've built your ToDoList app, you've followed an MVC-centric design. The interface you built in storyboards makes up the view layer. `AddToDoItemViewController` and `ToDoListTableViewController` are the controllers that manage your views. In Tutorial: Add Data (page 126), you'll be incorporating a data model to work with the views and controllers in your app. When you begin designing your own app, it's important to keep MVC at the center of your design.

Target-Action

Target-action is a conceptually simple design in which one object sends a message to another object when a specific event occurs. The *action message* is a selector defined in source code, and the *target*—the object that receives the message—is an object capable of performing the action, typically a view controller. The object that sends the action message is usually a control—such as a button, slider, or switch—that can trigger an event in response to user interaction such as tap, drag, or value change.

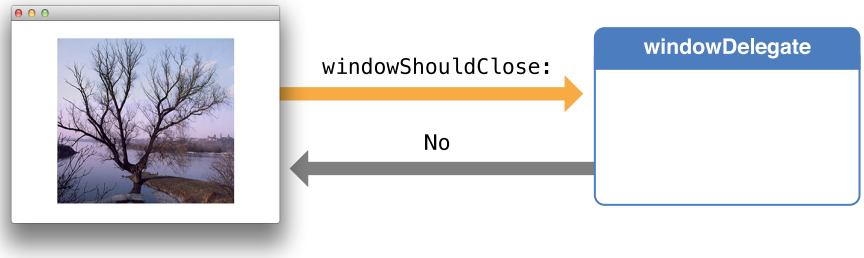
For example, imagine that you want to restore default settings in your app whenever a user taps the Restore Defaults button (which you create in your user interface). First, you implement an action, `restoreDefaults:`, to perform the logic to restore default settings. Next, you register the button's Touch Up Inside event to send the `restoreDefaults:` action message to the view controller that implements that method.



You've already used target-action in your ToDoList app. When a user taps the Save button in the `AddToDoItemViewController`, it triggers the `unwindToList:` action. In this case, the Save button is the object sending the message, the target object is the `ToDoListTableViewController`, the action message is `unwindToList:`, and the event that triggers the action message to be sent is a user tapping the Save button. Target-action is a powerful mechanism for defining interaction and sending information between different parts of your app.

Delegation

Delegation is a simple and powerful pattern in which one object in an app acts on behalf of, or in coordination with, another object. The delegating object keeps a reference to the other object—the delegate—and at the appropriate time, the delegating object sends a message to the delegate. The message informs the delegate of an event that the delegating object is about to handle or has just handled. The delegate may respond to the message by updating the appearance (or state) of itself or of other objects in the app, and in some cases it will return a value that affects how an impending event is handled.



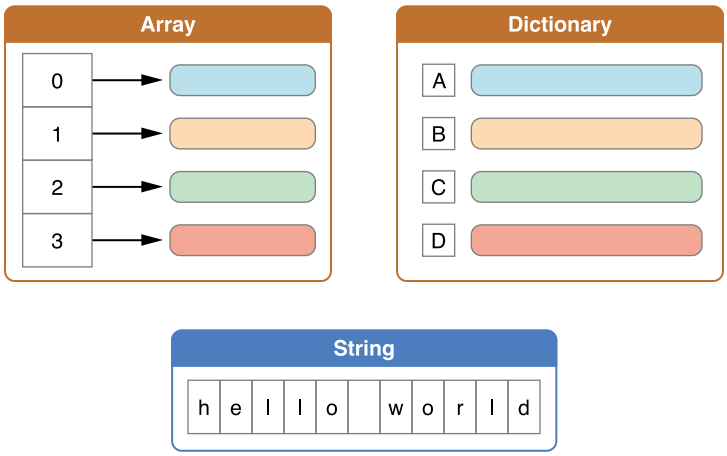
The delegate pattern is prevalent in existing framework classes, but you can also implement delegation between two custom objects in an app. A common design uses delegation to allow a child view controller to communicate a value (typically a user-entered value) to its parent view controller.

You haven't worked with delegation yet, but in Tutorial: Add Data (page 126), you'll see an example of it when you add additional behavior to your `ToDoListTableViewController` class.

These are a few of the most common design patterns that you'll encounter during iOS development, but there are many more. As you learn more about Objective-C, you'll spot other design patterns that you can apply in your app.

Working with Foundation

As you begin writing Objective-C code for your app, you'll find many frameworks that you can take advantage of. Of particular importance is the *Foundation framework*, which provides basic services for all apps. The Foundation framework includes **value classes** representing basic data types such as strings and numbers, as well as **collection classes** for storing other objects. You'll be relying on value and collection classes to write much of the code for your ToDoList app.



Value Objects

The Foundation framework provides classes that generate value objects for strings, binary data, dates and times, numbers, and other values.

A *value object* is an object that encapsulates a primitive value (of a C data type) and provides services related to that value. You frequently encounter value objects as the parameters and return values of methods and functions that your app calls. Different parts of a framework—or even different frameworks—can exchange data by passing value objects.

Some examples of value objects in the Foundation framework are:

- `NSString` and `NSMutableString`
- `NSData` and `NSMutableData`
- `NSDate`
- `NSNumber`

`NSValue`

Because value objects represent scalar values, you can use them in collections and wherever else objects are required. Value objects have an advantage over the primitive types they encapsulate: They let you perform certain operations on the encapsulated value simply and efficiently. The `NSString` class, for example, has methods for searching for and replacing substrings, for writing strings to files or (preferably) URLs, and for constructing file-system paths.

You create a value object from data of a primitive type. The `NSNumber` class provides an example of this approach.

```
int n = 5; // Value assigned to primitive type
NSNumber *numberObject = [NSNumber numberWithInt:n]; // Value object created from
 primitive type
```

Later, you can access the encapsulated data from the object.

```
int y = [numberObject intValue]; // Encapsulated value obtained from value object
 (y == n)
```

Most value classes create their instances by declaring both initializers and class factory methods. *Class factory methods*—implemented by a class as a convenience for clients—combine allocation and initialization in one step and return the created object. For example, the `NSString` class declares a `string` class method that allocates and initializes a new instance of the class and returns it to your code.

```
NSString *string = [NSString string];
```

In addition to creating value objects and letting you access their encapsulated values, most value classes provide methods for simple operations such as object comparison.

Strings

Objective-C and C support the same conventions for specifying strings: Single characters are enclosed by single quotes, and strings of characters are surrounded by double quotes. But Objective-C frameworks typically don't use C strings. Instead, they use `NSString` objects.

The NSString class provides an object wrapper for strings, offering advantages such as built-in memory management for storing arbitrary-length strings, support for different character encodings (particularly Unicode), and utilities for string formatting. Because you commonly use such strings, Objective-C provides a shorthand notation for creating NSString objects from constant values. To use this NSString literal, just precede a double-quoted string with the at sign (@), as shown in the following examples:

```
// Create the string "My String" plus newline.
NSString *myString = @"My String\n";
// Create the formatted string "1 String".
NSString *anotherString = [NSString stringWithFormat:@"%d %@", 1, @"String"];
// Create an Objective-C string from a C string.
NSString *fromCString = [NSString stringWithCString:"A C string"
encoding:NSUTF8StringEncoding];
```

Numbers

Objective-C offers a shorthand notation for creating NSNumber objects, removing the need to call initializers or class factory methods to create such objects. Simply precede the numeric value with the at sign (@) and optionally follow it with a value type indicator. For example, you can create NSNumber objects encapsulating an integer value and a double value:

```
NSNumber *myIntValue    = @32;
NSNumber *myDoubleValue = @3.22346432;
```

You can even use NSNumber literals to create encapsulated Boolean and character values.

```
NSNumber *myBoolValue = @YES;
NSNumber *myCharValue = @'V';
```

You can create NSNumber objects representing unsigned integers, long integers, long long integers, and float values by appending the letters U, L, LL, and F, respectively, to the notated value. For example, to create an NSNumber object encapsulating a float value, you can write:

```
NSNumber *myFloatValue = @3.2F;
```

Collection Objects

Most **collection objects** in Objective-C code are instances of a basic collection class—`NSArray`, `NSSet`, and `NSDictionary`. Collection classes are used to manage groups of objects, so any item you want to add to a collection must be an instance of an Objective-C class. If you need to add a scalar value, you must first create a suitable `NSNumber` or `NSValue` instance to represent it.

Any object you add to a collection will be kept alive at least as long as the collection is kept alive. That's because collection classes use strong references to keep track of their contents. In addition to keeping track of their contents, each collection class makes it easy to perform certain tasks, such as enumeration, accessing specific items, or finding out whether a particular object is part of the collection.

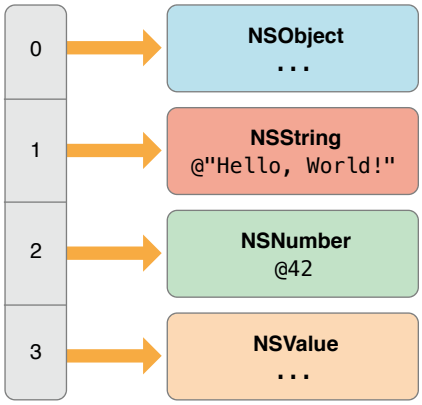
The contents of the `NSArray`, `NSSet`, and `NSDictionary` classes are set at creation. Because their contents can't be changed over time, these classes are called *immutable*. Each one also has a subclass that's *mutable* to allow you to add or remove objects at will. Different types of collections organize their contained objects in distinctive ways:

- `NSArray` and `NSMutableArray`. An array is an ordered collection of objects. You access an object by specifying its position (that is, its index) in the array. The first element in an array is at index 0 (zero).
- `NSSet` and `NSMutableSet`. A set stores an unordered collection of objects, with each object occurring only once. You generally access objects in the set by applying tests or filters to objects in the set.
- `NSDictionary` and `NSMutableDictionary`. A dictionary stores its entries as key-value pairs; the key is a unique identifier, usually a string, and the value is the object you want to store. You access this object by specifying the key.

Arrays

An **array** (`NSArray`) represents an ordered collection of objects. The only requirement is that each item be an Objective-C object—there's no requirement for each object to be an instance of the same class.

To maintain order in the array, each element is stored at a zero-based index.



Creating Arrays

You create an array through initialization or class factory methods. A variety of different initialization and factory methods are available, depending on the number of objects.

```
+ (id)arrayWithObject:(id)anObject;
+ (id)arrayWithObjects:(id)firstObject, ...;
- (id)initWithObjects:(id)firstObject, ...;
```

Because the `arrayWithObjects:` and `initWithObjects:` methods both take a `nil`-terminated, variable number of arguments, you must include `nil` as the last value.

```
id firstObject = @"someString";
id secondObject = @"secondString";
id thirdObject = @"anotherString";
NSArray *someArray =
[NSArray arrayWithObjects:firstObject, secondObject, thirdObject, nil];
```

The preceding example creates an array with three objects. The first object, `firstObject`, will have an array index of `0`; the last object, `thirdObject`, will have an index of `2`.

It's possible to create an array literal using a compact syntax.

```
NSArray *someArray = @[firstObject, secondObject, thirdObject];
```

When using this syntax, don't terminate the list of objects with `nil`—in fact, `nil` is an invalid value. For example, you'll get an exception at runtime if you try to execute the following code:

```
id nilObject = nil;
NSArray *someArray = @[firstObject, nilObject];
// exception: "attempt to insert nil object"
```

Querying Array Objects

After you've created an array, you can query it for information—such as how many objects it has or whether it contains a given item.

```
NSUInteger numberOfItems = [someArray count];
```

```
if ([someArray containsObject:secondObject]) {
    ...
}
```

You can also query the array for an item at a given index. If you attempt to request an invalid index, you'll get an out-of-bounds exception at runtime. To avoid getting an exception, always check the number of items first.

```
if ([someArray count] > 0) {
    NSLog(@"First item is: %@", [someArray objectAtIndex:0]);
}
```

This example checks to see whether the number of items is greater than zero. If it is, the Foundation function `NSLog` logs a description of the first item, which has an index of `0`.

As an alternative to using `objectAtIndex:`, query the array using a subscript syntax, which is just like accessing a value in a standard C array. The previous example can be rewritten like this:

```
if ([someArray count] > 0) {
    NSLog(@"First item is: %@", someArray[0]);
}
```

Sorting Array Objects

The `NSArray` class offers a variety of methods to sort its collected objects. Because `NSArray` is immutable, each method returns a new array containing the items in the sorted order.

For example, you can sort an array of strings by calling `compare:` on each string.

```
NSArray *unsortedStrings = @[@"gamma", @"alpha", @"beta"];
NSArray *sortedStrings =
        [unsortedStrings sortedArrayUsingSelector:@selector(compare:)];
```

Array Mutability

Although the `NSArray` class itself is immutable, it can nevertheless contain mutable objects. For example, if you create an immutable array that contains a mutable string, like this:

```
NSMutableString *mutableString = [NSMutableString stringWithString:@"Hello"];
NSArray *immutableArray = @[mutableString];
```

there's nothing to stop you from mutating the string.

```
if ([immutableArray count] > 0) {
    id string = immutableArray[0];
    if ([string isKindOfClass:[NSMutableString class]]) {
        [string appendString:@" World!"];
    }
}
```

If you want to add or remove objects from an array after initial creation, use `NSMutableArray`, which adds a variety of methods to add, remove, or replace one or more objects.

```
NSMutableArray *mutableArray = [NSMutableArray array];
[mutableArray addObject:@"gamma"];
[mutableArray addObject:@"alpha"];
[mutableArray addObject:@"beta"];

[mutableArray replaceObjectAtIndex:0 withObject:@"epsilon"];
```

This example creates an array made up of the objects `@"epsilon"`, `@"alpha"`, and `@"beta"`.

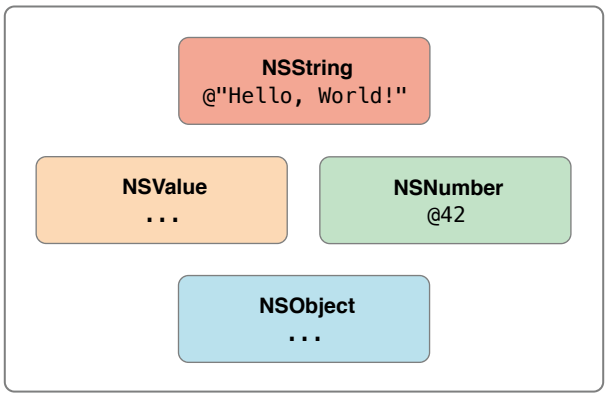
It's also possible to sort a mutable array in place, without creating a secondary array.

```
[mutableArray sortUsingSelector:@selector(caseInsensitiveCompare:)];
```

In this case, the contained items are sorted into the ascending, case-insensitive order `@"alpha"`, `@"beta"`, and `@"epsilon"`.

Sets

A *set* (`NSSet`) object is similar to an array, but it maintains an unordered group of distinct objects.



Because sets don't maintain order, they offer faster performance than arrays do when it comes to testing for membership.

Because the basic `NSSet` class is immutable, its contents must be specified at creation, using either an initializer or a class factory method.

```
NSSet *simpleSet =
  [NSSet setWithObjects:@"Hello, World!", @42, aValue, anObject, nil];
```

As with `NSArray`, the `initWithObjects:` and `setWithObjects:` methods both take a `nil`-terminated, variable number of arguments. The name of the mutable `NSSet` subclass is `NSMutableSet`.

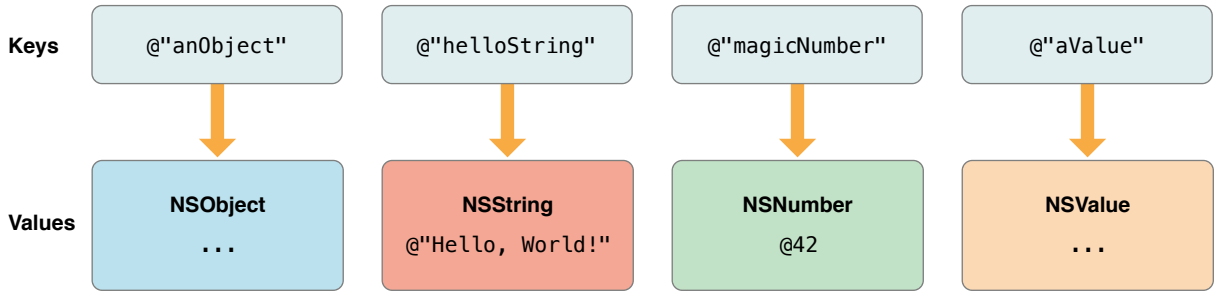
Sets store only one reference to an individual object, even if you try adding an object more than once.

```
    NSNumber *number = @42;
    NSSet *numberSet =
      [NSSet setWithObjects:number, number, number, number, nil];
    // numberSet only contains one object
```

Dictionaries

Rather than simply maintaining an ordered or unordered collection of objects, a *dictionary* (`NSDictionary`) stores objects associated with given keys, which can then be used for retrieval.

The best practice is to use string objects as dictionary keys.



Although you can use other objects as keys, keep in mind that each key is copied for use by a dictionary and so must support `NSCopying`. If you want to use key-value coding, however, you must use string keys for dictionary objects. (To learn more, see *Key-Value Coding Programming Guide*).

Creating Dictionaries

You can create dictionaries through initialization, or you can use class factory methods, like this:

```
NSDictionary *dictionary = [NSDictionary dictionaryWithObjectsAndKeys:
                anObject, @"anObject",
          @"Hello, World!", @"helloString",
                    @42, @"magicNumber",
                someValue, @"aValue",
                       nil];
```

For the `dictionaryWithObjectsAndKeys:` and `initWithObjectsAndKeys:` methods, each object is specified before its key, and the list of objects and keys must be `nil`-terminated.

Objective-C offers a concise syntax for dictionary literal creation.

```
NSDictionary *dictionary = @{
              @"anObject" : anObject,
           @"helloString" : @"Hello, World!",
           @"magicNumber" : @42,
                @"aValue" : someValue
};
```

For dictionary literals, the key is specified before its object, and the list of objects and keys is not `nil`-terminated.

Querying Dictionaries

After you've created a dictionary, you can ask it for the object stored against a given key.

```
NSNumber *storedNumber = [dictionary objectForKey:@"magicNumber"];
```

If the object isn't found, the `objectForKey:` method returns `nil`.

There's also a subscript syntax alternative to using `objectForKey:`.

```
NSNumber *storedNumber = dictionary[@"magicNumber"];
```

Dictionary Mutability

If you need to add or remove objects from a dictionary after creation, use the `NSMutableDictionary` subclass.

```
[dictionary setObject:@"another string" forKey:@"secondString"];
[dictionary removeObjectForKey:@"anObject"];
```

Representing nil with NSNull

It's not possible to add `nil` to the collection classes described in this section because `nil` in Objective-C means "no object." If you need to represent "no object" in a collection, use the `NSNull` class.

```
NSArray *array = @[ @"string", @42, [NSNull null] ];
```

With `NSNull`, the `null` method always returns the same instance. Classes that behave in this way are called *singleton classes*. You can check to see whether an object in an array is equal to the shared `NSNull` instance like this:

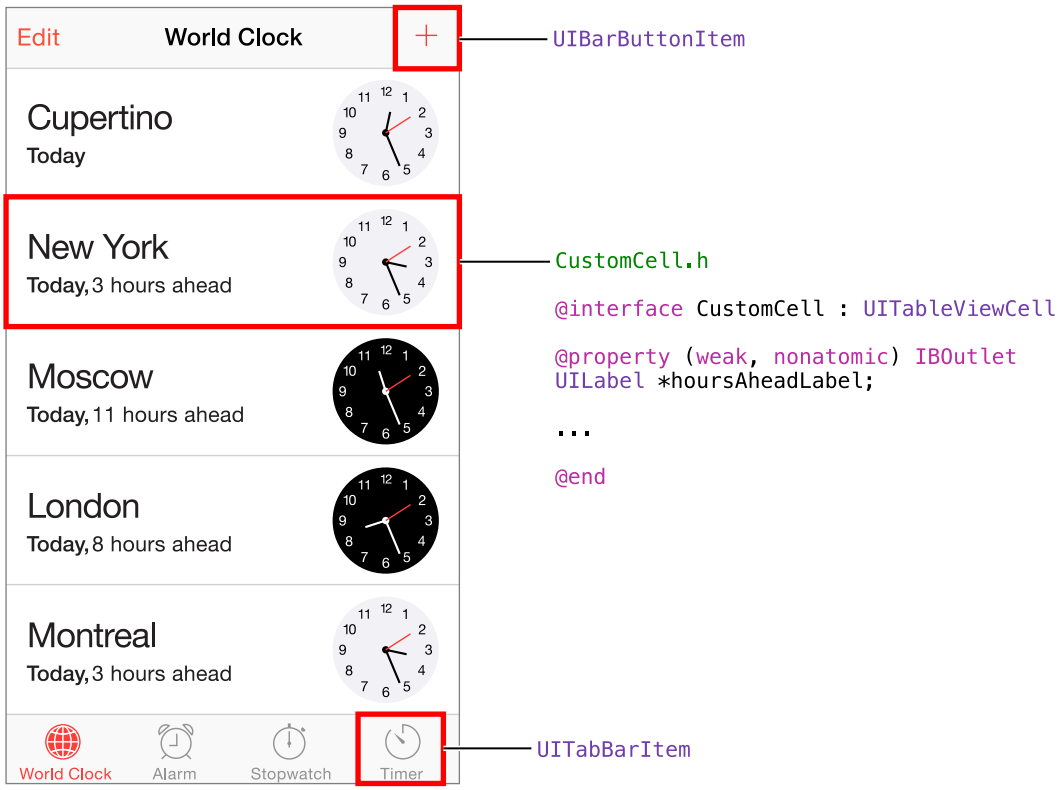
```
for (id object in array) {
    if (object == [NSNull null]) {
        NSLog(@"Found a null object");
    }
}
```

Although the Foundation framework contains many more capabilities than are described here, you don't need to know every single detail right away. If you do want to learn more about Foundation, take a look at *Foundation Framework Reference*. For now, you have enough information to continue implementing your ToDoList app, which you'll do by writing a custom data class.

Writing a Custom Class

As you develop iOS apps, you'll find many occasions when you need to write your own custom classes. Custom classes are useful when you need to package custom behavior together with data. In a custom class, you can define your own behaviors for storing, manipulating, and displaying your data.

For example, consider the World Clock tab in the iOS Clock app. The cells in this table view need to display more content than a standard table view cell. This is a good opportunity to implement a subclass that extends the behavior of `UITableViewCell` to let you display additional custom data for a given table view cell. If you were designing this custom class, you might add outlets for a label to display the hours ahead information and an image view to display the custom clock on the right of the cell.



This chapter teaches you what you need to know about Objective-C syntax and class structure to finish implementing the behavior of your ToDoList app. It discusses the design of `ToDoItem`, the custom class that will represent a single item on your to-do list. In the third tutorial, you'll actually implement this class and add it to your app.

Declaring and Implementing a Class

The specification of a class in Objective-C requires two distinct pieces: the interface and the implementation. The **interface** specifies exactly how a given type of object is intended to be used by other objects. In other words, it defines the public interface between instances of the class and the outside world. The **implementation** includes the executable code for each method declared in the interface.

An object should be designed to hide the details of its internal implementation. In Objective-C, the interface and implementation are usually placed in separate files so that you need to make only the interface public. As with C code, you define header files and source files to separate public declarations from the implementation details of your code. Interface files have a `.h` extension, and implementation files have a `.m` extension. (You'll actually create these files for the `ToDoItem` class in Tutorial: Add Data (page 126)—for now, just follow along as all of the pieces are introduced.)

Interface

The Objective-C syntax used to declare a class interface looks like this:

```
@interface ToDoItem : NSObject


@end
```

This example declares a class named `ToDoItem`, which inherits from `NSObject`.

The public properties and behavior are defined inside the `@interface` declaration. In this example, nothing is specified beyond the superclass, so the only behavior expected to be available on instances of `ToDoItem` is the behavior inherited from `NSObject`. All objects are expected to have a minimum behavior, so by default, they must inherit from `NSObject` (or one of its subclasses).

Implementation

The Objective-C syntax used to declare a class implementation looks like this:

```
#import "ToDoItem.h"


@implementation ToDoItem


@end
```

If you declare any methods in the class interface, you'll need to implement them inside this file.

Properties Store an Object's Data

Consider what information the to-do item needs to hold. You probably need to know its name, when it was created, and whether it's been completed. In your custom `ToDoItem` class, you'll store this information in **properties**.

Declarations for these properties reside inside the interface file (`ToDoItem.h`). Here's what they look like:

```
@interface ToDoItem : NSObject

@property NSString *itemName;
@property BOOL completed;
@property NSDate *creationDate;

@end
```

In this example, the `ToDoItem` class declares three public properties. Unless you specify otherwise, other objects can both read and change the values of the properties.

To declare a property read-only, you specify that in a **property attribute**. For example, if you don't want the creation date of `ToDoItem` to be changeable, you might update the `ToDoItem` class interface to look like this:

```
@interface ToDoItem : NSObject

@property NSString *itemName;
@property BOOL completed;
@property (readonly) NSDate *creationDate;

@end
```

Properties can be private or public. Sometimes it makes sense to make a property private so that other classes can't see or access it. For example, if you want to keep track of a property that represents the date an item was marked as completed without giving other classes access to this information, you make the property private by putting it in a *class extension* that you add after the `#import` statement the top of your implementation file (`ToDoItem.m`), as shown here.

```
#import "ToDoItem.h"

@interface ToDoItem ()
@property NSDate *completionDate;
@end

@implementation ToDoItem

@end
```

You access properties using getters and setters. A *getter* returns the value of a property, and a *setter* changes it. A common syntactical shorthand for accessing getters and setters is **dot notation**. For properties with read and write access, you can use dot notation for both getting and setting a property's value. If you have an object `toDoItem` of class `ToDoItem`, you can do the following:

```
toDoItem.itemName = @"Buy milk";                //Sets the value of itemName
NSString *selectedItemName = toDoItem.itemName;  //Gets the value of itemName
```

Methods Define an Object's Behavior

Methods define what an object can do. A *method* is a piece of code that you define to perform a task or subroutine in a class. Methods have access to data stored in the class and can use that information to perform an operation.

For example, to give a to-do item (`ToDoItem`) the ability to get marked as complete, you can add a `markAsCompleted` method to the class interface. Later, you'll implement this method's behavior in the class implementation, as described in Implementing Methods (page 124).

```
@interface ToDoItem : NSObject

@property NSString *itemName;
@property BOOL completed;
@property (readonly) NSDate *creationDate;
- (void)markAsCompleted;
```

```
@end
```

The minus sign (–) at the front of the method name indicates that it is an *instance method*, which can be called on an object of that class. This minus sign differentiates it from class methods, which are denoted with a plus sign (+). *Class methods* can be called on the class itself. A common example of class methods are class factory methods, which you learned about in Working with Foundation (page 108). You can also use class methods to access a piece of shared information associated with the class.

The `void` keyword is used inside parentheses at the beginning of the declaration to indicate that the method doesn't return a value. In this case, the `markAsCompleted` method takes in no parameters. Parameters are discussed in more detail in Method Parameters (page 123).

Method Parameters

You declare methods with *parameters* to pass along a piece of information when you call a method.

For example, you can revise the `markAsCompleted` method from the previous code snippet to take in a single parameter that determines whether the item is marked completed or uncompleted. This way, you can toggle the completion state of the item instead of setting it only as completed.

```
@interface ToDoItem : NSObject

@property NSString *itemName;
@property BOOL completed;
@property (readonly) NSDate *creationDate;
- (void)markAsCompleted:(BOOL)isComplete;

@end
```

Now, your method takes in one parameter, `isComplete`, which is of type `BOOL`.

When you refer to a method with a parameter by name, you include the colon as part of the method name, so the name of the updated method is now `markAsCompleted:`. If a method has multiple parameters, the method name is broken up and interspersed with the parameter names. If you wanted to add another parameter to this method, its declaration would look like this:

```
- (void)markAsCompleted:(BOOL)isComplete onDate:(NSDate *)date;
```

Here, the method's name is written as `markAsCompleted:onDate:`. The names `isComplete` and `date` are used in the implementation to access the values supplied when the method is called, as if these names were variables.

Implementing Methods

Method implementations use braces to contain the relevant code. The name of the method must be identical to its counterpart in the interface file, and the parameter and return types must match exactly.

Here is a simple implementation of the `markAsCompleted:` method discussed earlier:

```
@implementation ToDoItem
- (void)markAsCompleted:(BOOL)isComplete {
    self.completed = isComplete;
}
@end
```

Like properties, methods can be private or public. Public methods are declared in the public interface and so can be seen and called by other objects. Their corresponding implementation resides in the implementation file and can't be seen by other objects. Private methods have only an implementation and are internal to the class, meaning they're only available to call inside the class implementation. This is a powerful mechanism for adding internal behavior to a class without allowing other objects access to it.

For example, say you want to keep a to-do item's `completionDate` updated. If the to-do item gets marked as completed, set `completionDate` to the current date. If it gets marked as uncompleted, set `completionDate` to `nil`, because it hasn't been completed yet. Because updating the to-do item's `completionDate` is a self-contained task, the best practice is to write its own method for it. However, it's important to make sure that other objects can't call this method—otherwise, another object could set the to-do item's `completionDate` to anything at any time. For this reason, you make this method private.

To accomplish this, you update the implementation of `ToDoItem` to include the private method `setCompletionDate` that gets called inside `markAsCompleted:` to update the to-do item's `completionDate` whenever it gets marked as completed or uncompleted. Notice that you're not adding anything to the interface file, because you don't want other objects to see this method.

```
@implementation ToDoItem
- (void)markAsCompleted:(BOOL)isComplete {
    self.completed = isComplete;
    [self setCompletionDate];
```

```
}
- (void)setCompletionDate {
    if (self.completed) {
        self.completionDate = [NSDate date];
    } else {
        self.completionDate = nil;
    }
}
@end
```

At this point, you've designed a basic representation of a to-do list item using the `ToDoItem` class. `ToDoItem` stores information about itself—name, creation date, completion state—in the form of properties, and it defines what it can do—get marked as completed or uncompleted—using a method. Now it's time to move on to the next tutorial and add this class to your app.

Tutorial: Add Data

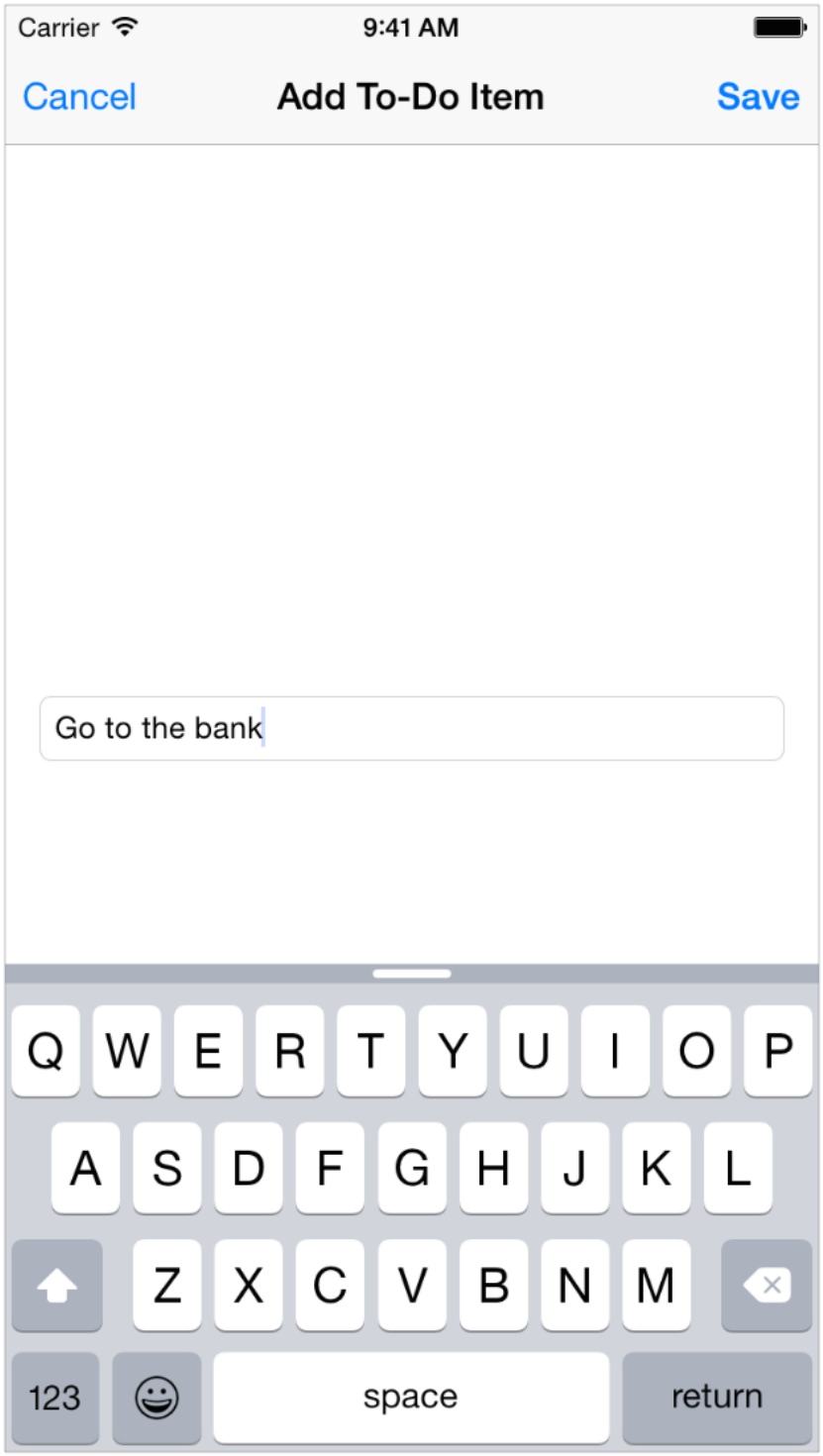
SwiftObjective-C

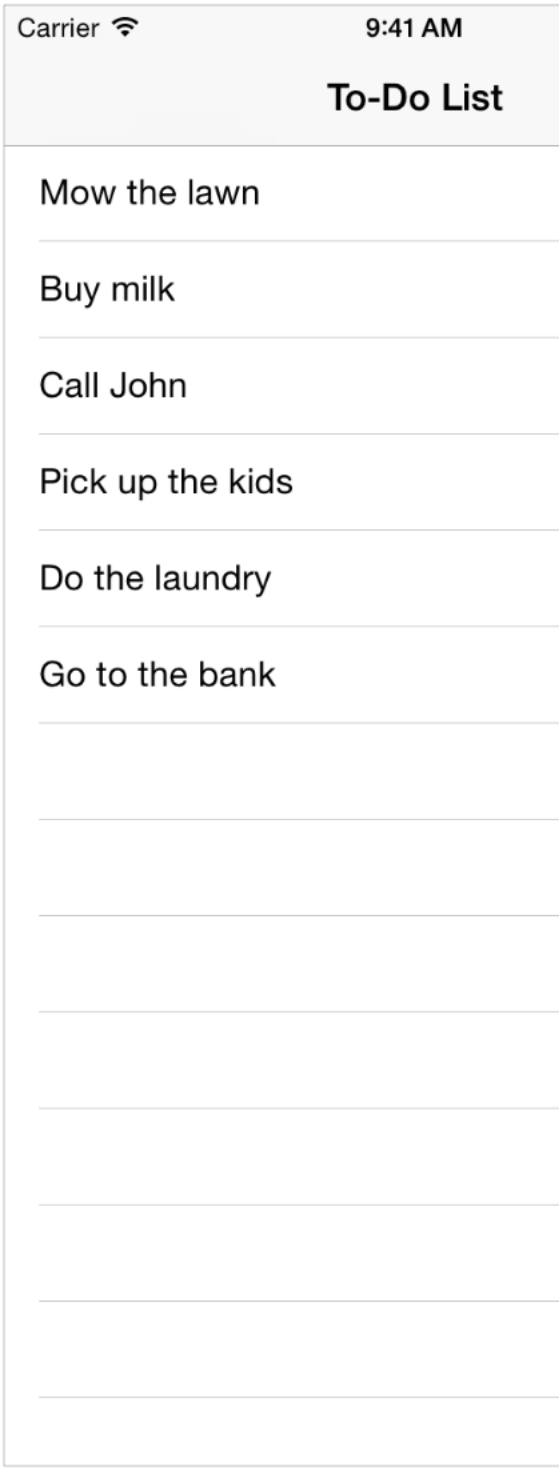
This tutorial builds on the project you created in the second tutorial (Tutorial: Storyboards (page 60)). You'll apply what you learned about using design patterns, working with Foundation, and writing a custom class to add support for dynamic data to your ToDoList app.

This tutorial teaches you how to:

- Work with common Foundation classes
- Create custom data classes
- Implement a data source and delegate protocol
- Pass data between view controllers

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





Create a Data Class

To get started, open your existing project in Xcode.

At this point, you have an interface and a navigation scheme for your ToDoList app using storyboards. Now, it's time to add data storage and behavior with model objects.

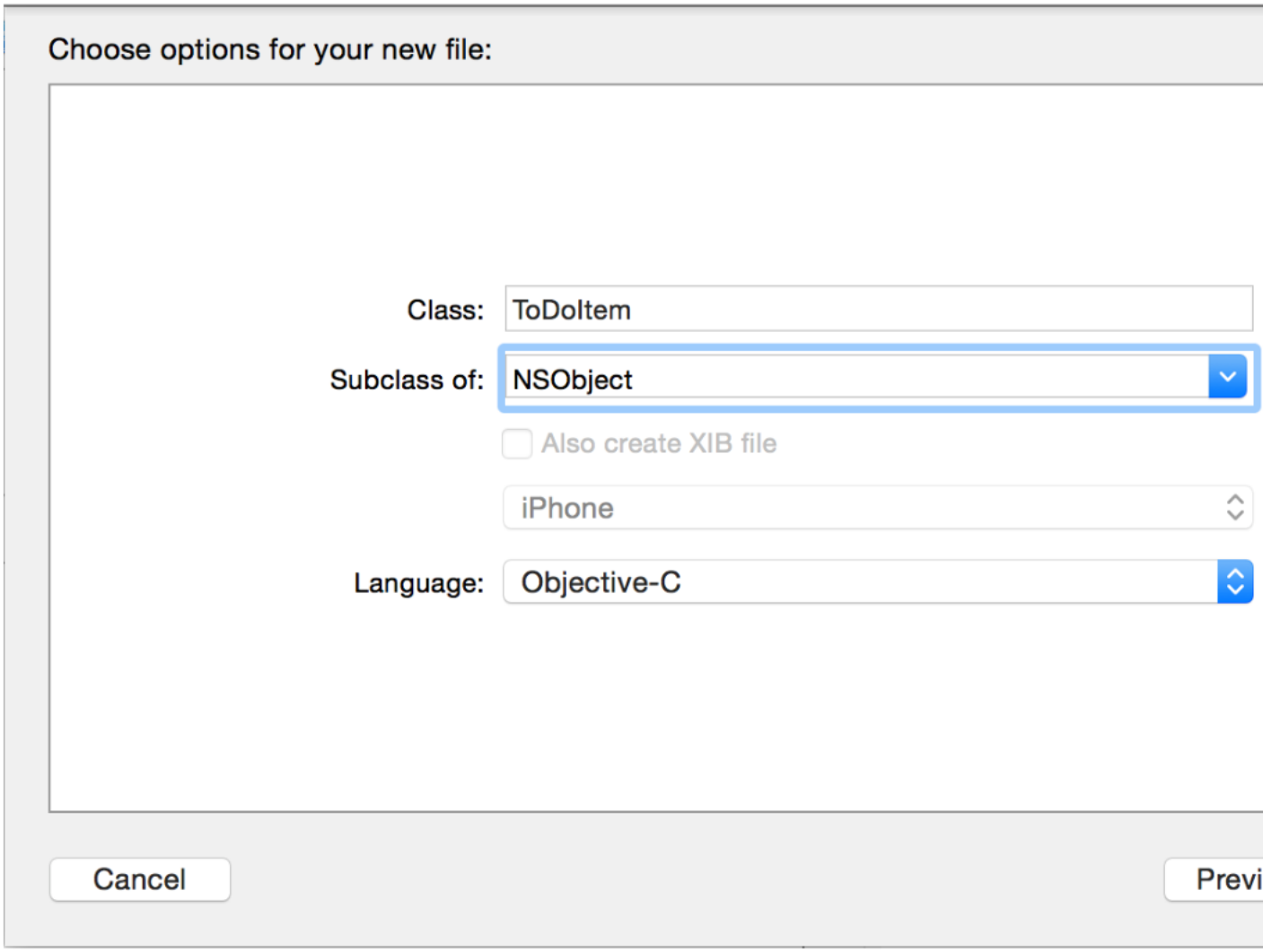
The goal of your app is to create a list of to-do items, so first you'll create a custom class, `ToDoItem`, to represent an individual to-do item. As you recall, the `ToDoItem` class was discussed in Writing a Custom Class (page 119).

To create the ToDoItem class

1. Choose File > New > File (or press Command-N).

   A dialog appears that prompts you to choose a template for your new file.
2. On the left, select Source under iOS.
3. Select Cocoa Touch Class, and click Next.
4. In the Class field, type `ToDoItem`.

5. Choose NSObject from the “Subclass of” pop-up menu.



6. Click Next.

   The save location defaults to your project directory.

   The Group option defaults to your app name, ToDoList.

   In the Targets section, your app is selected and the tests for your app are unselected.

7. Leave these defaults as they are, and click Create.

The `ToDoItem` class is straightforward to implement. It has properties for its name, creation date, and completion status. Go ahead and add these properties to the `ToDoItem` class interface.

To configure the ToDoItem class

1. In the project navigator, select `ToDoItem.h`.

2. Add the following properties to the interface so that the declaration looks like this:

```
@interface ToDoItem : NSObject

@property NSString *itemName;
@property BOOL completed;
@property (readonly) NSDate *creationDate;

@end
```

Checkpoint: Build your project by choosing Product > Build (or pressing Command-B). You're not using your new class for anything yet, but building it gives the compiler a chance to verify that you haven't made any typing mistakes. If you have, fix them by reading through the warnings or errors that the compiler provides, and then look back over the instructions in this tutorial to make sure everything looks the way it's described here.

Load the Data

You now have a class from which you can create and store the data for individual to-do items. You also need to keep a list of those items. The natural place to track this is in the `ToDoListTableViewController` class—view controllers are responsible for coordinating between the model and the view, so they need a reference to the model.

The Foundation framework includes a class, `NSMutableArray`, that works well for tracking lists of items. It's important to use a mutable array so that you can add items to the array. The immutable version, `NSArray`, doesn't let you add items to it after it's initialized.

To use an array you need to both declare it and create it. You do this by allocating and initializing the array.

To allocate and initialize the array

1. In the project navigator, select `ToDoListTableViewController.m`.

   Because the array of items is an implementation detail of your table view controller, you declare it in the `.m` file instead of the `.h` file. This makes it private to your custom class.

2. Add the following property to the interface category Xcode created in your custom table view controller class. The declaration should look like this:

```
@interface ToDoListTableViewController ()


@property NSMutableArray *toDoItems;


@end
```

3. Find the `viewDidLoad` method. The template implementation looks like this:

```
- (void)viewDidLoad {
    [super viewDidLoad];


    // Uncomment the following line to preserve selection between
presentations.
    // self.clearsSelectionOnViewWillAppear = NO;


    // Uncomment the following line to display an Edit button in the
navigation bar for this view controller.
    // self.navigationItem.rightBarButtonItem = self.editButtonItem;
}
```

The template implementation of this method includes comments that were inserted by Xcode when it created `ToDoListTableViewController`. Code comments like this provide helpful hints and contextual information in source code files, but you don't need them for this tutorial. Feel free to delete the comments.

4. Update the method implementation to allocate and initialize the `toDoItems` array in the `viewDidLoad` method:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.toDoItems = [[NSMutableArray alloc] init];
}
```

At this point, you have an array that you can add items to. You'll add a few initial items in a separate method, `loadInitialData`, which you'll call from `viewDidLoad`. This code goes in its own method because it's a modular task, and you can improve code readability by making this method separate. In a real app, this method would load the data from a persistent store, such as a file. For now, you'll create some test data to experiment with.

Create an item as you created the array: Allocate and initialize. Then, give the item a name. This name will be shown in the table view. Do this for a couple of items.

To load initial data

1. In `ToDoListTableViewController.m`, add a new method called `loadInitialData` below the `@implementation` line.

```
- (void)loadInitialData {


}
```

2. In this method, create a few list items, and add them to the array.

```
- (void)loadInitialData {
    ToDoItem *item1 = [[ToDoItem alloc] init];
    item1.itemName = @"Buy milk";
    [self.toDoItems addObject:item1];
    ToDoItem *item2 = [[ToDoItem alloc] init];
    item2.itemName = @"Buy eggs";
    [self.toDoItems addObject:item2];
    ToDoItem *item3 = [[ToDoItem alloc] init];
    item3.itemName = @"Read a book";
    [self.toDoItems addObject:item3];
}
```

3. Update the implementation of the `viewDidLoad` method to call the `loadInitialData` method.

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.toDoItems = [[NSMutableArray alloc] init];
    [self loadInitialData];
}
```

Checkpoint: Build your project by choosing Product > Build. You should see numerous errors for the lines of your `loadInitialData` method. The key to what's gone wrong is the first line, which should say "Use of undeclared identifier ToDoItem." This means that the compiler doesn't know about your `ToDoItem` class when it's compiling `ToDoListTableViewController`. Compilers are very particular and need to be told explicitly what to pay attention to.

To tell the compiler to pay attention to your ToDoItem class

1. Find the `#import "ToDoListTableViewController.h"` line near the top of the `ToDoListTableViewController.m` file.
2. Add the following line immediately below it:

```
#import "ToDoItem.h"
```

Checkpoint: Build your project by choosing Product > Build. It should build without errors.

Display the Data

At this point, your custom table view controller subclass, `ToDoListTableViewController`, has a mutable array that's prepopulated with some sample to-do items. Now you need to display the data in the table view that's managed by this class.

To display dynamic data, a table view needs two important helpers: a data source and a delegate. A table view *data source*, as implied by its name, supplies the table view with the data it needs to display. A table view *delegate* helps the table view manage cell selection, row heights, and other aspects related to displaying the data. By default, `UITableViewController`—and any of its subclasses—adopts the necessary protocols to make the table view controller both a data source (`UITableViewDataSource` protocol) and a delegate (`UITableViewDelegate` protocol) for its associated table view. Your job is to implement the appropriate protocol methods in your table view controller subclass so that your table view has the correct behavior.

A functioning table view requires three table view data source methods. The first of these is `numberOfSectionsInTableView:`, which tells the table view how many sections to display. Sections provide a way of visually grouping cells within table views, especially table views with a lot of data. For a simple table view like the one in the ToDoList app, you just need the table view to display a single section, so the implementation of the `numberOfSectionsInTableView:` data source method is straightforward.

To display a section in your table view

1. In the project navigator, select `ToDoListTableViewController.m`.

2. Find the `numberOfSectionsInTableView:` data source method. The template implementation looks like this:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
#warning Potentially incomplete method implementation.
    // Return the number of sections.
    return 0;
}
```

3. You want a single section, so change the return value from 0 to 1, and remove the warning line.

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    // Return the number of sections.
    return 1;
}
```

   You removed the warning line that says "Potentially incomplete method implementation" because you've completed the implementation.

The next data source method, `tableView:numberOfRowsInSection:`, tells the table view how many rows to display in a given section. You have a single section in your table, and each to-do item should have its own row in that section. That means that the number of rows should be the number of `ToDoItem` objects in your `toDoItems` array.

To return the number of rows in your table view

1. In `ToDoListTableViewController.m`, find the `tableView:numberOfRowsInSection:` data source method. The template implementation looks like this:

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
#warning Incomplete method implementation.
    // Return the number of rows in the section.
    return 0;
}
```

   You want to return the number of to-do items you have. Fortunately, `NSArray` has a handy method called `count` that returns the number of items in the array, so the number of rows is `[self.toDoItems count]`.

2. Change the `tableView:numberOfRowsInSection:` data source method to return the appropriate number of rows, and remove the warning line.

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    // Return the number of rows in the section.
    return [self.toDoItems count];
}
```

The last data source method, `tableView:cellForRowAtIndexPath:`, asks for a cell to display for a given row. Each row in a table view has one cell, and that cell determines the content that appears in that row and how that content is laid out.

Until now, you've worked with code only, but the cell to display for a row is very much part of your interface. Fortunately, Xcode makes it easy to design custom cells in your storyboard. The first task is to tell the table view that instead of using static content, it's going to be using prototype cells with dynamic content.

To configure your table view to use prototype cells

1. In the project navigator, select `Main.storyboard`.
2. In the outline view, select Table View under To-Do List Scene.
3. With the table view selected, open the Attributes inspector  in the utility area.
4. In the Attributes inspector, change the table view's Content field from Static Cells to Dynamic Prototypes.

   Interface Builder takes the static cells you configured and converts them all into prototypes.

Prototype cells, as the name implies, are cells that are configured with text styles, colors, images, or other attributes as you want them to be displayed but that get their data from the table view data source at runtime. The data source will load a prototype cell for each row and then configure that cell to display the data for the row.

To load the correct cell, the data source needs to know what it's called, and that name must also be configured in the storyboard.

While you're setting the prototype cell name, you'll also configure another property—the cell selection style, which determines a cell's appearance when a user taps it. You'll set the cell selection style so that the cell won't be highlighted when a user taps it. This is the behavior you want your cells to have when a user taps an item in the to-do list to mark it as completed or uncompleted—a feature you'll implement later in this tutorial.

To configure the prototype cell

1. Select the first table view cell in your table view.
2. In the Attributes inspector, locate the Identifier field and type `ListPrototypeCell`.
3. In the Attributes inspector, change the cell's Selection field from Default to None.
4. In the outline view, select every cell besides the first one and delete them.

   You only need one prototype cell from this point.

You could also change the font or other attributes of the prototype cell. The basic configuration is easy to work with, so you'll keep that.

The next step is to teach your data source how to configure the cell for a given row by implementing `tableView:cellForRowAtIndexPath:`. For table views with a small number of rows, all rows may be onscreen at once, so this method gets called for each row in your table. But table views with a large number of rows display only a small fraction of their total items at a given time. It's most efficient for table views to only ask for the cells for rows that are being displayed, and that's what `tableView:cellForRowAtIndexPath:` allows the table view to do.

For any given row in the table view, you configure the cell by fetching the corresponding item in the `toDoItems` array and then setting the cell's text label to that item's name.

To display cells in your table view

1. In the project navigator, select `ToDoListTableViewController.m`.
2. Find and uncomment the `tableView:cellForRowAtIndexPath:` data source method. (To uncomment the method, remove the `/*` and `*/` characters surrounding it.)

   After you do that, the template implementation looks like this:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:@"reuseIdentifier"
forIndexPath:indexPath];


    // Configure the cell...


    return cell;

}
```

   The template performs several tasks. It asks the table view for a cell with a placeholder identifier, adds a comment about where code to configure the cell should go, and then returns the cell.

To make this code work for your app, you'll need to change the placeholder identifier to the one you set in the storyboard and then add code to configure the cell.

3. Change the placeholder identifier to the identifier you set in your storyboard. To avoid typos, copy and paste from the storyboard to the implementation file. The line of code that returns the cell should now look like this:

```
UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:@"ListPrototypeCell"
forIndexPath:indexPath];
```

4. Immediately before the return statement, add the following lines of code:

```
ToDoItem *toDoItem = [self.toDoItems objectAtIndex:indexPath.row];
cell.textLabel.text = toDoItem.itemName;
```

These lines fetch the appropriate item in the `toDoItems` array and set the name of that item to display in the cell.

Your `tableView:cellForRowAtIndexPath:` method should look like this:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:@"ListPrototypeCell" forIndexPath:indexPath];
    ToDoItem *toDoItem = [self.toDoItems objectAtIndex:indexPath.row];
    cell.textLabel.text = toDoItem.itemName;
    return cell;
}
```

Checkpoint: Run your app. The list of items you added in `loadInitialData` should show up as cells in your table view.

Toggle Item Completion State

A to-do list isn't much good if you can never mark items as completed. Now, you'll add support for that. Table views allow you to implement a simple interface where the completion state of an item toggles when the user taps a cell. Specifically, a table view notifies its delegate when the user taps a cell. When the delegate receives this notification, you'll write code to respond to it in one of two ways:

- If the item in that cell has not been completed, then the code marks it as completed and displays the completed item with a checkmark next to it.
- If the item in that cell has been completed, then the code marks it as uncompleted and removes the checkmark next to it.

All you have to do is implement the `tableView:didSelectRowAtIndexPath:` delegate method to respond to user taps and update your to-do list items accordingly.

To toggle an item as completed or uncompleted

1. In the project navigator, select `ToDoListTableViewController.m`.
2. Add the following lines to the end of the file, just above the `@end` line:

```
#pragma mark – Table view delegate


– (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {


}
```

   Try typing the second line instead of just copying and pasting. You'll find that **code completion** is one of the great time-saving features of Xcode. When Xcode brings up the list of potential completions, scroll through the list until you find the one you want and then press Return. Xcode inserts the whole line for you.

3. In this method, you want to respond to a tap but not actually leave the cell selected. Add the following code to deselect the cell immediately after selection:

```
[tableView deselectRowAtIndexPath:indexPath animated:NO];
```

4. Add this code line to search for the `ToDoItem` in your `toDoItems` array that corresponds to the cell that was tapped.

```
ToDoItem *tappedItem = [self.toDoItems objectAtIndex:indexPath.row];
```

5. Toggle the completion state of the tapped item.

```
tappedItem.completed = !tappedItem.completed;
```

6. Tell the table view to reload the row whose data you just updated.

```
[tableView reloadRowsAtIndexPaths:@[indexPath]
withRowAnimation:UITableViewRowAnimationNone];
```

Your `tableView:didSelectRowAtIndexPath:` method should look like this:

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
*)indexPath {
    [tableView deselectRowAtIndexPath:indexPath animated:NO];
    ToDoItem *tappedItem = [self.toDoItems objectAtIndex:indexPath.row];
    tappedItem.completed = !tappedItem.completed;
    [tableView reloadRowsAtIndexPaths:@[indexPath]
withRowAnimation:UITableViewRowAnimationNone];
}
```

Checkpoint: Run your app. The list of items you added in `loadInitialData` is visible as cells in your table view. But when you tap items, nothing seems to happen. Why not?

The reason is that you haven't configured the table view cell to display the completion state of an item. To do so, you need to go back to the `tableView:cellForRowAtIndexPath:` data source method and configure the cell to display a checkmark when an item is completed.

Table view cells can have a cell accessory on the right side. By default, there's no accessory selected; however, you can change the cell to display a checkmark.

To display an item's completion state

1. Go to the `tableView:cellForRowAtIndexPath:` method.
2. Add the following code just below the line that sets the text label of the cell:

```
if (toDoItem.completed) {
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
} else {
    cell.accessoryType = UITableViewCellAccessoryNone;
}
```

This code checks the completion state of a to-do item and sets the cell accessory based on that.

Your `tableView:cellForRowAtIndexPath:` method should now look like this:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:@"ListPrototypeCell" forIndexPath:indexPath];

    ToDoItem *toDoItem = [self.toDoItems objectAtIndex:indexPath.row];

    cell.textLabel.text = toDoItem.itemName;

    if (toDoItem.completed) {

        cell.accessoryType = UITableViewCellAccessoryCheckmark;

    } else {

        cell.accessoryType = UITableViewCellAccessoryNone;

    }

    return cell;

}
```

Checkpoint: Run your app. The list of items you added in `loadInitialData` is visible as cells in your table view. When you tap an item, a checkmark should appear next to it. If you tap the same item again, the checkmark disappears.

Add New Items

The final step in creating the to-do list app's functionality is implementing the ability to add an item. When a user enters an item name in the text field in the add-to-do-item scene and taps the Save button, you want `AddToDoItemViewController` to create a new to-do item and pass it back to `ToDoListTableViewController` to display in the to-do list.

First, you need to have a to-do item to configure. Give `AddToDoItemViewController` a property to hold information about a new to-do item.

To add a to-do item property to AddToDoItemViewController

1. In the project navigator, select `AddToDoItemViewController.h`.

   Because you'll need to access the to-do item from your table view controller later on, it's important to make the to-do item property public. That's why you declare it in the interface file, `AddToDoItemViewController.h`, instead of in the implementation file, `AddToDoItemViewController.m`.

2. In `AddToDoItemViewController.h`, add an import declaration to `ToDoItem.h` right below the `#import <UIKit/UIKit.h>` line.

```
#import "ToDoItem.h"
```

3. Add a `toDoItem` property to the interface.

```
@interface AddToDoItemViewController : UIViewController

@property ToDoItem *toDoItem;

@end
```

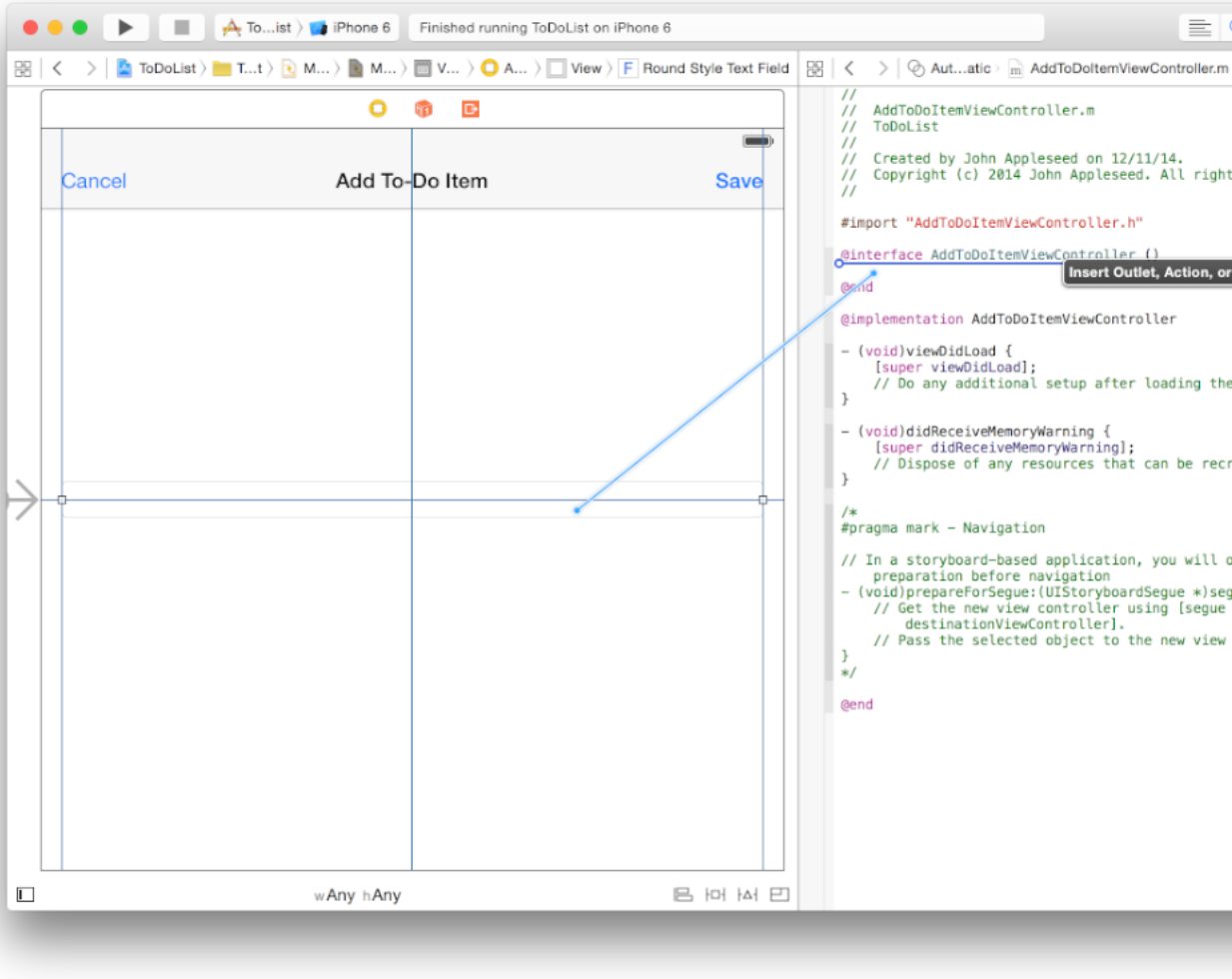
To get the name of the new item, `AddToDoItemViewController` needs access to the contents of the text field where the user enters the name. To do this, create a connection from `AddToDoItemViewController.m` to the text field in your storyboard.

To connect the text field to the AddToDoItemViewController code

1. In the project navigator, select `Main.storyboard`.
2. In the outline view, select Add To-Do Item under Add To-Do Item Scene.
3. Click the Assistant button in the Xcode toolbar to open the assistant editor.

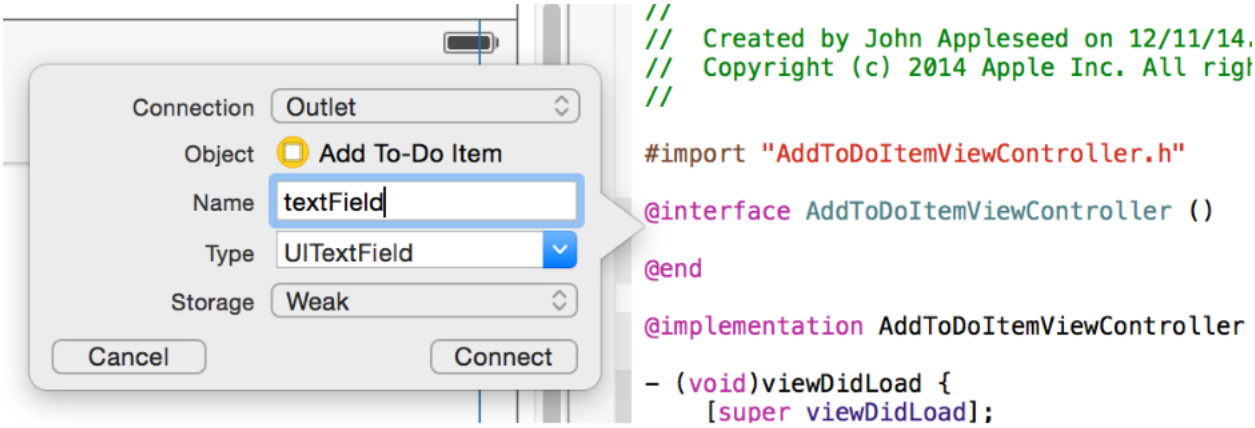
4. Change the assistant editor from Preview to Automatic > `AddToDoItemViewController.m`.
5. In your storyboard, select the text field.

6. Control-drag from the text field on your canvas to the code display in the editor on the right, stopping the drag at the line just below the `@interface` line in `AddToDoItemViewController.m`.



7. In the dialog that appears, for Name, type `textField`.

Leave the rest of the options as they are. Your dialog should look like this:



8. Click Connect.

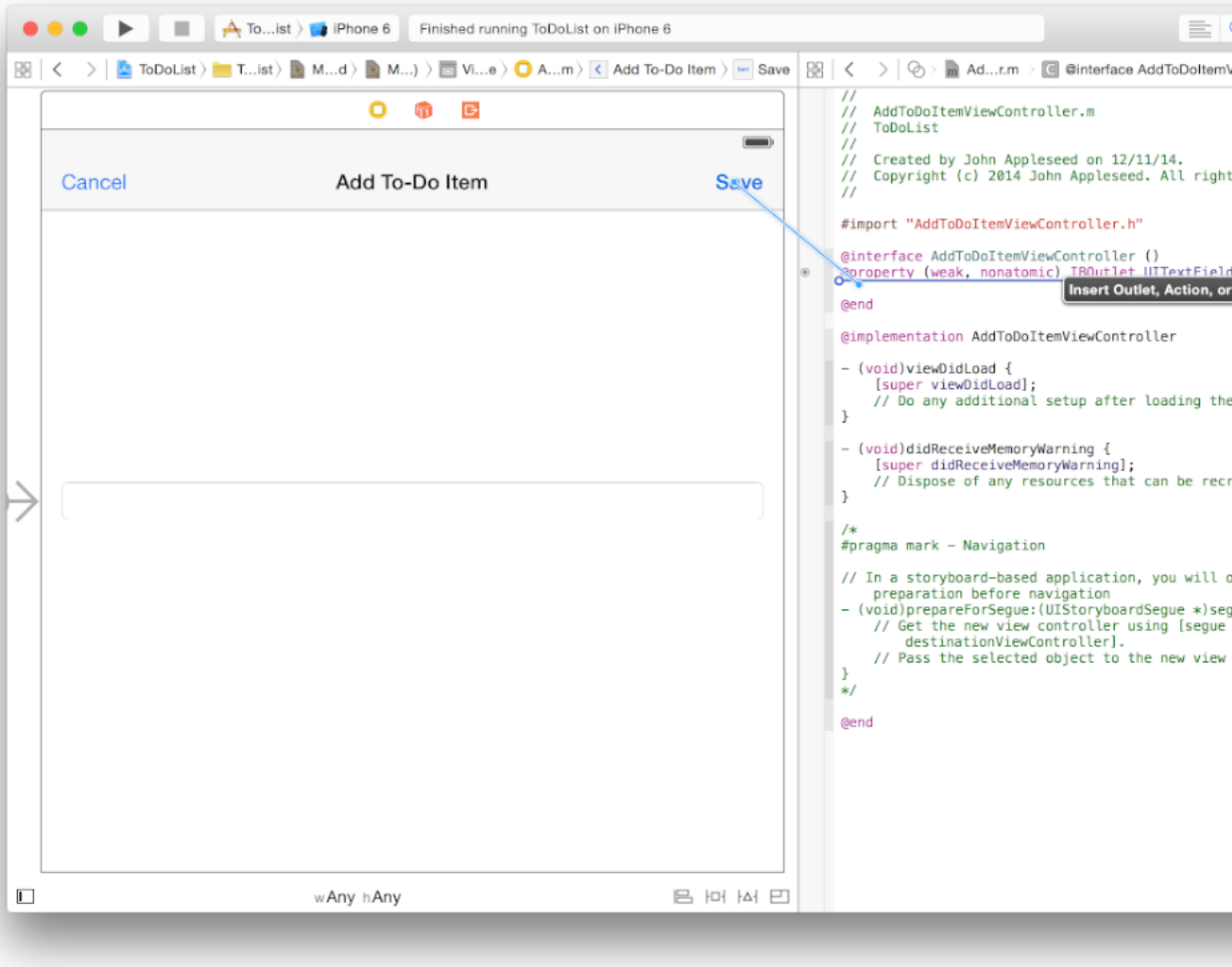
   Xcode adds the necessary code to `AddToDoItemViewController.m` to store a pointer to the text field and configures the storyboard to set up that connection.

Additionally, `AddToDoItemViewController` needs to know when to create a new to-do item. You want the item to be created only if the Save button was tapped. To be able to determine when this happens, add the Save button as an outlet in `AddToDoItemViewController.m`.

To connect the Save button to the AddToDoItemViewController code

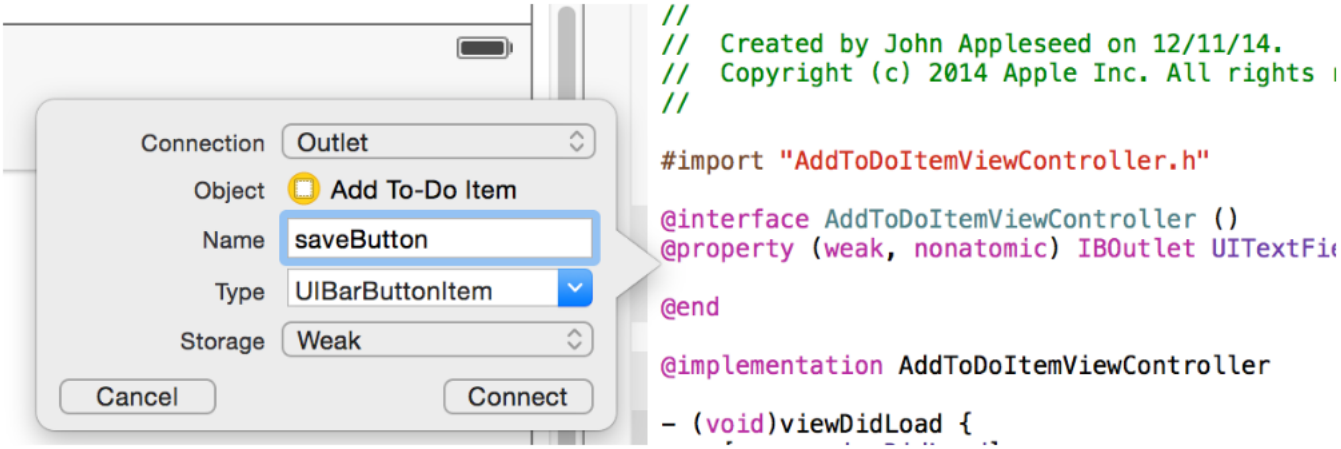
1. In your storyboard, select the Save button.

2. Control-drag from the Save button on your canvas to the code display in the editor on the right, stopping the drag at the line just below your `textField` property in `AddToDoItemViewController.m`.



3. In the dialog that appears, for Name, type `saveButton`.

Leave the rest of the options as they are. Your dialog should look like this:



4. Click Connect.

You now have a way to identify the Save button. Because you want to create an item when the Save button is tapped, you need to know when that happens.

Recall that when the user taps either the Cancel or the Save button, it kicks off an unwind segue back to the to-do list—that's something you configured in the second tutorial. Before a segue executes, the system gives the view controller involved a chance to prepare by calling `prepareForSegue:`. This is exactly the point at which the view controller should check which one of the buttons got tapped, and if it was the Save button, create a new to-do item. (If the Cancel button is tapped, you don't need to do anything to save the item.)

To tell AddToDoItemViewController to create an item only when the user taps the Save button

1. In the project navigator, select `AddToDoItemViewController.m`.

   If the assistant editor is still open, return to the standard editor by clicking the Standard button in the Xcode toolbar.

2. Find and uncomment the `prepareForSegue:` data source method. (To uncomment the method, remove the `/*` and `*/` characters surrounding it.)

   After you do that, the template implementation looks like this:

```
#pragma mark - Navigation


// In a storyboard-based application, you will often want to do a little
 preparation before navigation
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    // Get the new view controller using [segue destinationViewController].
    // Pass the selected object to the new view controller.
}
```

3. In the method, see whether the Save button was tapped.

   If the Save button wasn't tapped, instead of saving the item, you want the method to return without doing anything else, like this:

```
if (sender != self.saveButton) return;
```

4. Add this code to check whether the user entered any text in the text field.

```
if (self.textField.text.length > 0) {
}
```

5. If there's text, create a new item and give it the name of the text in the text field. Also, ensure that the completed state is set to `NO`.

```
self.toDoItem = [[ToDoItem alloc] init];
self.toDoItem.itemName = self.textField.text;
self.toDoItem.completed = NO;
```

   If there isn't text, you don't want to save the item, so you don't need to add code to do anything else.

Your `prepareForSegue:` method should look like this:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if (sender != self.saveButton) return;
    if (self.textField.text.length > 0) {
        self.toDoItem = [[ToDoItem alloc] init];
```

```
        self.toDoItem.itemName = self.textField.text;
        self.toDoItem.completed = NO;
    }
}
```

The new item needs to be passed back to `ToDoListTableViewController` so that it can add the item to the to-do list. To accomplish this, you need to revisit the `unwindToList:` method that you wrote in the second tutorial. This method gets called after the unwind segue triggers.

The `unwindToList:` method takes a segue as a parameter, like all methods that are used as targets for an unwind segue. The segue parameter is the segue that unwinds from `AddToDoItemViewController` back to `ToDoListTableViewController`. Because a segue is a transition between two view controllers, it is aware of its source view controller—`AddToDoItemViewController`. By asking the segue object for its source view controller, you can access any data stored in the source view controller in the `unwindToList:` method. In this case, you want to access the `toDoItem` property. If it's `nil`, the item was never created—either the text field had no text or the user tapped the Cancel button. If there's a value for `toDoItem`, you retrieve the item, add it to your `toDoItems` array, and display it in the to-do list by reloading the data in the table view.

To store and display the new item

1. In the project navigator, select `ToDoListTableViewController.m`.
2. In `ToDoListTableViewController.m`, add an import declaration to `AddToDoItemViewController.h` right below the `#import "ToDoItem.h"` line.

```
#import "AddToDoItemViewController.h"
```

3. Find the `unwindToList:` method you added in the second tutorial. It looks like this:

```
- (IBAction)unwindToList:(UIStoryboardSegue *)segue {


}
```

4. In this method, retrieve the source view controller—the controller you're unwinding from, `AddToDoItemViewController`.

```
AddToDoItemViewController *source = [segue sourceViewController];
```

5. Retrieve the value of the controller's `toDoItem` property.

```
ToDoItem *item = source.toDoItem;
```

If a user entered text in the text field of the add-to-do-item scene and then tapped the Save button, the value of the `toDoItem` property will contain an item. If a user didn't enter text in the text field or tapped the Cancel button to close the screen, the value of the `toDoItem` property won't contain an item. You need to check whether the item exists.

6. Add this code to check whether the item exists.

```
if (item != nil) {
}
```

If it's `nil`, there's no item in the `toDoItem` property, so you don't need to do additional work to save the item.

If it does exist, you need to add the item to your `toDoItems` array, like this:

```
[self.toDoItems addObject:item];
```

7. Reload the data in your table.

Because the table view doesn't keep track of its data, it's the responsibility of the data source—in this case, your table view controller—to notify the table view when there's new data for it to display.

```
[self.tableView reloadData];
```

Your `unwindToList:` method should look like this:

```
- (IBAction)unwindToList:(UIStoryboardSegue *)segue {
    AddToDoItemViewController *source = [segue sourceViewController];
    ToDoItem *item = source.toDoItem;
    if (item != nil) {
        [self.toDoItems addObject:item];
        [self.tableView reloadData];
    }
}
```

Checkpoint: Run your app. Now when you click the Add button (+) and create a new item, you should see it in your to-do list. Congratulations! You've created an app that takes input from the user, stores it in an object, and passes that object between two view controllers. This is the foundation of moving data between scenes in a storyboard-based app.

Recap

You're almost done with this introductory tour of developing apps for iOS. The final section gives you more information about how to find your way around the documentation, and it suggests some next steps you might take as you learn how to create more advanced apps.

Next Steps

iOS Technologies

You've just learned how to write an app with a simple user interface and basic behavior. At this point, you may be thinking about implementing additional behavior that will turn your project into a full-featured app.

As you consider which features you want to add, remember that you don't have to implement everything from scratch. iOS provides frameworks that define particular sets of functionality—from gaming and media to security and data management—which you can integrate into your app. You've already used the UIKit framework to design your app's user interface, and the Foundation framework to incorporate common data structures and behavior into your code. These are two of the most common frameworks used in iOS app development, but there are many more available to you.

This chapter is a high-level overview of technologies and frameworks that you might consider adopting in your app. Use this chapter as a starting point for possible technologies to explore. For a full overview of the technologies available in iOS, see *iOS Technology Overview*.

User Interface

UIKit. UIKit framework classes let you create a touch-based user interface. Because all iOS apps are based on UIKit, you can't ship an app without this framework. UIKit provides the infrastructure for drawing to the screen, handling events, and creating common user interface elements. UIKit also organizes a complex app by managing the content that's displayed onscreen. See *UIKit Framework Reference*.

Core Graphics. Core Graphics—a low-level, C-based framework—is the workhorse for handling high-quality vector graphics, path-based drawing, transformations, images, data management, and more. Of course, the simplest and most efficient way to create graphics in iOS is to use prerendered images with the standard views and controls of the UIKit framework, letting iOS do the drawing. Because UIKit, a higher-level framework, also provides classes for custom drawing—including paths, colors, patterns, gradients, images, text, and transformations—use it instead of Core Graphics whenever possible. See *Core Graphics Framework Reference*.

Core Animation. Core Animation interfaces are part of the Quartz Core framework. Use Core Animation to make advanced animations and visual effects. UIKit provides animations that are built on top of the Core Animation technology. If you need advanced animations beyond the capabilities of UIKit, use Core Animation directly. You can create a hierarchy of layer objects that you manipulate, rotate, scale, transform, and so forth. Core Animation's familiar view-like abstraction lets you can create dynamic user interfaces without having to use low-level graphics APIs such as OpenGL ES. See *Core Animation Programming Guide*.

Games

GameKit. The GameKit framework provides leaderboards, achievements, and other features to add to your iOS game. See *GameKit Framework Reference*.

SpriteKit. The SpriteKit framework provides graphics support for animating arbitrary textured images, or *sprites*. In addition to being a graphics engine, it also includes physics support to bring objects to life. SpriteKit is a good choice for games and other apps that require complex animation chains. (For other kinds of user interface animation, use Core Animation instead.) See *SpriteKit Programming Guide*.

OpenGL ES. OpenGL ES is a low-level framework that supports hardware-accelerated 2D and 3D drawing. Apple's implementation of the OpenGL ES standard works closely with the device hardware to provide high frame rates for full-screen, game-style apps. Because OpenGL ES is a low-level, hardware-focused API, it has a steep learning curve and a significant effect on the overall design of your app. (For apps that require high-performance graphics for more specialized uses, consider SpriteKit or Core Animation.) See *OpenGL ES Programming Guide for iOS*.

Game Controller. The Game Controller framework makes it easy to find controllers connected to a Mac or iOS device. When a controller is discovered on your device, your game reads control inputs as part of its normal gameplay. These controllers provide new ways for players to control your game. Apple has designed specifications for hardware controllers to ensure that all controllers have consistent sets of control elements that both players and game designers can rely on. See *Game Controller Framework Reference*.

Data

Core Data. The Core Data framework manages an app's data model. Use it to create model objects, known as managed objects. You manage relationships between those objects and make changes to the data through the framework. Core Data takes advantage of the built-in SQLite technology to store and manage data efficiently. See *Core Data Framework Reference*.

Foundation. You worked with Foundation earlier in this guide. The Foundation framework defines a set of useful base classes. Among other things, this framework includes classes representing basic data types, such as strings and numbers, and collection classes for storing other objects. See *Foundation Framework Reference*.

Media

AV Foundation. AV Foundation is one of several frameworks that you can use to play and create time-based audiovisual media. For example, use AV Foundation to examine, create, edit, or reencode media files. You can also use it to get input streams from devices and manipulate video during real-time capture and playback. See *AV Foundation Framework Reference*.

Finding Information

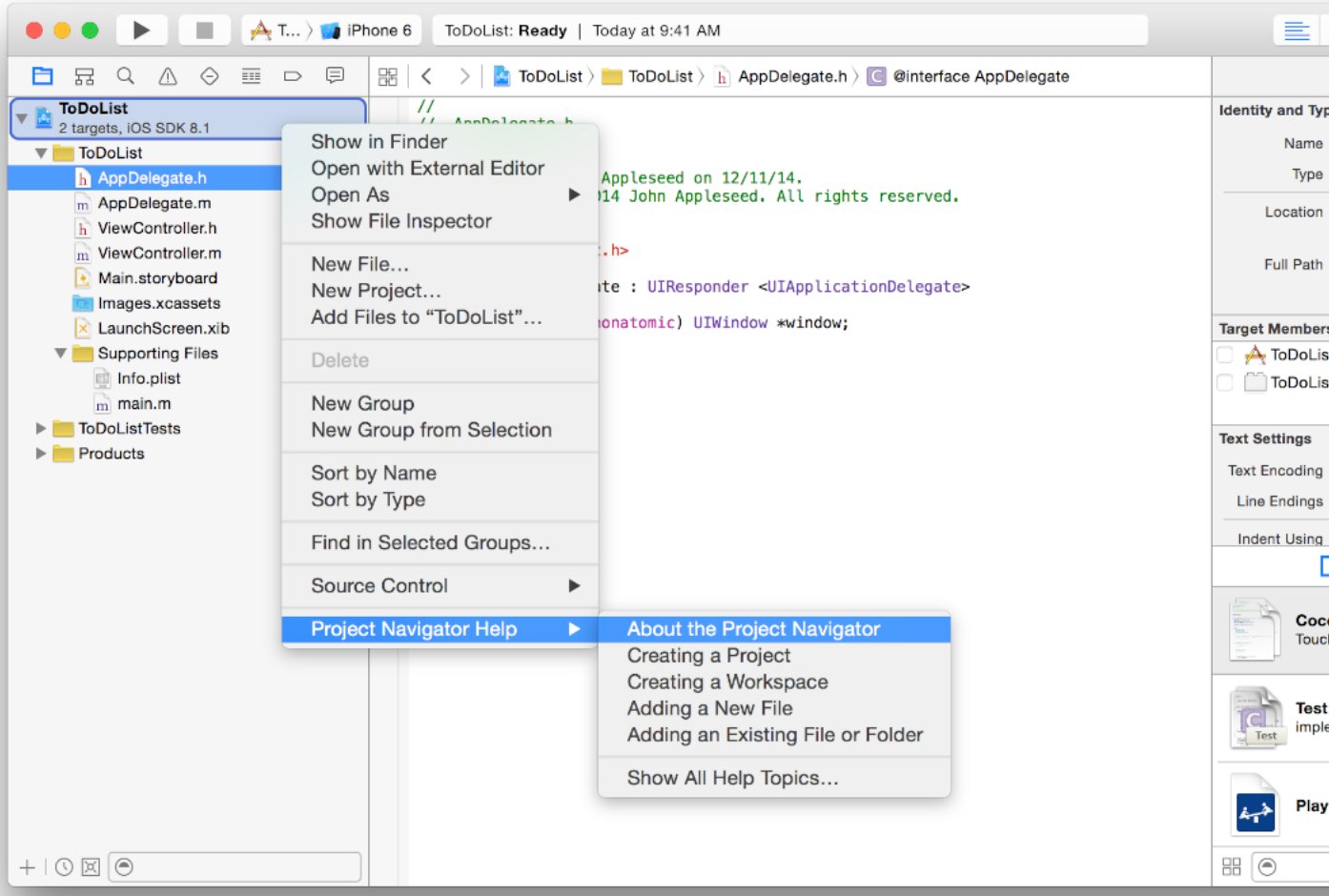
As you develop your app, you'll want the information you've learned—and more—at your fingertips. You can get all the information you need without leaving Xcode.

Xcode comes with a large documentation suite containing general and conceptual guides, framework and class references, focused help articles, and more. You can access this documentation by Control-clicking on a UI element of Xcode to learn how to use it, opening the Quick Help pane in the main project window for context-aware code help, or searching in the documentation viewer to find guides and full API reference. You can also view documentation online at developer.apple.com/library/ios.

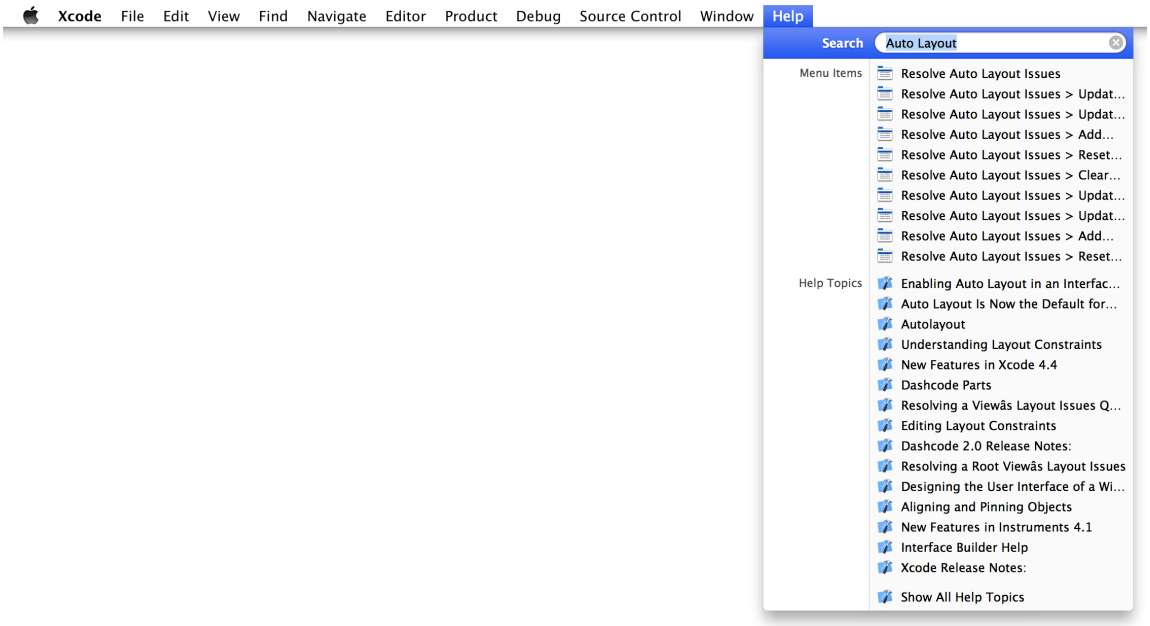
Use Contextual Help Articles for Xcode Guidance

To get help when using Xcode itself, take a look at help articles. **Help articles** show how to accomplish common tasks, such as creating a new class, setting a custom class in Interface Builder, and resolving issues with Auto Layout.

Depending on what you're trying to do, you can access some help articles by Control-clicking on a UI element in Xcode. Look for the last entry (in this image, Project Navigator Help) in the contextual menu.



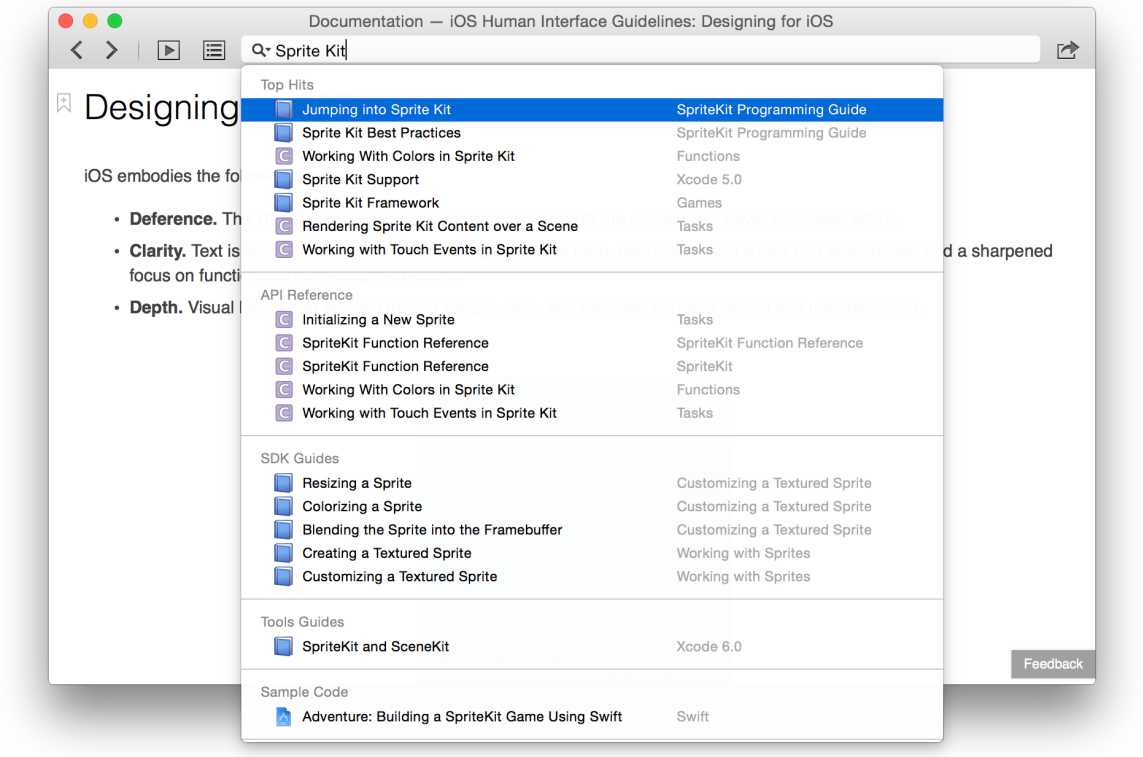
If you're looking for general help, or information about a task related to an element that doesn't support Control-clicking, you can also search for contextual help in the Xcode Help menu.



Use Guides for General and Conceptual Overviews

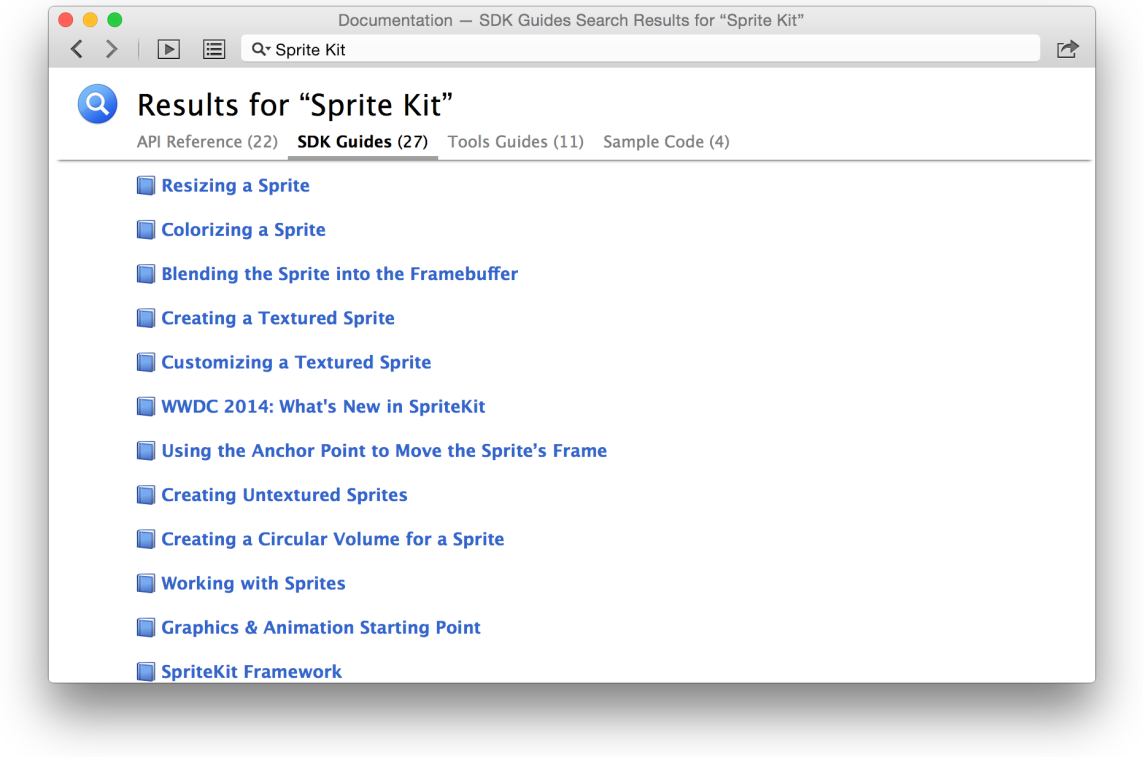
To learn about a new technology or to gain a deeper understanding of how the different classes in a framework work together, look at one of the **programming guides**. Most Cocoa frameworks and technologies have programming guides associated with them, such as *SpriteKit Programming Guide*, *Programming with Objective-C*, and *Location and Maps Programming Guide*.

To view these documents in Xcode, use the *documentation viewer*, which you can access by choosing Help > Documentation and API Reference (Shift–Command–0). Simply type the name of a technology, such as Sprite Kit.



The results are ordered to be most helpful when you are writing code. This means that you'll see top hits, followed by API reference, and then SDK and tools guides.

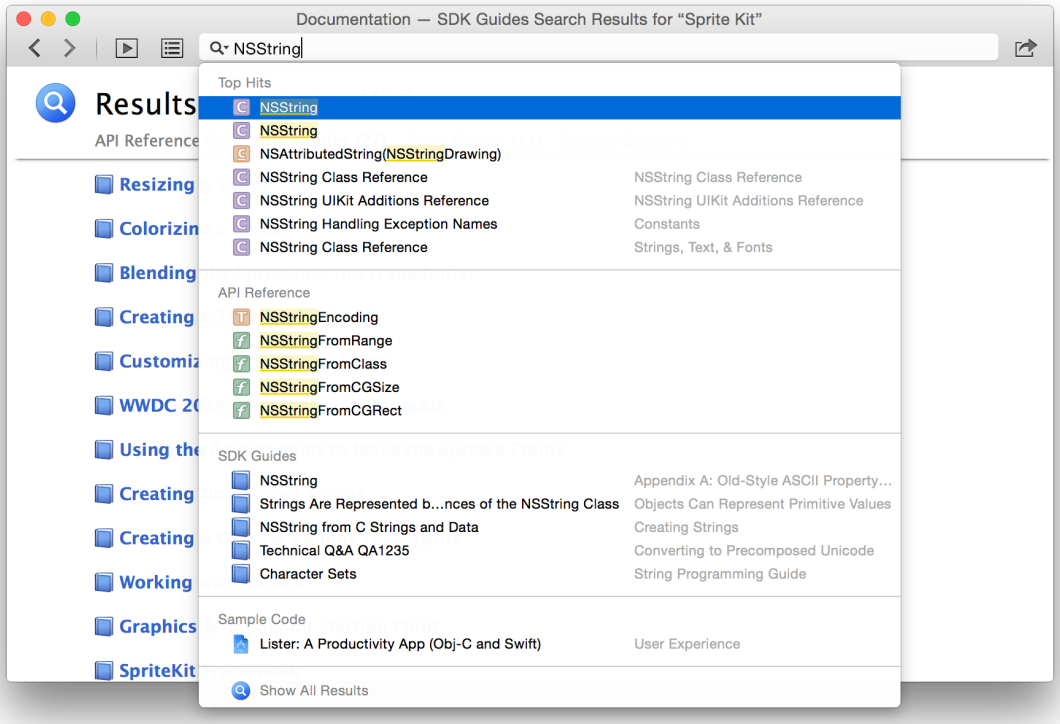
If a suitable result doesn't appear in the pop-up list, view all results by pressing Return without selecting anything. This displays a complete set of results that you can filter.



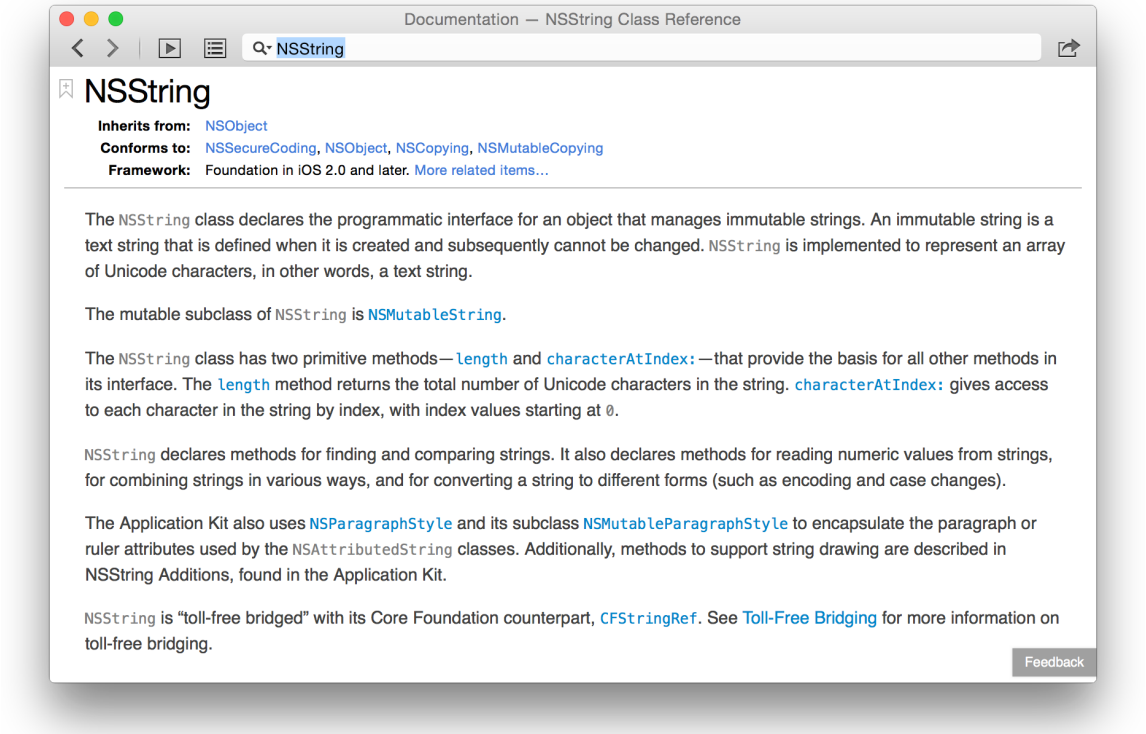
Use API Reference for Class Information

When you've read a guide to learn about the different parts of a technology and you start writing code that uses the technology, you'll probably find that you need to know more about what an individual class can do, or that you need to learn the correct way to call a particular method. This information is provided by **API reference** documents.

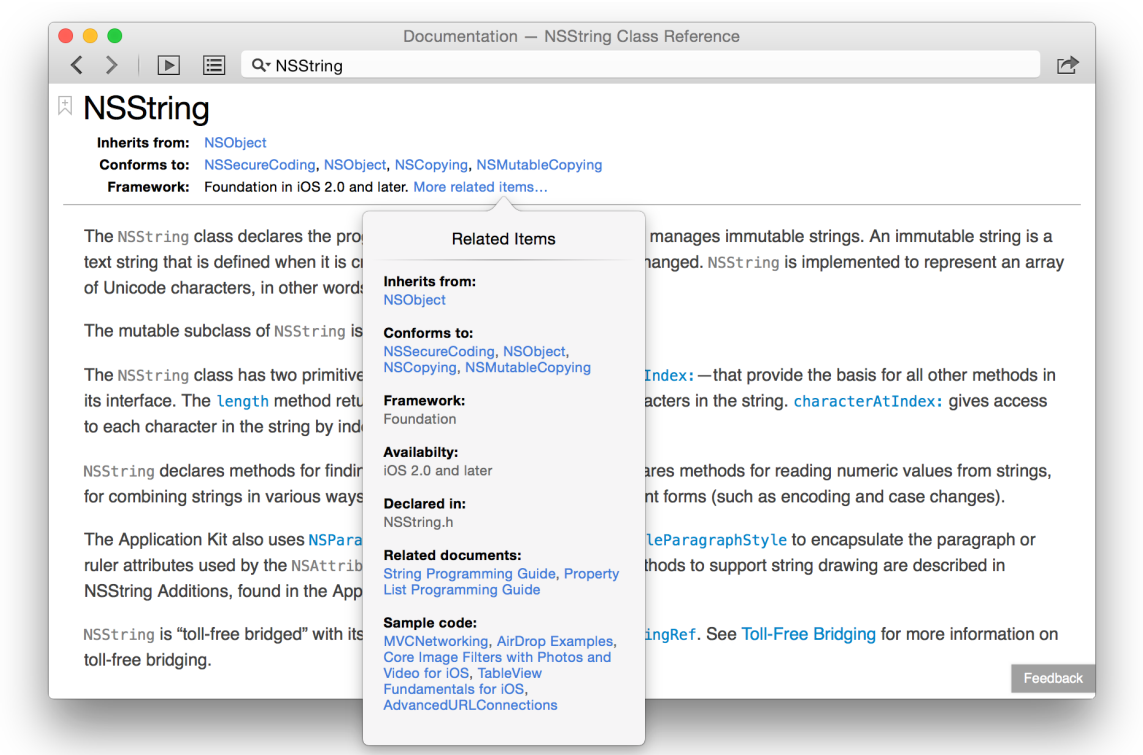
For example, for more information on the `NSString` class that you used in the earlier tutorials, just type the name of the class in the documentation viewer's search field.



The top hit is usually the one you want; press Return to select it, and you see the API reference for that class.



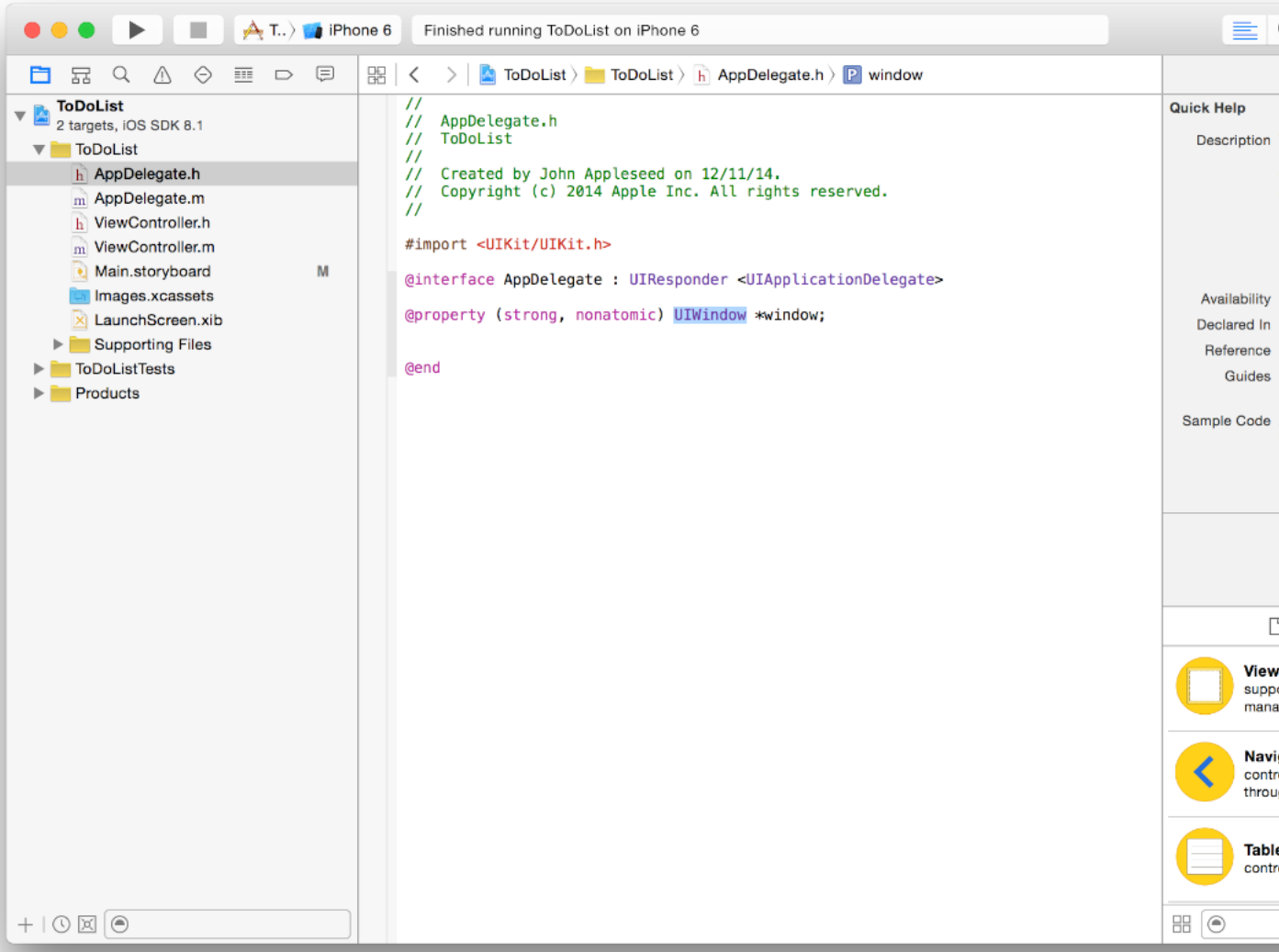
API reference documentation gives you quick access to information about individual classes, including methods, properties, parent class, adopted protocols, and tasks that the class can perform. Click “More related items” to see general information about the class.



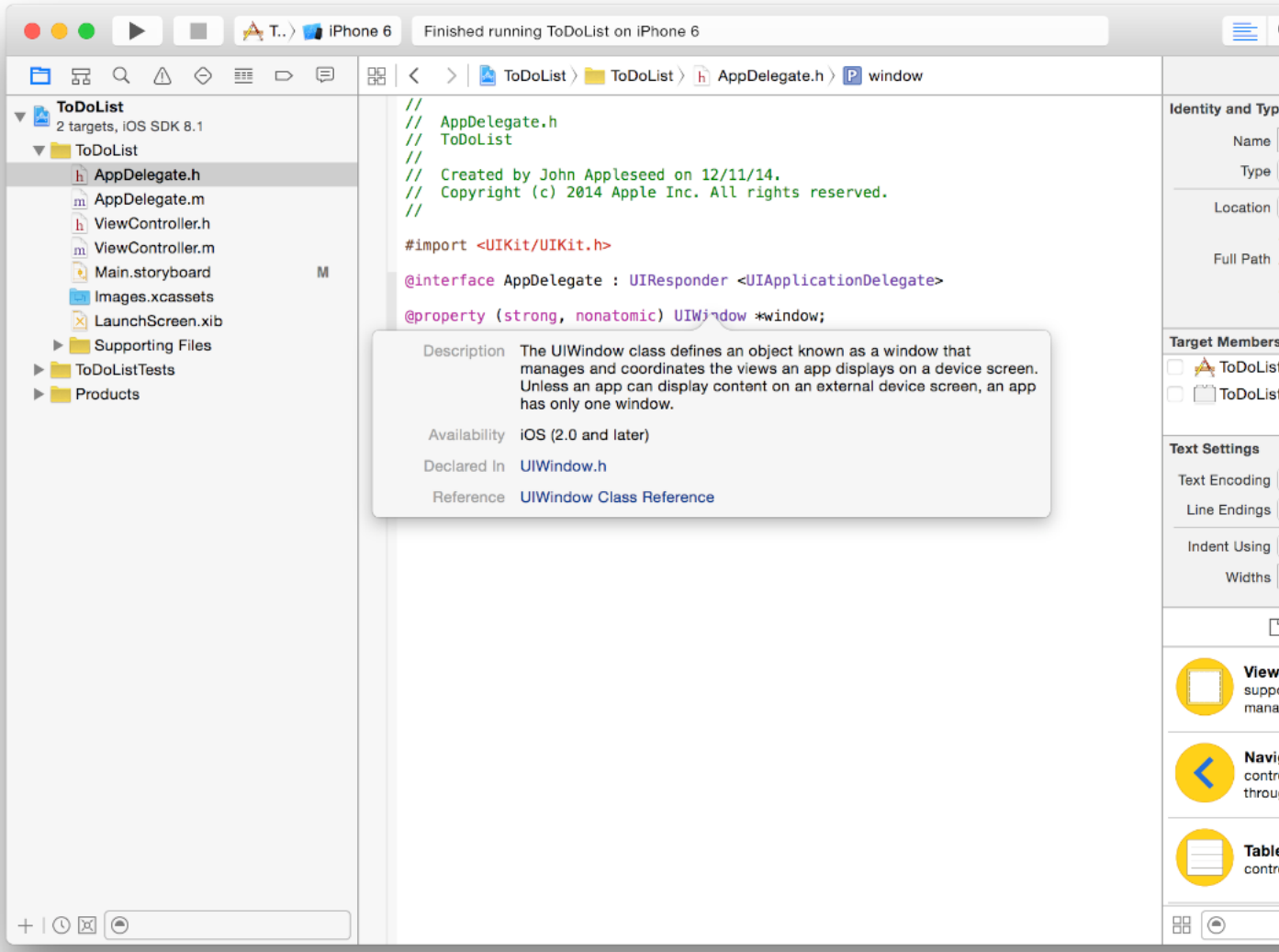
The Related Items popover also shows a list of related guides. For `NSString`, for example, you might want a more conceptual overview rather than delving into reference material, so you should read *String Programming Guide*.

Use Quick Help for Contextual Source Code Information

When you're writing code in the source editor, you have easy access to API reference documentation in the Quick Help pane (choose View > Utilities > Show Quick Help Inspector). The **Quick Help** pane updates as you write code, showing information about the symbol you're currently typing.



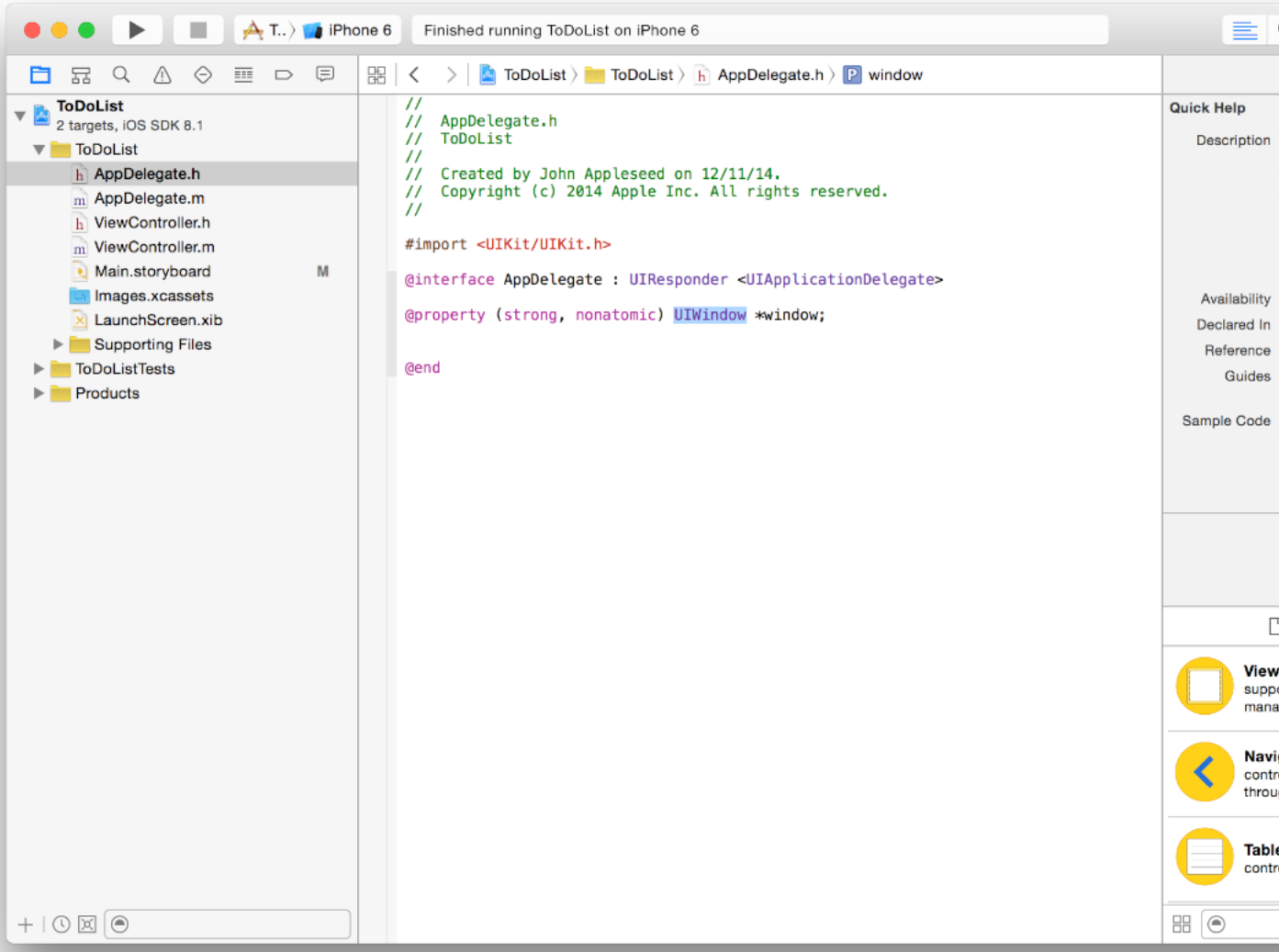
Alternatively, you can Option-click a symbol in the source editor to display a pop-up window with the Quick Help information.



From the Quick Help pane or pop-up window, open the API Reference in the documentation viewer, or view the original header file containing the declaration for the symbol you clicked.

Use Sample Code to See Real-World Usage

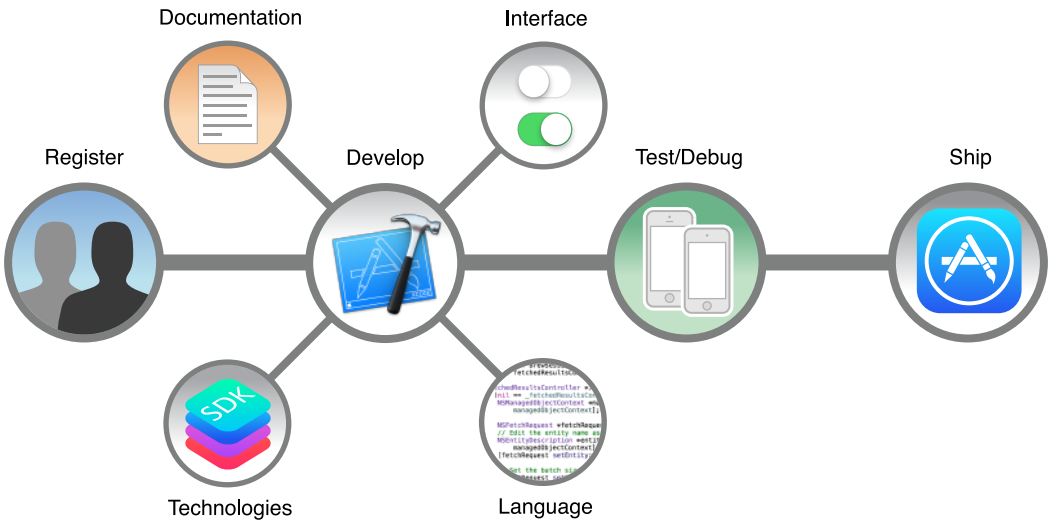
In addition to written documentation, you can access a library of **sample code**. Whenever you look at quick help, or guides and reference in the documentation viewer, you'll see entries showing relevant sample code projects for the given technology or class.



Click one of the sample code entries to download and open a project in Xcode so that you can examine the code.

Where to Go from Here

In *Start Developing iOS Apps Today*, you learned the basics of iOS app development. At this point, you're ready to start developing your first full-featured app. Although taking an app from a simple concept to the App Store isn't a small task, the process and practices you've learned in this document will guide you in the right direction.



Here are a few more pointers on where to go from here:

- **Register as a developer.**

  Managing Accounts walks you through the process of registering as an Apple developer.

- **Learn to design beautiful app interfaces.**

  iOS Human Interface Guidelines teaches you how to make your app consistent with the user interface conventions for iOS.

  Auto Layout Guide teaches you how to create adaptive, versatile user interfaces.

- **Learn the Objective-C language.**

  Programming with Objective-C describes how to define classes, send messages, encapsulate data, and accomplish a variety of other tasks with the Objective-C programming language.

- **Learn to develop great apps.**

  App Programming Guide for iOS explains the essential things you must know and do when developing an iOS app.

- **Learn about the technologies available to you.**

  iOS Technology Overview describes the frameworks and other technologies that are available to your app in iOS.

- **Access the documentation.**

  Finding Information (page 154) shows you how to make the most of the documentation available to you.

- **Debug and test your app.**

  Debug Your App teaches you how to thoroughly debug and test your app in Xcode.

- **Ship your app.**

  App Distribution Guide walks you through the process of provisioning devices for testing and submitting apps to the App Store.

Taking the ToDoList App to the Next Level

The to-do list app you just created benefits from numerous built-in behaviors. You can continue to experiment with this app to enhance your understanding, or you can start something new. If you do continue with the to-do list app, here are some areas to investigate:

- Your to-do list disappears when you quit and relaunch the app. You might want to explore ways to make the list persist over time.
- You're using the default appearance for all of the controls in your app. UIKit includes features for customizing the appearance of many controls. Explore different user interface options using this technology.
- You've given the user a way to add items to the list and mark them as completed, but there's no facility for deleting items. Table views have built-in behavior for supporting editing, including deletion and reordering of rows, which you might consider incorporating into your app.

As you continue developing iOS apps, you'll find many concepts and technologies to explore, including localization, accessibility, and appearance customization. Start by defining a direction that interests you. Remember to put concepts into practice as you learn them. When you encounter an interesting new technology, framework, or design pattern, don't be afraid to write a small app to test it out.

By adopting the approach shown in this document, you'll find that you ship your first app quickly. After you have an app in the App Store, you can continue to incorporate additional features incrementally. There are always new ways to keep your customers engaged and looking forward to the next great thing.

Document Revision History

This table describes the changes to *Start Developing iOS Apps Today*.

Date	Notes
2015-03-09	Applied minor edits throughout.
2014-12-19	Updated for new features in Xcode 6.1 and iOS 8.1.
2013-10-22	Rewritten as a multipart tutorial to provide the fundamental skills needed to create an iOS app.

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