FALL 2015 EDITION

INTRODUCTION TO PROGRAMMING WITH XOJO

IOS ADDENDUM

BRAD RHINE

Introduction to Programming with Xojo

IOS ADDENDUM

BY BRAD RHINE

Fall 2015 Edition

Table of Contents

Introduction

Welcome to the iOS Addendum for *Introduction to Programming with Xojo*!

For a long time, there were only two ways to build and deploy apps for iPhone and iPad. You could use Apple's Xcode, which is a powerful tool, but requires you to learn one of Apple's programming languages. Or you could build a web app in HTML and JavaScript, which works, but results in an app that can't be distributed through the App Store and that generally runs much more slowly.

This addendum has been created to provide you with an introduction to creating iPhone and iPad apps with Xojo, which means you can leverage your existing knowledge and skills in Xojo to build and deploy apps for one of today's biggest mobile platforms.

Right off the bat, there are a few issues that should be mentioned. First, Xojo currently only supports iOS, so Xojo apps for Android and other mobile platforms aren't possible at the moment. Second, because of the way that Apple's Developer Tools work, Xojo iOS apps can only be developed on the Mac. You'll learn more about that in a bit.

This addendum is meant to be an introduction to iOS development with Xojo. You'll learn how to create an iOS app and test it in Apple's iOS Simulator. Additional topics like distributing your app on the App Store are beyond the scope of this introduction, but there are some links at the end to help you with those advanced topics.

This addendum assumes that you have worked through *Introduction to Programming with Xojo* and are familiar with the basics of using Xojo.

1.1 OS X & Xcode

As mentioned earlier, developing for iOS requires a Mac. You should be running the latest version of OS X. You will also need to install Apple's Xcode, which is a free download from the Mac App Store.

Apple's Developer Program does have paid memberships, but you do not need to pay to test your iOS apps. If you decide to move forward and deploy an app on the App Store, you'll need to investigate a paid membership at that point.

Xcode is Apple's recommended development environment. It's a very powerful tool, but it has a steeper learning curve than Xojo. You will mostly only need Xcode because it supplies you with the iOS Simulator.

Note that the first time you launch Xcode (whether you launch it manually or Xojo launches it for you), you'll need to agree to Apple's terms.

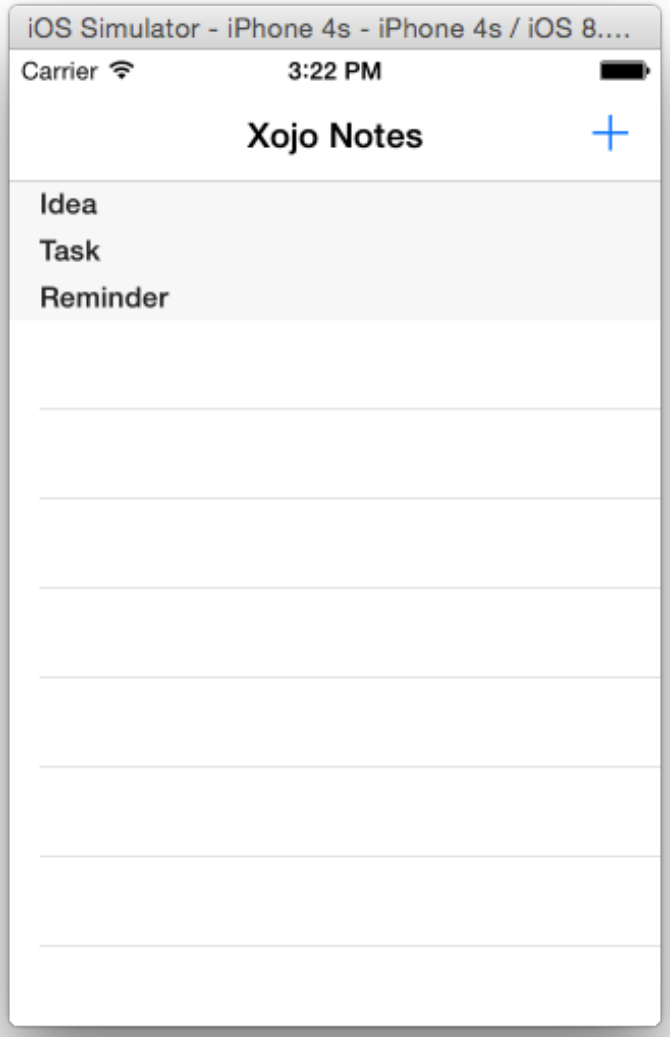
1.2 The iOS Simulator

The iOS Simulator is a Mac app that runs a stripped down version of iOS, which you can use to test your apps. If you think back to working with Xojo desktop apps, you could click the Run button and launch a temporary version of your app. For iOS apps, that temporary version runs inside the iOS Simulator.

In the early days of iOS, a developer could rely on every iPhone having the same screen size and same resolution. That started to change in 2010 with

the advent of retina displays, and in recent years, Apple has released an even wider variety of screen sizes and resolutions.

Because of this variety, you'll want to test your app on different devices. The iOS Simulator allows you to set up multiple virtual iOS devices (both iPhones and iPads of various sizes), which you'll learn about in the next section.



1.3 iOS Controls & Events

Before you jump into a quick and easy sample project, you need to know a few things about controls on iOS. They differ from desktop controls in a few significant ways.

First, because there is no mouse pointer on iOS, controls have fewer events. There is no MouseEnter event, or anything mouse-related. Most controls have events for Open, Close, and Action.

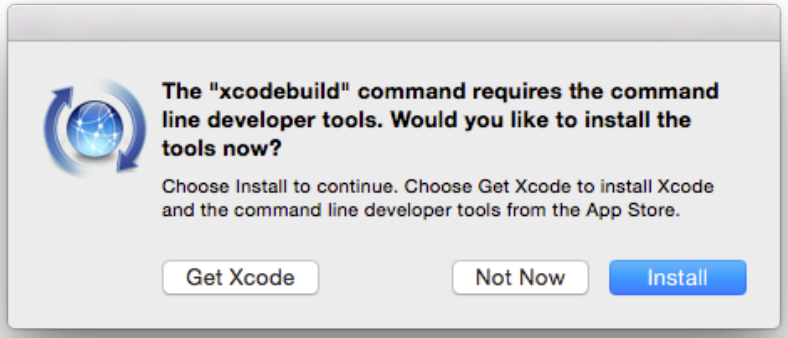
Second, while Xojo on the desktop keeps track of your mouse pointer's coordinates for you, iOS doesn't have that capability unless the user's finger is touching the screen.

Last, you'll notice that the list of available controls for iOS is much smaller than for the desktop. Many desktop controls have no place on a touch screen. Others may be coming in future versions of Xojo.

1.4 Hands on with iOS Controls

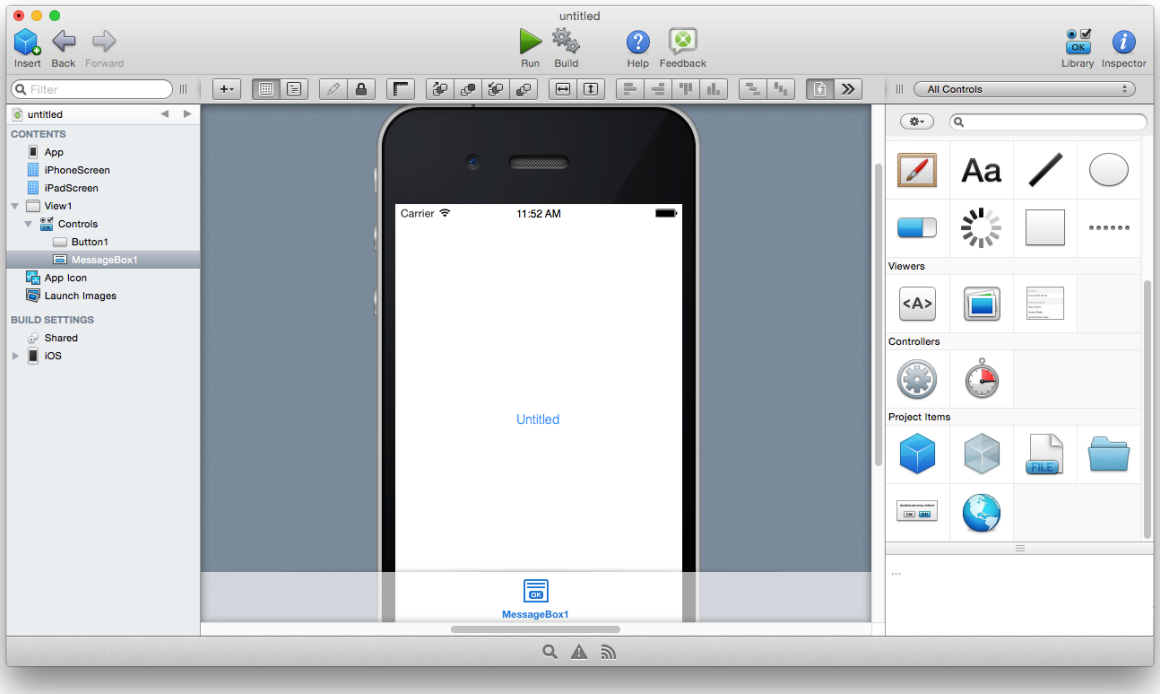
To create an iOS app, begin by opening the Xojo application. You will be prompted to choose a template for a new project. Select “iOS” and fill in the Application Name, Company Name, and Application Identifier.

Note that if Xcode has not been installed or properly configured, you may see an error message like this:



Assuming everything is installed, the Application Name can be whatever you like, as can the Company Name. The Application Identifier, however, should in Apple’s “reverse DNS” format. For example, if a company called FooBar created an app called DreamCatcher, the Application Identifier might be com.foobar.dreamcatcher (as you can see, it almost looks like a backwards website address, which is why it’s sometimes called reverse DNS format).

Once you’ve filled in all three fields, press the OK button. You’ll be greeted with a screen that looks largely familiar, but different in some subtle ways.



For starters, you'll see an iPhone screen in Xojo, instead of the usual desktop window. You'll also notice that the list of available controls is not as lengthy and varied as the desktop selection, as mentioned above.

1) Find the Button in the controls list and drag it onto View1.

Use the guides to center the Button on the View.

2) Find the MessageBox in the controls list and drag it onto View1.

The MessageBox control will automatically place itself at the bottom of the pasteboard, because it is a non-visual control that only appears when needed.

3) Using the Inspector, change your Button's caption to "Say Hello."

4) Using the Inspector, change the following properties of your MessageBox:

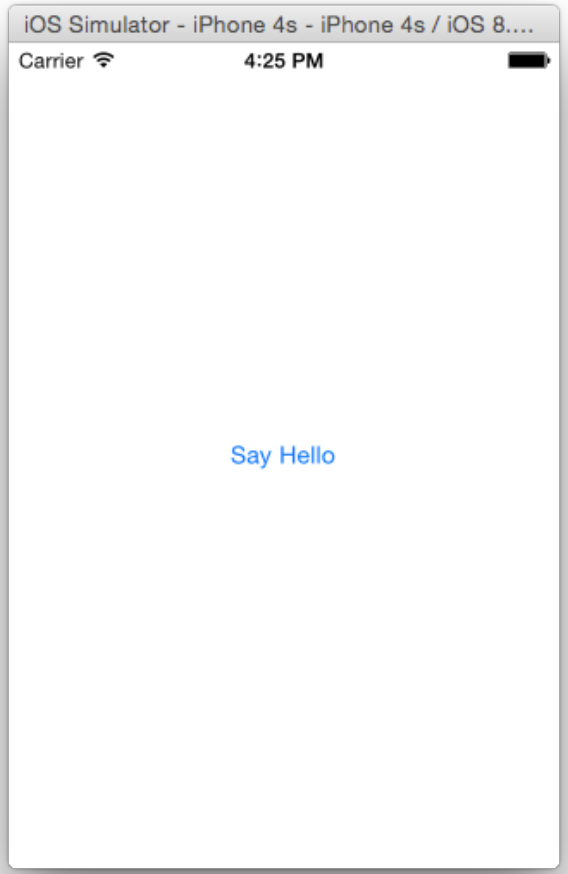
Message: "Nice to meet you!"
Title: "Hello, Mobile World!"
Left Button: "Cancel"
Right Button: "Okay"

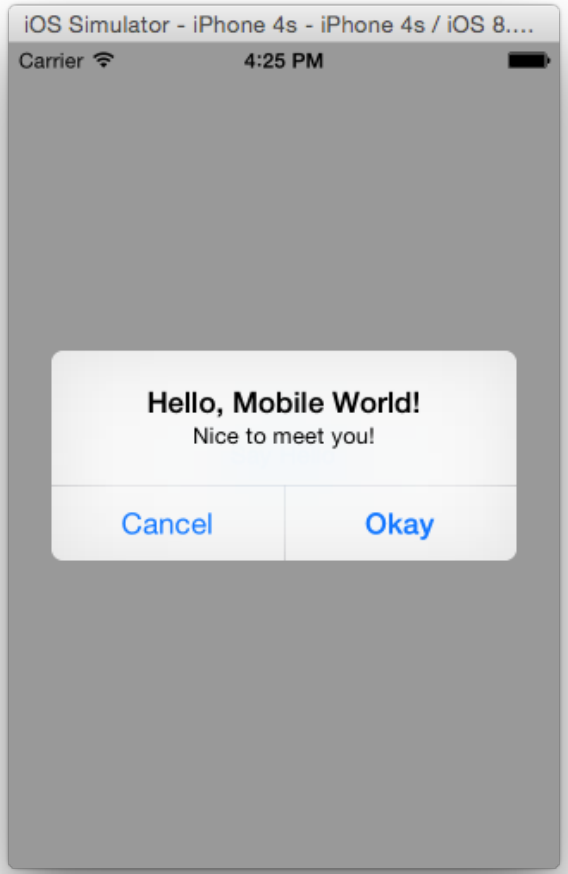
5) Add the following code to your Button's Action event:

```
MessageBox1.Show
```

6) Run your project.

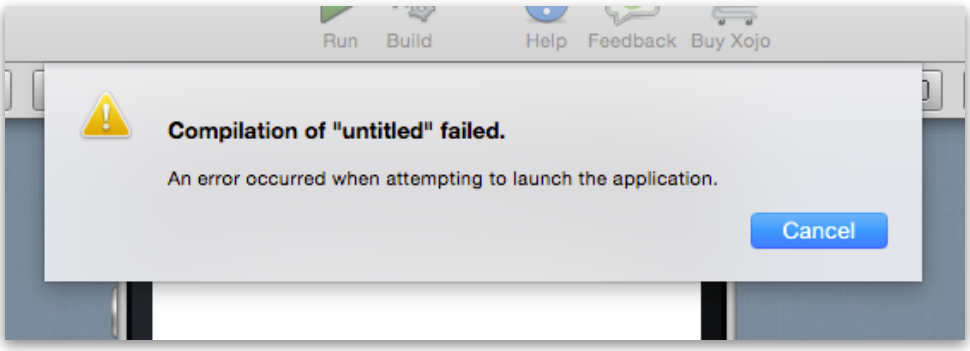
If your computer is set up correctly, the iOS Simulator will appear and launch your app. Click on the button to see the "Hello, Mobile World" MessageBox appear. It should look something like the images below.





7) Troubleshoot

If your computer is not set up correctly, you may see an error message. The following error message indicates that either Xcode is has never been run (which also means that the license hasn't been agreed to).



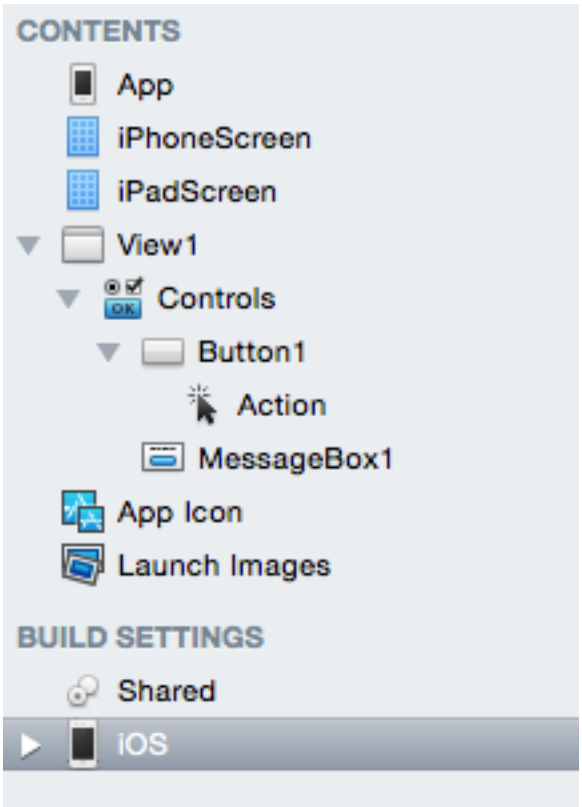
1.5 Devices & Build Settings

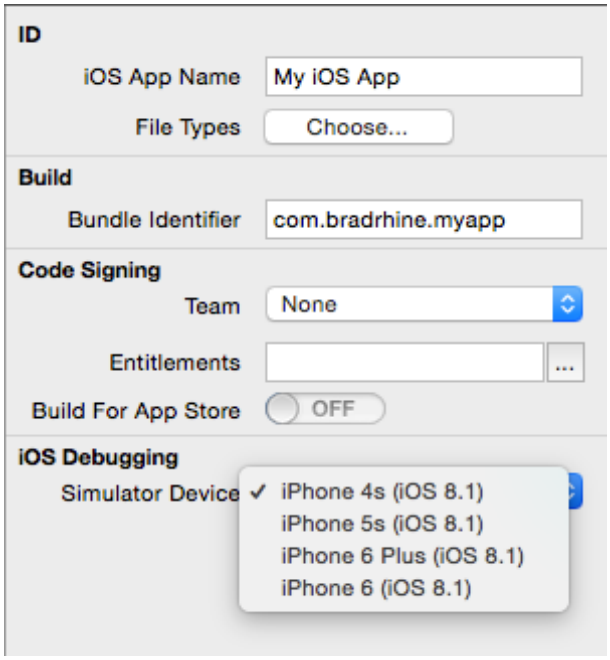
As mentioned earlier, the iOS Simulator can be used to test your app on a variety of virtual devices. The downside is that you have to configure these virtual devices yourself.

Hopefully you still have the iOS Simulator running from your Hello Mobile World test. If so, go to the iOS Simulator's Hardware menu. Under Hardware, go to Device, and then Manage Devices. This will launch Xcode and bring up the Device Management screen.

On the Device Management screen, you can create your own virtual iOS devices. The hardware devices you can choose are limited to the hardware that Apple's current build of iOS supports, but each will be a reliable way to determine what your app will look like and how it will behave on that device.

Back in Xojo, you can choose which device to use for debugging by going to the iOS Build Settings in the Navigator:





Whichever Simulator Device you choose here will be launched when you run your iOS app in the Debugger, and your app will be scaled to that device accordingly.

1.6 Screens & Views

One of the biggest ways that iOS development is different from desktop development is in how information is displayed to the user. On the desktop, your user sees a window, which is just a portion of the screen.

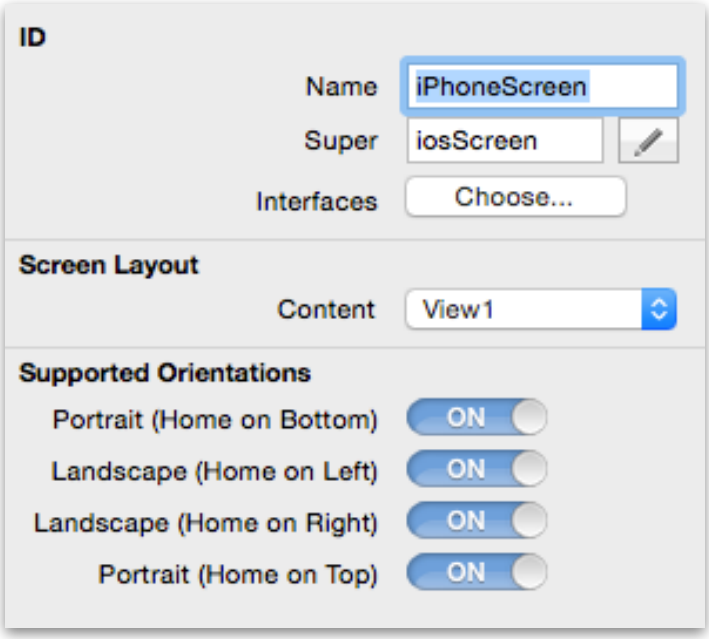
On iOS, there are no windows. The user is unable to resize the interface, and his or her entire screen is always active. Since iOS can run on a variety of devices and screen sizes, your app needs to be able to adapt to different layouts and resolutions. This is accomplished using Screens and Views.

It’s helpful to think of a View as the iOS version of a window. They're not exactly the same, but a View is where you build your interface and define how your app responds to the user, much like a desktop Window.

A Screen is a bit trickier, because there’s no real desktop feature that compares to it. In your iOS app, you define a Screen by device and orientation. An iOS app in Xojo will have two Screens by default: an iPhone Screen and iPad Screen.

If you’re building an iPhone-only app, it’s safe to delete the iPad Screen, and vice-versa.

Recall that in a desktop app, you select the default window seen by the user. In an iOS app, you select the default View for the user, but you must do so for each Screen:



Notice that by combining your Screen and View with the different orientations available, you can create a very customized interface for your iOS app. For example, someone using your app on an iPad in landscape orientation could see an entirely different interface from someone using your app on an iPad in portrait orientation. And you can even further refine the interface based on whether the user has the iOS device's Home button at the top, at the bottom, on the right, or on the left.

But switching from one View to another on iOS is not as simple as showing a new Window on the desktop. Because the current View will fill up the user interface, you must manage which View is in front. This is best demonstrated with a quick sample project.

1) Create a new iOS project.

Since this is a quick demonstration project, you do not need to worry about the Application name and other related details. A View called View1 will already exist by default.

2) Create a new View.

Go to the Insert menu and choose View. The new View will be called View2 by default. In the Inspector, set View2's NavigationBarVisible property to true. This gives View1's Back button a place to be.

3) Add an iOSButton to View1.

Drag an iOSButton onto View1 and position it in the center of the View. Change its Caption to "This is View1." Give the iOSButton an Action event handler with the following code:

```
Dim V As New View2
PushTo(V)
```

4) Set View1's BackButtonTitle property.

Every View has a property called "BackButtonTitle." This property determines the caption of the button that will return the user to the View. Set View's BackButtonTitle to "Back." This will turn on the system-level toolbar for View2 and add a Back button. Remember that the Back button's caption on View2 is set by View1's BackButtonTitle property.

5) Add an iOSLabel to View2.

Drag an iOSLabel onto View2 and position it in the center of the View. Change its Text property to "This is View2."

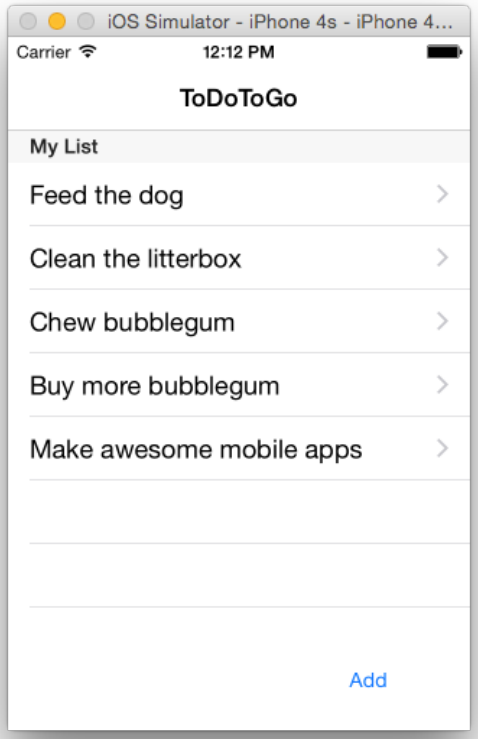
6) Run your project.

Experiment with navigating back and forth between View1 and View2. Notice in particular how View2 "slides" on top of View1 and then "slides" back out of it way.

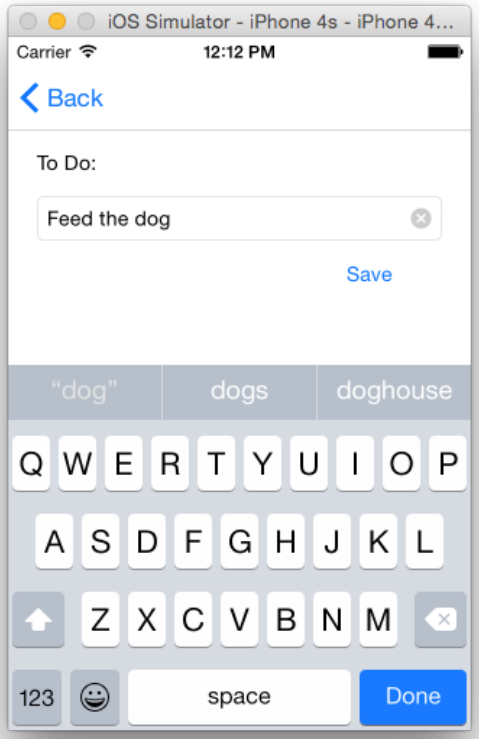
Managing these interactions between Views is a critical part of developing a great iOS app because they give user a sense of “place” within the interface. The PushTo method in particular, causes a View to “slide in” or “push on top of” another View, creating a sense for the user that these Views are stacked and that their positions are relative to one another.

1.7 Hands On With iOS

Your sample project for iOS is a variation on the To Do List Manager that you created in Chapter 5. You will be able to add items to a to do list, edit those items, and delete them. The finished app will something like this:



ListView



ToDoView

1) If you haven't already done so, launch Xojo and create a new iOS Application. Save it as "ToDoToGo".

To start, you'll build the interface, then add your code later. First, rename your default View to "ListView". Make sure ListView's BackButtonTitle is set to "Back" and that its NavigationBarVisible property is set to True. Set ListView's title to "ToDoToGo." From the Library, add two controls to ListView: an iOSTable named "ToDoTable" and a button named "AddButton."

2) Add a new module to your project by going to the Insert menu and choosing Module.

The module should be named "ToDoList." Give it a new property: ToDo(-1) As Text (note that in iOS projects, a Text property is analogous to a String property in desktop and web apps). The parentheses after the property name make this property an array (the value of -1 makes it an empty array). This is where your app will store the user's to do items. Make sure the property's scope is set to Global.

3) Add a new method to ListView.

Your app will need a way to display the current list of items to the user, so you will need to add a method that loops through the array of to do items and places each one in the iOSTable. Add a new method to ListView called "PopulateTable."

4) Add this code to the PopulateTable method:

```
Dim i As Integer
Dim cell As iOSTableCellData
ToDoTable.RemoveAll
ToDoTable.AddSection( "My List" )

For i = 0 To UBound( ToDo )
     cell = New iOSTableCellData
     cell.Text = ToDo( i )
     cell.AccessoryType = ¬
          iOSTableCellData.AccessoryTypes.Disclosure
     ToDoTable.AddRow( 0, cell )
Next
```

Note that working with an iOSTable is not quite the same as working with a ListBox in a desktop app. One major difference is that an iOSTable has sections, which are discrete areas of the table, each with its own title. For example, if you wanted to break your to do items into categories, you could have a section of the iOSTable for each category. To keep things simple in this example project, the above code adds one section called “My List” to the iOSTable.

Adding cell data to an iOSTable is also quite different. Remember that on the desktop, you simply need to use the AddRow method and feed the ListBox a string value. On iOS, you will create a new instance of the class iOSTableCellData and pass that to the iOSTable’s AddRow method. That method also takes an integer indicating which section the cell should belong to. In this example, since there is only one section, just use zero as the first parameter.

The iOSTableCellData data type has one property that should always have a value: text. This is the data that will be displayed on the list that the user sees. The above method also assigns an AccessoryType to the iOSTableCellData. The AccessoryType provides a graphical cue to the cell’s purpose. In this case, the Disclosure type adds a small arrow at the end of the cell indicating that the user can touch that cell to trigger a new interaction.

Finally, note that before you fill the table with data, you must clear out any existing data, to avoid presenting the user with multiple copies of the same data and potentially causing confusion.

5) Give ListView an Event Handler for the Activate event.

Whenever the user sees ListView, the data should be refreshed and up to date, so the Activate event handler should include this line of code:

```
PopulateTable
```

6) Add a new View called “ToDoView.”

This is the View where the user will enter and edit to do items. Make sure its NavigationBarVisible property is set to True. It will need four controls, which you may position as you see fit (again, using the screenshot above as a guide): a label, a

TextField named “ToDoField,” a button named “SaveButton,” and a button named “DeleteButton.” SaveButton and DeleteButton’s captions should be “Save” and “Delete,” respectively. Set DeleteButton’s Visible property to False; it will only be needed when the user is editing a to do item, so your app should only show it when needed. Finally, giveToDoView a new property: ToDoItem as Text.

7) Add a new method to ToDoView called “Populate.”

The method should take one parameter: Item as Text. The code below will take that value and assign to the View’s ToDoItem property.

When the user edits a to do item, your app will need to show the text of that item in ToDoField. The Populate method will take care of that. Here is the code for the method:

```
ToDoItem = Item
ToDoField.Text = ToDoItem
DeleteButton.Visible = True
```

This code sets ToDoView’s ToDoItem property to the text of the item that the user wishes to edit. It also places that value into ToDoField’s text property. Finally, it makes DeleteButton visible to allow the user to remove the to do item from his or her list.

8) Add the following code to the Action even of AddButton on ListView.

Give AddButton on ListView a handler for its Action event and add this code:

```
Dim V As New ToDoView
PushTo( V )
```

This code should look familiar from the navigation example earlier in this chapter. It simply brings ToDoView onto the screen to allow the user to create a new to do item.

9) Add the following code to the Action even of ToDoTable on ListView.

Give AddButton on ListView a handler for its Action event and add this code:

```
Dim item As Text
```

```
Dim V As New ToDoView
item = Me.RowData( section, row ).Text
V.Populate( item )
PushTo( V )
```

This code is triggered when the user touches a row on ToDoTable. It determines which to do item the user is touching and passes that to the Populate method of ToDoView. Note that rather than simply accessing the string value of the selected row as you would on the desktop, you must access it via the RowData property.

10) Add the following code to the Action even of SaveButton on ToDoView.

Give SaveButton on ToDoView a handler for its Action event and add this code:

```
If ToDoField.Text <> "" Then
     If ToDoItem <> "" Then
          Dim i As Integer
          For i = 0 To UBound( ToDo )
               If ToDo( i ) = ToDoItem Then
                    ToDo( i ) = ToDoField.Text
               End If
          Next
     Else
          ToDo.Append( ToDoField.Text )
     End If
End If
Self.Close
```

When the user touches SaveButton, your app will need to either create a new to do item or update an existing one. To determine which action to take, this code checks to see if ToDoView's ToDoItem property contains any text. If it does not, it is safe to assume that the user intends to create a new to do item. This is done by appending the text property of ToDoField to the global ToDo array.

If, however, ToDoView's TodoItem does contain text, then the existing item must be updated. To do so, this code loops through the global array of ToDo items, looking for one that matches the one that was given to ToDoView via the Populate method. If it finds a match, it updates that item in the array.

When either task is complete, the method closes the window and returns the user to ListView.

11) Add the following code to the Action even of DeleteButton on ToDoView.

Give DeleteButton on ToDoView a handler for its Action event and add this code:

```
If ToDoItem <> "" Then
     Dim i As Integer
     For i = 0 To UBound( ToDo )
          If ToDo( i ) = ToDoItem Then
               ToDo.Remove( i )
          End If
     Next
End If
Self.Close
```

When the user touches DeleteButton, the app will loop through the global ToDo array, searching for an item that matches the matches the one that was given to ToDoView via the Populate method. If a match is located, it is deleted from the array, and the user is returned to ListView.

12) Run your project.

The iOS Simulator should appear and your iOS app should launch. Try adding, editing, and deleting some to do items.

If your project fails to run, make sure that you have selected a Simulator Device in your project's iOS Build Settings and that Xcode is properly installed.

1.8 Debugging

Even though your iOS project seems like it's bouncing through various applications before you see it running, debugging it is almost exactly the same as debugging a desktop app. You can still set breakpoints and step through code.

For more information on debugging, see sections 1.5 and 2.6 in *Introduction to Programming with Xojo.*