Tech Note

embarcadero

Using Delphi Prism XE to Develop for iPhone, iPod Touch and iPad

Brian Long Consultancy & Training Services Ltd

February 2011

Americas Headquarters
100 California Street, 12th Floor
San Francisco, California 94111

EMEA Headquarters
York House
18 York Road
Maidenhead, Berkshire
SL6 1SF, United Kingdom

Asia-Pacific Headquarters
L7. 313 La Trobe Street
Melbourne VIC 3000
Australia

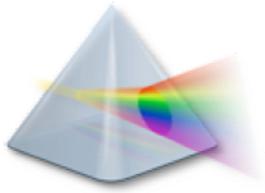
EXECUTIVE SUMMARY

The iPhone® is clearly a very successful smart-phone and the ability to develop applications for iPhones opens up a new sector for developers. Initially, iPhone application development was only within the reach of Objective-C programmers directly using Apple's CocoaTouch framework, but this is no longer the case.

Embarcadero's Delphi Prism®, in conjunction with Mono® and MonoTouch from Novell®, provides all you need to develop and debug native applications for deployment to iPhone. This white paper looks at the process of building iPhone applications with Delphi Prism, exploring various techniques and common application features.

CONTENT

INTRODUCTION



Delphi Prism XE is the latest release of Embarcadero's development environment for .NET and cross-platform Mono. It is available as either a standalone product or as part of Embarcadero RAD Studio XE. Delphi Prism includes an Object Pascal compiler (the RemObjects Oxygene compiler) that targets Microsoft's .NET platform on Windows® (and the Mac® if you are building Silverlight® applications). Delphi Prism's support for .NET constructs is very much up-to-date with all the current features in the C# language and offers various extensions to the familiar Delphi implementation of Object Pascal.

When building .NET applications on Windows, Delphi Prism is typically used within Visual Studio® after installation. If a Visual Studio installation is already present when Delphi Prism is installed, it integrates with that, else the Visual Studio shell is installed.

As well as targeting Microsoft's .NET, Delphi Prism also compiles applications for Novell's Mono platform (http://www.mono-project.com), meaning .NET programming skills can progress your application code base from solely targeting Windows to working against Linux and Mac OS X as well, on various hardware platforms. Mono includes and supports various toolkits and libraries to support the UIs and technologies available on these platforms.

When developing Mono projects you can work within Visual Studio, as above, or you can work within the dedicated Mono development environment, MonoDevelop (http://monodevelop.com). Whilst this is a free and open source tool (originally based on the SharpDevelop open source editor), you are advised not to download it directly, as Delphi Prism has been integrated into a specific build (see Installing MonoDevelop later).

The traditional route to iPhone development involves using Apple development tools on a Mac. This means programming in Objective-C in Xcode® in combination with UI development in Interface Builder to build a native iOS[1] application that can be tested in

[1] Where OS X is the operating system on a Mac, iOS is the operating system on an iPhone, iPod Touch and iPad, so an iOS device could be an iPhone, an iPad or an iPod Touch. In this paper, use of the term iPhone typically means any iOS device.

the iPhone simulator and then deployed to an iPhone for further testing before optionally going to Apple's App Store℠.

MonoTouch To bypass the learning curve of Objective-C and retain your .NET programming skills, an alternative path involves building your application using a .NET language with Mono (still employing the Interface Builder UI step) and using Novell's MonoTouch toolkit.

MonoTouch was originally launched in September 2009. It offers several things to facilitate generating iPhone applications from Mono applications. Firstly it provides the managed bindings to Apple's Cocoa Touch® API (CocoaTouch is the touch-oriented version of Apple's Cocoa® UI library, as used on the iPhone), sometimes referred to as CocoaTouch.NET. Secondly it provides a number of C# application templates targeting iPhones (and iPad® and iPod Touch®). Finally it incorporates an AOT compiler[2] that turns the managed, normally JIT-compiled Mono code into native ARM code, stripping out as much of the Mono library code that can be identified as never being called (what the Delphi compiler calls smart linking) and combining the executable and all the dependant libraries into one single executable.

This AOT compilation and native code generation is necessary as Apple prohibits dynamic code generation, JIT compilation and shared libraries on iOS devices.

Note: your application starts off as a managed Mono application, with the Mono runtime environment and all the trimmings. The smart linker will get rid of a lot of redundant code, but it is fair to say your MonoTouch application will still be noticeably larger than an equivalent application written directly in Objective-C.

At one point Apple also prohibited any application not built with Apple development tools (essentially anything other than Objective-C) but fortunately that is all in the past now and MonoTouch applications are welcomed onto the App Store.

MonoTouch itself offers support for C# applications but Delphi Prism adds in templates to help kick-start development of various types of applications on the iPhone, iPhone Touch and iPad.

Note: Delphi Prism can run on Windows within Visual Studio or MonoDevelop, however MonoTouch (and the Interface Builder tool from Apple's Xcode suite) requires you to be working on a Mac.

[2] AOT is Ahead Of Time as opposed to JIT or Just In Time

GETTING STARTED

Assuming you have an Apple Mac you will need to install several things to start developing iPhone applications with Delphi Prism.

INSTALLING IOS SDK

The iPhone SDK (also known as the iOS SDK) is free, but you are required to register yourself on the site first. Registering involves answering some questions on what markets and platforms you develop for and then clicking a link in a verification email you'll receive. You are then taken to the iOS Dev Center and sent another email confirming your Apple ID.

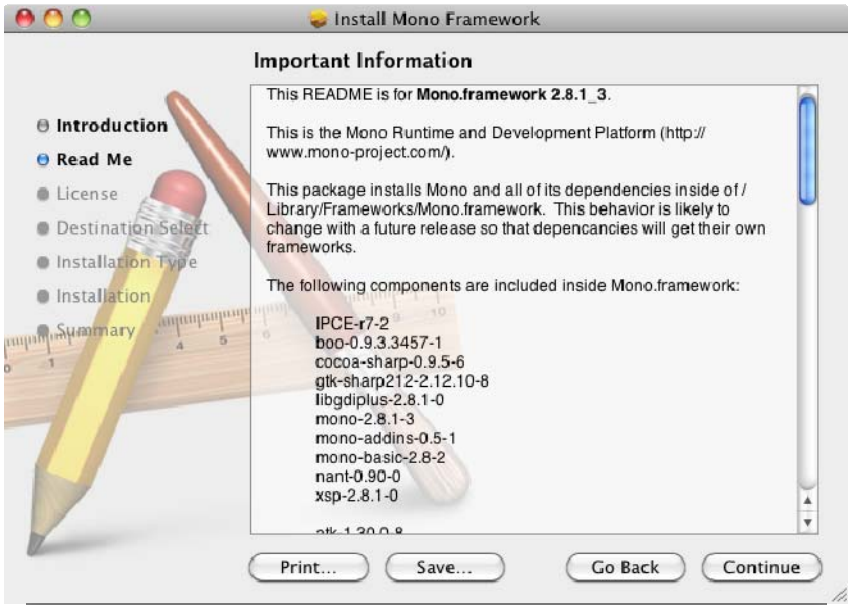
From this point on, you can get back to the Apple Dev Center at http://developer.apple.com/iphone, which you will likely want to in order to access the various reference materials and guides. Of course any programming documentation or sample code will be in Objective-C, but that needn't be a complete stumbling block.

On this page, you can download the iOS SDK. The download is listed as *Xcode and the iOS SDK* combining two development kits: the Xcode development environment (which includes Interface Builder) as well as the iOS SDK (versions 3.2.5 and 4.2 respectively, at the time of writing). The fact that this is a combined download adds some unnecessary weight to the file if you already have Xcode installed, as it's a 3.5GB download.

Note: Whilst the development tools and SDK are free, in order to deploy to a device or to the App Store you must be enrolled in Apple's iPhone Developer Program (http://developer.apple.com/programs/ios), which costs $99 per year. It costs nothing to run your applications in the iPhone simulator though.

INSTALLING MONO

You can find Mono for OS X at http://mono-project.com/Downloads - the latest stable version at the time of writing is 2.8.1_3. The installation is, as is to be expected on a Mac, trivial.



INSTALLING MONODEVELOP

MonoDevelop's home page is at http://monodevelop.com, however if your goal is to develop with Delphi Prism then you should ignore this site, as the Delphi Prism download includes a dedicated copy of MonoDevelop with Delphi Prism integrated into it.

INSTALLING DELPHI PRISM

You can download the Delphi Prism Mac installer from the link at either https://downloads.embarcadero.com/free/delphi_prism (trial version - you fill in a form and are sent a registration code) or http://cc.embarcadero.com/reg/delphi_prism (if you hold a valid license for Delphi Prism XE). In the zip file is the customized MonoDevelop application, so just copy it to ~/Applications, but don't launch it just yet!

INSTALLING MONOTOUCH SDK

The next step is to get the MonoTouch SDK from http://monotouch.net/Store. MonoTouch is a commercial product and you need to buy a license in order to deploy onto a device, though a trial version will let you run on the iPhone simulator. A single user license for the professional version currently costs $399 and includes all released updates for 1 year.

Download and install the trial version, which requires you to supply your email address.

BEFORE MOVING ON

Now it's time to launch MonoDevelop for the first time. You will be asked to follow a link[3] in order to register a serial number (trial or full) along with the registration code (such as the one you were sent when requesting the trial version). Register and download the license file, import it into the waiting MonoDevelop 2.4 dialog and it will start up.

Note: when MonoDevelop starts it will check, by default, to see if it is the latest version, which it won't be. The Delphi Prism archive contains MonoDevelop 2.4 but the latest version (at the time of writing) is the bug fix release, version 2.4.1. You can download the update if you wish (it is a standard .dmg disk image) but it is important to **not** proceed to install it over the old version otherwise you will lose the Delphi Prism integration.

Instead, first quit MonoDevelop and rename it from MonoDevelop to MonoDevelop2. Now you can install the update into your home directory's Applications subdirectory. In order to copy the Delphi Prism integration files from the original MonoDevelop to the updated version you can run a couple of commands in a terminal window:

```
cd ~/Applications
cp -R MonoDevelop2.app/Contents/MacOS/lib/monodevelop/AddIns/Oxygene ↵
      MonoDevelop.app/Contents/MacOS/lib/monodevelop/AddIns
```

MonoDevelop is now Prism-enabled and MonoDevelop2 can be deleted (or ignored).

Next, to ensure you have all the latest templates you should follow the instructions at http://www.remobjects.com/oxygene/prismextras.aspx that show how to have MonoDevelop pull the updates from the RemObjects MonoDevelop repository.

If you come from a Windows programming background you might have trouble getting used to some of the keystrokes in the MonoDevelop text editor. If so you should use the

[3] The next release of Delphi Prism will use a dialog at this point within MonoDevelop instead of taking you to a web page.

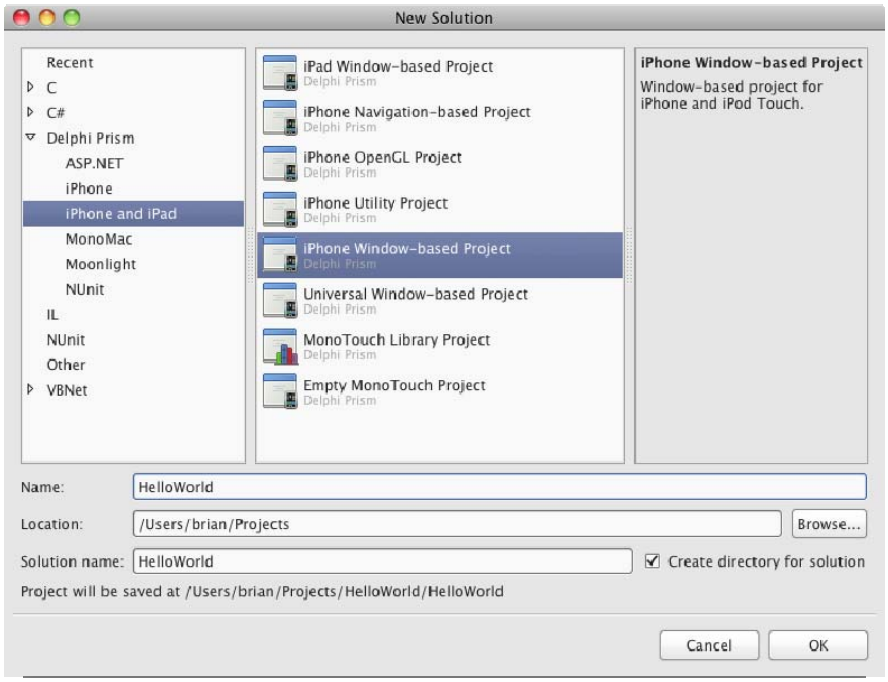
MonoDevelop preferences dialog and remap some of the editor commands to keystrokes you find more natural.

Also, after a few hours use, sometimes the Solution window in the open source MonoDevelop environment becomes unresponsive or the editor may throw an exception causing Code Completion, etc. to stop working. If this happens, just close MonoDevelop and restart it.

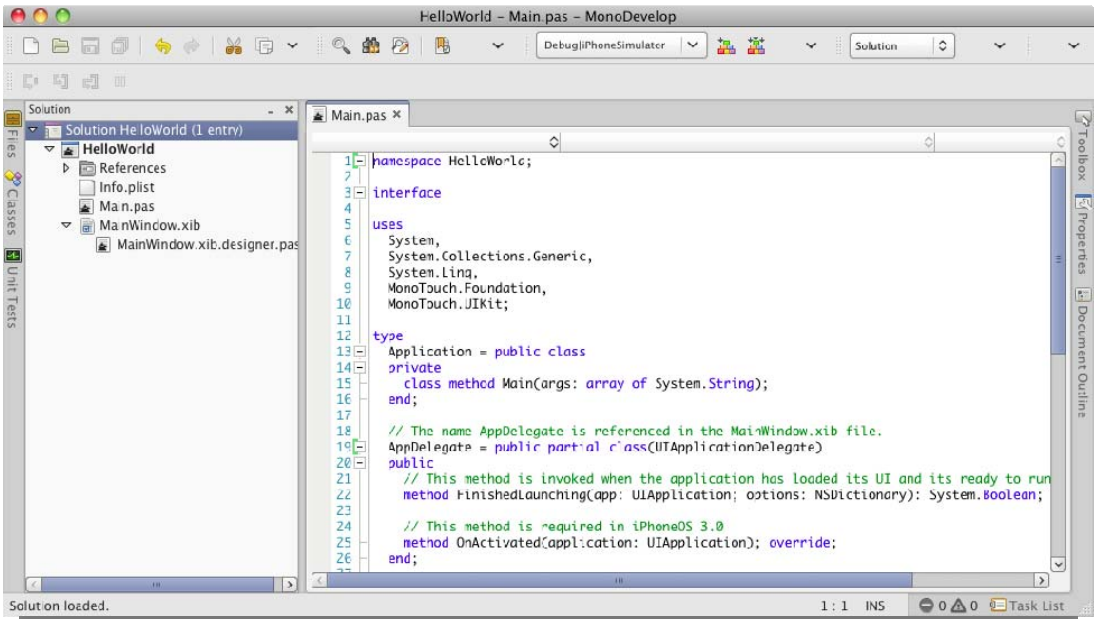
Now at last you're set, so let's get started!

Developing Applications With MonoTouch

For our first foray into iPhone application development, tradition dictates we do a Hello World program. In MonoDevelop choose File, New, Solution… (or press ⇧⌘**N**) or press the *Start a New Solution* link on the welcome screen. Here you can see the various Delphi Prism templates available. We'll start off with a window-based project, since a Hello World will just have a single screen.



This creates a solution[4] containing a single project made up of various files.



Info.plist is a property list file used to set various properties of interest to iOS. We'll ignore this for now.

The important file is Main.pas; this is where we will be writing code. The simple `Application` class contains a class method `Main` that is the application's entry point. `AppDelegate` is a bit more interesting – it represents a delegate for the underlying CocoaTouch Application object and can respond to Application events, such as the `FinishedLaunching` event that triggers when the app has loaded up and which typically contains initialization code.

CocoaTouch operates with an MVC model and this delegate model crops up regularly, for example with event handlers for UI controls.

Talking of UI controls, we need to set up a UI. The MainWindow.xib[5] file represents the UI; when double-clicked it will be opened in Interface Builder, a tool you'll be spending some time with as you work with MonoTouch projects.

---

[4] As with Visual Studio a *solution* is a means of managing potentially multiple projects, much as a Delphi project group is.

Just before looking at Interface Builder I'll point out the other source file in the project. MainWindow.xib.designer.pas is an auto-generated file that contains code necessary to access parts of the UI, referred to as the .xib's *code-behind file*. We'll take a look at it again presently.

INTERFACE BUILDER AND THE UI

Interface Builder is one of the tools from Apple's Xcode development tool suite and is where we build the UI design part of the iOS application. Which solution/project template you start with dictates the size of the window you get to design, as the resolutions are different. iPhone and iPod Touch are 320x480[6] whereas iPad is 1024x768.

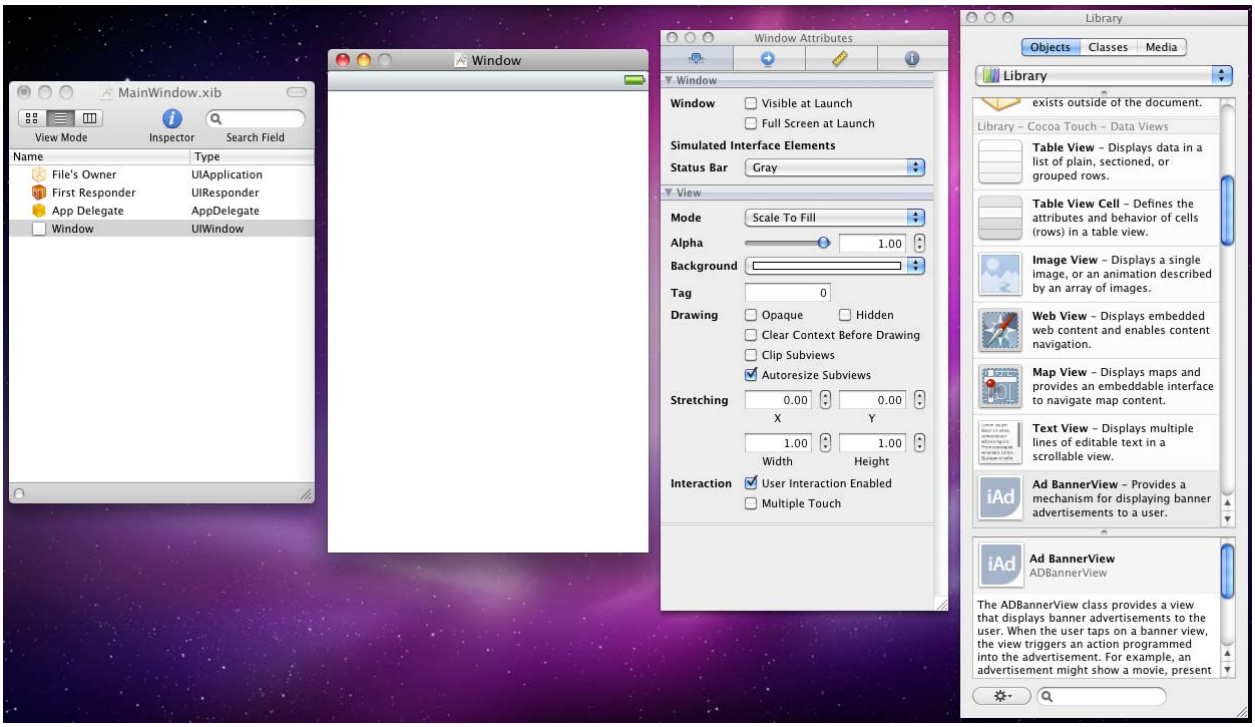
When your .xib file opens in Interface Builder you are presented with four windows, as illustrated in the screenshot below. We will need to gain familiarity with each of these so let's look at them one at a time.

The first one is the Document Window and lists the key items from the .xib file and allows them to be selected. The important ones to note right now are the App Delegate, which tallies with the `AppDelegate` class in the source file from earlier, and also the Window, which represents your application window (of which you have one in a new window-based project).

The second one is your application's window and it is made active if you double-click the Window item in the Document Window. This window is sized appropriate for your iOS target device (in this case the window is 320x480). If you look carefully at the screenshot, you'll see that at the top of the window is a representation of the iPhone's status bar. This window is essentially a form designer, and will reflect our UI as we build it up by adding controls and views.

[5] Originally, Apple UI files were binary and used a .nib extension. When building iOS applications the UI files are XML and use the .xib extension. It is common for OS X and iOS developers to refer to both .xib and .nib files simply as *nib files*.

[6] iPhone 4 has a higher resolution screen than iPhone 3 at 640x960. That is an increase in pixel resolution but, when measured in points, all iPhones offer a 320x480 resolution. This means you only need to design the UI once and it works on all iPhones.



The third window is the Inspector and serves the same purpose as an Object Inspector or Properties window. It has 4 pages, selectable by the buttons at the top or by menu items. If you check the Tools menu you'll see that ⇧⌘**1** selects the Inspector window, with whatever tab is currently active. However ⌘**1**, ⌘**2**, ⌘**3** and ⌘**4** select the Attributes Inspector, Connections Inspector, Size Inspector and Identity Inspector respectively, as the four buttons also do.

The Attributes Inspector offers up miscellaneous properties and the Size Inspector lets you play with the size and anchoring (as Delphi would call it) of the selected control. The Connections Inspector is where we'll hook up outlets and actions, as we'll see momentarily.

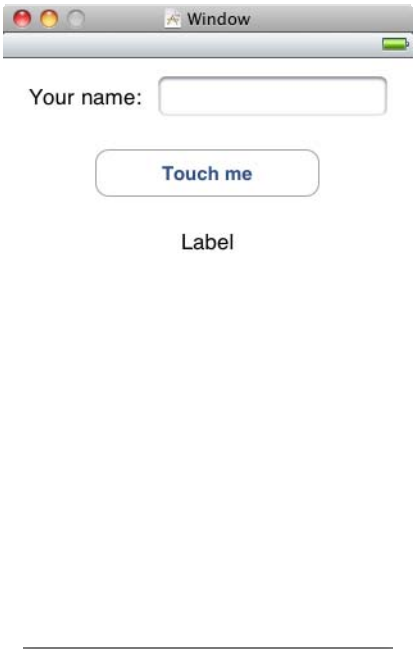
The fourth window is the Library. Again, buttons at the top switch its mode. The *Objects* button shows all the objects you can add to the window (rather like Delphi's Component Palette) and the *Classes* button lists all the classes available (the *Media* button is unimportant for our purposes). In either mode there is a Search box at the bottom of the window that filters the long lists shown by default.

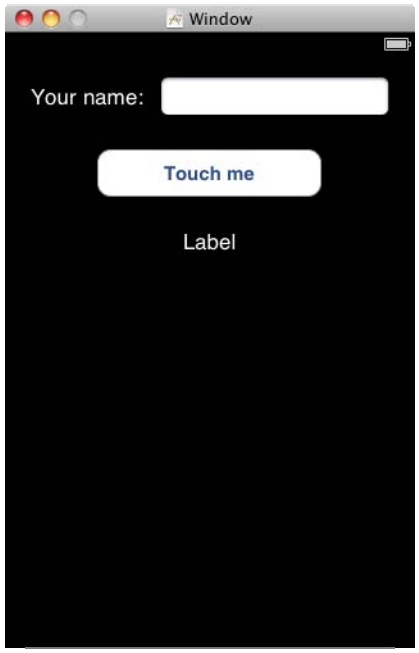
The UI of this starting app will require two Labels (`UILabel` controls), a Round Rect Button (`UIButton`) and a Text Field (`UITextField`). To find these input controls in the Library window ensure the *Objects* button is selected then use the drop-down control to show only *Inputs & Values* (as opposed to the full library of CocoaTouch and custom objects,

which is selected by default). This cuts down the list considerably and so you should be able to find the controls readily – they are all adjacent in the list. Again, you can also use the search box to search for the class name or the description (though that resets the drop down filter to show the whole library once more).

Drag the controls to the form and arrange them as below. You can edit the text in all these controls either by double-clicking the control, or using the Text or Title attributes, as appropriate, in the Attributes Inspector (⌘1). If you prefer dark backgrounds, you can also set the text color for the labels and the background color of the window itself. The window attributes also let you set the color of the iPhone's status bar[7].

The text field will automatically pop up a keyboard when tapped[8], and there are various attributes we can configure in the Text Input Traits section of the Attributes Inspector. In this case set it to capitalize words and set the Return Key attribute to Done, which changes the normal *Return* button on the keyboard to be a highlighted *Done* button instead.





[7] That setting is described in Interface Builder as a Simulated Interface Element, so shows you what it might look like, but won't have an effect at runtime. To finish the job we need to do it in code, and we will.

[8] We shall see later that, whilst the keyboard will automatically pop up, it is down to the programmer to dismiss it.

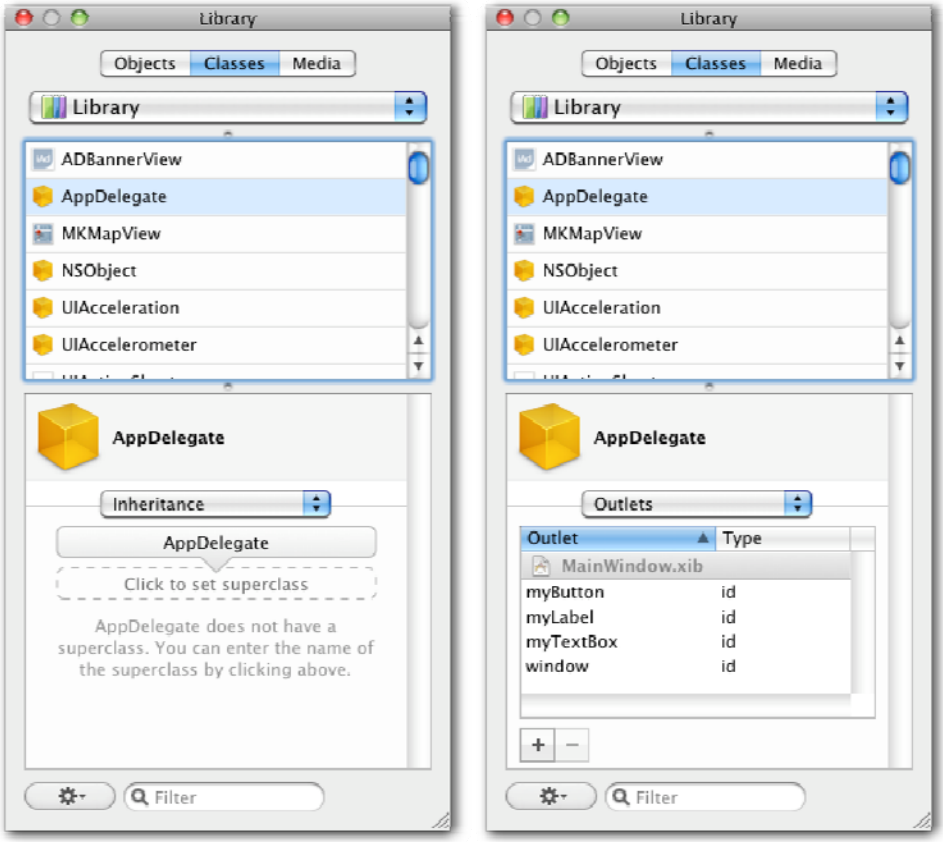
That's the UI designed, but before leaving Interface Builder we need to cater for the programming that comes next.

OUTLETS

The code will need to read from the text field and write to the bottom label (and also, just to prove we can, we'll be writing to the button as well). In order to access the controls we need to define the variables that will refer to them, which are not created by default. These variables are called *outlets* and are defined in the Library window on the *Classes* page.

You'll recall that in this application the App Delegate is where the code will be added, so we need to add the outlets to this class. To locate the App Delegate on the *Classes* page of the Library window, either use the drop-down box, which lists all available classes, and scroll around till you find it, or scroll up and down the main list on the window, or alternatively you can use the search box.

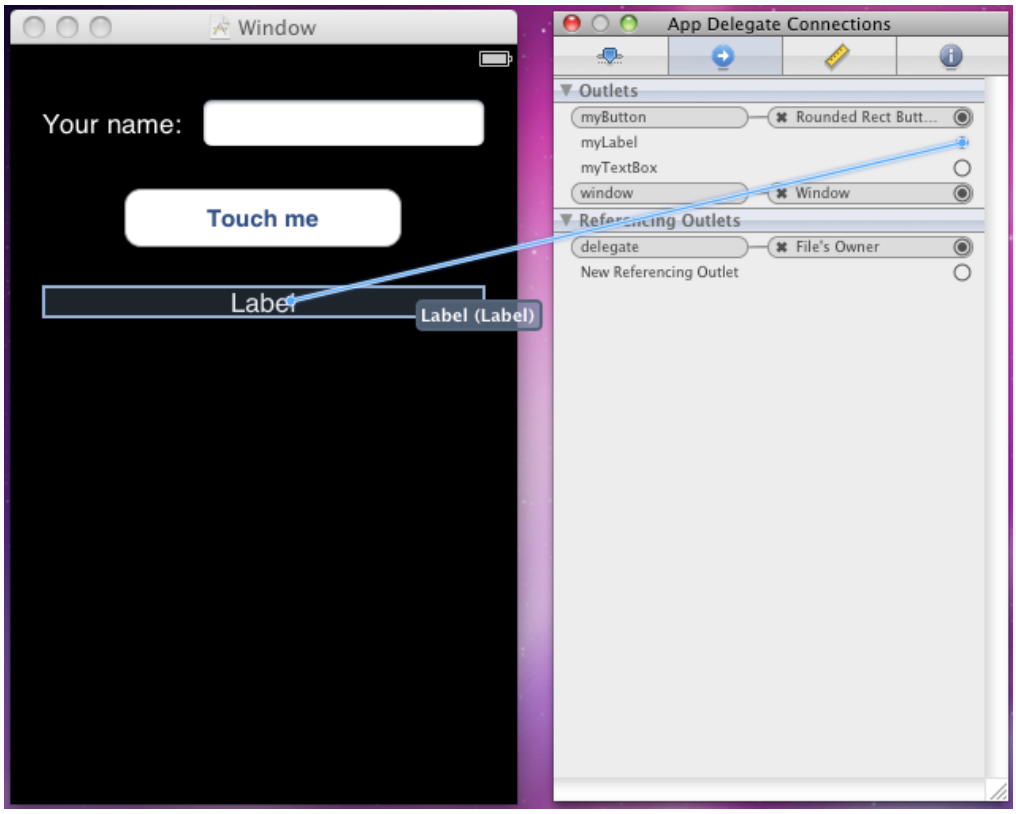
When selected you see an uninformative inheritance diagram in the lower half of the window. Using the drop down box in the center of the window switch to the Outlet view, where you'll see one outlet already defined for the window object. Now use the **+** button to add in an outlet for each control of interest.



The variables are now defined but we haven't told Interface Builder what they each represent so we need to connect the outlets to the controls. You do this by selecting the object that defines the outlets (select the App Delegate in the Document Window) and then use the Connections Inspector (⌘2). You'll see all the outlets listed in the Connections Inspector and the window outlet shows it is connected, unlike our new outlets.

To connect an outlet to a control you move your mouse over the little circle to the right of the outlet whereupon it turns into a plus, then you drag from there and drop onto the control. You can see the label outlet being connected below.

Set the three outlets up and then we can consider what happens with events.



CocoaTouch Actions and Events

The CocoaTouch controls available in Interface Builder have various events that can be responded to, as you'd probably expect. However when using MonoTouch we have a choice of two ways to set up the event handler. One way requires an outlet to be set up in Interface Builder and then uses normal .NET-style events in the source code. The second way matches the traditional Cocoa programming model and revolves around *actions* and we'll look at this approach just now.

An action represents a method that gets implemented in your class but is connected to a control's event in Interface Builder. You set up an action in Interface Builder in a similar way to defining an outlet - the Library window allows you to choose an Actions page for a selected class. We'll need to add an action to the App Delegate.

Note: these actions are sent around from within the underlying CocoaTouch library (the event handler method in your code maps onto it) and so have the same rules as when programming in Objective-C. Since all these events tend to have parameters the requirement is to ensure the action has a colon as a suffix character, so in this case you could add an action called `myButtonPressed:` and that will work out okay. Omitting the colon won't cause an error, but the code won't execute as you would expect.

To hook the action up to an event you can do it in one of 2 ways. Firstly, you can locate the action in the Connections Inspector for the App Delegate (it's in the Received Actions list) and then drag it to the relevant control (the button). This will produce a popup list of all the events (see screenshot to the left) and the one we probably want is the Touch Up Inside event, such that it triggers when the user touches the button and then moves their finger away.

The other way to hook up the action is to select the button, so that the Connections Inspector shows all the available events in a list, and then drag the required event to the App Delegate on the Document Window. This produces a popup containing the list of available actions, in this case containing just the one item.

Either way, if you make the connection and save the .xib file (⌘**S**) then we can switch over to MonoDevelop and get on with the code-writing side of things.

Event Handlers For CocoaTouch Actions

Switching back to MonoDevelop causes the code behind file to be regenerated so let's take a look at it now (listed below with some unimportant lines removed for brevity). What we have here in the code behind file is a partial class definition for the `AppDelegate` class that is also partly declared in Main.pas. A MonoTouch attribute has been used to ensure

this Mono class is bound to the Objective-C `AppDelegate` class down at the CocoaTouch level.

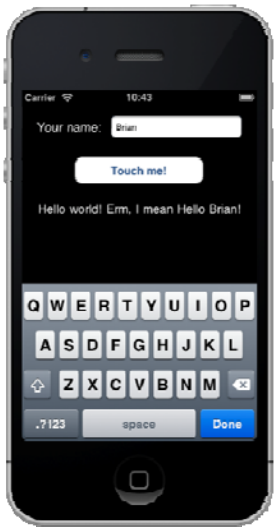
You can see a declaration for the event handler method `myButtonPressed` is there, with another attribute to bind it to the Objective-C action. Note the partial and empty keywords that allow the declaration to be left here without an implementation. We will implement the method in Main.pas. The rest of the class consists of the properties that represent the outlets (along with setters, getters and private variables in the real code) along with yet more binding attributes.

```
[MonoTouch.Foundation.Register('AppDelegate')]
AppDelegate = public partial class
Private
  [MonoTouch.Foundation.Export('myButtonPressed:')]
  method myButtonPressed(sender: MonoTouch.UIKit.UIButton); partial; empty;
  [MonoTouch.Foundation.Connect('window')]
  property window: MonoTouch.UIKit.UIWindow
    read get_window write set_window;
  [MonoTouch.Foundation.Connect('myButton')]
  property myButton: MonoTouch.UIKit.UIButton
    read get_myButton write set_myButton;
  [MonoTouch.Foundation.Connect('myLabel')]
  property myLabel: MonoTouch.UIKit.UILabel
    read get_myLabel write set_myLabel;
  [MonoTouch.Foundation.Connect('myTextBox')]
  property myTextBox: MonoTouch.UIKit.UITextField
    read get_myTextBox write set_myTextBox;
end;
```

Switching over to Main.pas we need to add in the declaration and implementation of the event handler in this partial class:

```
type
  AppDelegate = public class
  ...
  private
    method myButtonPressed(sender: MonoTouch.UIKit.UIButton); partial;
  end;

method AppDelegate.myButtonPressed(sender: MonoTouch.UIKit.UIButton);
begin
  myLabel.Text := 'Hello world! Erm, I mean Hello ' + myTextBox.Text + '!';
end;
```

Straightforwardly the code looks like regular Delphi code. You can check the project builds using the options on the Build menu (also ⌘**K** builds the active project in the solution while ⌘**B** builds all projects in the solution).



iPhone Simulator

Now is the time to test the application. You can run the application from MonoDevelop with the Run menu or ⌥⌘↩ (that's **Option-Command-Enter**, or **Alt-Apple-Enter** for those new to the Mac keyboard) and it will be launched in the iPhone Simulator. You can interact with the application using the mouse instead of your finger. If you were implementing multi-touch you can mimic two finger touches by holding down the **Option** (**Alt**) key.

Note: The iPhone Simulator is a *simulator*, not an *emulator*. As a consequence you are running regular Intel code, not ARM code, and for this and various other reasons the performance and behavior of your application may be very different than when deployed on a real device.

When you tap the text field a keyboard obligingly pops up with a highlighted button inviting you to press *Done* when you have entered your name. Unfortunately nothing yet happens when you press it – that's something we have to take responsibility for. However the *Touch me* button works fine.

We should add in some initialization code to blank out the greeting label on startup (or clear it in Interface Builder. We'll also alter the caption of the button (just add a character on the end) and deal with this keyboard. All this code is going in the overridden `FinishedLaunching` method. The call to `window.MakeKeyAndVisible()` is where the screen is told to display so the startup code will be placed just before that. Clearing the label is trivial but the button caption (or title) is a little more obscure. It turns out that you need to get the current 'normal state' title and then separately set an updated 'normal state' title.

```
myLabel.Text := '';
myButton.SetTitle(myButton.Title(UIControlState.Normal) + '!',
  UIControlState.Normal);
```

Sometimes you need to browse through the Code Completion window to see which methods or properties are available. Sometimes you need to trawl through the MonoTouch documentation (a pretty good work in progress at http://monotouch.net/Documentation). Sometimes you need to go back to the real documentation for CocoaTouch on the Apple web site (http://developer.apple.com/library/ios). The syntax will look a little odd at times, but it's all about methods, properties and so on, so acting as a reference it's still invaluable when required.

One thing before moving on; if you tried to set the iPhone status bar to be opaque black in Interface Builder you will have noticed that the simulator ignored that setting (for no good reason), unlike the screenshot above. This can be overcome by setting it in code instead. In the same `FinishedLaunching` method add this code in towards the top:

```
UIApplication.SharedApplication.StatusBarStyle := UIStatusBarStyle.BlackOpaque;
```

Note: `UIApplication.SharedApplication` is how you access the underlying Application object from anywhere in your code.

Text Entry Keyboards

As mentioned we need to tell the text field to close the keyboard when we decide it is necessary. But how do we find out how to deal with this? Well, just searching the Internet is a good start as there is quite a large amount of information on common problems that stump MonoTouch programmers and of course even more for CocoaTouch programmers. But let's walk through a search of the documentation for how to find the answer and for the sake of it we'll start with Apple's reference site.

Using the Documentation

At http://developer.apple.com/library/ios you will find a list of all the iOS frameworks (CocoaTouch is an umbrella term for a few of them). The class we are looking at is `UITextField` and the prefix letters tell us it is part of the UIKit framework, so clicking on *UIKit* gives a whole list of potential reference topics to choose from including *UITextField Class Reference*. Clicking on that gives a big, detailed reference page, but in the Overview section it says:

"A text field object supports the use of a delegate object to handle editing-related notifications. You can use this delegate to customize the editing behavior of the control and provide guidance for when certain actions should occur. For more information on the methods supported by the delegate, see the UITextFieldDelegate protocol."

We'll need to follow the link as we are looking for an editing-related thing that we want to customize. Typically the MonoTouch layer will merge this type of optional delegate object functionality into the main object in question, so the methods of `UITextFieldDelegate` will be implemented in MonoTouch's `UITextField` object as delegate properties to save you the chore of building a delegate class to customize the text field object. However, the delegate class is still available as an option if you want to use a separate delegate object.

The `UITextFieldDelegate` page describes the methods (or messages as they are called in Objective-C) supported by the delegate. If you browse the information you'll see the message `textFieldShouldreturn:` is the one.

Now let's have a look at the MonoTouch version of this. At http://www.go-mono.com/docs start expanding the reference tree along this node path: *MonoTouch Framework*, *MonoTouch.UIKit*, *UITextFieldDelegate Class*, *Methods*. Of the methods listed the one we need is called `ShouldReturn` – as you see, MonoTouch also simplifies some member names. Select this method and the page tells you that the method signature is a function that takes a `UITextField` and returns a Boolean indicating if the keyboard should do its default behavior when *Return* (or *Done*) is pressed.

So this shows the Apple and MonoTouch documentation of the same delegate object property, but as mentioned MonoTouch absorbs the delegate object into the text field. Check the help for the *UITextField Class* and you'll find a `ShouldReturn` property that can reference a method and this is what we'll use.

The help indicates that we can assign a regular method to this property (with the right signature), but it also says an anonymous method can be used. Anonymous methods are convenient as they save a bit of typing. If the functionality is not excessive and not required elsewhere an anonymous method is a sensible option. We can also potentially reduce typing a little further by using a lambda, but before worrying about what that is let's consider what the `ShouldReturn` functionality needs to in order to dismiss the keyboard.

First Responders

The closest iOS equivalent to a focused control that receives input in Windows is a first responder. When you tap the text field it becomes first responder and displays the keyboard. Dismissing the keyboard is simply a matter of dropping this first responder state.

```
myTextBox.ShouldReturn := method(textField: UITextField): Boolean
                          begin Result := textField.ResignFirstResponder end;
```

`ResignFirstResponder` returns True if its first responder status was lost and so this value is returned from the anonymous method, meaning the text field should process the press of *Return*. The anonymous method above could be laid out to look more like a traditional method implementation, but there is no real need or benefit. It can also be compressed a little and turned into a lambda:

```
myTextBox.ShouldReturn := (textField) ->
  begin Result := textField.ResignFirstResponder end;
```

The body of the lambda contains a statement so the `begin`/`end` pair is required. However if the body is an expression whose type is the same as the target function return type we can simplify even further.

```
myTextBox.ShouldReturn := (textField) ->  textField.ResignFirstResponder;
```

All three options are identical in meaning, although you'll notice that explicit parameter type information was removed in the lambda – the compiler is able to work this out through the type of the delegate property on the left hand side.

Event Handlers For MonoTouch Events

Just before re-launching the application in the simulator to check it works let's add in another event handler. You may recall from earlier that there were 2 ways of setting up event handlers for UI controls and we looked at the approach that used actions. We'll add another event handler onto the same *Touch me* button, but this time solely in code, just using the event of the text field object.

```
method AppDelegate.InfoAlert(Msg: String);
begin
  using av := new UIAlertView('Info', Msg, nil, 'OK', nil) do
    av.Show
end;
...
  myButton.TouchUpInside +=
    method begin InfoAlert('Hello ' + myTextBox.Text) end;
```

So event handlers (which, of course, are multi-casting in .NET and Mono) can be added using the **+=** operator in conjunction with a method, anonymous or otherwise. This event handler pops up an alert (the closest equivalent of a message box) via a **`UIAlertView`** object.

Note: the **`using`** statement, like C#'s, is an abbreviated way of ensuring that **`Dispose`** is called, to free up the Objective-C resources when we know we are done with them. The above method is exactly the same as this slightly longer version.

```
method AppDelegate.InfoAlert(Msg: String);
begin
  with av := new UIAlertView('Info', Msg, nil, 'OK', nil) do
    try
      av.Show
    finally
      av.Dispose
    end
end;
```

Note: Objective-C typically requires explicit memory management, much like regular Delphi Win32 programming. Mono and .NET, however, have a garbage collector (GC) and so tidy up after you. You can merrily create your objects and leave them for the GC to collect when it gets round to it, but if you want to be more responsible with the limited

memory on the device you can call the `Dispose` method on any objects you know are no longer required. This will cause the underlying Objective-C object to be released.

If you test the application, it now behaves more as expected. The keyboard's *Done* button operates and the button now does 2 things (updates the label and pops up an alert). Interestingly, pressing the button does not dismiss the keyboard, as the text box remains the first responder. You can 'fix' this by adding this condition into one of the button's event handlers

```
if myTextBox.IsFirstResponder then
  myTextBox.ResignFirstResponder;
```

VIEW CONTROLLERS

The simple application above has all the functionality in the App Delegate, which purists might suggest is best left to act as a delegate for the CocoaTouch `Application` object. Typical applications are more likely to use one or more view controllers (`UIViewController` or a descendant) as delegates for views on the various windows in the application. You get a view controller in the application if you start with the iPhone Navigation-based Project template or iPhone Utility Project template in MonoDevelop. Let's make a new Navigation-based project.

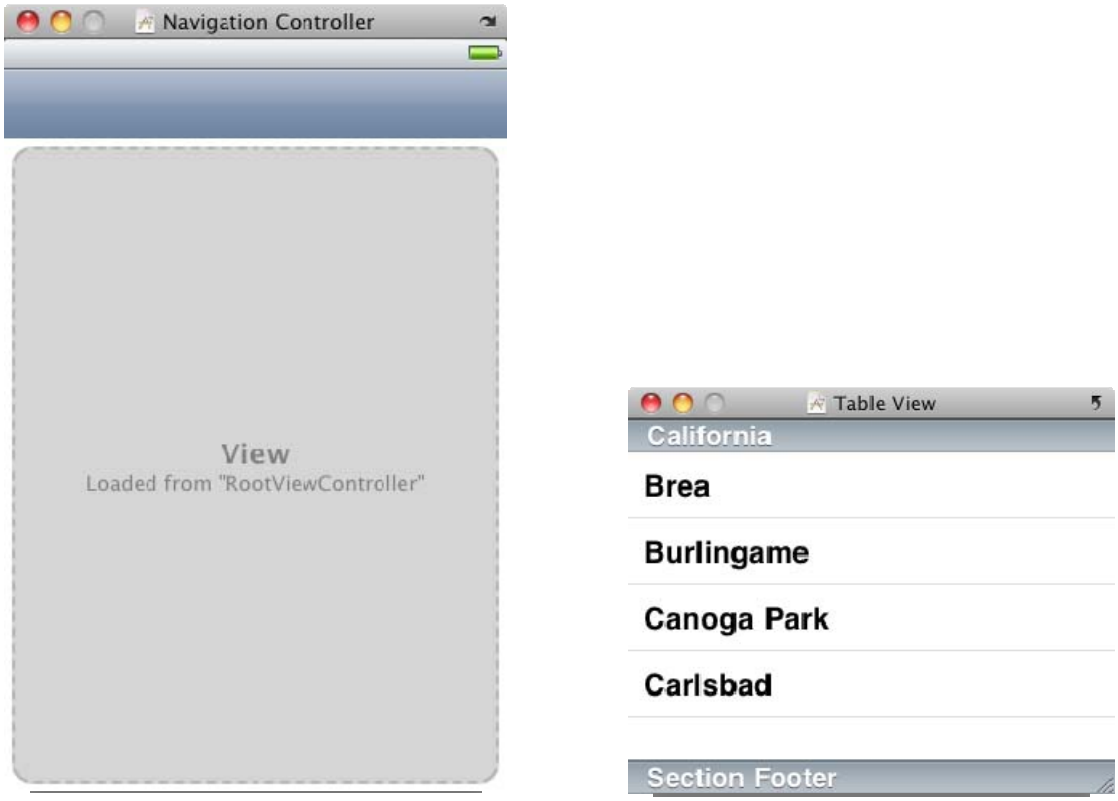
The project we get from this template has a window and an App Delegate as before, but importantly also has a Navigation Controller, which works with a Navigation Bar. The idea of this is to support the common workflow in an application of going from one screen to another, and then maybe to another, etc., and being able to readily navigate back to any of those earlier screens. iPhones facilitate this using a Navigation Bar under the control of a Navigation Controller. The Navigation Bar reflects which screen you are on, where each of the navigable screens is actually a `UIView` descendant.

Note: when you double-click MainWindow.xib there are potentially two UI windows opened up by Interface Builder, as the .xib file defines both the main window, which is completely blank, and also the Navigation Controller, which has the Navigation bar etc. on. You can readily open up whichever one you choose using the Document Window.

The template sets us up a `UITableView` as a starting view with a corresponding `UITableViewController`, suitable for showing a very customizable list in a manner iPhone users will be very familiar with. As items are selected in the table (or list) the application has the option to navigate to other pages.

When you look at the two .xib files in Interface Builder you see the blue Navigation Bar at the top of the main window (you can give it some text by double-clicking it) as well as an

indication that the rest of the window content comes from RootViewController.xib. This latter .xib file just contains a Table View, which is shown populated with sample data.



We'll see how this `UITableView` works by displaying some information from an SQLite database. The coding will take place in the source file that is associated with the Table View .xib: RootViewController.xib.pas (not to be confused with the code behind file, RootViewController.xib.designer.pas).

USING SQLITE

Before worrying about the table, we'll get some code in place to create a database, a table and some sample data when the main Table View is loaded. To keep things tidy we'll also delete the database when it unloads, though clearly a real application may need to keep its database around between invocations. The contents of the database table will be read from the database and stored in a strongly typed list. Again, consideration should be given to memory requirements in a real application; in this sample there will only be a handful of records.

Since the list is to be strongly typed we'll need a type to represent the data being read:

```
Customer = public class
public
  constructor;
  property CustID: Integer;
  property FirstName: String;
  property LastName: String;
  property Town: String;
end;
```

The `ViewDidLoad` and `ViewDidUnload` overridden methods are already present in the template project so here's the extra code that uses standard ADO.NET techniques with the Mono SQLite database types:

```
uses
  ...
  System.Collections.Generic,
  System.Data,
  System.IO,
  Mono.Data.Sqlite;
...
  connection: SqliteConnection;
  dbPath: String;
  customerList: List<Customer>;
...
method RootViewController.ViewDidLoad;
const
  TblColDefs = ' Customers (CustID INTEGER NOT NULL, FirstName ntext,
                LastName ntext, Town ntext)';
  TblCols = ' Customers (CustID, FirstName, LastName, Town) ';
begin
  inherited ViewDidLoad();

  //Create the DB and insert some rows
  var documents := Environment.GetFolderPath(
    Environment.SpecialFolder.Personal);
  dbPath := Path.Combine(documents, 'NavTestDB.db3');
  var dbExists := File.Exists(dbPath);
  if not dbExists then SqliteConnection.CreateFile(dbPath);
  connection := new SqliteConnection('Data Source=' + dbPath);
  try
    connection.Open;
    using cmd := connection.CreateCommand() do
    begin
      cmd.CommandType := CommandType.Text;
      if not dbExists then
      begin
        var statements: array of String := [
          "CREATE TABLE" + TblColDefs,
          "INSERT INTO" + TblCols +
          "VALUES (1, 'John', 'Smith', 'Manchester')",
           ...
           ];
        for statement in statements do
        begin
```

```
          cmd.CommandText := statement;
          cmd.ExecuteNonQuery;
        end;
      end;
      customerList := new List<Customer>;
      cmd.CommandText := 'SELECT CustID, FirstName, LastName, Town ' +
        'FROM Customers ORDER BY LastName';
      using reader := cmd.ExecuteReader do
      begin
        //read customers from DB into customerList
        while reader.Read do
        begin
          var cust := new Customer;
          cust.CustID := Convert.ToInt32(reader['CustID']);
          cust.FirstName := String(reader['FirstName']);
          cust.LastName := String(reader['LastName']);
          cust.Town := String(reader['Town']);
          customerList.Add(cust);
        end
      end
    end
  finally
    connection.Close
  end;
  self.TableView.Source := new CustomerDataSource(self)
end;

method RootViewController.ViewDidUnload;
begin
  //Delete the sample DB. Pointlessly kill table in the DB first.
  using cmd := connection.CreateCommand() do
  begin
    cmd.CommandText := "DROP TABLE IF EXISTS Customers";
    cmd.CommandType := CommandType.Text;
    connection.Open;
    cmd.ExecuteNonQuery;
    connection.Close;
  end;
  File.Delete(dbPath);
  inherited ViewDidUnload()
end;
```

TABLE VIEW DATA SOURCE

After all that code that's the Table View itself done. The remaining work is done in the Table View's `CustomerDataSource` class, a descendant of `UITableViewsource`. You'll notice a `CustomerDataSource` object being set up at the end of the template code in `ViewDidLoad`. The data source class is set up in the template as a nested class defined within the Table View with a number of its virtual methods already overridden for you.

Tables can be split into multiple sections, each (optionally) with its own header. Our customer list will not need additional sections so `NumberOfSections` should return 1. To tell the Table View how many rows should be displayed in this single section, `RowsInSection` should return `controller.customerList.Count` (`controller` is

set in the constructor, giving access to the view controller). To give the section a header you need to override the method `TitleForHeader`.

Overriding virtual methods is easy in MonoDevelop; start typing the declaration in the public section of the data source class. Once you've typed in method and the first couple of characters of the method name press **Ctrl+Space** and Code Completion will let you select the method and fill in the declaration. You'll have to enter the implementation yourself though. Have it return the string *Customers*.

To populate the cells we use the `GetCell` method, whose parameters are the Table View and the cell's index path (the section number and row number within the section given by the `Section` and `Row` properties). The first thing to note about the code below is the innate support for virtual lists through reusable cells. If you wanted to display a very long list it may not be practical to create a `UITableViewCell` for every item due to the memory usage required. Instead you can take advantage of the Table View offering any cell that is scrolled off-screen as reusable. You can have various categories of reusable cells by simply using different cell identifiers.

```
method RootViewController.CustomerDataSource.GetCell(tableView: UITableView;
  indexPath: MonoTouch.Foundation.NSIndexPath): UITableViewCell;
begin
  var cellId: System.String := 'Cell';
  var cell := tableView.DequeueReusableCell(cellId);
  if cell = nil then
  begin
    cell := new UITableViewCell(UITableViewCellStyle.Subtitle, cellId);
    // Add in a detail disclosure icon to each cell
    cell.Accessory := UITableViewCellAccessory.DetailDisclosureButton;
  end;
  // Configure the cell.
  with cust := controller.customerList[indexPath.Row] do
  begin
    cell.TextLabel.Text :=
      String.Format('{0} {1}', cust.FirstName, cust.LastName);
    cell.DetailTextLabel.Text := cust.Town;
  end;
  exit cell
end;
```

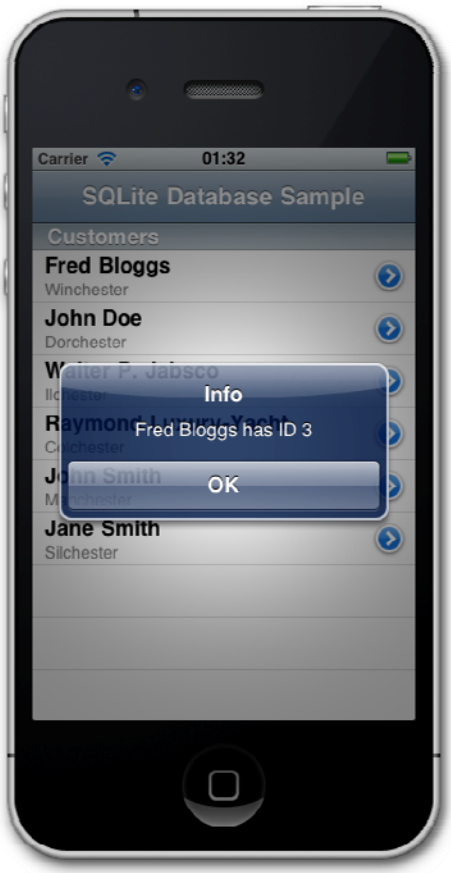
This code creates cells that permit a text value and an additional smaller piece of text (a subtitle). These are accessed through the `TextLabel` and `DetailTextLabel` properties respectively.

During the cell setup a detail disclosure button is also added in. This adds in a little arrow in a circle on the right side of each cell. This then gives us two possible actions from the user: they can tap the row in general, which triggers `RowSelected`, or tap the disclosure button, which triggers `AccessoryButtonTapped`. Often, `RowSelected` is used take you to another screen, so in this case we will leave `RowSelected` doing nothing and just

support the disclosure button, which issue's an alert displaying some information about the selected customer.

```
method RootViewController.CustomerDataSource.AccessoryButtonTapped(
  tableView: UITableView; indexPath: NSIndexPath);
begin
  var cust := controller.customerList[indexPath.Row];
  InfoAlert(string.Format ("{0} {1} has ID {2}",
    cust.FirstName, cust.LastName, cust.CustID))
end;
```

All of which gives us this application:



NAVIGATION CONTROLLERS

Let's start another iPhone Navigation-based project. This time we'll focus more on the navigation support than the table support; the table view will simply act as a menu for some other pages. The menu will contain three items and, just to show how menu items can be grouped, we'll have some sections in the list/menu. In this case each item will be in

its own section, so three sections, but it is down to your application how you apportion list items within the sections.

Make the `NumberOfSections` method return 3 and `RowsInSection` return 1 and then add in a `TitleForHeader` method as above by using Code Completion in the public part of the class declaration (type in `method Ti` and press **Ctrl+Space**). This should be implemented thus:

```
method RootViewController.DataSource.TitleForHeader(tableView: UITableView;
  section: Int32): String;
begin
  case section of
    0: exit 'UIKit example';
    1: exit 'CoreLocation & MapKit example';
    2: exit 'Device information example';
  end;
end;
```

The different sections of the menu offer different types of choices exemplifying different parts of the CocoaTouch library. To populate the table cells (or menu items) we need to add code to `GetCell` as before. This time, though, we know we will only have a small number of cells in the list and so the reusable cell facility described earlier is not required:

```
method RootViewController.DataSource.GetCell(tableView: UITableView; indexPath:
MonoTouch.Foundation.NSIndexPath): UITableViewCell;
begin
  var cell := new UITableViewCell(UITableViewCellStyle.Default, '');
  if (indexPath.Section = 0) and (indexPath.Row = 0) then
    cell.TextLabel.Text := 'Web browser';
  if (indexPath.Section = 1) and (indexPath.Row = 0) then
    cell.TextLabel.Text := 'GPS information';
  if (indexPath.Section = 2) and (indexPath.Row = 0) then
    cell.TextLabel.Text := 'Device information';
  exit cell
end;
```

To give us somewhere to implement these examples we require 3 additional pages. You can add a new file to the solution's active project either using File, New, File from the main menu (or ⌘**N**) or by right-clicking your project in the Solution window and choose Add, New File... Either way, click on iPhone and iPad in the dialog that pops up and choose iPhone View with Controller. The first one should be called *BrowserPage*, then do the same and add *GPSPage* and finally add a last page called *InfoPage*.

Each of these new files contains a `UIViewController` descendant named as you specified the file should be named. When you choose an item from the table view we'll launch one of these new views so before leaving RootViewController.xib.pas we should fill in the `RowSelected` method.

```
method RootViewController.DataSource.RowSelected(tableView: UITableView;
  indexPath: MonoTouch.Foundation.NSIndexPath);
```

```
begin
  if (indexPath.Section = 0) and (indexPath.Row = 0) then
    controller.NavigationController.PushViewController(
      new BrowserPage(), true);
  if (indexPath.Section = 1) and (indexPath.Row = 0) then
    controller.NavigationController.PushViewController(new GPSPage(), true);
  if (indexPath.Section = 2) and (indexPath.Row = 0) then
    controller.NavigationController.PushViewController(new InfoPage(), true);
end;
```

As you can see, the navigation controller can have a new view controller pushed onto its stack of view controllers. This new view controller's view is displayed and the navigation bar will contain a button that takes you back to the previous page making return navigation straightforward.

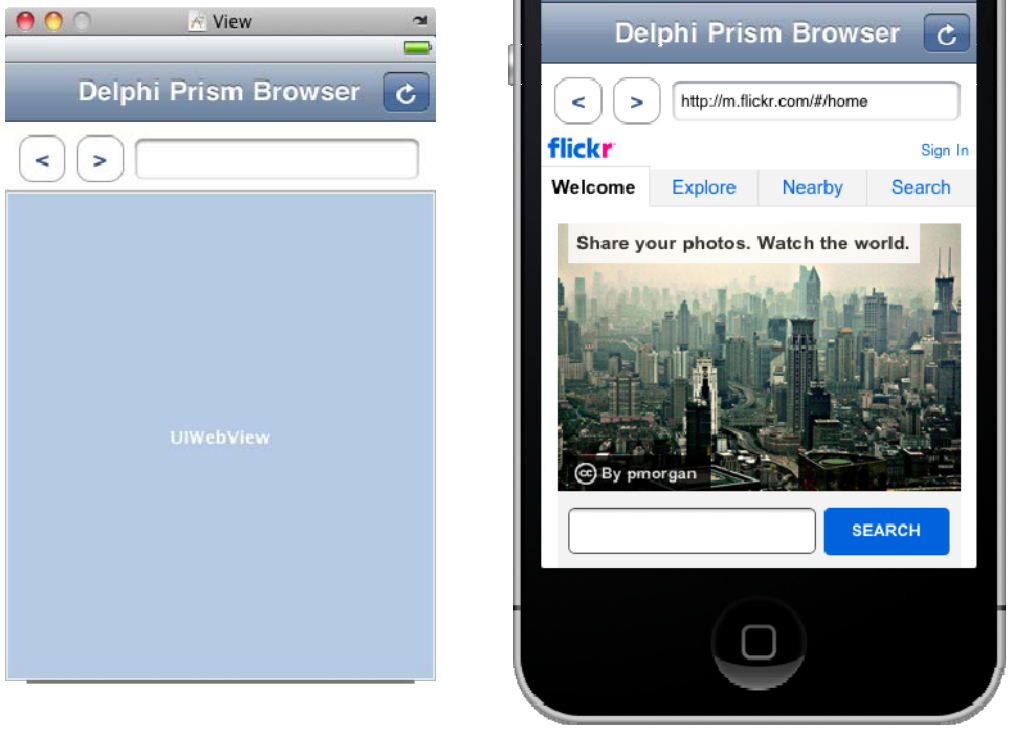
Finally, to edit the Navigation Bar's text, double-click MainWindow.xib, ensure you can see the Navigation Controller (double-click it in the Document Window if need be) and then you can edit the Navigation Bar text by double-clicking it. Add in any text; the sample project simply says: *Brian's Stuff*.

Note: back in the example program that used SQLite we put setup and teardown code in `ViewDidLoad` and `ViewDidUnload` respectively. This was fine as there was just the one view that was loaded into memory as the application started and unloaded whenever the view is removed from memory, such as when the application exits or when memory gets low. In this application there is a menu view and then three secondary views, each of which may require setup and teardown code. Certainly when each view is first opened the `ViewDidLoad` code will execute but when you navigate back to the menu `ViewDidUnload` will not execute. Similarly when you go back to the same secondary view `ViewDidLoad` will not execute again as it will still be loaded in memory.

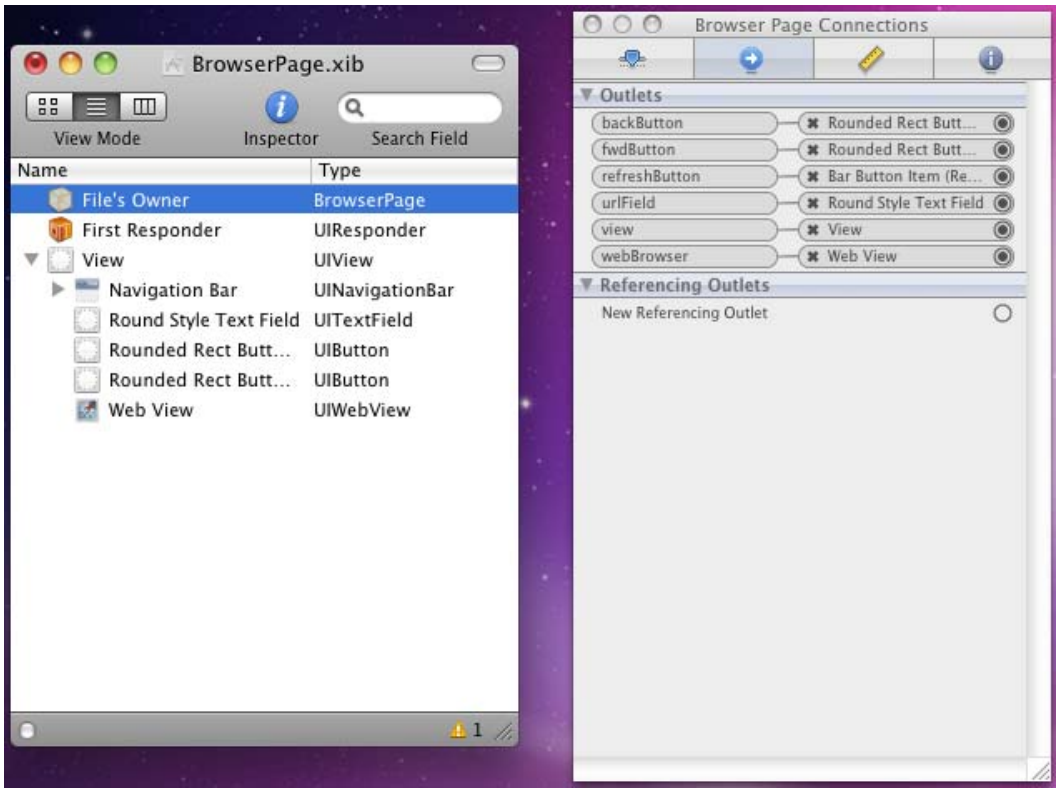
It is important to decide whether the teardown code is important to execute as soon as the view is no longer visible (for performance reasons, for example) or whether it is okay to leave it until the view is eventually removed from memory. This will dictate whether you should continue to use the `ViewDidLoad`/`ViewDidUnload` methods or maybe switch to the `ViewDidAppear`/`ViewDidDisappear` methods.

WEB BROWSING WITH UIWEBVIEW

Let's get the web browser page under way. The ingredients for this include a Navigation Bar (`UINavigationBar`) with a Bar Button Item (`UIBarButtonitem`) placed on it, both found in the Windows, Views and Bars subset of Objects on the Library window. We also need two Round Rect Buttons (`UIButton`), a Text Field (`UITextField`) and a Web View (`UIWebView`), the last of which is in the Data Views subset of Objects. They can be laid out according to the Interface Builder screenshot below, which is followed by an image of the target application page running (remember that the top navigation bar there comes from the main window). The Navigation Bar button can have its image selected by its Identifier attribute.



Outlets will need to be set up for some of these controls so we can access them at runtime. In the case of a page based on a `UIViewController` you'll see the controls you add to the page under the View in the Document Window. The outlets need to be added to the *BrowserPage* object, which can be selected on the Document Window as File's Owner. The screenshot below shows the names of the five new outlets. It should be mentioned that the button outlets are only being added so event handlers can be set up using anonymous method syntax instead of setting up actions in Interface Builder, which would necessitate implementing the action method to house the event handler code.



The idea of the UI controls is for the left and right arrow to be *Back* and *Forward* history navigation buttons, and for the Button on the lower Navigation Bar to refresh the current page. The Text Field is meant to be the same as a browser address bar, so use the Attributes Inspector to set the Keyboard attribute to URL and the Return Key attribute to Go.

Notice on the top Navigation Bar (in the running app) there is a button to take you back to the main page.

The implementation of this page is fairly simply. When the view loads the various UI controls need initializing and a default web page is loaded:

```
method BrowserPage.ViewDidLoad;
begin
  inherited;
  InitButtonsAndTextField;
  InitBrowser;
  //Load a default page
  LoadPage("flickr.com");
end;
```

Helper functions are used for all these jobs. The initialization functions set up events for the controls. The Buttons are trivial; they simply call corresponding methods in the

`UIWebView`. The Text Field `ShouldReturn` property ensures the keyboard will close when Go is pressed, as we've seen before, but also then loads the URL that was entered into the field.

```
method BrowserPage.Alert(Caption, Msg: String);
begin
  using av := new UIAlertView(Caption, Msg, nil, 'OK', nil) do
    av.Show;
end;

method BrowserPage.InitButtonsAndTextField;
begin
  backButton.TouchUpInside += method begin webBrowser.GoBack end;
  fwdButton.TouchUpInside += method begin webBrowser.GoForward end;
  refreshButton.Clicked += method begin webbrowser.Reload end;
  urlField.ShouldReturn := textField -> begin
    Result := textField.ResignFirstResponder; //Hide keyboard
    LoadPage(textField.Text.ToString);
  end;
end;

method BrowserPage.InitBrowser;
begin
  webbrowser.LoadStarted += method
  begin
    UIApplication.SharedApplication.NetworkActivityIndicatorVisible := true
  end;
  webbrowser.LoadFinished += method
  begin
    urlField.Text := webBrowser.Request.Url.AbsoluteString;
    UIApplication.SharedApplication.NetworkActivityIndicatorVisible := false
  end;
  webbrowser.LoadError += method(sender: Object; e: UIWebErrorArgs)
  begin
    UIApplication.SharedApplication.NetworkActivityIndicatorVisible := false;
    Alert('Browser error', 'Web page failed to load: ' + e.Error.ToString());
  end;
end;
```

The web browser control also has event handlers set up, with the main purpose of activating and deactivating the network activity indicator on the iPhone status bar, which oddly is not an automatic response to network activity, and also that reports any navigation errors in an alert. Additionally, when a web request has finished loading a page it may ultimately resolve to a different URL than was requested. This is very common on mobile devices as pages redirect to a mobile-specific version. The `LoadFinished` event handler reflects the final URL back to the Text Field.

The remaining method is `LoadPage`, which ensures the required http:// prefix is present in the URL and then builds an `NSUrlRequest` from an `NSUrl` and passes it to the `UIWebView`'s `LoadRequest` method.

```
method BrowserPage.LoadPage(url: String);
begin
  if url <> '' then
  begin
    if not url.StartsWith('http') then
      url := string.Format('http://{0}', url);
    webbrowser.LoadRequest(new NSUrlRequest(new NSUrl(url)));
  end;
  //Show the URL that was requested
  urlField.Text := url;
end;
```

And there we have the fully working web browser shown earlier that defaults to displaying http://flickr.com (which is then redirected to the mobile version of Flickr).

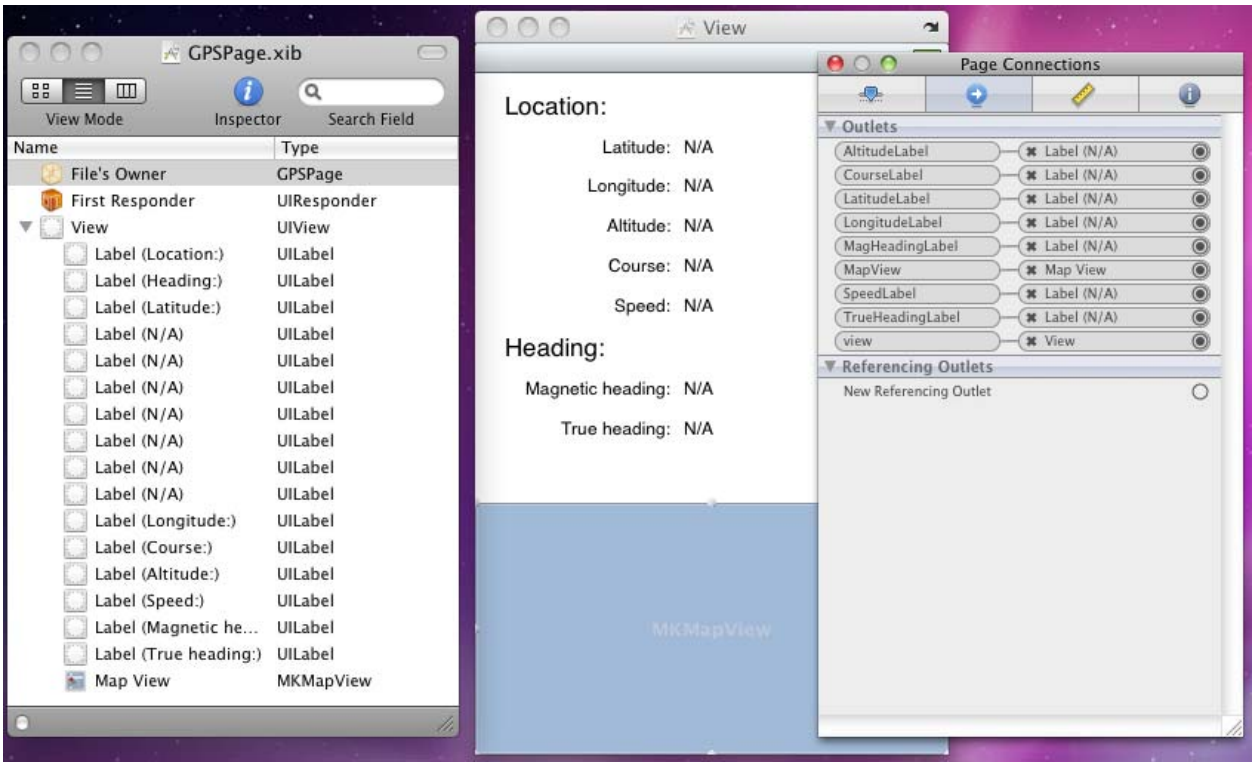
LOCATION/HEADING SUPPORT WITH CORELOCATION AND MAPKIT

One of the very neat features of the iPhone and other current smartphones is the in-built GPS and compass support. There are many handy applications that can chart your progress during running or cycling, or just record your travelled route, built using this capability.

Basic GPS/compass support is offered through the CoreLocation API and a location-aware map control is found in the MapKit: the `MKMapView`[9]. The *GPSPage* view in this sample application will use CoreLocation and an `MKMapView` to show the current location, heading, altitude and speed. To build the UI in Interface Builder you need to lay down 16 labels with text on as shown in the screenshot below and a Map View. All the labels that say *N/A*, as well as the Map View, should be connected to outlets defined in *GPSPage* as per the Connections Inspector in the screenshot.

Next we start on the code.

[9] The `MKMapView` control uses Google's services to do its work and by using it you acknowledge that you are bound by their terms, available online at: http://code.google.com/apis/maps/iphone/terms.html



The starting point for location-based functionality is the `CLLocationManager` class so declare a variable `locationManager` of this type in your `GPSPage` class. This object offers us GPS-based information about the location (position, course, speed and altitude from the GPS hardware[10]) and the compass-based heading (the direction the device is pointing). The GPS-dependant information will be of varying accuracy, as is the nature of GPS data (you will be locked onto a varying number of satellites).

The location manager offers callback facilities that triggers as the heading and location changes, allowing your journey to be tracked. Depending on the type of application you build you can control how accurate you would *like* the data to be and you can also control how often your application will be notified of heading and/or location changes. If you weren't required to track a detailed route, then being notified for every single location change would be excessive. It may be more appropriate to be notified when the location changes by 50 meters, say. Requiring less accuracy and being notified less often is helpful in the context of battery usage.

[10] If GPS signal or hardware is not available the device will provide coarse-grained location information based on cell phone towers or your WiFi hotspot

This callback mechanism is implemented in CoreLocation using the common approach of supporting a delegate object (inherited from type `CLLocationManagerDelegate`), which has methods to override for location and heading changes. You create an instance of such a class and assign it to the location manager's `Delegate` property. An example delegate class might look like the following code (notice that the main view, *GPSPage*, is passed into the constructor and is to be stored in the `Page` variable, so it can access controls on the view:

```
CoreLocationManagerDelegate nested in GPSPage =
  class(CLLocationManagerDelegate)
private
  Page: GPSPage;
public
  constructor (page: GPSPage);
  method UpdatedHeading(manager: CLLocationManager;
    newHeading: CLHeading); override;
  method UpdatedLocation(manager: CLLocationManager;
    newLocation, oldLocation: CLLocation); override;
end;
```

As we have seen before, the MonoTouch approach is to absorb such delegate objects and their optional methods and expose them as events in the main object. So the location manager actually has properties called `UpdatedHeading` and `UpdatedLocation`. In this code, we'll use those instead.

The signatures of these methods fit in with the standard .NET event signature:

```
method UpdatedHeading(sender: Object; args: CLHeadingUpdatedEventArgs);
method UpdatedLocation(sender: Object; args: CLLocationUpdatedEventArgs);
```

where `sender` refers to the location manager and the `args` parameters contains properties matching the remaining parameters that are sent to the matching delegate object method.

In `GPSPage.ViewDidAppear` we'll initialize the location manager:

```
locationManager := new CLLocationManager();
locationManager.DesiredAccuracy := -1; //Be as accurate as possible
locationManager.DistanceFilter := 50; //Update when we have moved 50 m
locationManager.HeadingFilter := 1; //Update when heading changes 1 degree
locationManager.UpdatedHeading += UpdatedHeading;
locationManager.UpdatedLocation += UpdatedLocation;
locationManager.StartUpdatingLocation();
locationManager.StartUpdatingHeading();
```

You should also clean up in `ViewDidDisappear`:

```
locationManager.StopUpdatingHeading();
locationManager.StopUpdatingLocation();
locationManager.Dispose;
```

```
locationManager := nil;
```

Note: the setup/teardown code in this page is done in `ViewDidAppear` and `ViewDidDisappear` (as opposed to `ViewDidLoad` and `ViewDidUnload`) to avoid the GPS hardware continuing to report information to the view when you have navigated back to the menu.

We'll need to look at the event handlers referenced here, but first we should also initialize the Map View. Above the location manager initialization code in the `ViewDidAppear` method we need this:

```
MapView.WillStartLoadingMap += method
begin
  UIApplication.SharedApplication.NetworkActivityIndicatorVisible := true
end;
MapView.MapLoaded += method
begin
  UIApplication.SharedApplication.NetworkActivityIndicatorVisible := false
end;
MapView.LoadingMapFailed += method
begin
  UIApplication.SharedApplication.NetworkActivityIndicatorVisible := false;
end;
MapView.MapType := MKMapType.Hybrid;
MapView.ShowsUserLocation := True;
//Set up the text attributes for the user location annotation callout
MapView.UserLocation.Title := 'You are here';
MapView.UserLocation.Subtitle := 'YA RLY!';
```

You can see we have Map View events that mirror the `UIWebView` events and do a similar job (though this time we simply ignore any errors). The `MapType` and `ShowsUserLocation` properties could actually have been set in Interface Builder in the Attributes Inspector but instead are set in code. `MapType` allows you to make the usual display choice that maps such as Google or Bing offer: standard (map), satellite, or hybrid (satellite plus road markings). `ShowUserLocation` controls whether the map will display the user's location (using an annotation), assuming it can be determined. The final property being set, `UserLocation`, customizes this map annotation. When clicked on, the annotation can produce a callout displaying extra information consisting of a title and subtitle, and that's what we are setting here.

Now back to the callback events. The heading change callback is short and simple, since there are only two new heading values offered. The `NewHeading` object inside `args` has `TrueHeading` (heading relative to true north) and `MagHeading` (heading relative to magnetic north) properties. It also offers `HeadingAccuracy` that indicates how many degrees, one way or the other, the heading values might be. If this accuracy value is negative, then heading information could not be acquired, as is the case in the iPhone Simulator. The Simulator has some GPS functionality, but no emulated compass.

```
method GPSPage.UpdatedHeading(sender: Object;
  args: CLHeadingUpdatedEventArgs);
begin
  if args.newHeading.HeadingAccuracy >= 0 then
  begin
    MagHeadingLabel.Text := string.Format('{0:F1}° ± {1:F1}°',
      args.NewHeading.MagneticHeading, args.NewHeading.HeadingAccuracy);
    TrueHeadingLabel.Text := string.Format('{0:F1}° ± {1:F1}°',
      args.NewHeading.TrueHeading, args.NewHeading.HeadingAccuracy);
  end
  else
  begin
    MagHeadingLabel.Text := 'N/A';
    TrueHeadingLabel.Text := 'N/A';
  end
end;
```

The location change callback is a little longer, but only because there are more values available from the GPS hardware. This time `args` has both a `NewLocation` and an `OldLocation` `CLLocation` object, so you could work out the distance travelled between the two (`CLLocation` offers a `DistanceFrom` method) if you chose:

```
method GPSPage.UpdatedLocation(sender: Object; args:
CLLocationUpdatedEventArgs);
const
  LatitudeDelta = 0.002; //no. of degrees to show in the map
  LongitudeDelta = LatitudeDelta;
begin
  var PosAccuracy:= args.NewLocation.HorizontalAccuracy;
  if PosAccuracy >= 0 then
  begin
    var Coord := args.NewLocation.Coordinate;
    //In simulator, MapKit's user location is fixed on Apple's HQ but
    //CoreLocation will happily detect current location via network
    //(contrary to Apple docs)
    LatitudeLabel.Text := string.Format(
      '{0:F6}° ± {1} m', Coord.Latitude, PosAccuracy);
    LongitudeLabel.Text := string.Format(
      '{0:F6}° ± {1} m', Coord.Longitude, PosAccuracy);
    if Coord.IsValid then
    begin
      var region: MKCoordinateRegion := new MKCoordinateRegion(
        Coord, new MKCoordinateSpan(LatitudeDelta, LongitudeDelta));
      MapView.SetRegion(region, False);
      MapView.SetCenterCoordinate(Coord, False);
      MapView.SelectAnnotation(MapView.UserLocation, False);
    end;
  end
  else
  begin
    LatitudeLabel.Text := 'N/A';
    LongitudeLabel.Text := 'N/A';
  end;
  if args.NewLocation.VerticalAccuracy >= 0 then
    AltitudeLabel.Text := string.Format(
      '{0:F6} m ± {1} m', args.NewLocation.Altitude,
                          args.NewLocation.VerticalAccuracy)
```

```
  else
    AltitudeLabel.Text := 'N/A';
  if args.NewLocation.Course >= 0 then
    CourseLabel.Text := string.Format('{0}°', args.NewLocation.Course)
  else
    CourseLabel.Text := 'N/A';
  SpeedLabel.Text := string.Format('{0} m/s', args.NewLocation.Speed);
end;
```

Breaking the code up, the first big condition deals with the position, updating the latitude and longitude labels with the relevant position and the accuracy achieved, and the Map View position. If the accuracy value is negative then a position has not been obtained and so *N/A* is written to the labels.

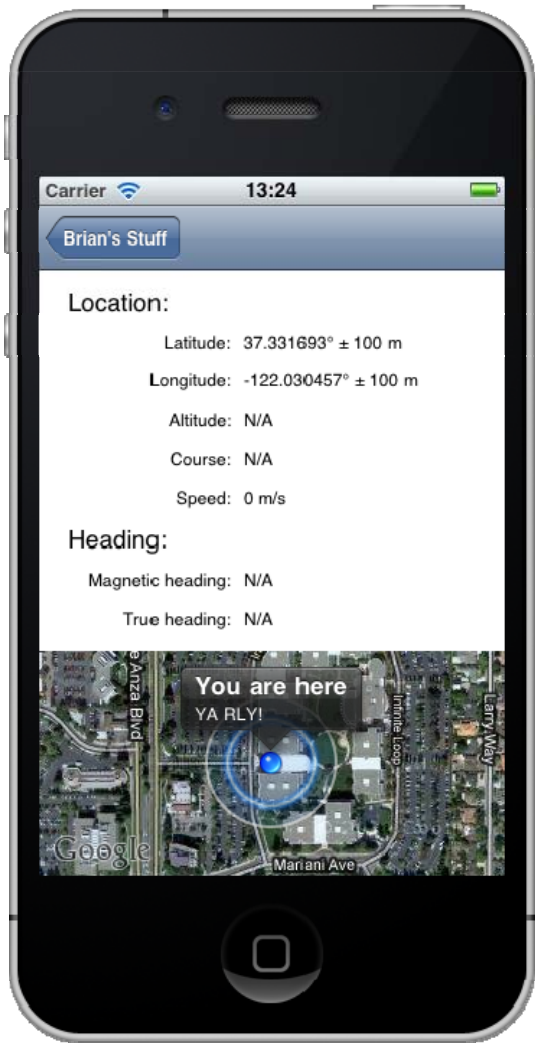
You might notice the comment in the code that talks about the GPS functionality in the Simulator. All references I found, in forums and in the Apple documentation, state that CoreLocation will always return a fixed location in the iPhone Simulator, the location being the Apple HQ at 1 Infinite Loop, Cupertino, CA 95014 with an accuracy of 100m. In my tests this was true of the Map View – if not forced to do otherwise it will always report the user's location as being at Apple HQ. However, CoreLocation would correctly identify my location and return co-ordinates to my office. This seems to contradict various statements and shows some in-Simulator inconsistency between MapKit and CoreLocation.

To keep things consistent the code takes the CoreLocation coordinate as the true location and forces the Map View to use it by specifying a region to display and centering the map on that coordinate (we lose the user location annotation this way, but at least we see where we really are). The map display region is set up in terms of a coordinate and a pair of X and Y deltas, which dictate how much of the earth to display in terms of degrees. A small value has been used for both deltas to show a vaguely recognizable piece of the local territory. This control of the Map View only takes place if the CoreLocation's coordinate is deemed to be valid. On the first few callbacks it is common for the coordinate to start as invalid while the GPS system gets on top of its communication.

The final thing done with the Map View is a call to `SelectAnnotation` made against the annotation at the user's location. This is the equivalent of clicking the annotation and will cause the callout (with the title and subtitle) to be displayed. Of course, if the app is showing your actual location and the Map View has the user location annotation in Cupertino, you are unlikely to see it. In the sample code source (not shown in the listing above) there is a conditional define called `SHOW_FAKE_POSITION_IN_SIMULATOR` that you can define to overcome this and ensure the Map View's notion of the user location is used for both the information labels and also the map position, and so showing the user location annotation.

The remaining code performs familiar looking tasks for the altitude and course – displaying the values if they are valid – and also displays the current speed as ascertained by the GPS observations.

The screenshot below shows the GPS Page operating, though it was taken with the aforementioned conditional compilation symbol defined, so the image looks consistent with the Apple documentation. The user location annotation is actually dynamic. As well as the blue marble in the centre and the outer circle indicating the possible inaccuracy radius, the blue circle in between pulses out from the center to the outer circle in a manner pleasing to the eye.



DEVICE ROTATION

You'll doubtless be aware that many iPhone applications respond to you rotating the phone 90 degrees by reorganizing their UI to display appropriately in a landscape manner instead of portrait. The main underpinning to this support is the `ShouldAutorotateToInterfaceOrientation` virtual method of the view controller class. When the phone is rotated, this method is called with the new orientation (`Portrait`, `LandscapeLeft`, `PortraitUpsideDown` or `LandscapeRight`) and the method returns `True` or `False` depending on whether the app should be rotated to that orientation. Returning `True` regardless means the app will rotate around as the phone is rotated, but won't reorganize per se. In the case of a simple form like this app's main form, just containing a Table View, that would be enough as the Table View will fill up the available space:

However, something like *GPS Page* will require more attention given the variety of controls. If we leave things as they are the Map View will not be visible and there would be blank space on the right.

We still need `ShouldAutorotateToInterfaceOrientation` to return `True` but we also need to implement `WillAnimateRotation` to act on the rotation and re-jig the control layout. A helper routine, `SetupUIForOrientation`, will be used that takes the new UI orientation.

```
SetupUIForOrientation(toInterfaceOrientation)
```

Additionally, we will need to call the helper routine from `ViewDidAppear` given the phone could be in any orientation when the *GPS Page* is invoked.

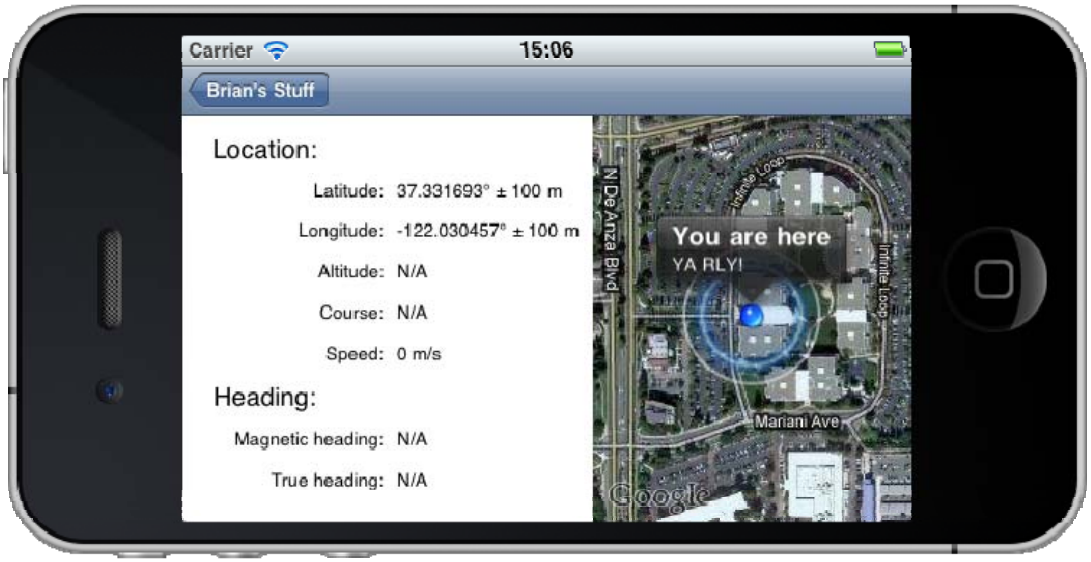
At the start of `ViewDidAppear` add:

```
if InterfaceOrientation not in [UIInterfaceOrientation.Portrait,
                UIInterfaceOrientation.PortraitUpsideDown] then
    SetupUIForOrientation(InterfaceOrientation);
```

This helper routine will look at the orientation and locate the Map View accordingly, either below the labels in portrait modes or to the right of them in landscape modes.

```
method GPSPage.SetupUIForOrientation(orientation: UIInterfaceOrientation);
const
  NavBarHghtPortrait = 44;
  NavBarHghtLandscape = 32;
  TextLabelsWidth = 270; //horizontal screen extent occupied by labels
  TextLabelsHeight = 257; //vertical screen extent occupied by labels
begin
  var DeviceHeight := Integer(UIScreen.MainScreen.Bounds.Height);
  var DeviceWidth := Integer(UIScreen.MainScreen.Bounds.Width);
  with AppFrame := UIScreen.MainScreen.ApplicationFrame do
    if orientation in [UIInterfaceOrientation.Portrait,
                       UIInterfaceOrientation.PortraitUpsideDown] then
      MapView.Frame := RectangleF.FromLTRB(0, TextLabelsHeight,
        DeviceWidth, AppFrame.Height - NavBarHghtPortrait)
    else
      MapView.Frame := RectangleF.FromLTRB(TextLabelsWidth, 0,
        DeviceHeight, AppFrame.Width - NavBarHghtLandscape);
end;
```

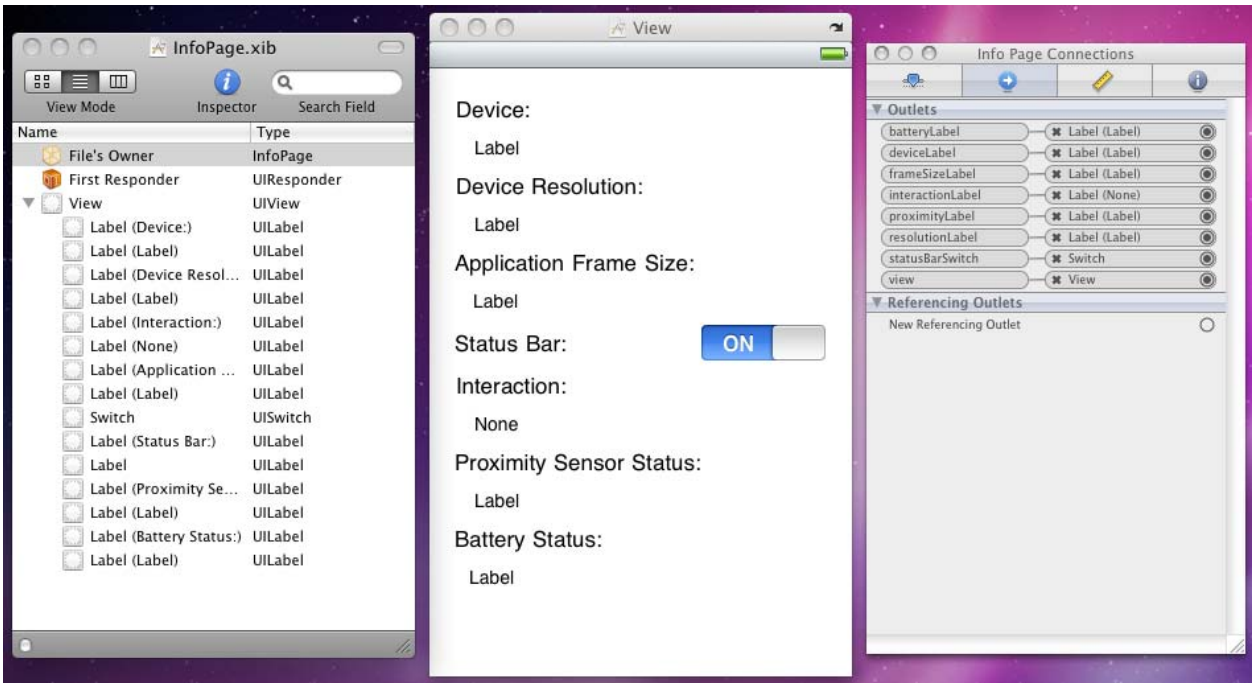
The constants have been worked out manually. The full screen resolution (in points) is given by `UIScreen.MainScreen.Bounds` (320 wide by 480 high in iPhones) and the available space on screen, taking into account the status bar, is given by `UIScreen.MainScreen.ApplicationFrame`. The dimensions of the Map View are calculated using these various dimensions.



For example, in landscape mode (shown above) the Map View needs its left border placed after the full width of the text labels area and its top at the topmost pixel, 0, which will be immediately below the status bar and Navigation bar. Its right needs to be the rightmost pixel on the screen, given by the screen 'height', and the bottom border will be at the bottom of the landscape screen, which is the application frame width (screen 'width' minus status bar height) minus the height of the Navigation Bar.

Device Information

The last page in this application is intended to display various pieces of information about the device. This requires another collection of labels to be laid out in Interface Builder, as well as a Switch (`UISwitch`). The six smaller labels and the Switch are needed in the code so outlets need connecting to them as shown here:



Some of the information required to populate the labels can be attained as soon as the view is loaded, such as what device it is and what the screen resolution is. Some information will be updated as and when necessary, such as *Application Frame Size* (that will change when the status bar is toggled on and off), *Proximity Sensor Status* and *Battery Status*. *Interaction* is another one that will get updated by the user interacting with the phone – it will update to show when the phone is rotated or shaken, and when the user taps. Let's tackle these one at a time.

The specific device is identified using a helper class `DeviceHardware` that offers a class method, `Version`, which returns a value from an enumerated type:

```
type
  HardwareVersion = public (
    iPhone,
    iPhone3G,
    iPhone3GS,
    iPhone4,
    iPod1G,
    iPod2G,
    iPod3G,
    iPod4G,
    iPad,
    iPhoneSimulator,
    iPhone4Simulator,
    iPadSimulator,
    Unknown);
```

The code in the class (which employs native interop) is not important here but is included with the sample projects. It is adapted and enhanced from some existing C# code on the Mono wiki at http://wiki.monotouch.net/HowTo/Device/Detect_the_Hardware_Type. The class also has a `VersionString` class method that returns a descriptive string for the current device. To display the device details `ViewDidAppear` contains:

```
deviceLabel.Text := String.Format('{0}, iOS v{1}',
  DeviceHardware.VersionString, UIDevice.CurrentDevice.SystemVersion);
```

This is followed by a call to a helper routine that emits the screen resolution and application frame size.

```
method InfoPage.UpdateUIMetrics;
begin
  with scrn := UIScreen.MainScreen do
  begin
    //iPhone 4 doubles pixel count, but point count remains same
    resolutionLabel.Text := string.Format('{0}x{1} points, {2}x{3} pixels',
      scrn.Bounds.Width, scrn.Bounds.Height,
      scrn.Bounds.Width * scrn.Scale, scrn.Bounds.Height * scrn.Scale);
    frameSizeLabel.Text := string.Format('{0}x{1} points',
      scrn.ApplicationFrame.Width, scrn.ApplicationFrame.Height);
  end;
end;
```

The status bar switch needs to be set to the correct value to start with and then requires an event handler:

```
  statusBarSwitch.On := not UIApplication.SharedApplication.StatusBarHidden;
  statusBarSwitch.ValueChanged += StatusBarValueChanged;
...
method InfoPage.StatusBarValueChanged(Sender: Object; E: EventArgs);
begin
  UIApplication.SharedApplication.StatusBarHidden := not statusBarSwitch.On;
  // NOTE: it's required to call the inherited View property from inside a
  // ViewController, as the autogenerated 'view' property is nil, and Pascal
  // isn't case sensitive.
  // From outside the class, the View property works fine.
  if (inherited View <> nil) and ((inherited View).Window <> nil) then
    (inherited View).Window.Frame := UIScreen.MainScreen.ApplicationFrame;

  //Without this, the nav bar is lazy about moving to the right place
  //Required a public property to be added to the AppDelegate
  var AppDel := AppDelegate(UIApplication.SharedApplication.Delegate);
  var NavController := AppDel.NavController;
  NavController.SetNavigationBarHidden(True, False);
  NavController.SetNavigationBarHidden(False, False);
  UpdateUIMetrics
end;
```

There are a few noteworthy things in here. Firstly the status bar's visibility is simply controlled via `UIApplication.SharedApplication.StatusBarHidden`.

Secondly, just hiding the status bar does little to our current view. In order to fill the new amount of space on the screen the underlying window's `Frame` property is set to match the screen's `ApplicationFrame` property. Remember we are in a view controller descendant class at the moment. The underlying window is a property of the view, for which `UIViewController` defines a property, `View` (upper case `V`). However the code behind file for this class also happens to define a property `view` (lower case `v`), which always returns `nil`. Whenever you wish to access the view associated with your view controller, it is vital to remember to use `inherited View`. When using MonoTouch from C# you also get this extra (apparently pointless) `view` property in the partial class, but it's not such an issue there as C# is a case-sensitive language, unlike Delphi Prism.

Finally, even after we resize the view's window into the available space the Navigation Bar often doesn't respond when it should (its decision on when to move up with the window seems unpredictable). The best way I found to overcome the issue is to hide and then re-show the Navigation Bar after the status bar has been toggled. However, gaining access to the Navigation Bar requires some thought.

It's useful to know that the Navigation Bar is exposed through properties and methods of the navigation controller but how do we access the Navigation Controller from a secondary view?

There is a property added to the `AppDelegate` class in the code behind file for the main window, MainWindow.xib.designer.pas, that exposes the navigation controller. In the auto-generated partial class the property `navigationController` is defined with a getter and setter called `get_navigationController` and `set_navigationController` respectively. The problem is that all these symbols are private and so are inaccessible from where we are writing code. The most direct way to overcome this is to define a new public property in the `AppDelegate` class in the main code file Main.pas, e.g.

```
public
  property NavController: UINavigationController
    read get_navigationController write set_navigationController;
```

Of course, we then have the issue of how to talk to the `AppDelegate` object from the *Info Page*, but this is readily solved. You may recall we saw earlier how to access the Application object using `UIApplication.SharedApplication`. The Application object's delegate object (i.e. the `AppDelegate`) is available through the Application object's `Delegate` property. Once we gain access to the navigation controller its `SetNavigationBarHidden` method can be used to toggle the visibility of the Navigation Bar.

After all this, the application frame size will have changed so the `UpdateUIMetrics` helper is invoked again.

When this page was initially displayed the status bar was present, but it may be the case that the switch has toggled it off. Before the view exits back to the menu screen we should restore the natural order. We do this in `ViewDidDisappear`:

```
statusBarSwitch.On := True;
StatusBarValueChanged(statusBarSwitch, new EventArgs);
```

PROXIMITY SENSOR AND NOTIFICATIONS

The iPhone's Proximity Sensor is used to turn the phone's display off when you answer a call and move the phone next to your face. It can doubtless be employed in various other useful scenarios and so it is good to see how we can be notified of proximity state changes. The mechanism is simple; proximity to something is either detected or not and there will be a state change notification when the situation changes. This code is in `ViewDidAppear`:

```
UIDevice.CurrentDevice.ProximityMonitoringEnabled := True;
if UIDevice.CurrentDevice.ProximityMonitoringEnabled then
  NSNotificationCenter.DefaultCenter.AddObserver(
    UIDevice.ProximityStateDidChangeNotification,
    method
    begin
      proximityLabel.Text := iif(UIDevice.CurrentDevice.ProximityState,
        'Proximity detected', 'Proximity not detected');
    end)
else
  proximityLabel.Text := 'Proximity sensor not available';
```

Not all devices have a proximity sensor (the simulator doesn't, for example) so the advice is to turn proximity monitoring on and then check to see if it successfully turned on. If not there is no proximity sensor.

If you have the appropriate hardware then you need to arrange to respond to state changes. This is one of the cases that do not use the familiar delegate object (or event property alternative) approach. Instead it uses notifications orchestrated from a notification centre that requires observer methods to notice. Every application has a notification centre accessible with `NSNotificationCenter.DefaultCenter` and of the various overloads `AddObserver` offers, the simplest one takes the notification identifier and a delegate that is passed an `NSNotification` object (which we ignore in the code and so don't need to declare in the anonymous method). The declaration in the MonoTouch documentation looks like this:

```
AddObserver(string, Action<NSNotification>) : NSObject
```

`Action<T>` is a standard .NET generic delegate type declared in the System namespace. It represents a function that returns no value but takes a parameter of type `T`.

In Objective-C, the notification identifiers are literal strings and you could very well use this as the first parameter to `AddObserver`:

```
new NSString('UIDeviceProximityStateDidChangeNotification')
```

However the `UIDevice` class has a number of these notification identifiers set up as properties for your convenience, for example:

`UIDevice.OrientationDidChangeNotification`,
`UIDevice.BatteryLevelDidChangeNotification`,
`UIDevice.BatteryStateDidChangeNotification`.

Don't forget that we enabled proximity state monitoring so in `ViewDidDisappear` it is appropriate to turn it off:

```
if UIDevice.CurrentDevice.ProximityMonitoringEnabled then
   UIDevice.CurrentDevice.ProximityMonitoringEnabled := False;
```

Battery Status and Timers

The battery status monitoring operates quite similar to the proximity state monitoring. Not all devices support battery status monitoring (for example the iPhone Simulator does not) and to see if it's supported you again enable monitoring and then check whether monitoring is still enabled. If not, then it's not supported.

Whilst battery status monitoring can be done with notifications, that is only appropriate if you want monitoring to be on all the time. To be a bit more battery-friendly we can instead just check once every so often, say once a minute or two. Each time we want to check we turn battery monitoring on, if possible, check the battery level and battery state and then turn monitoring off. This is the method that does the checking:

```
method InfoPage.ReadBatteryStatus;
begin
  with dev := UIDevice.CurrentDevice do
  begin
    dev.BatteryMonitoringEnabled := True;
    if dev.BatteryMonitoringEnabled then
      try
        batteryLabel.Text := string.Format('{0}% - {1}',
          Math.Round(dev.BatteryLevel * 100), dev.BatteryState);
      finally
        dev.BatteryMonitoringEnabled := False
      end
    else
    begin
      batteryLabel.Text := 'Battery level monitoring not available';
      UpdateBatteryStatusTimer.Invalidate
    end
  end
end;
```

To run this code at fixed intervals, we need a scheduled repeating timer. Timers only work when they are scheduled on a run loop (the iOS equivalent of a Windows message loop) and need to be repeating to fire more than once.

In `ViewDidAppear`, the timer is set up to trigger every 60 seconds and the battery check code executed an initial time:

```
UpdateBatteryStatusTimer := NSTimer.CreateRepeatingScheduledTimer(
  60, new NSAction(ReadBatteryStatus));
ReadBatteryStatus;
```

`NSAction` is rather like `Action<T>` described earlier, but is a Mono delegate type that represents a function that returns no value and takes no parameters. This looks a little different to the anonymous method we passed in when setting up the proximity state notification, but we could make it look more similar by writing it like this:

```
UpdateBatteryStatusTimer := NSTimer.CreateRepeatingScheduledTimer(
  60, method begin ReadBatteryStatus end);
ReadBatteryStatus;
```

It can be made more intuitive as:

```
UpdateBatteryStatusTimer := NSTimer.CreateRepeatingScheduledTimer(
  60, @ReadBatteryStatus);
ReadBatteryStatus;
```

Note that in the timer event handler earlier, if battery monitoring is not available, then the timer is cancelled using its `Invalidate` method. It is also important to remember to cancel the timer in `ViewDidDisappear` by calling the same method.

iPhone Interaction

The final label to address in the *Info Page* relates to interaction and is designed to give information about taps, swipes, shakes and rotations. The latter case is straightforward – we've already looked at device rotation earlier. We need to make `ShouldAutorotateToInterfaceOrientation` return `True` regardless of orientation passed in and then write some code in `WillAnimateRotation`:

```
method InfoPage.WillAnimateRotation(
  toInterfaceOrientation: UIInterfaceOrientation; duration: Double);
begin
  interactionLabel.Text := case toInterfaceorientation of
    UIInterfaceOrientation.Portrait: 'iPhone is oriented normally';
    UIInterfaceOrientation.LandscapeLeft: 'iPhone has been rotated right';
    UIInterfaceOrientation.PortraitUpsideDown: 'iPhone is upside down';
    UIInterfaceOrientation.LandscapeRight: 'iPhone has been rotated left';
  end;
```

```
  SetTimerToClearMotionLabel;
  UpdateUIMetrics
end;
```

Note: a recent feature of the Delphi Prism language is used in this code snippet: a `case` expression (as opposed to the more typical `case` statement).

Note: since we are not actually reorganizing the UI and are just responding to altered rotation, we might be better to opt for use of the notification center and register an observer to be called in response to the orientation change notification.

```
NSNotificationCenter.DefaultCenter.AddObserver(
  UIDevice.OrientationDidChangeNotification, @OrientationChanged);
...
method InfoPage.OrientationChanged(notification: NSNotification);
begin
  interactionLabel.Text := case InterfaceOrientation of
    UIInterfaceOrientation.Portrait: 'iPhone is oriented normally';
    UIInterfaceOrientation.LandscapeLeft: 'iPhone has been rotated right';
    UIInterfaceOrientation.PortraitUpsideDown: 'iPhone is upside down';
    UIInterfaceOrientation.LandscapeRight: 'iPhone has been rotated left';
  end;
  SetTimerToClearMotionLabel;
  UpdateUIMetrics
end;
```

As any interaction is noticed the code updates the interaction label, but then calls a helper routine that uses a timer to reset the label text after a short time interval of three seconds. Also the resolution labels are updated with another call to `UpdateUIMetrics`.

```
method InfoPage.SetTimerToClearMotionLabel;
begin
  if ClearMotionLabelTimer <> nil then
    ClearMotionLabelTimer.Invalidate;
  ClearMotionLabelTimer := NSTimer.CreateScheduledTimer(3,
  method
  begin
    interactionLabel.Text := 'None';
    ClearMotionLabelTimer := nil;
  end);
end;
```

This code should look familiar, as it is very similar to the timer code used for the proximity sensor. In this case, however, we require a one-shot timer to reset the label instead of one that keeps firing, so a scheduled timer is created rather than a scheduled repeating timer.

SHAKES

The iPhone recognizes when you shake it (if the application chooses to enable this). Standard use of this feature is for undo/redo but clearly this is down to the imagination of the programmer. But how do you enable shake recognition in the first place? Well, a few things need to be done. It is usually considered that the shake gesture is directed at the

view but we can pick it up in our view controller if we configure it correctly. We need to have the view controller assert that it can successfully become a first responder and follow that up by making it so. Then we need to enable shake support and finally we need to override the `MotionEnded` method where shake detection can take place.

By default, a view controller's `CanBecomeFirstResponder` property returns False, rather stymieing our efforts to detecting shakes. However, as luck would have it, the getter for this property was declared virtual and so can be overridden.

```
method InfoPage.get_CanBecomeFirstResponder: Boolean;
begin
  exit True
end;
```

With calls to `BecomeFirstResponder` and `ResignFirstResponder` in `ViewDisAppear` and `ViewDidDisappear` respectively, that covers the first step. To enable shake support, add this to `ViewDidAppear`:

```
UIApplication.SharedApplication.ApplicationSupportsShakeToEdit := True;
```

That leaves the last step of actually responding to a detected shake:

```
method InfoPage.MotionEnded(motion: UIEventSubtype; evt: UIEvent);
begin
  if motion = UIEventSubtype.MotionShake then
  begin
    interactionLabel.Text := 'iPhone was shaken';
    SetTimerToClearMotionLabel
  end
end;
```

If you were using the more common intent of a shake, meaning undo or redo, then you would need to look into the `NSUndoManager` class to help implement this.

TAPS

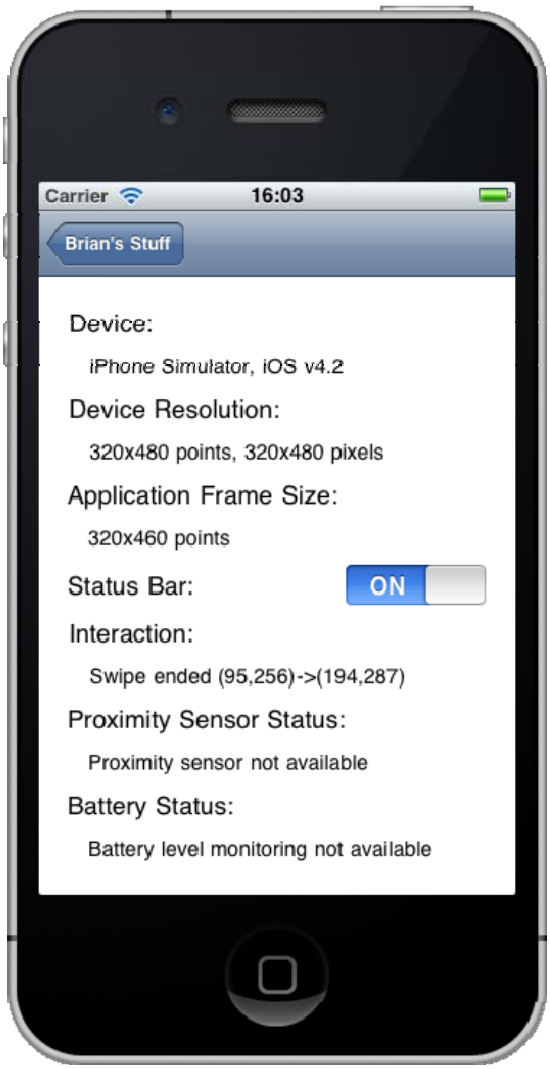
Responding to the user tapping on the screen can be done in a few ways. In this application, we'll take the approach of overriding the `TouchesBegan`, `TouchesMoved` and `TouchesEnded` methods to report what is going on. This will allow taps, double-taps and multi-taps to be detected and, potentially, swipe gestures also[11]. `TouchesCancelled` finishes the set of virtual methods that support touch operations and is useful in that it informs you when a touch operation ends abruptly due to something like a low memory condition.

[11] In general, for recognizing gestures, as opposed to simple taps, you would be well advised to look into *gesture recognizers* that take care of the hard work of tracking the sequence of coordinates for you.

```
method InfoPage.DescribeTouch(touch: UITouch): String;
begin
  Result := case touch.TapCount of
    0: 'Swipe';
    1: 'Single tap';
    2: 'Double tap';
  else 'Multiple tap' end +
  case touch.Phase of
    UITouchPhase.Began: ' started';
    UITouchPhase.Moved: ' moved';
    UITouchPhase.Stationary: " hasn't moved";
    UITouchPhase.Ended: ' ended';
    UITouchPhase.Cancelled: ' cancelled' end;
end;

method InfoPage.TouchesBegan(touches: NSSet; evt: UIEvent);
begin
  var touchArray := touches.ToArray<UITouch>;
  if touches.Count > 0 then
  begin
    var Coord := touchArray[0].LocationInView(touchArray[0].View);
    if touchArray[0].TapCount < 2 then
      StartCoord := Coord;
    interactionLabel.Text := string.Format('{0} ({1},{2})',
      DescribeTouch(touchArray[0]), Coord.X, Coord.Y);
  end;
end;

method InfoPage.TouchesMoved(touches: NSSet; evt: UIEvent);
begin
  var touchArray := touches.ToArray<UITouch>;
  if touches.Count > 0 then
  begin
    var Coord := touchArray[0].LocationInView(touchArray[0].View);
    interactionLabel.Text := string.Format('{0} ({1},{2})',
      DescribeTouch(touchArray[0]), Coord.X, Coord.Y);
  end;
end;

method InfoPage.TouchesEnded(touches: NSSet; evt: UIEvent);
begin
  var touchArray := touches.ToArray<UITouch>;
  if touches.Count > 0 then
  begin
    var Coord := touchArray[0].LocationInView(touchArray[0].View);
    interactionLabel.Text := string.Format('{0} ({1},{2})->({3},{4})',
      DescribeTouch(touchArray[0]),
      StartCoord.X, StartCoord.Y, Coord.X, Coord.Y);
    SetTimerToClearMotionLabel
  end;
end;
```

The description-making helper uses a concatenation of case expressions and also takes advantage of the Delphi Prism support for either single or double quotes for string delimiters.

These methods are passed a set of `UITouch` objects and can potentially support multi-touch operations. In the program as it is this will not happen, as multi-touch has not been enabled, and so `touchArray` in each case will hold a single `UITouch` object. Notice that the first tap coordinate is recorded into `StartCoord`, declared in the view controller class, allowing the message describing a double-tap or swipe to include this starting coordinate as well as the ending coordinate, as shown in the screenshot below.



If you need to support multi-touch in your iPhone application then you'll need to add this to one of your start-up methods, such as `ViewDidLoad` or `Initialize`:

```
(inherited View).MultipleTouchEnabled := True;
```

To simulate multi-touch in the iPhone Simulator, hold down the **Option** (or **Alt**) key.

UTILITY APPLICATIONS

Another of the available project templates lets you build a Utility (iPhone Utility Project template). This is intended to be a simple application with two views. There's a main view that comes up by default with a button that allows you to go to the other view. The main view has a little information button on it that when pressed flips over to reveal the other view (called the flipside view) in a nice animated manner. The flipside view has a navigation bar with a button that lets you go back to the main view, again using a nice flip animation.

There are a number of files in a Utility project so we should briefly run through them. An empty window and the App Delegate are defined in Main.pas, MainWindow.xib and the code behind file MainWindow.designer.xib.pas. The App Delegate creates an instance of the main view controller, sizes its view appropriately and puts the main view in the window. This main view and view controller are defined in MainView.pas and MainViewController.pas, backed by MainView.xib and the code behind file MainView.xiv.designer.pas. If you load MainView.xib into Interface Builder, you will see the Info Button and can locate the action, `showInfo:`, defined in the controller. There is a corresponding method in the main view controller that creates the flipside view controller and flips it into view, as well as setting up a custom event handler for the *Done* button on the flipside view's Navigation Bar that closes it.

This Utility project makes an interesting departure from the style of the previous projects here, in that the main view and main view controller (defined in the same .xib file) have their own source files defining the classes. This is because we have custom classes for both the view controller and the view (for both the main view and also the flipside view), whereas in previous projects the view was a predefined type, typically a `UITableView`. That said, it is still common to do the miscellaneous UI work (event handlers for the various controls) in the view controller descendant, leaving the view descendant dedicated to any custom drawing or display that it requires.

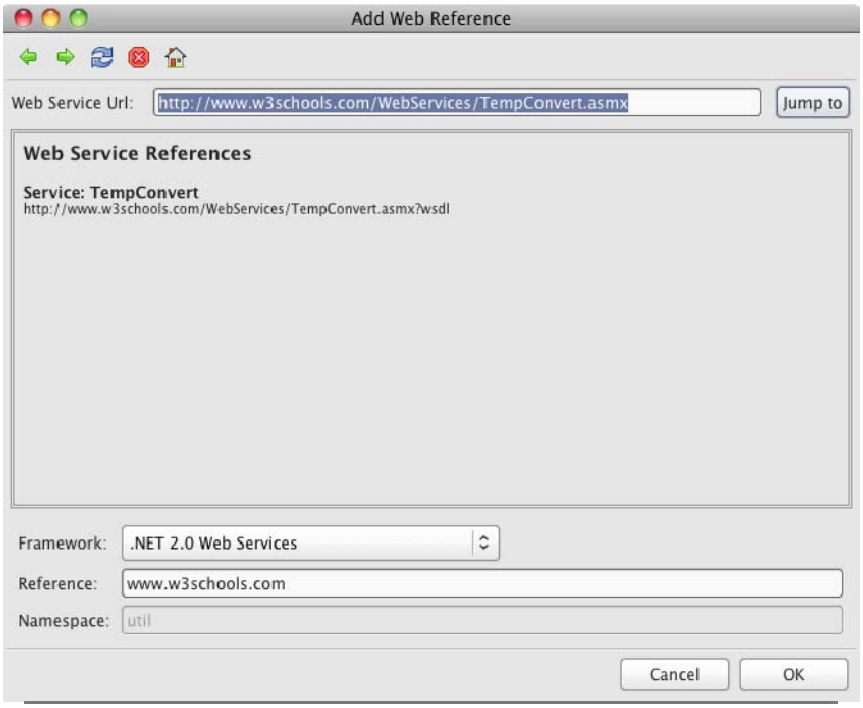
SOAP-BASED WEB SERVICES

Let's make use of a Utility project to build a web service consumer. We'll use a simple SOAP-base web service that's readily available to convert temperatures between degrees Celsius (or centigrade) and degrees Fahrenheit.

Make a new iPhone Utility Project, then in the Solution window, right-click on the project name and choose Add, Add Web Reference. This brings up the Web Services import dialog, which already has the URL of the simple temperature conversion web service filled in: http://www.w3schools.com/WebServices/TempConvert.asmx. Be sure to set the Framework to .NET 2.0 Web Services and then press *OK* to import the web service into the project.

If you use a web browser and enter that web services address you can see the capabilities of this `TempConvert` web service are fairly meager, but adequate, offering `CelsiusToFahrenheit` and `FahrenheitToCelsius`. And we'll surface these to the UI of this application.

Anyway, back to the results of using the web service import dialog:



This adds a Web References directory to the solution project. Oddly, the node contains only the reference name – www.w3schools.com – and there seems to be no direct way of opening the generated source file. However, you can see the code by opening it manually from the directory from a file called Reference.pas. The generated proxy class is found therein. Its namespace is a combination of the project name and the Reference value, so if you saved your project as WebServices, say, and left the Reference value unchanged, then the namespace that defines this proxy class will be WebServices.www.w3schools.com.

SYNCHRONOUS VS. ASYNCHRONOUS

The web service proxy class contains a plethora of methods offering different ways of calling the two web service methods, which can be simply divided into synchronous and asynchronous calls. The synchronous calls are as straightforward as this:

```
method FahrenheitToCelsius(Fahrenheit: String): String;
method CelsiusToFahrenheit(Celsius: String): String;
```

Straightforward and tempting, but you should beware of synchronous web service calls as they will block while working and freeze the UI. If the web service calls are slow or time-consuming then your application will become unresponsive and users will not enjoy that aspect of your user experience. It is much better to use asynchronous (async) calls.

There are two different sets of async declarations offered. One set follows the typical .NET asynchronous programming model.

```
method BeginFahrenheitToCelsius(Fahrenheit: String; callback: AsyncCallback;
  asyncState: Object): IAsyncResult;
method EndFahrenheitToCelsius(asyncResult: IAsyncResult): String;
method BeginCelsiusToFahrenheit(Celsius: String; callback: System.AsyncCallback;
asyncState: Object): IAsyncResult;
method EndCelsiusToFahrenheit(asyncResult: IAsyncResult): String;
```

To initiate the async web service method invocation you call either `BeginFahrenheitToCelsius` or `BeginCelsiusToFahrenheit`, passing in the textual input value, a reference to a callback that will be called when the web service method completes and also an optional piece of state information that will be passed through to the callback. Either method then returns an `IAsyncResult` object. Your application then remains responsive while the web service method executes. When it is done the callback will be invoked with the `IAsyncResult` object passed in with the optional state information available in its `AsyncState` property. The callback will not be called on the main UI thread but on the thread that the web service method was invoked on.

The result of the web service method call is obtained by calling `EndFahrenheitToCelsius` or `EndCelsiusToFahrenheit` as appropriate, passing in the `IAsyncResult` object you got earlier. This is typically done inside the callback (on that secondary thread), but can also be done in the main thread if required. For example, you could call `BeginFahrenheitToCelsius` and then continue normal UI processing in order to do some necessary operations. When the main thread has done all it needs to, or perhaps all it *can* do without the result of the web service call, you can then call `EndFahrenheitToCelsius`. If the web service call has concluded you will immediately get the result. If it is still executing then the call will block until the result is available.

The other set of methods in the web service proxy class offer asynchronous invocation in an event-driven fashion.

```
event FahrenheitToCelsiusCompleted: FahrenheitToCelsiusCompletedEventHandler;
method FahrenheitToCelsiusAsync(Fahrenheit: String);
method FahrenheitToCelsiusAsync(Fahrenheit: String; userState: Object);
event CelsiusToFahrenheitCompleted: CelsiusToFahrenheitCompletedEventHandler;
method CelsiusToFahrenheitAsync(Celsius: String);
method CelsiusToFahrenheitAsync(Celsius: String; userState: Object);
```

Here you can see two options for invoking each method, either with or without some arbitrary state information. To get the results of the calls you hook an event handler up to either `FahrenheitToCelsiusCompleted` or `CelsiusToFahrenheitCompleted` as appropriate. These events are defined with the normal .NET event signature. The event handler takes two parameters, the first being the object that triggered the event and the

seconds being an `EventArgs` descendant, either `FahrenheitToCelsiusCompletedEventArgs` or `CelsiusToFahrenheitCompletedEventArgs`. Both of these offer the web service call result in the `Result` property and the optional state information in the `UserState` property. As with the previous async callbacks these completion event handlers will be called on a secondary thread, not the main UI thread.

Accessing the UI From a Secondary Thread

Given the async completion callbacks and event handlers execute in a secondary thread, it is important to note that you will not be able to directly access the UI controls from there. Code that accesses controls will compile but will do very little.

To access the UI from a secondary thread, you must call `InvokeOnMainThread`, which takes an `NSAction`-compatible delegate (an anonymous method or lambda will work fine) containing the UI code.

Using a Web Service

Okay, after that preamble, let's build the UI for this web service and move on to calling it. In the Utility app, we'll leave the main view alone for now and add controls onto the flipside view.

We need a Text Field for entering a temperature and a Label to describe it. Give the Text Field a default Text attribute of 100 or some other temperature value, set the Return Key attribute to Done and perhaps set the Keyboard attribute to Number Pad as we only want to enter numeric temperatures[12]. To choose between Celsius to Fahrenheit or vice versa we'll use a Segmented Control (`UISegmentedControl`), which can be used as two buttons in one, or like a pair of radio buttons. You can edit each segment in turn so give the two segments Title attributes of ℃ → ℉ and ℉ → ℃.

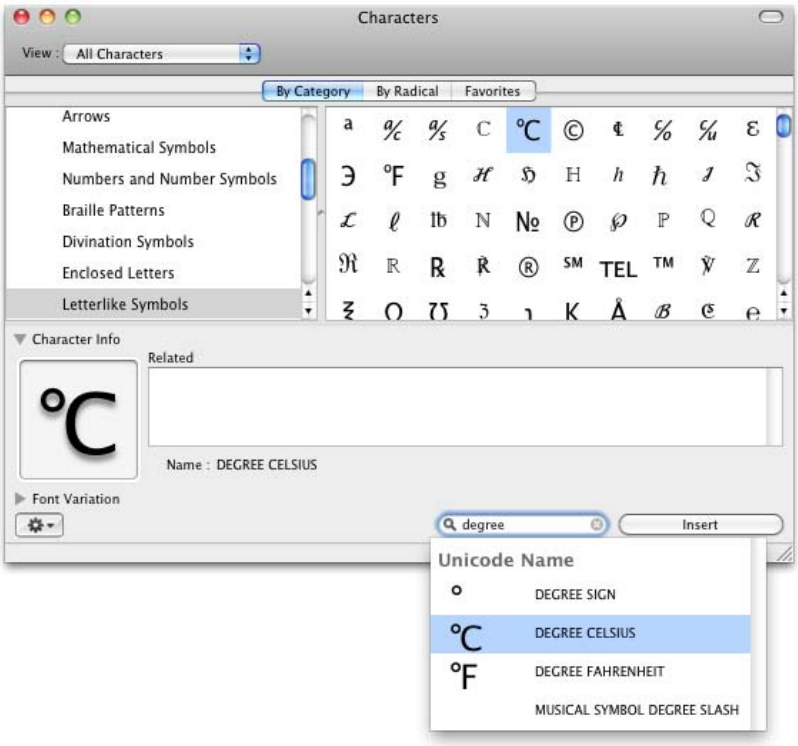
All three of the characters in those Titles are plucked from the large spread of options in the Unicode character set. To enter the characters in interface Builder, when you click in the Title attribute field choose Edit, Special Characters... from the menu (or press ⌥⌘T) to invoke the Characters window[13]. Using the lists and dropdowns on this window, you can browse around a large number of different characters, but if looking for something specific

---

[12] The number pad keyboard is just that: a numeric keypad. This means we will not be able enter fractional temperatures or negative temperatures thanks to the lack of − and . on this keyboard.

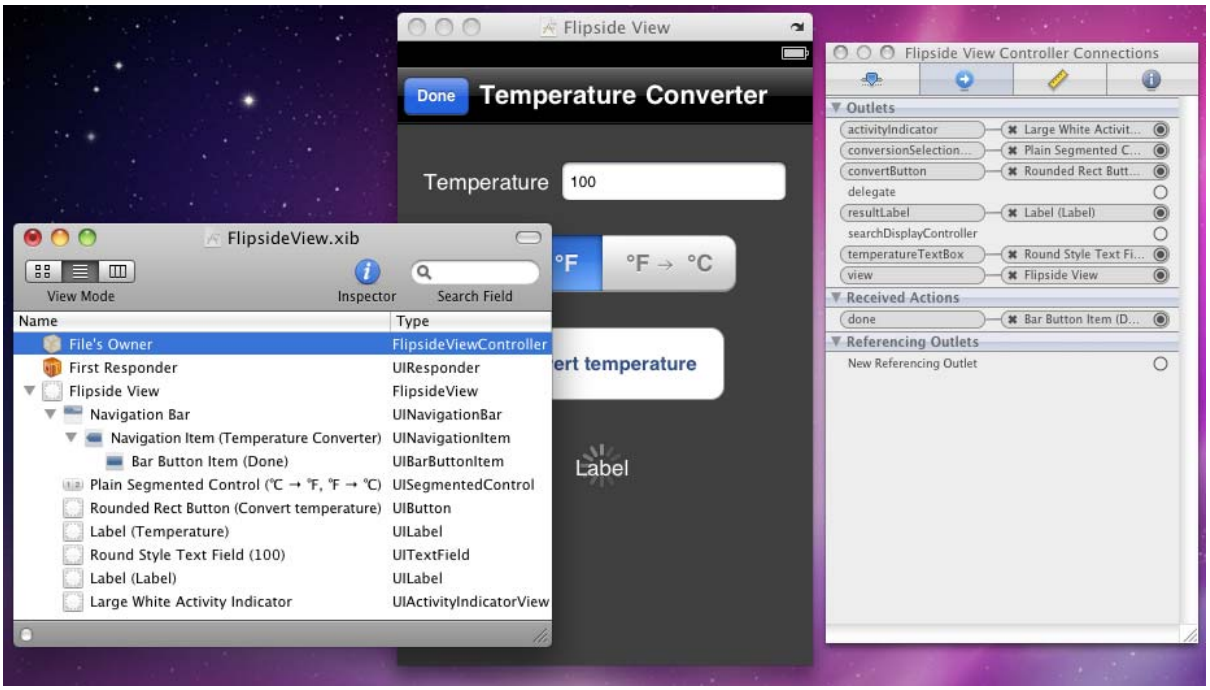
[13] You can find this menu option in many applications that allow text editing, although I notice Microsoft Word 2008 for the Mac is an exception (instead it has its own character selection option accessible through Insert, Symbol...).

the search box at the bottom is helpful. In the screenshot below, it is being used to locate the two symbols for temperature units in the two scales we are working with. Similarly, searching for *arrow* gives a large number of arrows including the rightwards arrow symbol used in the two button segment titles.



Also needed on the view in Interface Builder is a Button, with a suitable Title, to initiate the conversion and a Label to display the result. Additionally, given the web service might take some time to execute, add on an Activity Indicator View (`UIActivityIndicatorView`) and check both the Hide When Stopped and Hidden attributes.

To have the code work you will need to add five outlets to the FlipsideViewController and connect them as shown here:



They would be `activityIndicator`, `conversionSelectionButton`, `convertButton`, `resultLabel` and `temperatureTextBox`.

Don't forget to add the name of the proxy class's namespace into the `uses` clause (WebServices.www.w3schools.com if you saved the project as WebServices before importing the web service reference). Once that is done, the code can be added.

In the `ViewDidLoad` method we'll clear the result label, create an instance of the web service proxy class and set up an event handler to respond to the conversion button being touched:

```
resultLabel.Text := '';
Converter := new TempConvert();
convertButton.TouchUpInside += ConvertButtonTouched;
```

`Converter` is a variable declared in the view controller class. When the convert button gets touched, this is the code that executes:

```
method FlipsideViewController.ConvertButtonTouched(sender: Object;
  e: EventArgs);
begin
  resultLabel.Text := '';
  var InputTemp: Double;
  if Double.TryParse(temperatureTextBox.Text, out InputTemp) then
  begin
    UIApplication.SharedApplication.NetworkActivityIndicatorVisible := True;
    activityIndicator.StartAnimating;
    if conversionSelectionButton.SelectedSegment = 0 then
```

```
      Converter.BeginCelsiusToFahrenheit(temperatureTextBox.Text,
        @FinishedTemperatureConversion, True)
    else
      Converter.BeginFahrenheitToCelsius(temperatureTextBox.Text,
        @FinishedTemperatureConversion, False);
  end
  else
    resultLabel.Text := 'Invalid input temperature'
end;
```

Assuming the text in the Text Field is numerical (given we specified a number pad then it will either be a number or a blank string) then we start off by turning on the network activity indicator on the status bar and also initiating our activity indicator control's animation. This way the user knows the application is busy doing something and that the network is being accessed. The segmented control is checked to see in which direction the conversion should be made and this controls whether `BeginCelsiusToFahrenheit` or `BeginFahrenheitToCelsius` is called. In this implementation a single method is used to act as a callback; in case we need to know which direction the conversion was requested, a Boolean value is passed through as the optional state information, and is designed to indicate if it was a Celsius to Fahrenheit conversion (or not). The callback looks like this:

```
method FlipsideViewController.FinishedTemperatureConversion(
  AsyncResult: IAsyncResult);
begin
  var FromCelsius := Boolean(Asyncresult.AsyncState);
  var Answer :=
    iif(FromCelsius, Converter.EndCelsiusToFahrenheit(AsyncResult),
                     Converter.EndFahrenheitToCelsius(AsyncResult));
  // Since we are running on a separate thread, we can not access UIKit
  // objects from here, so we need to invoke those on the main thread:
  InvokeOnMainThread(() ->
  begin
    resultLabel.Text := String.Format(
      iif(FromCelsius, '{0}℃ is {1:F2}℉', '{0}℉ is {1:F2}℃'),
      temperatureTextBox.Text, Convert.ToDouble(Answer));
    UIApplication.SharedApplication.NetworkActivityIndicatorVisible := False;
    activityIndicator.StopAnimating;
  end);
end;
```

The optional state information is extracted and restored to Boolean type. This is used to decide whether to call `EndCelsiusToFahrenheit` or `EndFahrenheitToCelsius` to extract the result of the temperature conversion, after passing in the received `IAsyncResult` object. In order to write the result out and turn off the activity indicators, we execute the next bit of code indirectly via `InvokeOnMainThread`, since we aren't running in the UI thread (as discussed earlier).

That works quite well except for the keyboard, which has not been encouraged to disappear after you've entered your chosen temperature. That can be addresses by

hooking event handlers to trigger when any other control is used and implementing them to make the keyboard go away. These two statements can be added to `ViewDidLoad` – note that the convert button now has two `TouchUpInside` event handlers:

```
convertButton.TouchUpInside += AnyNonTextControlTouched;
conversionSelectionButton.ValueChanged += AnyNonTextControlTouched;
```

The handler itself is straightforward and matches what we have seen in a previous example:

```
method FlipsideViewController.AnyNonTextControlTouched(Sender: Object;
  E: EventArgs);
begin
  temperatureTextBox.ResignFirstResponder
end;
```

Now we have a functioning web services application with the functionality resident on the flipside of a blank main screen. Before looking to address that, it might be good to see what the code looks like when using the event-based async web service methods as opposed to the `Begin`/`End` async methods we just employed. These statements need adding to the `ViewDidLoad` method immediately after constructing the web service proxy:

```
Converter.CelsiusToFahrenheitCompleted += FinishedCelsiusToFahrenheit;
Converter.FahrenheitToCelsiusCompleted += FinishedFahrenheitToCelsius;
```

Notice we now have two separate event handlers as they take a different `EventArgs`-descendant parameter. The button event handler changes to look like this:

```
method FlipsideViewController.ConvertButtonTouched(sender: Object;
  e: EventArgs);
begin
  resultLabel.Text := '';
  var InputTemp: Double;
  if Double.TryParse(temperatureTextBox.Text, out InputTemp) then
  begin
    UIApplication.SharedApplication.NetworkActivityIndicatorVisible := True;
    activityIndicator.StartAnimating;
    if conversionSelectionButton.SelectedSegment = 0 then
      Converter.CelsiusToFahrenheitAsync(temperatureTextBox.Text)
    else
      Converter.FahrenheitToCelsiusAsync(temperatureTextBox.Text);
  end
  else
    resultLabel.Text := 'Invalid input temperature'
end;
```

Notice that since we have two separate web service method completion event handlers there is no need for the optional state parameter to be passed through. Here are those event handlers:

```
method FlipsideViewController.FinishedCelsiusToFahrenheit(sender: Object;
  args: CelsiusToFahrenheitCompletedEventArgs);
begin
  InvokeOnMainThread(() ->
  begin
    resultLabel.Text := String.Format('{0}℃ is {1:F2}℉',
      temperatureTextBox.Text, Convert.ToDouble(args.Result));
    UIApplication.SharedApplication.NetworkActivityIndicatorVisible := False;
    activityIndicator.StopAnimating;
  end);
end;

method FlipsideViewController.FinishedFahrenheitToCelsius(sender: Object;
  args: FahrenheitToCelsiusCompletedEventArgs);
begin
  InvokeOnMainThread(() ->
  begin
    resultLabel.Text := String.Format('{0}℉ is {1:F2}℃',
      temperatureTextBox.Text, Convert.ToDouble(args.Result));
    UIApplication.SharedApplication.NetworkActivityIndicatorVisible := False;
    activityIndicator.StopAnimating;
  end);
end;
```

IMAGES

On the main view of your Utility Project you can clearly put what you like. In this case we will add an image. We could use Interface Builder to add an appropriate control to the main view and set its attributes, but this time we'll skip Interface Builder and do it solely in code.

To add an image to an iOS project you can copy the file into the source directory tree; mine is called Fire.jpg (whose dimensions are smaller than that of an iPhone screen) and has been placed in the project's main directory, alongside the source. In MonoDevelop you notify the project of the image's existence firstly by adding the image to the project (right-click the project and choose Add, Add Files... or press ⌥⌘**A**), and secondly by right-clicking the image in the project and choosing Build Action, Content.

Now in the main view controller's `ViewDidLoad` you can write:

```
method MainViewController.ViewDidLoad;
begin
  inherited ViewDidLoad();
  //Deal with image
  var image := new UIImage("Fire.jpg");
  var imageView := new UIImageView(image);
  with appF := UIScreen.MainScreen.ApplicationFrame, imgS := image.Size do
    imageView.Frame := new RectangleF((appF.Width - imgS.Width) div 2,
      (appF.Height - imgS.Height) div 2, imgS.Width, imgS.Height);
  (inherited View).AddSubview(imageView);
end;
```

This loads the image file into an Image control, adds it to an image view, sets the image view frame to be centered in the application frame and finally adds the image view into the main view. The main view of the application can now be seen below. Something a little more helpful and informative would be better in a real application, but there is a loose reason why the picture was chosen. This is a temperature conversion program and fires imply high temperatures.

DRAGGABLE CONTROLS

If you want to play about you could make this image draggable. Let's build a new class `UIDraggableImageView` that inherits from `UIImageView` and supports being dragged around. The class is actually quite simple and operates by overriding the methods `TouchesBegan`, `TouchesMoved` and `TouchesEnded` to shift the image view's frame around based upon where the user drags their finger. You may recall we used those methods to track touch movement in an earlier project.

The class definition looks like this:

```
type
  UIDraggableImageView = public class(UIImageView)
```

```
  private
    StartLocation: PointF;
    method MoveImage(touches: NSSet);
  public
    constructor(image: UIImage);
    method TouchesBegan(touches: NSSet; evt: UIEvent); override;
    method TouchesMoved(touches: NSSet; evt: UIEvent); override;
    method TouchesEnded(touches: NSSet; evt: UIEvent); override;
  end;
```

The constructor and methods are implemented thus:

```
constructor UIDraggableImageView(image: UIImage);
begin
  inherited;
  UserInteractionEnabled := True;
end;

method UIDraggableImageView.MoveImage(touches: NSSet);
begin
  // Move relative to the original touch point
  var pt := (touches.AnyObject as UITouch).LocationInView(Self);
  // Use a separate frame var to ensure both new coordinates are set at once
  var frame := Self.Frame;
  with appFrame := UIScreen.MainScreen.ApplicationFrame do
  begin
    frame.X := frame.X + pt.X - StartLocation.X;
    if frame.X < 0 then frame.X := 0;
    if frame.X + Image.Size.Width > appFrame.Width then
      frame.X := appFrame.Width - Image.Size.Width;
    frame.Y := frame.Y + pt.Y - StartLocation.Y;
    if frame.Y < 0 then frame.Y := 0;
    if frame.Y + Image.Size.Height > appFrame.Height then
      frame.Y := appFrame.Height - Image.Size.Height;
  end;
  Self.Frame := frame;
end;

method UIDraggableImageView.TouchesBegan(touches: NSSet; evt: UIEvent);
begin
  var pt := (touches.AnyObject as UITouch).LocationInView(Self);
  StartLocation := pt;
  Superview.BringSubviewToFront(Self);
end;

method UIDraggableImageView.TouchesMoved(touches: NSSet; evt: UIEvent);
begin
  MoveImage(touches)
end;

method UIDraggableImageView.TouchesEnded(touches: NSSet; evt: UIEvent);
begin
  MoveImage(touches)
end;
```

Since the image is intended for single touch, `TouchesBegan` just plucks any of the touch objects from the passed in set; typically there will only be one anyway. The location of that

touch is then recorded as the starting point. As the user's finger is moved around the `MoveImage` helper routine does the job of incrementing or decrementing the x and y coordinates of the frame and ensuring it is kept inside the screen boundaries.

This class was based on a code snippet from http://monotouchexamples.com.

LAUNCH SCREENS

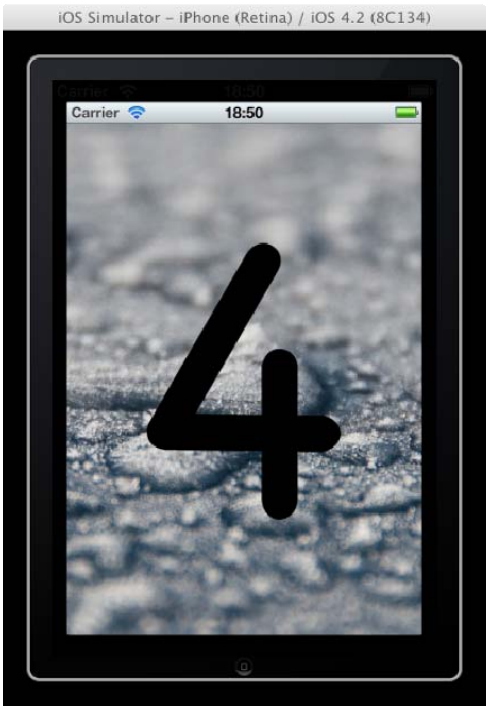
The example above includes an image but it is embedded in one of the views. A common aspect of commercial applications is a launch screen (or splash screen), which serves to distract the user from the length of time the application takes to start up and initialize. The Apple UI guidelines suggest that a splash screen should be an image of the application running but this is often ignored. Applications typically include company marketing information, logos for the application and the company, etc. But how do we include a splash screen? Moreover, how do we include icons so our applications stand out a little more than they currently do?

The answer to the questions is to add various specially named files to the project and set their Build Action to Content. Let's look at a launch screen first.

To incorporate a launch screen, it is simply a matter of adding a file called Default.png to your project, set to a size of 320x460 (the iPhone resolution is 320x480 and this image size takes account of the 20 pixel high status bar). That image will work for iPhone 3 and iPod Touch. It will also work with iPhone 4 but you may recall that the pixel resolution of iPhone 4 has doubled. To cater better for the improved display (and avoid the original image being stretched to fit) you can also include a 640x920 file called Default@2x.png and this will be used automatically when the application is launched on an iPhone 4.

This Utility Project we have been working on is included in the files that accompany this paper. In the project are two launch images as per the specification above. They are the same image, a picture of some ice (cold temperature), but Default.png has been shrunk to the smaller resolution. In order to prove the point and be sure the right file is used, a big number 3 has been drawn on the smaller file and a big number 4 on the larger file. The following screenshot shows this same application being launched in the iPhone 4 Simulator.

You can switch between simulating the various hardware options in the Simulator by choosing Hardware, Device and then either selecting iPhone (for the iPhone 3 and iPod Touch Simulator), iPhone (Retina) (for the iPhone 4 Simulator) or iPad.



SUPPORTING THE IPAD

If you run any of your applications in the iPad Simulator they won't look like they fit in very well as all our windows and views have been hardcoded to fit to a resolution of 320x480. If you want an application to run on the iPad then you really ought to design it for the iPad and the various features it offers, not least of which, the 1024x768 screen resolution.

If you start with the iPad Window-based project then your UI will be sized sensibly. The Universal Window-based project covers both options by having two nib files, one for the iPhone resolution and one for the iPad resolution. However you choose to go, there is support for the iPad.

If iPad is a valid target for your application you should also explore the various project options available (right-click on your project in the Solution window and choose Options), especially on the iPhone Build and iPhone Application pages.

Space doesn't really allow much on the specifics of iPad programming but it's safe to say that all the techniques we've looked at thus far are perfectly applicable to programming an iPad application. However, with regard to launch screens there are various different files required to cater for the different orientations the iPad application may be started in. You can find full details in the post at http://phunkwerk.posterous.com/ipad-managing-multiple-launch-images-aka-defa.

ICONS

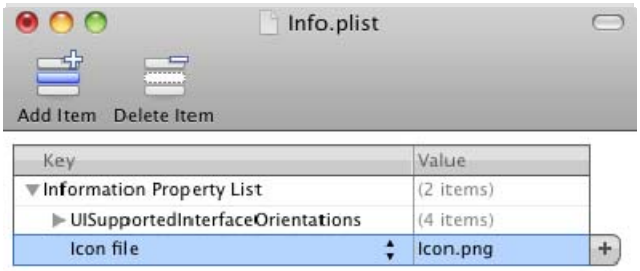
There are three places that an icon is used in iOS: the Home screen, the Settings screen and the Spotlight (search) screen. Each of these places may require an icon of different dimensions. Depending on what devices you are targeting you may have to consider additional dimensions of each image again as iPhone 3 and iPod Touch use one set of resolutions, iPhone 4 uses another and iPad uses yet another.

In the simple case of targeting iPhone 3 and iPod Touch, you would include two image files for these icons. Icon.png (57x57) is used on the Home screen and Icon-Small.png (29x29) is used on the Settings screen and also on the Spotlight screen.

If you were building a Universal application targeting all iOS devices then you would need to include all these icon files:

File name	Image dimensions	Purpose
Icon.png	57x57	iPhone/iPod Touch Home screen
Icon@2x.png	114x114	iPhone 4 Home screen
Icon-72.png	72x72	iPad Home screen
Icon-Small.png	29x29	iPhone/iPod Touch Spotlight and Settings, iPad settings
Icon-Small@2x.png	58x58	iPhone 4 Spotlight and Settings
Icon-Small-50.png	50x50	iPad Spotlight

Unlike the launch screen image, these icons are not automatically used by iOS. Instead you have to specify the root file name in an entry in your project's Info.plist file. This is an XML Information Property List file processed by iOS to set things up correctly for your application. Double-clicking on Info.plist invokes the property list editor. By default, you'll see the file has very little in it. You must add in a new entry in using the *Add Child* button (or *Add Item* button, depending on what is selected). The new entry (the key) should be Icon file (this can be selected from the drop down list) and the value should be the root icon file name as shown below.



The web services example contains the iPhone and iPhone 4 icons and also has a customized display name on the Home screen thanks to one of the project options. On the iPhone Application page of the options, the Display Name value in the Application Bundle section controls what text is written under the icon on the Home screen.

Note: The release version of Delphi Prism XE has an occasional bug when running in MonoDevelop. After browsing through the options, you will sometimes find that subsequent compilations will fail with an unexpected error like:

Error CE19: Cannot open file "/Users/brian/Projects/Prism/MyProject" (CE19) (MyProject)

This will typically be due to this intermittent issue where somehow the Win32 Icon option (found in the project options dialog in the Build, General section) has been given the project directory as a value. Unfortunately, it seems this cannot be rectified from within MonoDevelop. Instead you should close MonoDevelop and open the project file (it has an

.oxygene extension, so Myproject.oxygene for example) in a text editor. Locate the line that looks like this (with your project directory):

```
<ApplicationIcon>/Users/brian/Projects/Prism/Pages</ApplicationIcon>
```

and simply delete the whole line.

DEBUGGING

When running MonoTouch applications in the iOS Simulator from MonoDevelop, you have access to a very capable debugger in the MonoDevelop IDE. It has the usual features of breakpoints, stepping into calls, stepping over calls, watches, local variables, call stack, thread information, etc. All that is needed is some time to get used to where the windows and controls are and what shortcuts are available.

Some important things to note, however, include the fact that when debugging an application it takes up much more space than a release version and will be much slower due to the way debugging is implemented.

Additionally, you need to take care when trying to debug code that executes during the application startup process, for example in your main view controller's `ViewDidLoad` method or the App Delegate's `FinishedLaunching` method. iOS keeps an eye on an application as it starts and checks to see it makes it past startup to be usable by the user in a timely fashion. If it gets held up in the startup process for more than 10-20 seconds, iOS will assume the application has hung and so will kill it. So if you place a breakpoint in one of those startup methods, you get very little time to debug anything before the application is removed before your very eyes.

To debug code in the main view's startup methods, consider making it a secondary view launched from a button on a temporary main view. Alternatively, use the very common process of logging information to the Application Output window in order to track down your bug. Calls to `Console.WriteLine` get listed in the MonoDevelop Application Output window and this proves to be a very handy tracking mechanism.

TECHNICAL RESOURCES

Delphi Prism extends the regular Delphi language in various ways. You can get a heads-up at these pages: http://prismwiki.embarcadero.com/en/Language, http://prismwiki.embarcadero.com/en/The_Prism_Primer, http://prismwiki.embarcadero.com/en/Anonymous_Methods_and_Delegates.

The MonoTouch API Reference is at http://www.go-mono.com/docs (towards the bottom of the hierarchy on the left of the page). Tutorials (C# biased) can be found at

http://monotouch.net/Tutorials, How-To documents at http://wiki.monotouch.net/HowTo and other articles at http://monotouch.info.

The iOS Reference Library is at http://developer.apple.com/library/ios.

A recommended book is *Professional iPhone Programming with MonoTouch and .NET/C#* by Wallace B. McClure, Rory Blyth, Craig Dunn, Chris Hardy, Martin Bowling (http://www.amazon.com/dp/047063782X).

SQLite SQL syntax is documented at http://www.sqlite.org/lang.html.

CONCLUSION

This white paper has shown how Delphi Prism can be utilized to leverage .NET programming knowledge and start building applications for iPhone, iPod Touch and iPad in a very capable manner.

The MonoTouch SDK provides a convenient means of starting to write iPhone apps without having to learn an entirely new programming language. Of course, an understanding of CocoaTouch (or at least CocoaTouch.Net) needs to be built up in order to do the job well, but working with a language you are familiar with helps ease the transition to a new UI framework.

Business logic can potentially be reused in applications running on iOS if partitioned sensibly, but clearly an entirely different UI needs to be developed for this type of application.

About the Author

Brian Long has spent the last 1.5 decades as a trainer, trouble-shooter and mentor focusing on the Delphi, C# and C++ languages, and the Win32, .NET and Mono platforms, recently adding iOS and Android onto the list. In his spare time, when not exploring the Chiltern Hills on his mountain-bike or pounding the pavement in his running shoes, Brian has been re-discovering and re-enjoying the idiosyncrasies and peccadilloes of Unix-based operating systems. Besides writing a Pascal problem-solving book in the mid-90s he has contributed chapters to several books, written countless magazine articles, spoken at many international developer conferences and acted as occasional Technical Editor for Sybex. Brian has a number of online articles that can be found at http://blong.com.

Embarcadero Technologies, Inc. is the leading provider of software tools that empower application developers and data management professionals to design, build, and run applications and databases more efficiently in heterogeneous IT environments. Over 90 of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero's award-winning products to optimize costs, streamline compliance, and accelerate development and innovation. Founded in 1993, Embarcadero is headquartered in San Francisco with offices located around the world. Embarcadero is online at www.embarcadero.com.