iOS PROGRAMMING FOR .NET DEVELOPERS

a field guide to the other side

JOSH SMITH

iOS Programming for .NET Developers
A Field Guide to the Other Side
By Josh Smith

Thank you for checking out this sample of my book!

The Author

Josh Smith is a mobile software artisan who sharpened his teeth for many years in the trenches of .NET programming. Four of those years were spent as a Microsoft MVP thanks to his technical and written contributions in the Client Application Development space. His passion for iOS programming and the future of mobile software compels him to do crazy things like write books in his free time.

Josh is a Senior Experience Developer with Cynergy, an agency that designs and develops excellent mobile software experiences. He is blessed with the opportunity to architect and develop complex mobile software solutions for Fortune 500 companies.

When he isn't being a computer geek Josh spends time playing the piano, riding his bicycle, studying whatever topic tickles his fancy, and enjoying quality time with his lovely wife Sarah and their dog Thor. Visit Josh online at iJoshSmith.com or get in touch with him over email at iOSArtisan@gmail.com

The Technical Reviewers

Jordan Nolan is the Founder and Principal of Buttercup Mobile, a Boston-based consulting firm focused on enterprise mobile software development. He is a seasoned software developer, architect, and manager with extensive experience designing and delivering software solutions for Fortune 500 companies. Before discovering the beauty of iOS Jordan worked extensively with .NET technologies such as WPF, Silverlight, and ASP.NET. If your company is looking for a mobile solution feel free to contact him at jordan.nolan@buttercupmobile.com

Josh Wagoner is a software developer with fourteen years of experience. He has been developing iOS applications since 2009. Before moving to iOS development he worked with numerous Microsoft technologies; including WPF, Silverlight, ASP.NET and COM. Currently he is a co-founder of Kairos and does freelance development consulting. Contact Josh to discuss iOS contract work at josh@theotherdingo.com

[Part I: Prepare for Landing]

The structure of this book parallels the process of colonizing another planet. The first few chapters prepare you for landing on the planet iOS, with a review of its ecosystem and the tools needed to survive. The second part shows how to speak with the locals in their native language of Objective-C, and relate their unusual ideas to your .NET worldview. Once you have acclimatized and gotten a lay of the land, the third and final part of the book explains how to start building your software civilization with their strange, yet elegant, technology.

In other words, this book teaches .NET developers how to understand and work with a radically different programming platform. First things first, let's review what iOS is all about and how to find your way around Apple's development environment.

Chapter 1: Greetings, Earthling!

We are witnessing the dawn of the mobile revolution. Mobile devices have been part of the computing landscape for quite some time, but only in the past two or three years have they become ubiquitous. Demand for mobile applications in the consumer space is exploding. Many people now expect companies to have a mobile presence. Similarly, there is large and rapidly growing demand for mobile software in the business world, which is where most professional software developers earn a living. The majority of professional software developers have little to no experience creating mobile applications. While the iron is in the fire, the wise developer prepares to strike.

Why learning iOS matters

It should come as no surprise to hear that Apple has had enormous success with their iOS operating system and the devices on which it runs. The iPhone redefined the term "smartphone." The iPad is the world's first successful tablet computer, and it created a market in which it is unrivaled. These computing devices, and the software running on them, have propelled Apple into a position of fabulous wealth and power.

The world has changed. Microsoft is no longer the supreme ruler in the realms of personal and business computing. There is a new contender in living rooms, coffeehouses, and office buildings around the world. People love their Apple devices and are obsessed with apps. Business executives are becoming aware of the unique advantages and capabilities that mobile devices offer. The world is willing and eager to pay for iOS apps.

If that isn't a good reason to learn iOS programming, what is?

Why you should read this book

After having spent many years doing only .NET development, this is the book I wished for when I started learning native iOS programming. There have been countless books published about every conceivable topic in iOS. Unfortunately,

none of them were designed to help you get over the steep iOS learning curve by leveraging your existing knowledge of .NET development. I wanted a book that would help me map what I already knew to what I was learning. In essence, I wanted a tour guide who could also translate. If that is how you would like to learn iOS programming, this book is for you.

This book is not the one and only thing you should read to master iOS programming. It is a field guide kept in your pocket while exploring the foreign lands of Apple. Along the way I recommend books and online resources that dive deep into topics covered here at a high level.

The sample code is not available online

I firmly believe that typing code is a much better way to gain experience with a programming platform than pasting code copied from a sample project. This book contains many short code snippets, but is not accompanied by a source code download. To use this book's code snippets in an app, you will need to type them by hand.

Native vs. MonoTouch vs. Hybrid

The subject of this book is native iOS programming using Objective-C, Xcode, and Cocoa Touch. It is not about how to use MonoTouch, nor will I discuss making apps that are simple shells around Web pages imitating the look and feel of an iOS app. With that said, it is worthwhile to briefly review some of the options available for making apps that run on an iOS device.

- **Native** development means creating iOS apps using the platform, frameworks, and tools published and supported by Apple. This typically involves using the Xcode Integrated Development Environment (IDE) to edit and compile Objective-C source code that makes use of frameworks published by Apple and third parties. This is the only kind of iOS app guaranteed to have full access to all the publicly available features of the device's operating system. Objective-C code runs very fast and is supported by a lightweight runtime, somewhat analogous to the .NET Common Language Runtime.

- The major advantages of "going native" are that your code runs very fast and has full access to iOS platform features, the platform is well documented by Apple, and it is actively discussed and explained on many blogs and community sites like StackOverflow.

- The major disadvantage is the daunting task of learning a different programming language, platform, and IDE.

- **MonoTouch** is a product previously developed by Novell, but now developed by a small company named Xamarin. It provides an SDK for Mac OS X that lets you use .NET programming languages to create apps for iOS devices. MonoTouch is based on the Mono implementation of the .NET framework. The code in a MonoTouch app is compiled down to native ARM assembly code, rather than the Intermediate Language bytecode of normal .NET applications.

    - The big appeals of MonoTouch are that it reduces the learning curve for a .NET developer getting into iOS programming, and some of your .NET code can be reused on other mobile platforms.

    - The big drawback is that you depend on Xamarin to expose a stable subset of the iOS SDK that includes the platform features your apps need. You also need to hope that Xamarin does not stop supporting MonoTouch, or go out of business.

- **Hybrid** apps consist of a thin native shell around a UIWebView control, which renders HTML and executes JavaScript. A hybrid application is deployed as a set of Web pages made available by a native iOS host app that can be downloaded from the App Store. This approach is commonly implemented using products such as PhoneGap or Sencha Touch. These products involve an interop system that allows your JavaScript code and the mobile device to communicate with each other.

    - The major benefits of taking the hybrid approach are that Web developers can leverage their HTML, CSS, and JavaScript skills, and these Web pages can, in theory, run on other mobile platforms (e.g. Android).

    - The major downsides are that your app will have limited access to the host device platform, custom plugins that access unexposed device features must be written in native code (not JavaScript), the UI will not look and feel quite like a native iOS app, and some of these products are known to have significant breaking changes from release to release. It is Web programming, after all!

How iOS exposes platform features

In the previous section I mentioned that native applications are guaranteed to have complete access to all the publicly available features of iOS. Let's take a moment to see how the operating system partitions and exposes those features to an application.

Frameworks are the iOS equivalent of .NET assemblies. They are reusable code libraries that expose a set of related functionality through an application programming interface (API). When a new iOS project is created in Xcode, it links to a few of the most essential frameworks by default. If an application needs functionality in another framework, such as using EventKit to gain access to the user's calendar event data, add a link to it in Xcode and then your app can use its API.

Certain low-level or less commonly used features of the operating system are not exposed through a framework. Instead, these specialty features are made available through a *dynamic library*, which has the file extension of *dylib*. Most iOS apps don't directly use a dynamic library; except for *libsqlite3.dylib* which is often used to manage local SQLite databases.

Xcode vs. AppCode

In this book I show screenshots of and make references to Apple's IDE, named Xcode. There is another IDE, published by JetBrains, that some people prefer for writing Objective-C code, called AppCode. At the time of this writing, AppCode has richer support for refactoring, quick-fix, and unit testing than Xcode. This makes sense considering JetBrains is the company that built the ReSharper plug-in for Visual Studio, which adds many very popular refactoring and quick-fix features to that IDE.

On the other hand, AppCode lacks some fundamental features such as integration with Interface Builder (the topic of Chapter 8), integration with the Core Data model editor (reviewed in Chapter 10), and the ability to create an IPA file (which is submitted to the App Store). Working in AppCode typically involves

having an instance of Xcode open as well. I mention this to make you aware that Xcode is not the only IDE you can use for iOS development. By the time you read this it is quite possible that AppCode will have matured considerably so it is definitely worth investigating, if you are interested.

Setting up a development environment

You can be up and running with a fully functional iOS development environment in less than an hour. You will need a decent Apple computer, but nothing too fancy. If you're on a budget consider buying a Mac Mini, which is a relatively inexpensive desktop option that can be plugged into any monitor.

Upgrade to the latest version of the OS X operating system, then download and install Xcode using the Mac App Store. The App Store ships with the operating system and allows you to browse, buy, and install applications on your Mac.

iOS simulator vs. iOS device

iOS apps can be run and debugged without using an iOS device. Xcode's installer includes the iOS Simulator, which is a desktop application that simulates an iOS device. When running an app in the simulator Xcode will attach a debugger to it so that you can stop at breakpoints in your code, just like in Visual Studio. The simulator is quite good at what it does, and can help you hit the ground running when learning how to write iOS programs. Eventually, however, apps should be tested on a real iOS device. As discussed in the next section, you will need to join Apple's iOS Developer Program and follow their instructions to prepare a development environment for running apps on an iOS device. Once that chore is out of the way simply plug an iOS device into your Mac via a USB port and Xcode will detect it, allowing the app to be run on that device.

There are major differences between the iOS Simulator and a real iOS device. For example, the simulator runs on the CPU of your desktop or laptop, which is much more powerful than the processor in a mobile device. This causes the simulator to run apps faster than a real iOS device. Likewise, the simulator has a huge amount of RAM compared to what is available on, say, an iPhone. Running

in the simulator will prevent an app from ever using up all of its memory, which is a serious concern when running on an iOS device.

The simulator does not support several major APIs available in iOS. For example, the camera APIs do not work and push notifications cannot be received. A real iOS device must be used to test and debug the parts of your application that leverage these features.

On the other hand, the simulator has many useful features for testing and debugging apps that are not available on a real iOS device. It can simulate low-memory conditions by sending an app warnings to reduce its memory footprint. It can slow down animations, making them easier to troubleshoot. Also, the iOS Simulator can simulate multiple iOS device types, including iPhones and iPads with and without a Retina (hi-def) display.

Join the iOS Developer Program

To run your apps on an iOS device, and submit them to the App Store, you must be a registered iOS developer. Sign up with Apple and pay a $99 yearly fee to be in the iOS Developer Program. Once part of the program you will also have full access to Apple's developer resources in the iOS Developer Center. The details of how to sign up are easy to find online, so I won't bother explaining it here.

Some people moan and groan about having to pay $99 per year to be able to run their apps on an iPhone or iPad. That is a silly way to think about it. Consider it an investment in your career. Plus, if you publish an app or two in the App Store, you might earn that money right back, and then some. I did!

Summary

The software development world has no choice but to adapt to what is now a full-scale mobile revolution. It's here, it's happening, and it's unstoppable. Apple's mobile platform is the dominant player and should not be ignored. This book was designed to help you leverage your existing .NET knowledge and skills in the exotic landscape of native iOS programming. It is easy to set up a development

environment and immediately begin learning this valuable new skill set. So, why wait? Let's get started...

Chapter 2: From Windows to OS X

Writing software for Apple's mobile platform requires fluency with their desktop operating system, OS X. There are plenty of books and Web sites that provide all the details for learning how to master OS X, but much of that information is unnecessary to get started with developing iOS apps. This chapter provides an OS X newbie with tips and tricks for the most common and necessary things needed to be productive in OS X. If you are already fluent with a Mac, feel free to skip this chapter.

Chapter 3: From Visual Studio to Xcode

As developers we tend to be passionate about the tools we use to write and debug code. Most .NET developers are accustomed to working in an Integrated Development Environment (IDE). The vast majority of us know and love, to varying degrees, Microsoft's Visual Studio (VS). When working on iOS software, however, you cannot use VS. In this chapter, I show how to use Apple's IDE called Xcode. At the time of this writing the current version of Xcode is 4.3.3.

I have met people who try out Xcode and immediately say how terrible it is. They point out all the ways that it is not like VS, or whatever IDE they know best. Those who take the time to learn Xcode typically end up liking it. It's certainly not perfect, but it is a powerful and flexible tool used to create many great applications.

This chapter is a high-level overview of how to use Xcode, pointing out parallels it has with VS. I present enough practical knowledge so that you can start using Xcode right away without feeling like you've suddenly found yourself on Mars. There are whole books devoted entirely to Xcode 4, and Apple has published great documentation on how to use their IDE. It's a big topic, worthy of serious study.

[Part II: Ways of the World]

Learning a language, such as Italian or French, requires studying separate aspects of it in isolation. One must study grammar, vocabulary, and usage before combining them to form sentences and paragraphs that make sense to a native speaker.

In this part of the book I present how the Objective-C language can be used to make an iOS device bend to your will. The first step is to learn the "grammar" of Objective-C; it's syntax, parts of speech, etc. Chapters 4 and 5 explain the fundamentals of Objective-C and compare it to the C# language, with which so many .NET developers are fluent. Chapters 6 and 7 focus on the "vocabulary" of iOS development; namely, foundational classes and user interface controls used by almost all apps.

In the third and final part of the book, Chapters 9 and 10 contain lessons about "usage." They include code examples that combine Objective-C's grammar and vocabulary to solve realistic programming problems.

Chapter 4: From C# to Objective-C

The name "Objective-C" reveals two important facts about the language; it is an *object*-oriented superset of the *C* programming language. It is possible to write a program in Objective-C that does not use any classes, only C structures and functions. In practice, most Objective-C programs make heavy use of the language's object-oriented features, and include old-school C code when necessary.

.NET developers are familiar with a style of object-orientation strongly influenced by C++ and Java. Objective-C's style of object-orientation comes from a different lineage, strongly influenced by Smalltalk. The core concepts of classes, objects, instance methods, polymorphism, etc., are the same regardless of language. There are, however, enough differences in how the concepts are actualized by Objective-C to merit review.

All code examples in this chapter make use of Automatic Reference Counting (ARC), which prevents the code from being encumbered with memory management concerns. Memory management and ARC are the topic of the next chapter.

Chapter 5: From Garbage Collection to ARC

The .NET runtime uses a garbage collector to reclaim memory by deleting unreferenced objects. This introduces ambiguity about exactly when an object will be deleted. The technical term for this ambiguity is *nondeterministic finalization*. When a .NET object is garbage collected its finalizer method is invoked so that it can perform cleanup work, such as releasing unmanaged resources. This will only occur if an object has a finalizer method which has not been explicitly suppressed at run-time.

There is no guarantee that a .NET object will ever be garbage collected. Once the GC frees up enough memory to meet its quota it goes to sleep, potentially leaving behind objects eligible for collection. If a .NET program must have control over when an object performs cleanup work, it might use the IDisposable interface to implement the Dispose pattern. That pattern can have far-reaching effects on how classes are designed and used.

All of this complexity was introduced for the sake of making software development easier. Whether or not that goal was achieved is debatable. Whether or not it engenders a false sense of security in many developers, which can lead to intellectual laziness and sloppy coding, is also debatable. What cannot be debated is that having a garbage collector makes software run less quickly and predictably than it otherwise could.

The Objective-C runtime in iOS does not have a garbage collector. When designing iOS, Apple decided not to include the garbage collector used by many OS X desktop apps. The processing power and battery life of today's mobile devices pale in comparison to desktop and laptop computers, making garbage collection a luxury not worth the price.

For a competent iOS developer memory management is, out of necessity, integral to the thought process of writing code. Mobile apps that work with large amounts of data are practically guaranteed to run out of memory and crash if

memory is improperly managed. This problem becomes even more apparent if an app needs to support running on older device models, which have much less memory than the latest and greatest iDevices.

This chapter shows how memory management works and can be properly implemented in iOS software. It also reviews how memory management has become almost entirely automated starting in iOS 5, but why a professional developer must be able to manage memory without assistance.

Chapter 6: From System.* to NS*

Software development is all about small tasks, such as adding an object to a collection or comparing two strings. A developer's productivity is a function of the speed at which he or she can write code to perform such tasks. Naturally, the first step to becoming productive is learning the language in which a program must be written. Learning a language is largely an intellectual exercise, reinforced with hands-on experience. The next step is gaining familiarity with the APIs needed to get real work done. Many APIs must be learned to create even modest production software. This requires memorization through practical experience more than abstract contemplation.

Moving from the familiar APIs of .NET to the foreign APIs of iOS can be frustrating at first. A seasoned .NET developer might spend a few seconds on some trivial programming task in a .NET app, while in iOS it might require five to thirty minutes of searching the Web and browsing books, just to find the right method or class. Once found the task may take no more than a few seconds to complete. Good times!

The crux of the matter is that iOS terminology is often different from .NET terminology. Both platforms include the same concepts but give them different names. Overcoming this "vocabulary barrier" is one of the most significant challenges to becoming a productive iOS developer. This chapter provides terminology and concept mappings that make it easier for a .NET developer to understand and use the iOS APIs.

The title of this chapter indicates that it relates information about classes in the System.* namespaces to equivalent classes with the NS prefix. Apple uses the NS prefix for classes in the Foundation framework (it is also used by the Core Data framework, the topic of Chapter 10). The Foundation framework contains the most commonly used classes in iOS apps. In case you're curious, the NS class prefix was inherited by Apple when they acquired the NeXT company and its **N**eXT**S**TEP operating system, on which OS X and, by extension, iOS are based.

The code examples in this chapter include comments that show equivalent C# code using .NET objects, where applicable.

Object vs. NSObject

System.Object is the root base class for every class written in C#. It provides a handful of core methods used by almost all applications. In iOS, NSObject serves the same purpose. It is almost always included in the class hierarchy of other classes, though technically it is optional. NSObject, too, provides commonly used methods, as seen in **Figure 1**.

```
// Comments show the C# equivalent.

// Object obj = new Object();
NSObject *obj = [[[NSObject alloc] init] autorelease];

// string desc = obj.ToString();
NSString *desc = [obj description];

// int hsh = obj.GetHashCode();
NSUInteger hsh = [obj hash];

// Object obj2 = new Object();
NSObject *obj2 = [[[NSObject alloc] init] autorelease];

// bool eq = obj.Equals(obj2);
BOOL eq = [obj isEqual:obj2];

// Type cls = obj.GetType();
Class cls = [obj class];
```

Figure 1 - NSObject

Chapter 7: From XAML to UIKit

People expect iOS apps to be beautiful, intuitive, and responsive. It is in Apple's best interest to help application developers meet those expectations. A significant aspect of creating a great iOS app is making it visually consistent with Apple apps; such as Settings, Calendar, and Mail. The iOS development platform makes it easy for developers to use the same UI controls Apple uses. This is certainly not a foolproof means for ensuring an app will have an excellent user experience, but it is a big step in the right direction.

This chapter reviews UIKit, the most popular and high-level UI programming platform for iOS. It explains UIKit to a .NET developer familiar with WPF or Silverlight (hereafter referred to as WPF/SL). Don't worry if you have no experience with those UI platforms, there is plenty of information to absorb in this chapter regardless of your UI programming background. The following chapter builds upon knowledge gained here by showing how to build UIKit-based user interfaces via drag-and-drop.

Before starting our guided tour of UIKit, I want to point out that Apple is very restrictive about allowing UI controls to be customized by application developers. This is in stark contrast to WPF/SL, which support almost unlimited UI customization. Apple's controls are highly opinionated whereas Microsoft's are simply suggestive. Perhaps their emphasis on consistency explains, in part, why Apple's mobile platform has become so popular. Food for thought...

Chapter 8: Building UIs with Interface Builder

While understanding the design principles and programmatic interface of UIKit is essential to being a competent iOS developer, it is equally important to have the know-how needed to make use of it. This chapter shifts focus from a conceptual overview of UIKit to a more practical guide to the tool in which user interfaces are created: Interface Builder (IB).

For many years IB was a separate program, loosely integrated with Xcode. Starting in Xcode 4 the two merged, with IB now seamlessly incorporated into the IDE. For a .NET developer familiar with Expression Blend, this can be thought of as Blend being made part of Visual Studio.

[Part III: Alien Technology]

Creating complex software requires more than just knowledge of a programming language, an IDE, and the common libraries. Most applications need to share data with other computers over the Internet, store and retrieve complex data quickly, and a host of other complex tasks. As the size and complexity of an application grows, so does the job of troubleshooting and fixing problems. This part of the book shows how these tasks can be accomplished with Apple's tools and APIs, along with some third-party code libraries.

Chapter 9: Calling Web Services

Most mobile apps are not deployed with all the data they will ever display to a user. They rely on other computers to provide them with interesting and relevant data for the user to view, hear, touch, and edit. Many apps provide the user with a means of creating data that must be transmitted from their iDevice and received by another computer for further processing. This client-server architecture is nothing new for most .NET application developers.

The .NET framework and tooling provide layer after layer of abstraction that make interacting with server data dead simple. That is not the case with iOS programming. Apple does not appear to have made this aspect of their development platform a high priority. For example, it was not until iOS 5 that the platform supported reading and writing JSON, a very popular lightweight data format used by Web services for mobile apps. Fortunately, there are some community projects that can help make life easier. This chapter shows how to work with Web services using APIs from Apple and community projects.

Chapter 10: Overview of Core Data

Manually incorporating a database into an app is boring, tedious, error-prone work. Imagine, for example, having to write properties in eighty-four classes so that they exactly match the schema of eighty-four database tables used by an app. Then suppose all eighty-four of those classes must have custom code that saves their properties to, and retrieves them from, a database. In that scenario the likelihood of a developer introducing mistakes, or leaping out of the nearest window, is extremely high. Computers are excellent at quickly performing boring, tedious, error-prone work. Developers, being such a clever bunch, have leveraged that fact by writing frameworks to automate as much of the tedium as possible.

This chapter reviews the Core Data framework that Apple provides to iOS developers to simplify working with data. It is somewhat similar to the Entity Framework and NHibernate framework used by many .NET developers. Every large, enterprise-scale iOS app I have worked on uses Core Data to manage the data layer. Even smaller apps, such as my pet projects, benefit from Core Data because it reduces the amount of code I must write, test, and maintain. Using the Core Data framework is certainly not a requirement, but it has so much to offer to application developers that to not learn about it is doing yourself a disservice.

Chapter 11: Debugging Techniques

Writing software is easy. Writing software that works properly is not, which is why most programming platforms include APIs and programs designed to assist developers with troubleshooting their code. This chapter examines the debugging tools provided by Apple. Developers accustomed to working in Visual Studio might initially find the support in Xcode for debugging limited and inconvenient. While Xcode certainly does not have the same rich visual debugging aides available in VS, it does provide a powerful set of tools that can be used to quickly track down bugs. This chapter also introduces Instruments, a tool that collects information about a running application which can be visually analyzed to identify the cause of performance problems.

Chapter 12: Unit Testing

Many developers write programs that test other programs. Having a computer verify the correctness of a program is generally referred to as *automated testing*. There are different kinds of automated tests. Automated UI testing involves evaluating how an app responds to input created by scripts that interact with its user interface. This form of testing can be done using the Automation instrument of the Instruments program. Automated UI testing is not covered in this book.

Another form of automated testing is known as *unit testing*, in which test code exercises a small, independent functional unit of an application. Unit testing is not only used for automated testing. Some developers write unit tests before the code that passes those tests, to keep their classes focused and simple. This style of programming is called *Test-Driven Development* (TDD). This chapter shows how unit tests are created in Xcode, whether for TDD or automated testing purposes.

Thank you for checking out this sample of

iOS Programming for .NET Developers.

To buy the full version of the book please visit:

http://iosfordotnetdevs.com