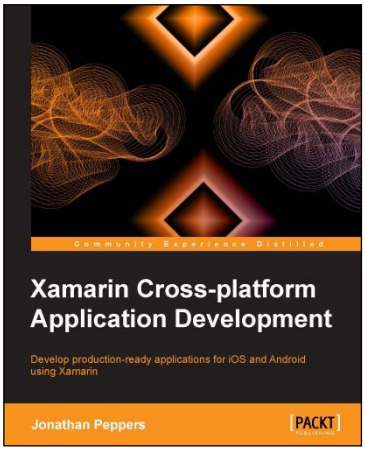
[PACKT] PUBLISHING

Xamarin Cross-platform Application Development

Jonathan Peppers



Chapter No. 3
"Code Sharing Between iOS and Android"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "Code Sharing Between iOS and Android"

A synopsis of the book's content

Information on where to buy this book

About the Author

Jonathan Peppers is a Xamarin MVP and the lead developer of the popular cross-platform game, Draw a Stickman: EPIC. Jon works for Hitcents, a software development company based in Bowling Green, Kentucky. He has been working with the C# programming language for over 7 years. He has also been working with technologies such as WinForms, WPF, ASP.NET WebForms, and ASP.NET MVC. In recent years, Hitcents has been heavily investing in mobile development with Xamarin and has developed nearly 40 mobile applications across multiple platforms.

Xamarin Cross-platform Application Development

Xamarin has built three core products for developing iOS and Android applications in C#: Xamarin Studio, Xamarin.iOS, and Xamarin.Android. Xamarin gives you direct access to the native APIs on each platform and the flexibility to share C# code among platforms. Using Xamarin and C#, the productivity you get is better than that of Java or Objective-C while maintaining a greater performance output compared to a HTML or JavaScript solution.

In this book, we will develop a real-world sample application to demonstrate what you can do with Xamarin technologies, and build on core platform concepts for iOS and Android. We will also cover advanced topics such as push notifications, retrieving contacts, using a camera, and GPS location. Finally, we will walk through what it takes to submit your application to the Apple App Store and Google Play.

What This Book Covers

Chapter 1, *Xamarin Setup*, covers the process of installing the appropriate Xamarin software and native SDKs required for performing cross-platform development.

Chapter 2, *Hello Platforms!*, covers creating your first "Hello World" application on iOS and Android, which covers some basic concepts on each platform.

Chapter 3, *Code Sharing Between iOS and Android*, introduces code-sharing techniques and strategies to set up projects that can be used with Xamarin.

Chapter 4, *XamChat – a Cross-platform App*, introduces a sample application that we will be building throughout the book. In this chapter, we will write all the shared code for the application, complete with unit tests.

Chapter 5, *XamChat for iOS*, covers the technique of implementing the iOS user interface for XamChat and various iOS development concepts.

Chapter 6, *XamChat for Android*, covers the technique of implementing the Android version of XamChat and introduces Android-specific development concepts.

Chapter 7, *Deploying and Testing on Devices*, explains the painful process of deploying your first application on a device. We also cover why it is important to always test your application on real devices.

Chapter 8, *Web Services with Push Notifications*, explains the technique of implementing a real backend web service for XamChat using Azure Mobile Services.

Chapter 9, *Third-party Libraries*, covers the various options to use third-party libraries with Xamarin and how you can even leverage native Java and Objective-C libraries.

Chapter 10, *Contacts, Camera, and Location*, introduces the library Xamarin. Mobile as a cross-platform library for accessing users' contacts, camera, and GPS location.

Chapter 11, *App Store Submission*, explains the process of submitting your app to the Apple App Store and Google Play.

3
Code Sharing Between iOS and Android

Xamarin Studio tools promise to share a good portion of your code between iOS and Android while taking advantage of the native APIs on each platform where possible. This is the equivalent of an exercise in software engineering more than a programming skill or having the knowledge of each platform. To architect a Xamarin Studio application to enable code sharing, it is a must to separate your application into distinct layers. We'll cover the basics as well as specific options to consider certain situations.

In this chapter, we will cover:

- The MVVM design pattern for code sharing
- Project and solution organization strategies
- Portable Class Libraries (PCLs)
- Preprocessor statements for platform specific code
- Dependency injection (DI) simplified
- Inversion of Control (IoC)

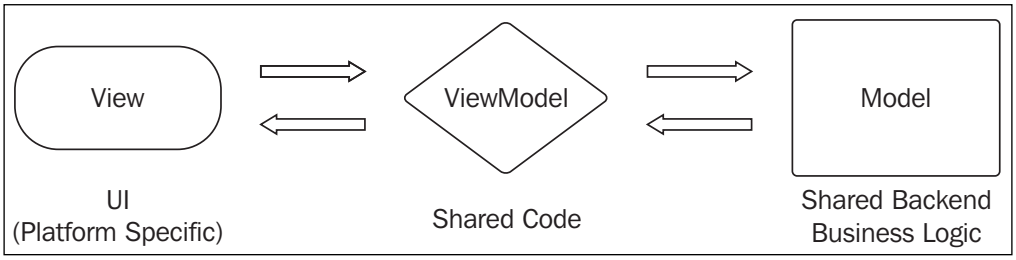
Learning the MVVM design pattern

The **Model-View-ViewModel** (**MVVM**) design pattern was originally invented for **WPF** (**Windows Presentation Foundation**) applications using **XAML** for separating the UI from business logic and taking full advantage of **data binding**. Applications architected in this way have a distinct ViewModel layer that has no dependencies on its user interface. This architecture in itself is optimized for unit testing as well as cross-platform development. Since an application's ViewModel classes have no dependencies on the UI layer, you can easily swap an iOS user interface for an Android one and write tests against the ViewModel layer. The MVVM design pattern is also very similar to the MVC design pattern discussed in the previous chapters.

The MVVM design pattern includes the following :

- **Model**: The model layer is the backend business logic driving the application and any business objects to go along with it. This can be anything from making web requests to a server to using a backend database.
- **View**: This layer is the actual user interface seen on the screen. In case of cross-platform development, it includes any platform-specific code for driving the user interface of the application. On iOS, this includes controllers used throughout an application and on Android, an application's activities.
- **ViewModel**: This layer acts as the glue in MVVM applications. The ViewModel layers coordinate operations between the View and Model layers. A ViewModel layer will contain properties that the view will get or set, and functions for each operation that can be made by the user on each view. The ViewModel will also invoke operations on the Model layer if needed.

The following figure shows the MVVM design pattern:



It is important to note that the interaction between the View and ViewModel layers is traditionally created by data binding with WPF. However, iOS and Android do not have built-in data binding mechanisms, so our general approach throughout the book will be to manually call the ViewModel layer from the view. There are a few frameworks out there that provide a data binding functionality such as **MVVMCross**.

To understand this pattern better, let's implement a common scenario. Let's say we have a search box on the screen and a search button. When the user enters some text and clicks on the button, a list of products and prices will be displayed to the user. In our example, we use the **async** and **await** keywords that are available in C# 5 to simplify asynchronous programming.

To implement this feature, we would start with a simple `model` class (also called a `business` object) as follows:

```
public class Product
{
  public int Id { get; set; } //Just a numeric identifier
  public string Name { get; set; } //Name of the product
  public float Price { get; set; } //Price of the product
}
```

Next, we would implement our Model layer to retrieve products based on the searched term. This is where the business logic is performed, expressing how the search needs to actually work. This is seen in the following lines of code:

```
// An example class, in the real world would talk to a web
// server or database.
public class ProductRepository
{
  // a sample list of products to simulate a database
  private Product[] products = new[]
  {
    new Product { Id = 1, Name = "Shoes", Price = 19.99m },
    new Product { Id = 2, Name = "Shirt", Price = 15.99m },
    new Product { Id = 3, Name = "Hat", Price = 9.99m },
  };

  public async Task<Product[]> SearchProducts(
    string searchTerm)
  {
    // Wait 2 seconds to simulate web request
    await Task.Delay(2000);

    // Use Linq-to-objects to search, ignoring case
    searchTerm = searchTerm.ToLower();
    return products.Where(p =>
      p.Name.ToLower().Contains(searchTerm))
      .ToArray();
  }
}
```

It is important to note here that the `Product` and `ProductRepository` classes are both considered as part of the Model layer of a cross-platform application. Some may consider `ProductRepository` as a service that is generally a self-contained class for retrieving data of some kind. It is a good idea to separate this functionality into two classes. The `Product` class's job is to hold information about a product, while `ProductRepository` is in charge of retrieving products. This is the basis for the **single responsibility principle**, which states that each class should only have one job or concern.

Next, we would implement a `ViewModel` class as follows:

```
public class ProductViewModel
{
  private readonly ProductRepository repository =
    new ProductRepository();

  public string SearchTerm
  {
    get;
    set;
  }

  public Product[] Products
  {
    get;
    private set;
  }

  public async Task Search()
  {
    if (string.IsNullOrEmpty(SearchTerm))
      Products = null;
    else
      Products = await repository.SearchProducts(SearchTerm);
  }
}
```

From here, your platform-specific code starts. Each platform would handle managing an instance of a `ViewModel` class, setting the `SearchTerm` property, and calling `Search` when the button is clicked. When the task completes, the user interface layer would update a list displayed on the screen.

If you are familiar with the MVVM design pattern used with WPF, you might notice that we are not implementing `INotifyPropertyChanged` for data binding. Since iOS and Android don't have the concept of data binding, we omitted this functionality. If you plan on having a WPF or Windows 8 version of your mobile application, you should certainly implement support for data binding where needed.

Comparing project organization strategies

You might be asking yourself at this point, how do I set up my solution in Xamarin Studio to handle shared code and also have platform-specific projects? Xamarin.iOS applications can only reference Xamarin.iOS class libraries; so, setting up a solution can be problematic. There are actually three main strategies for setting up a cross-platform solution, each with its own advantages and disadvantages.

Options for cross-platform solutions are as follows:

- **File Linking**: For this option, you would start with either a plain .NET 4.0 or .NET 4.5 class library containing all the shared code. You would then have a new project for each platform you want your app to run on. Each platform-specific project would have a subdirectory with all of the files linked in from the first class library. To set this up, add the existing files to the project, and select the **Add a link to the file** option. Any unit tests can run against the original class library. The advantages and disadvantages of file linking are as follows:
    - Advantages: This approach is very flexible. You can choose to link or not link certain files and can also use preprocessor directives such as `#if IPHONE`. You can also reference different libraries on Android versus iOS.
    - Disadvantages: You have to manage a file's existence in three projects: core library, iOS, and Android. This can be a hassle if it is a large application or if many people are working on it.
- **Cloned Project Files**: It is very similar to file linking, the main difference being that you have a class library for each platform in addition to the main project. By placing the iOS and Android projects in the same directory as the main project, the files can be added without linking. You can easily add files by right-clicking on the solution and selecting **Display Options** | **Show All Files**. Unit tests can run against the original class library or the platform-specific versions.
    - Advantages: This approach is just as flexible as file linking, but you don't have to manually link any files. You can still use preprocessor directives and reference different libraries on each platform.
    - Disadvantages: You still have to manage a file's existence in three projects. There is additionally some manual file arranging required to set this up. You also end up with an extra project to manage on each platform.

- **Portable Class Libraries**: This is the most optimal option; you begin the solution by making a **portable class library** (**PCL**) project for all your shared code. This is a special project type that allows multiple platforms to reference the same project, allowing you to use the smallest subset of C# and the .NET framework available in each platform. Each platform-specific project would reference this library directly, as well as any unit test projects.
    - Advantages: All your shared code is in one project, and all platforms use the same library. This is the cleanest and preferred approach for most cross-platform applications.
    - Disadvantages: You cannot use preprocessor directives or reference different libraries on each platform. Use of the **dependency injection** is the recommended way around this. You also could be limited to a subset of .NET depending on how many platforms you are targeting. This option has also been in beta or experimental version for some time. At the time of writing this book, PCLs are well on their way to become officially supported.

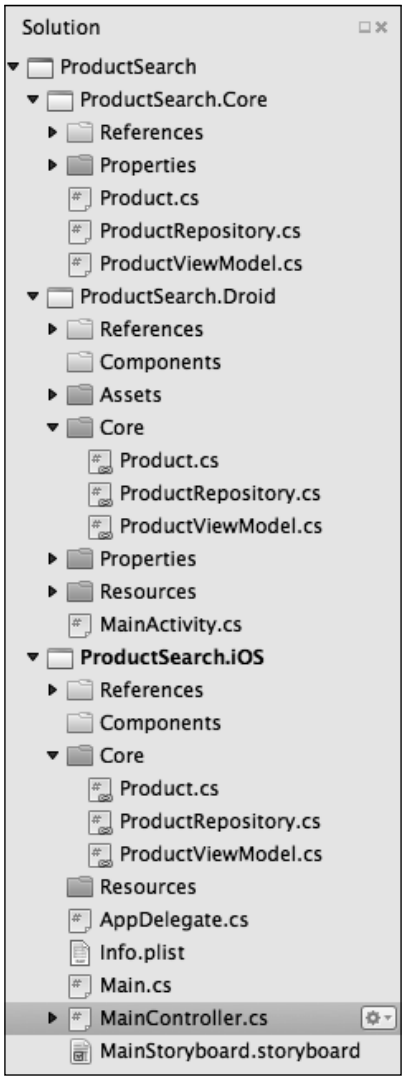
To understand each option completely and what situations call for, let's define a solution structure for each cross-platform solution. Let's use the product search example from earlier in the chapter and set up a solution for each approach.

To begin with file linking, perform the following steps:

1. Open Xamarin Studio and start a new solution.
2. Select a new **Library** project under the general **C#** section.
3. Name the project `ProductSearch.Core`, and name the solution `ProductSearch`.
4. Right-click on the newly created project and select **Options**.
5. Under **Build | General**, set the **Target Framework** option to **.NET Framework 4.5**.

6. Add the `Product`, `ProductRepository`, and `ProductViewModel` classes to the project used earlier in this chapter. You will need to add `using System.Threading.Tasks;` and `using System.Linq;` where needed.
7. Click on **Build** | **Build All** from the menu at the top to be sure that everything builds properly.
8. Now, let's create a new iOS project by right-clicking on the solution and selecting **Add** | **Add New Project**. Create a new project by navigating to **iOS** | **iPhone Storyboard** | **Single View Application** and name it `ProductSearch.iOS`.
9. Create a new Android project by right-clicking on the solution and selecting **Add** | **Add New Project**. Create a new project by navigating to **Android** | **Android Application** and name it `ProductSearch.Droid`.
10. Add a new folder named `Core` to both the iOS and Android projects.
11. Right-click on the new folder for the iOS project and select **Add** | **Add Files from Folder**. Select the root directory for the `ProductSearch.Core` project.
12. Check the three C# files in the root of the project. An **Add File to Folder** dialog will appear.
13. Select **Add a link to the file** and make sure the **Use the same action for all selected files** checkbox is selected.
14. Repeat this process for the Android project.
15. Click on **Build** | **Build All** from the menu at the top to double-check everything, and you have successfully set up a cross-platform solution with file linking.

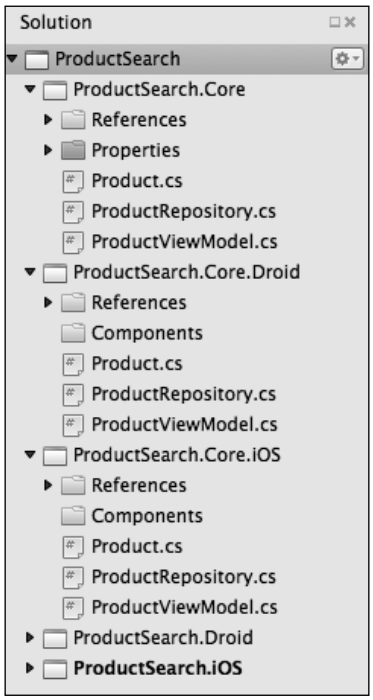
When all is done, you will have a solution tree that looks something like what you can see in the following screenshot:



You should consider using this technique when you have to reference different libraries on each platform. You might consider using this option if you are using `ReactiveUI`, `MonoGame`, or other frameworks that require you to reference a different library on iOS versus Android.

Setting up a solution with the cloned project files approach is similar to file linking, except that you will have to create an additional class library for each platform. To do this, create an Android library project and an iOS library project in the same directory: `ProductSearch.Core`. You will have to create the projects and move them to the proper folder manually, then re-add them to the solution. Right-click on the solution and select **Display Options** | **Show All Files** to add the required C# files to these two projects. Your main iOS and Android projects can reference these projects directly.

Your project will look like what is shown in the following screenshot, with `ProductSearch.iOS` referencing `ProductSearch.Core.iOS`, and `ProductSearch.Droid` referencing `ProductSearch.Core.Droid`:



Working with portable class libraries

A **Portable Class Library** (**PCL**) is a C# library project that can be supported on multiple platforms including iOS, Android, Windows, Windows Store apps, Windows Phone, Silverlight, and Xbox 360. PCLs have been an effort by Microsoft to simplify development across different versions of the .NET framework. Xamarin has also added support for iOS and Android in the newer versions of their products. Many popular cross-platform frameworks and open source libraries are starting to develop PCL versions such as Json.NET and MVVMCross.

Let's set up a solution using PCLs. Before starting, make sure you at least have Mono 3.2.1 installed (look under **Xamarin Studio** | **About Xamarin Studio** | **Show Details**). At the time of writing this book, one could easily switch to the beta channel in Xamarin Studio to update to the version supporting PCLs.

Let's begin setting up our portable class library:

1. Open Xamarin Studio and start a new solution.
2. Select a new **Portable Library** project under the general **C#** section.
3. Name the project `ProductSearch.Core`, and name the solution `ProductSearch`.
4. Add the `Product`, `ProductRepository`, and `ProductViewModel` classes to the project used earlier in this chapter. You will need to add `using System.Threading.Tasks;` and `using System.Linq;` where needed.
5. Click on **Build** | **Build All** from the menu at the top to be sure everything builds properly. You may wish to remove all references this project has set, as Xamarin Studio will display them with errors (this is a known issue with PCLs).
6. Now, let's create a new iOS project by right-clicking on the solution and selecting **Add** | **Add New Project**. Create a new project by navigating to **iOS** | **iPhone Storyboard** | **Single View Application** and name it `ProductSearch.iOS`.
7. Create a new Android project by right-clicking on the solution and selecting **Add** | **Add New Project**. Create a new project by navigating to **Android** | **Android Application** and name it `ProductSearch.Droid`.
8. Simply add a reference to the portable class library from the iOS and Android projects.
9. Click on **Build** | **Build All** from the top menu and you have set up a simple solution with a portable library.

Each solution type has their distinct advantages and disadvantages. PCLs are generally better, but there are certain cases where they can't be used. For example, if you were using a library such as **MonoGame**, which is a different library for each platform, you would be much better off using file linking or cloned project files. Similar issues would arise if you needed to use a preprocessor statement such as `#if IPHONE` or a native library such as the Facebook SDK on iOS or Android.

Using preprocessor statements

When using file linking or cloned projects files, one of your most powerful tools is the use of preprocessor statements. If you are unfamiliar with them, C# has the ability to define preprocessor variables such as `#define IPHONE`, and then using `#if IPHONE` or `#if !IPHONE`.

The following is a simple example of using this technique:

```
#if IPHONE
  Console.WriteLine("I am running on iOS");
#elif ANDROID
  Console.WriteLine("I am running on Android");
#else
  Console.WriteLine("I am running on ???");
#endif
```

In Xamarin Studio, you can define preprocessor variables in your project's options under **Build** | **Compiler** | **Define Symbols**, delimited with semicolons. These will be applied to the entire project. Be warned that you must set up these variables for each configuration setting in your solution (**Debug** and **Release**); it can be an easy step to miss. You can also define these variables at the top of any C# file by declaring `#define IPHONE`, but they will only be applied within the C# file.

Let's go over another example, assuming we want to implement a class to open URLs on each platform:

```
public static class Utility
{
  public static void OpenUrl(string url)
  {
    //Open the url in the native browser
  }
}
```

The preceding example is a perfect candidate for using preprocessor statements, since it is very specific to each platform and is a fairly simple function. To implement the method on iOS and Android, we will need to take advantage of some native APIs. Refactor the class to look like the following:

```
#if IPHONE
  //iOS using statements
  using MonoTouch.Foundation;
  using MonoTouch.UIKit;
#elif ANDROID
  //Android using statements
```

```
    using Android.App;
    using Android.Content;
    using Android.Net;
  #else
    //Standard .Net using statement
    using System.Diagnostics;
  #endif

  public static class Utility
  {
    #if ANDROID
      public static void OpenUrl(Activity activity, string url)
    #else
      public static void OpenUrl(string url)
    #endif
    {
      //Open the url in the native browser
      #if IPHONE
        UIApplication.SharedApplication.OpenUrl(
          NSUrl.FromString(url));
      #elif ANDROID
        var intent = new Intent(Intent.ActionView,
          Uri.Parse(url));
        activity.StartActivity(intent);
      #else
        Process.Start(url);
      #endif
    }
  }
```

The preceding class supports three different types of projects: Android, iOS, and a standard Mono or .NET framework class library. In the case of iOS, we can perform the functionality with static classes available in Apple's APIs. Android is a little more problematic, and requires an `Activity` object for launching a browser natively. We get around this by modifying the input parameters on Android. Lastly, we have a plain .NET version that uses `Process.Start()` to launch a URL. It is important to note that using the third option would not work on iOS or Android natively, which necessitates our use of preprocessor statements.

Using preprocessor statements is not normally the cleanest or the best solution for cross-platform development. They are generally best used in a tight spot or for very simple functions. Code can easily get out of hand and can become very difficult to read with many `#if` statements, so it is always better to use it in moderation. Using inheritance or interfaces is generally a better solution when a class is mostly platform specific.

Simplifying dependency injection

Dependency injection at first seems like a complex topic, but for the most part it is a simple concept. It is a design pattern aimed at making your code within your applications more flexible so that you can swap out certain functions when needed. The idea builds around setting up dependencies between classes in an application so that each class only interacts with an interface or base/abstract class. This gives you the freedom to override different methods on each platform when you need to fill in native functionality.

The concept originated from the **SOLID** object-oriented design principles, which is a set of rules you might want to research if you are interested in software architecture. The **D** in SOLID stands for **dependencies**. Specifically, the principle declares that a program should depend upon abstractions, not concretions (concrete types).

To build upon this concept, let's walk through the following example:

1. Let's assume we need to store a setting in an application that determines if the sound is on or off.
2. Now let's declare a simple interface for the setting: `interface ISettings { bool IsSoundOn { get; set; } }`.
3. On iOS, we'd want to implement this interface using the `NSUserDefaults` class.
4. Likewise, on Android, we would implement this using `SharedPreferences`.
5. Finally, any class that needs to interact with this setting would only reference `ISettings` so that the implementation could be replaced on each platform.

For reference, the full implementation of this example would look like the following snippet:

```
public interface ISettings
{
  bool IsSoundOn
  {
    get;
    set;
  }
}

//On iOS
using MonoTouch.UIKit;
using MonoTouch.Foundation;

public class AppleSettings : ISettings
```

```
{
  public bool IsSoundOn
  {
    get
    {
      return NSUserDefaults.StandardUserDefaults
      .BoolForKey("IsSoundOn");
    }
    set
    {
      var defaults = NSUserDefaults.StandardUserDefaults;
      defaults.SetBool(value, "IsSoundOn");
      defaults.Synchronize();
    }
  }
}

//On Android
using Android.Content;

public class DroidSettings : ISettings
{
  private readonly ISharedPreferences preferences;

  public DroidSettings(Context context)
  {
    preferences = context.GetSharedPreferences(
      context.PackageName, FileCreationMode.Private);
  }

  public bool IsSoundOn
  {
    get
    {
      return preferences.GetBoolean("IsSoundOn", true");
    }
    set
    {
      using (var editor = preferences.Edit())
      {
        editor.PutBoolean("IsSoundOn", value);
        editor.Commit();
      }
    }
  }
}
```

Now you would potentially have a `ViewModel` class that would only reference `ISettings` when following the MVVM pattern. It can be seen in the following snippet:

```
public class SettingsViewModel
{
  private readonly ISettings settings;

  public SettingsViewModel(ISettings settings)
  {
    this.settings = settings;
  }

  public bool IsSoundOn
  {
    get;
    set;
  }

  public void Save()
  {
    settings.IsSoundOn = IsSoundOn;
  }
}
```

Using a ViewModel layer for such a simple example is not necessarily needed, but you can see it would be useful if you needed to perform other tasks such as input validation. A complete application might have a lot more settings and might need to present the user with a loading indicator. Abstracting out your setting's implementation has other benefits that adds flexibility to your application. Let's say you suddenly need to replace `NSUserDefaults` on iOS with the iCloud function; you can easily do so by implementing a new `ISettings` class and the remainder of your code will remain unchanged. This will also help you target new platforms such as Windows Phone, where you may choose to implement `ISettings` in a platform-specific way.

Implementing Inversion of Control

You might be asking yourself at this point in time, how do I switch out different classes such as the `ISettings` example? **Inversion of Control** (**IoC**) is a design pattern meant to complement dependency injection and solve this problem. The basic principle is that many of the objects created throughout your application are managed and created by a single class. Instead of using the standard C# constructors for your `ViewModel` or `Model` classes, a service locator or factory class would manage them throughout the application.

There are many different implementations and styles of IoC, so let's implement a simple service locator class to use through the remainder of this book as follows:

```
public static class ServiceContainer
{
  static readonly Dictionary<Type, Lazy<object>> services =
    new Dictionary<Type, Lazy<object>>();

  public static void Register<T>(Func<T> function)
  {
    services[typeof(T)] = new Lazy<object>(() => function());
  }

  public static T Resolve<T>()
  {
    return (T)Resolve(typeof(T));
  }

  public static object Resolve(Type type)
  {
    Lazy<object> service;
    if (services.TryGetValue(type, out service)
    {
      return service.Value;
    }
    throw new Exception("Service not found!");
  }
}
```

This class is inspired by the simplicity of XNA/MonoGame's `GameServiceContainer` class, and follows the **service locator** pattern. The main differences are the heavy use of generics and the fact that it is a static class.

To use our `ServiceContainer` class, we would declare the version of `ISettings` or other interfaces that we want to use throughout our application by calling `Register` as seen in the following lines of code:

```
//iOS version of ISettings
ServiceContainer.Register<ISettings>(() => new AppleSettings());

//Android version of ISettings
ServiceContainer.Register<ISettings>(() => new DroidSettings());

//You can even register ViewModels
ServiceContainer.Register<SettingsViewModel>(() =>
  new SettingsViewModel());
```

On iOS, you could place this registration code in either your `static void Main()` method or in the `FinishedLaunching` method of your `AppDelegate` class. These methods are always called before the application is started.

On Android, it is a little more complicated. You cannot put this code in the `OnCreate` method of your activity, set as the main launcher. In some situations, the OS can close your application but restart it later in another activity. This situation is likely to cause an exception to be raised. The guaranteed safe place to put this is in a custom Android `Application` class, which has an `OnCreate` method that is called prior to any activities being created in your application. The following lines of code show the use of the `Application` class:

```
[Application]
public class Application : Android.App.Application
{
  //This constructor is required
  public Application(IntPtr javaReference, JniHandleOwnership
    transfer): base(javaReference, transfer)
  {

  }

  public override void OnCreate()
  {
    base.OnCreate();

    //IoC Registration here
  }
}
```

To pull a service out of the `ServiceContainer` class, we could rewrite the constructor of the `SettingsViewModel` class like the following lines of code:

```
public SettingsViewModel()
{
  this.settings = ServiceContainer.Resolve<ISettings>();
}
```

Likewise, you would use the generic `Resolve` method to pull out any `ViewModel` classes you would need to call from within controllers on iOS or activities on Android. This is a great, simple way to manage dependencies within your application.

There are, of course, some great open source libraries out there that implement IoC for C# applications. You might consider switching to one of them if you need more advanced features for service location, or just want to graduate to a more complicated IoC container.

Here are a few libraries that have been used with Xamarin projects:

- **TinyIoC**: `https://github.com/grumpydev/TinyIoC`
- **Ninject**: `http://www.ninject.org/`
- **MvvmCross**: `https://github.com/slodge/MvvmCross` includes a full MVVM framework as well as IoC
- **Simple Injector**: `http://simpleinjector.codeplex.com`
- **OpenNETCF.IoC**: `http://ioc.codeplex.com`

Summary

In this chapter, we learned about the MVVM design pattern and how it can be used to better architect cross-platform applications. We compared several project organization strategies for managing a Xamarin Studio solution containing both iOS and Android projects. We went over portable class libraries as the preferred option for sharing code and how to use preprocessor statements as a quick and dirty way to implement platform-specific code.

After completing this chapter, you should be able to speed up with several techniques for sharing code between iOS and Android applications using Xamarin Studio. Using the MVVM design pattern will help you divide your shared code between code that is platform specific. We also covered the three main options for setting up Xamarin Studio solutions and projects. You should also have a firm understanding of using the dependency injection and Inversion of Control to give your shared code access to the native APIs on each platform. In the next chapter, we will begin with writing a cross-platform application and dive into using these techniques.

Where to buy this book

You can buy Xamarin Cross-platform Application Development from the Packt Publishing website: `http://www.packtpub.com/xamarin-cross-platform-application-development/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/xamarin-cross-platform-application-development/book