

Binding a Java Library (.jar)

Consuming Java Libraries from C#

Sample Code:

- [OsmDroidBindingExample.zip](#)

Related Articles:

- [Working with JNI](#)
- [API Metadata Reference](#)
- <http://www.mono-project.com/GAPI - Metadata>
- [Library Projects](#)

Xamarin.Android 4.2 introduced support for binding arbitrary Java libraries (.jar files) and introduced a new project template called the Java Bindings Library Project to augment this new functionality. This guide covers how to use this new project template to bind a Java library. The tutorial portion recreates the bindings for osmdroid – a Java library to interact with OpenStreetMap.

Overview

The third-party library ecosystem for Android is massive. Because of this, it frequently makes more sense to use an existing library than to create a new one. Xamarin.Android has two ways to use these libraries, either use the *Java Native Interface (JNI)* to invoke the calls directly, or create a *binding* project that automatically wraps the library with C# wrappers based on a declarative approach.

A binding is simply a wrapper class, known as a *Managed Callable Wrapper*, which allows a .NET application to interop with non-.NET technology. In this case, it allows a Xamarin.Android application to call a Java class or interface from C#. Mono for Android ships with bindings for the types included in Android.jar, as well as bindings for the Google add-on APIs. However, for libraries that are not already bound, you need to create the bindings yourself.

Prior to Xamarin.Android 4.2, creating bindings meant working directly with the *Java Native Interface (JNI)* to call Java classes directly. To learn more about JNI, see the document [Working with JNI](#).

However, with Xamarin.Android 4.2, a new project template is available, called the *Java Bindings Library*, which moves closer toward complete automation of binding by providing a declarative approach to binding Java libraries. If you use this project template, then you'll follow this binding workflow:

- Create a binding project.
- Add the .jar file that you want to bind.
- Generate the binding.
- Resolve binding issues by declaring how to create the binding.
- Add support classes to make the binding conform to .NET patterns and usage.
- Regenerate the binding with the resolved issues and support classes.

The tooling automates much of the binding creation. However, in many cases, it still needs manual modifications to deal with places where Xamarin.Android maps the Android API to different types in C#, such as where Java int constants are replaced with C# enums.

In order to illustrate and explain the details involved in creating a Java library binding, this article walks through the creation of one for the Google Maps API. It illustrates both Visual Studio and Xamarin Studio implementations and covers how to address any errors that may occur during automated generation. Additionally, this document covers how to modify the API design of a binding and shape it more like .NET.

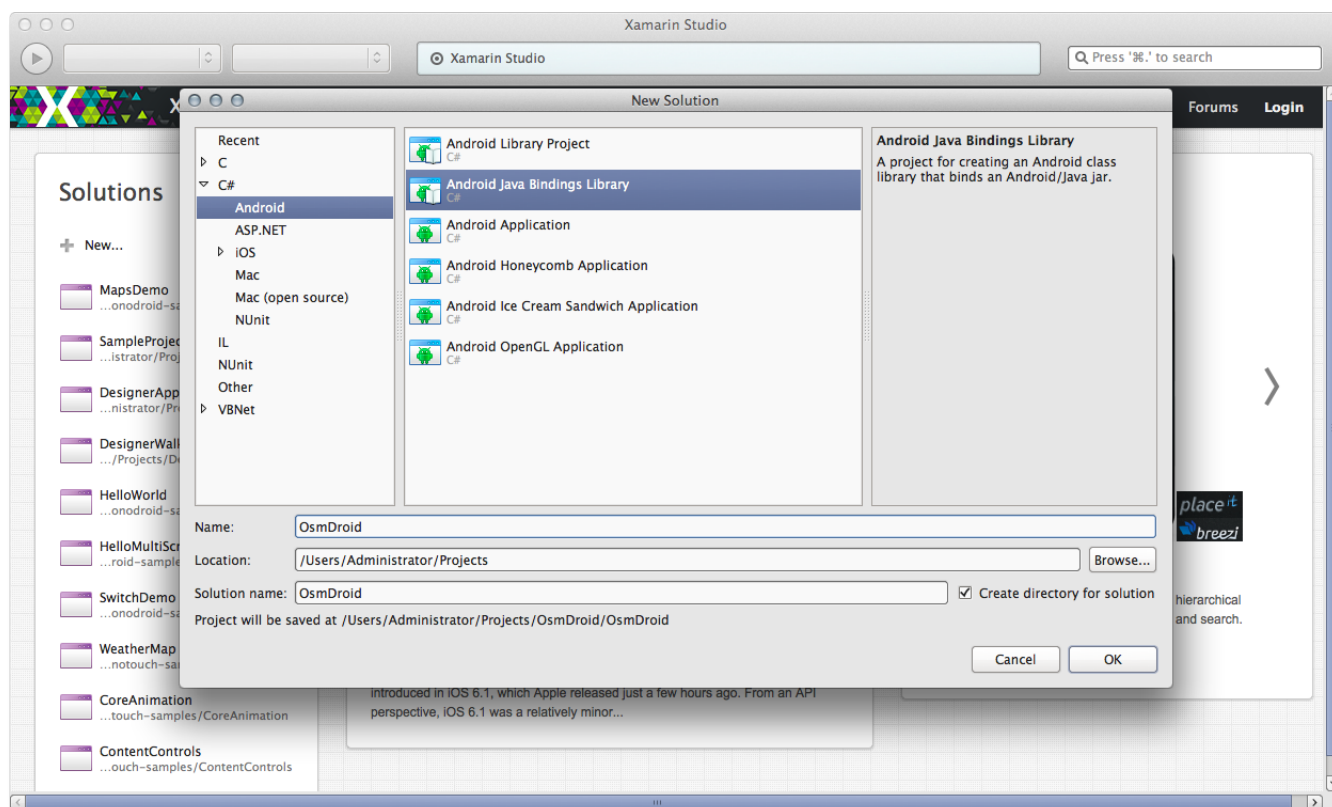
Java Library Binding Walkthrough

So, let's get started creating a binding.

Creating the Project

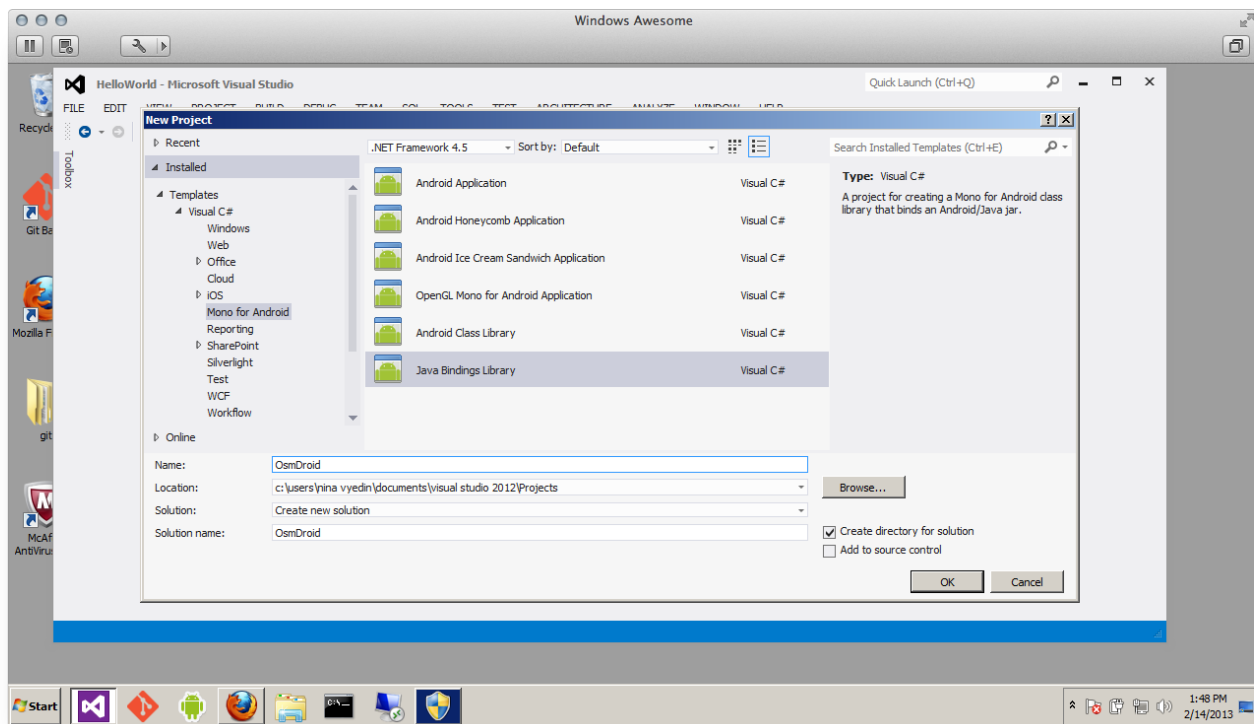
The first thing we need to do is create a new project for the binding.

USING VISUAL STUDIO



1. Select File > New > Project.
2. Select the Java Bindings Library project template and name it OsmDroid.

USING Xamarin Studio



1. Select File > New Solution.
2. Select the Java Bindings Library project template, and then name it OsmDroid.

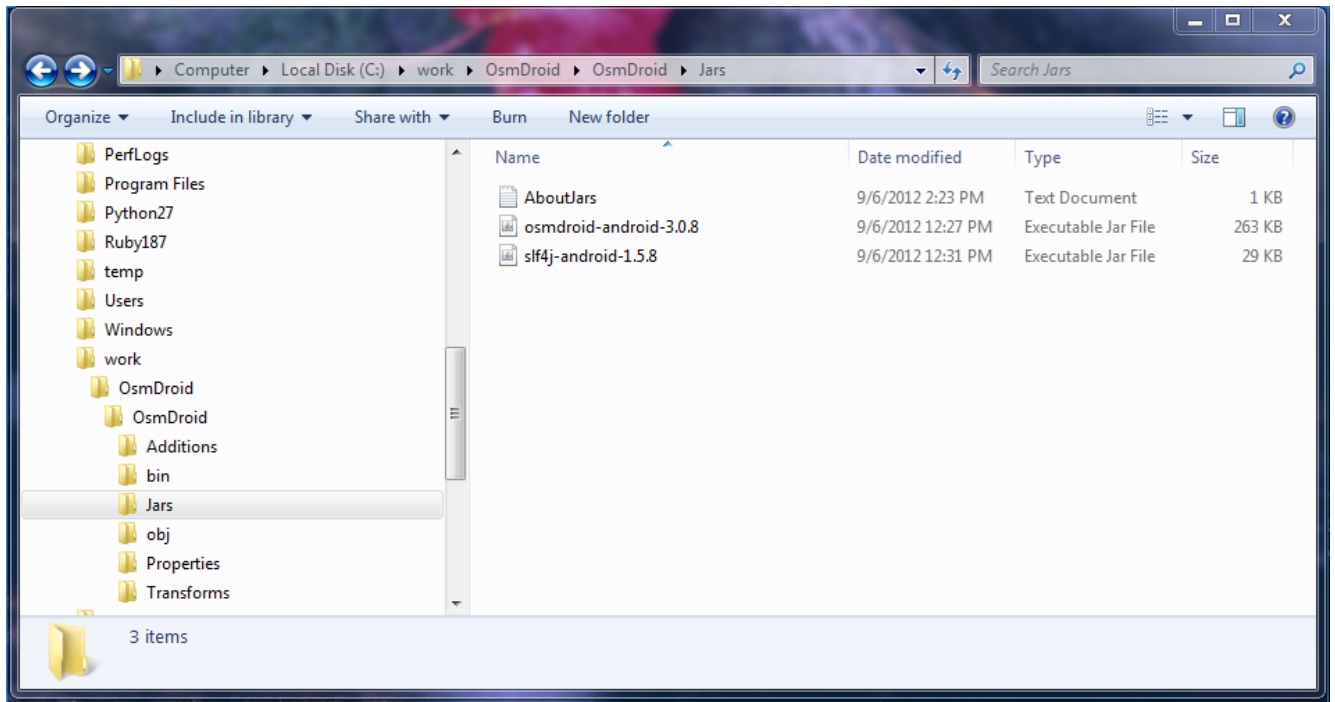
Adding The OsmDroid Java Libraries

Creating the project gives us an empty bindings library. But, we need to add the actual *Java Library* (.jar) file to the binding project so that it can be bound.

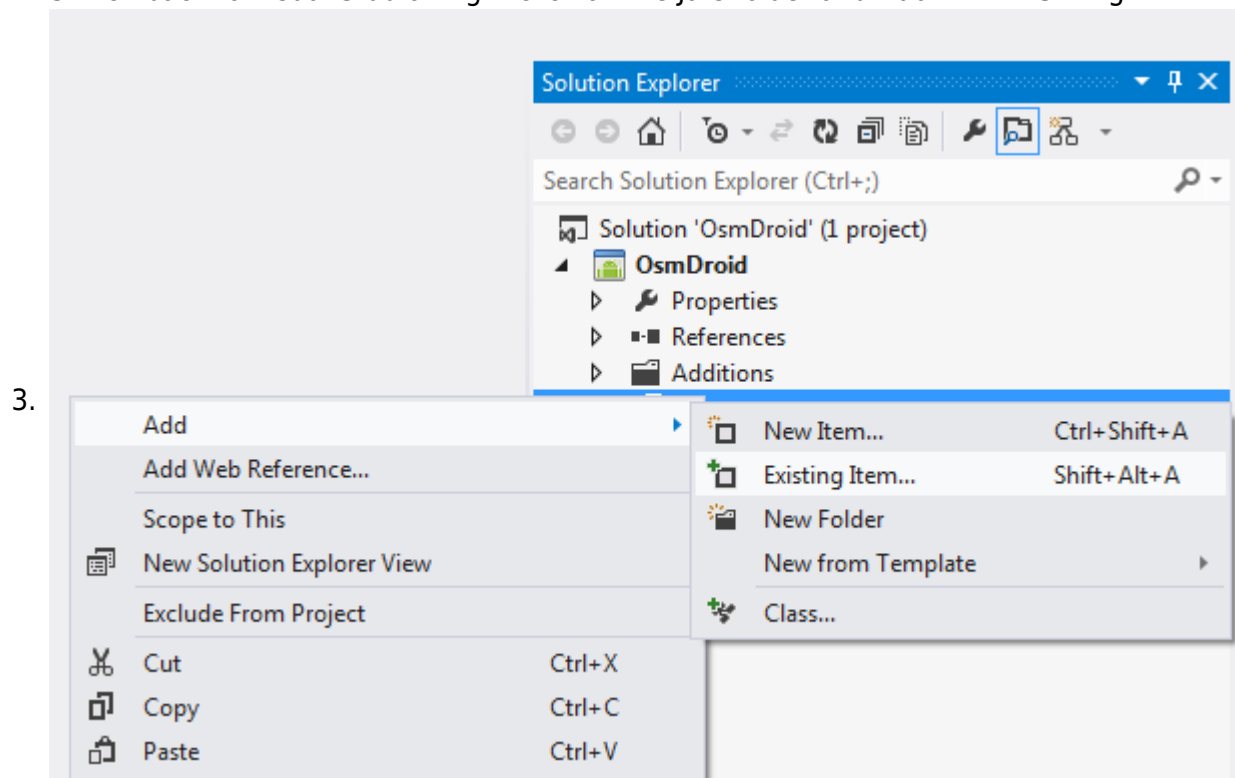
Using Visual Studio

1. osmdroid requires two .jar files which must be added to the project.

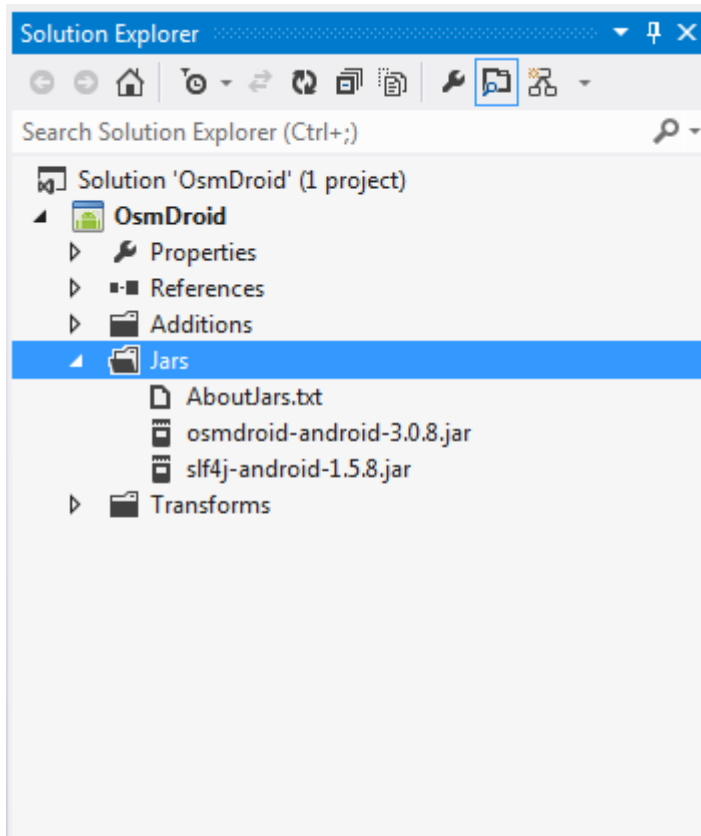
Download the two .jar's, and copy them into the Jars folder of the project using Windows Explorer.



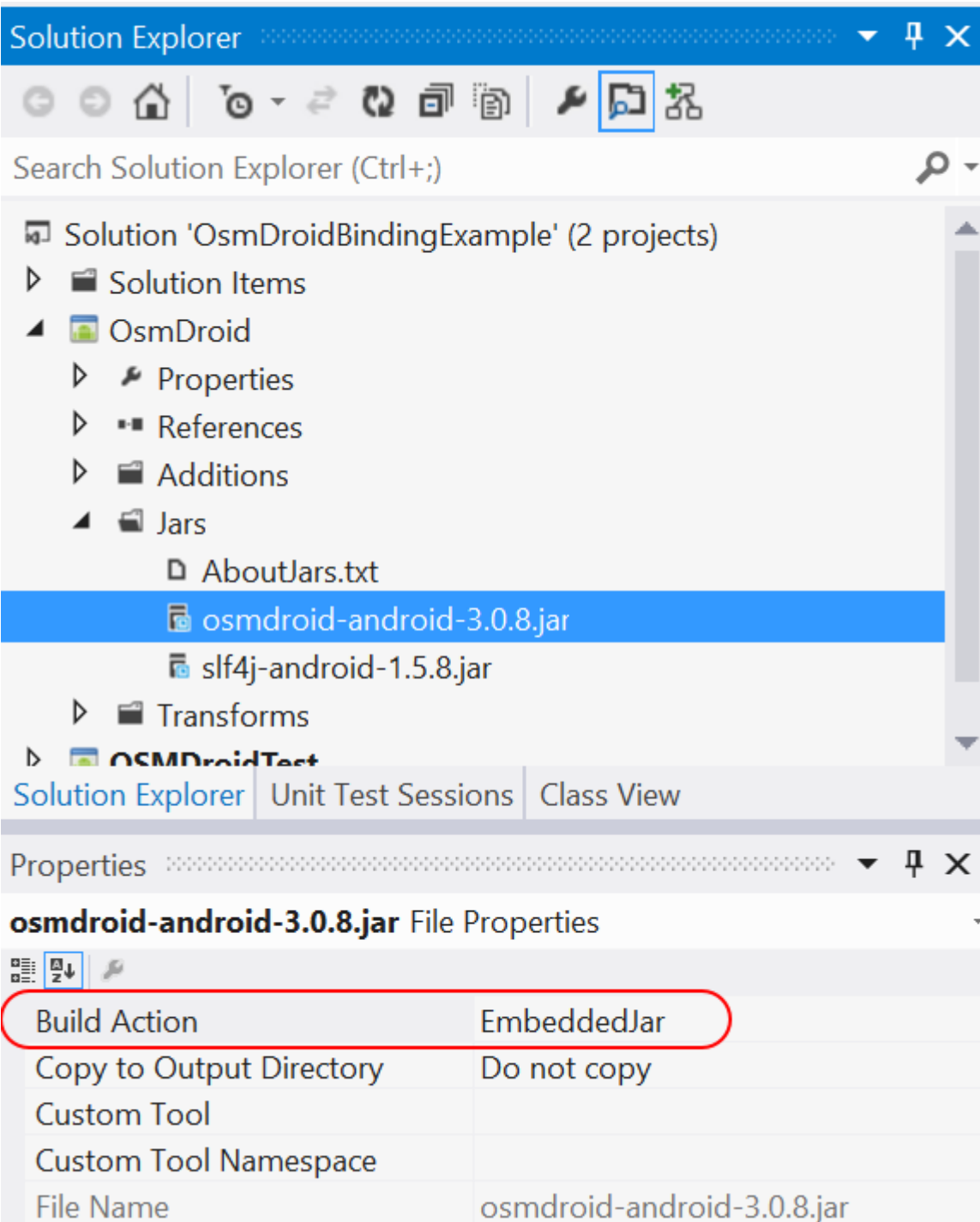
2. Switch back to Visual Studio. Right click on the Jars folder and Add > Existing.



4. Add the files osmdroid-android-3.0.8.jar and slf4j-android-1.5.8.jar to the project.



5. Next, we need to tell the project which .jar file that we want to create the bindings for. We do this by setting the Build Action of the file osmdroidandroid-3.0.8.jar to EmbeddedJar as shown below:

6. The screenshot shows the Visual Studio interface. The Solution Explorer at the top displays a solution named 'OsmDroidBindingExample' containing two projects. Under the 'OsmDroid' project, the 'Jars' folder is expanded, showing several jar files. The file 'osmdroid-android-3.0.8.jar' is selected and highlighted in blue. Below the Solution Explorer, the Properties window is open, showing the 'File Properties' for the selected jar file. A red circle highlights the 'Build Action' property, which is set to 'EmbeddedJar'. Other properties like 'Copy to Output Directory' (Do not copy), 'Custom Tool', 'Custom Tool Namespace', and 'File Name' (osmdroid-android-3.0.8.jar) are also visible.

osmdroid-android-3.0.8.jar File Properties	
Build Action	EmbeddedJar
Copy to Output Directory	Do not copy
Custom Tool	
Custom Tool Namespace	
File Name	osmdroid-android-3.0.8.jar

7. The file step is to tell the project that the file slf4j-android-1.5.8.jar is a *reference jar* that is required by osmdroid. A reference jar is a jar file is required by another jar (such as the input jar), but we do not want to generate bindings for it. We do this by setting the Build Action of the file slf4j-android-1.5.8.jar to EmbeddedReferenceJar as shown below:

8.

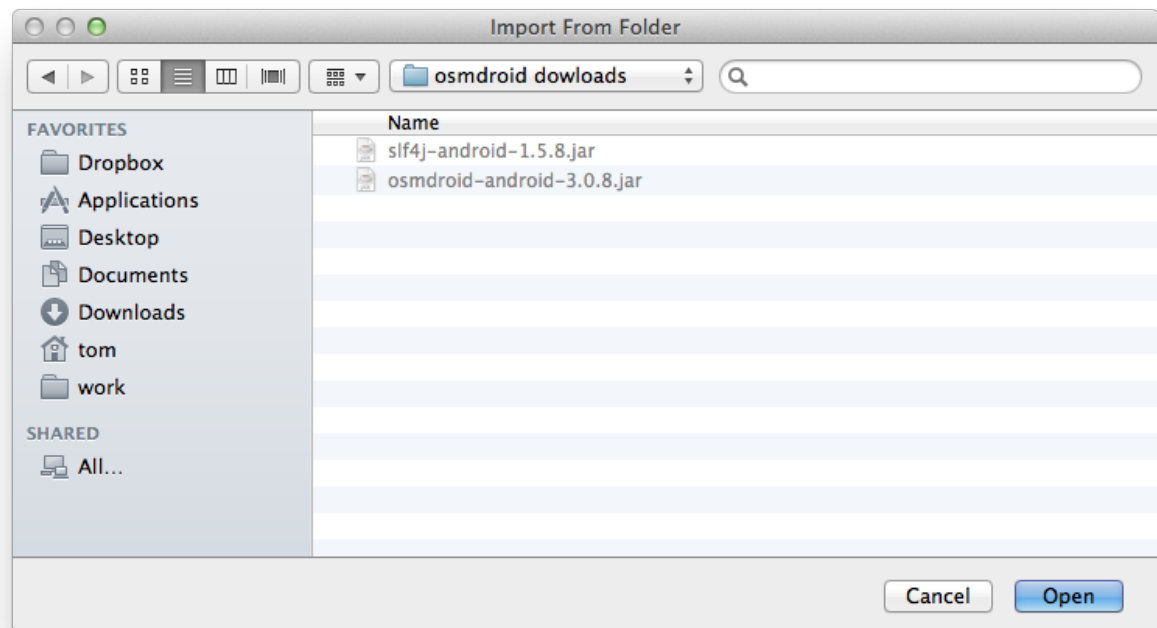
The screenshot shows the Visual Studio Solution Explorer for a project named 'OsmDroidBindingExample'. The 'Jars' folder is expanded, and 'slf4j-android-1.5.8.jar' is selected. The 'Properties' window at the bottom shows the 'Build Action' for 'slf4j-android-1.5.8.jar' is set to 'EmbeddedReferenceJar'.

slf4j-android-1.5.8.jar File Properties	
Build Action	EmbeddedReferenceJar
Copy to Output Directory	Do not copy
Custom Tool	
Custom Tool Namespace	
File Name	slf4j-android-1.5.8.jar

Using Xamarin Studio

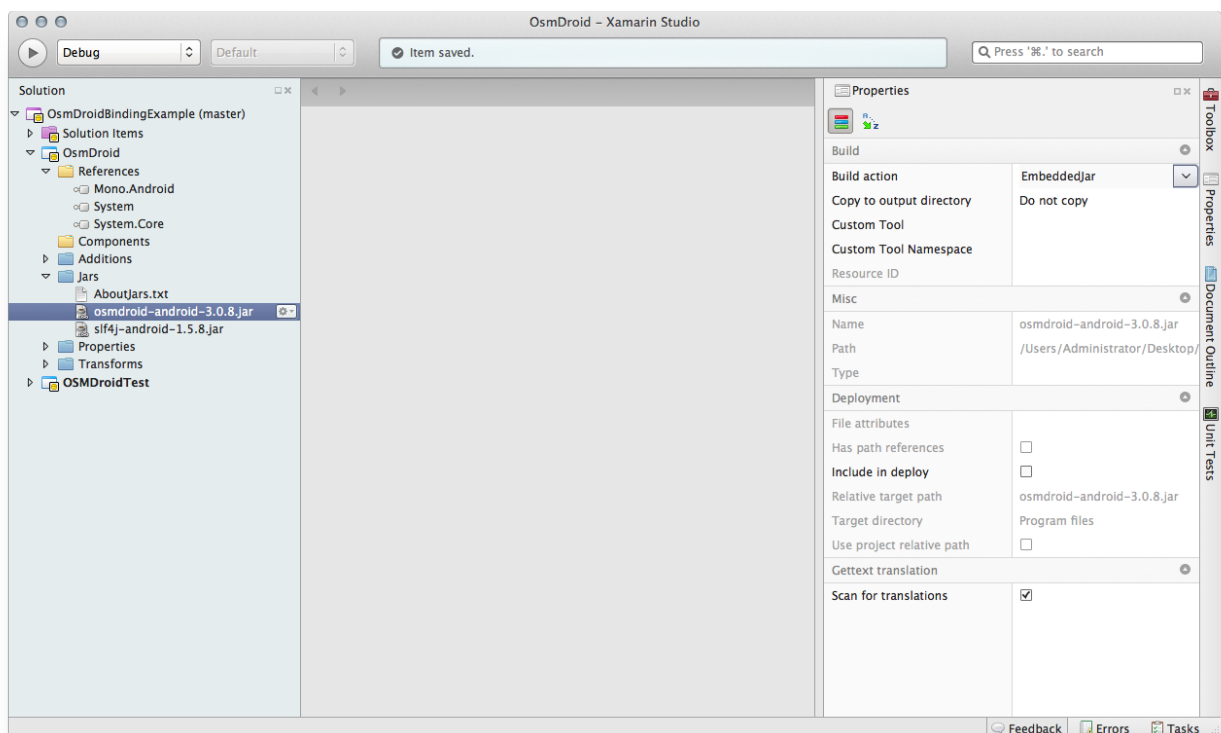
1. Right-click on the Jars folder, and then select Add >Add Files.
2. Navigate to the folder containing *osmdroid-android-3.0.8.jar* and it's dependent *slf4j-android-1.5.8.jar*, and add both those files to the project.

3.

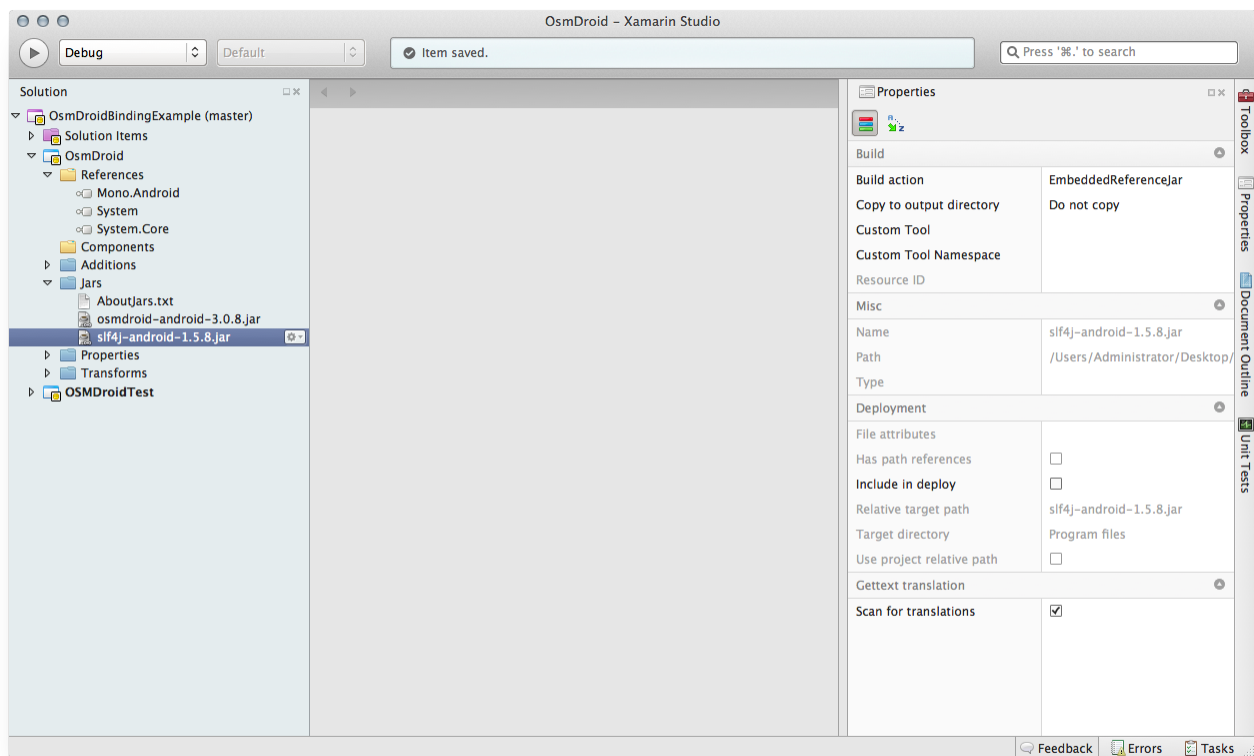


4. Next, we need to tell the project which .jar file that we want to create the bindings for. We do this by setting the Build Action of the file osmdroid-android-3.0.8.jar to EmbeddedJar as shown below:

5.



6. The file step is to tell the project that the file slf4j-android-1.5.8.jar is a *reference jar* that is required by osmdroid. A reference jar is a jar file is required by another jar, but we do not want to generate bindings for it. We do this by setting the Build Action of the file slf4j-android-1.5.8.jar to EmbeddedReferenceJar as shown below:



In this example there is one **EmbeddedJar** and one **EmbeddedReferenceJar**. However multiple EmbeddedJar and EmbeddedReferenceJars can be specified as needed.

At this point, we have a basic setup for creating the binding. However, for most configurations, including this case, to successfully build the binding we'll need to fine-tune it with some additional manual steps.

Resolving API Differences

When creating a binding, the binding project will:

- Read all the information from the jar.
- Generate a .NET assembly containing the Managed Callable Wrapper.

Transform Files

The binding project contains a Transforms folder that has three files that are used to control how the binding is generated.

- *EnumFields.xml* – Maps Java int constants to C# enums.
- *EnumMethods.xml* – Allows changes method parameters and return types from Java int constants to C# enums.
- *MetaData.xml* – Allows changes to be made to the final API, such as changing the namespace of the generated binding.

After the bindings project reads out all the information from the jar, we can use the files listed above to modify the API before it becomes an assembly. There are two reasons to do this:

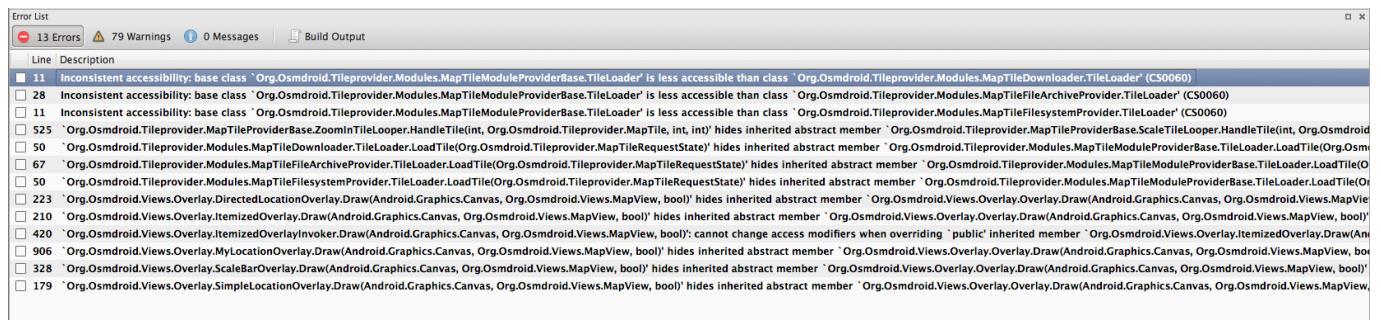
1. To fix issues with the binding.
2. To customize the API design by changing names or types, by removing unused pieces, etc.

Fixing Build Issues

Sometimes a binding will build without modification when we create it as we did in the previous section. In many cases, though, some modification will be needed before it will build correctly.

To resolve possible build issues, we need to modify the mappings used to generate the binding. We can then account for any differences between the Java API and the bindings that ship with Xamarin.Android, such as in the case described earlier, where Java int constants are replaced by C# enums in the Xamarin.Android.

For example, in the case of the osmdroid binding, if we try to compile the binding project as-is, we'll get the errors shown below, (Xamarin Studio is shown here but Visual Studio will look similar to this):



These errors are occurring for a variety of reasons:

- Some classes are less accessible than their sub-classes
- Some classes provide overrides that hide methods in base classes

Double-clicking on an error will take us to the generated file. The first error reports a problem in the generated code for the class Org.Osmdroid.Tileprovider.Modules.TileLoader, a snippet of which is shown below:

```
[global::Android.Runtime.Register ("org/osmdroid/tileprovider/modules/MapTileDownloader$TileLoader",
DoNotGenerateAcw=true)]
public new partial class TileLoader : global::Org.Osmdroid.Tileprovider.Modules.MapTileModuleProviderBase.
TileLoader {
```

The problem is that the base class, Org.Osmdroid.Tileprovider.Modules.MapTileModuleProviderBase.TileLoader, has an accessibility of protected, and the subclass is public. We can correct this by editing the file Metadata.xml, and changing the visibility of the generated C# class with the following:

```
<attr path=
"/api/package[@name='org.osmdroid.tileprovider.modules']/class[@name='MapTileModuleProviderBase.TileLoader']"
name="visibility">public</attr>
```

For the osmdroid.jar, it is necessary to do this several more times in order to adjust these visibility discrepancies. For example, if you examine Transforms\Metadata.xml, you will notice this mapping: <attr path="/api/package[@name='org.osmdroid.views.overlay']/class[@name='Overlay']/method[@name='draw']" name=

```
"visibility">public</attr>
```

This mapping will ensure that the visibility of the method `Overlay.Draw` is also public.

Correcting Enums

In order to get osmdroid to build, we need to change the two methods to use .NET enums instead of int constants of the Java code. To do this, open up the *EnumMethods.xml* file under the *Transforms* directory in the project. The following mapping can be seen:

```
<mapping jni-interface="org/osmdroid/api/IMapView">
  <method jni-name="setBackgroundColor" parameter="p0" clr-enum-type="Android.Graphics.Color" />
</mapping>
```

In the above example, we need the binding generator to map the parameter `p0` to be the enum `Android.Graphics.Color` in the `setBackgroundColor` method of the Java interface `org.osmdroid.api.IMapView`.

If we rebuild again, everything should now succeed and we'll have a `MyGoogleMaps.dll` that is ready to use.

Obfuscated Types

Beginning in Xamarin.Android 4.4, the binding generator treats any classes classes and interfaces whose name only consists of lower case letters, numbers, or the `$` as obfuscated. The obfuscated types will be filtered out by the by the binding generator and not included in the Java Binding library.

It is possible to modify *Metadata.xml* to force an obfuscated class to be included in Java Binding Library as follows:

```
<metadata>
  <attr path="/api/package[@name='{package_name}']/class[@name='{name}']" name="obfuscated">false</attr>
</xmetadata>
```

Normalizing the API

Changing intenum is just one way to modify an API to make it more familiar looking for .NET users. Some other things we might want to do to ensure that the API follows .NET patterns are:

- Rename namespaces, classes, methods, fields, or parameter names to follow .NET conventions.
- Remove namespaces, classes, methods, or fields that aren't needed.
- Move classes to different namespaces.
- Add additional support classes to make the design of the binding look more like .NET framework classes.

API Metadata

Changes to the final API can be specified in the *Transforms\Metadata.xml* file. For example, the

namespace of our Google maps library is generated as Org.Osmdroid. This type characterization reflects the capitalized Java namespace, but looks awkward to a .NET developer.

Let's add some XML to Metadata.xml that changes the namespace to OsmDroid:

```
<attr path="/api/package[@name='org.osmdroid']" name="managedName">OsmDroid</attr>
```

This causes the generator to find the Java package org.osmdroid and to change its .NET name to OsmDroid. It will not adjust any nested namespaces – so for each namespace that we wish to change we must have one mapping.

The Metadata.xml file gives us the power to modify an API in any way we want. For more information about the supported metadata operations, see [API Metadata Reference](#).

Fixing Up Parameter Names

When generating the managed callable wrapper, the generated C# code does not retain the names of parameters on Java methods. For example, the Java method org.osmdroid.views.MapView() takes only one parameter. However the generated C# code will name this parameter as p0. This is not a very meaningful name. To fix this, we can add the following mapping to Metadata.xml:

```
<attr path=
"/api/package[@name='org.osmdroid.views']/class[@name='MapView']/method[@name='setTileSource']/parameter[@name='p0']"
name="name">tileSource</attr>
```

Limitations

When creating a binding, we may add additional supporting classes to design the binding. However, if we add classes to any partial classes that we generated in the binding, we must not declare any instance fields that can reference a Java.Lang.Object subclass at runtime. This includes types such as:

- Java.Lang.Object (and subclasses).
- System.Object
- Interface fields

This restriction prevents the garbage collector from prematurely collecting objects. If a partial class must declare an instance field that can refer to a Java.Lang.Object subclass at runtime, it must use either WeakReference or GCHandle to refer to this subclass.

However, if we were working with a partial class that exposes a Java interface through .NET events, we would need to include a reference to a Java.Lang.Object.

For example, consider the [ViewPager](#) class, which is part of the Android v4 support package. This class has a [ViewPager.OnPageChangeListener](#) interface that receives callbacks. A portion of the generated binding is shown below, (with RegisterAttributes omitted for brevity):

```
public partial class ViewPager : global::Android.Views.ViewGroup {
    public partial interface IOnPageChangeListener : IJavaObject {
        void OnPageScrollStateChanged (int p0);
        void OnPageScrolled (int p0, float p1, int p2);
        void OnPageSelected (int p0);
    }
    public virtual void SetOnPageChangeListener (
        global::Android.Support.V4.View.ViewPager.IOnPageChangeListener p0)
```

```
}
```

In order to expose the callback methods in the interface as .NET events, we need to create a helper class that implements the interface and maps the methods to the events, which we can do by implementing the methods in terms of `EventHandler<T>` delegates, as shown below for the `OnPageScrollStateChanged` method.

```
internal partial class OnPageChangeEventDispatcher : Java.Lang.Object,
    ViewPager.IOnPageChangeListener
{
    ViewPager sender;

    public OnPageChangeEventDispatcher(ViewPager sender)
    {
        this.sender = sender;
    }

    internal EventHandler<PageScrollStateChangedEventArgs>
        PageScrollStateChanged;

    public void OnPageScrollStateChanged(int p0)
    {
        var h = PageScrollStateChanged;
        if (h != null)
            h(sender, new PageScrollStateChangedEventArgs()
                { State = p0 });
    }

    // OnPageScrolled and OnPageSelected omitted for brevity
}

public class PageScrollStateChangedEventArgs : EventArgs
{
    public int State { get; internal set; }
}
```

Now, we need to add our implementation to the `ViewPager` partial class that contains the actual events. This class needs to keep an instance of the `OnPageEventDispatcher` that is dispatching the events from the callback methods. However, we must use a `WeakReference` (as mentioned earlier), so that the garbage collector does not collect the object prematurely, as shown below:

```
partial class ViewPager
{
    WeakReference dispatcher;
    OnPageChangeEventDispatcher EventDispatcher
    {
        get
        {
            if (dispatcher == null || !dispatcher.IsAlive)
            {
                var d = new
                    OnPageChangeEventDispatcher(this);
                SetOnPageChangeListener(d);
                dispatcher = new WeakReference(d);
            }
            return(OnPageChangeEventDispatcher)
                dispatcher.Target;
        }
    }

    public event EventHandler<PageScrollStateChangedEventArgs>
        PageScrollStateChanged
    {
        add
        {
            EventDispatcher.PageScrollStateChanged += value;
        }

        remove
        {

```

```

        }
        Dispatcher.PageScrollStateChanged -= value;
    }
}

```

The OnPageScrolled and OnPageSelected methods can be implemented in a similar fashion.

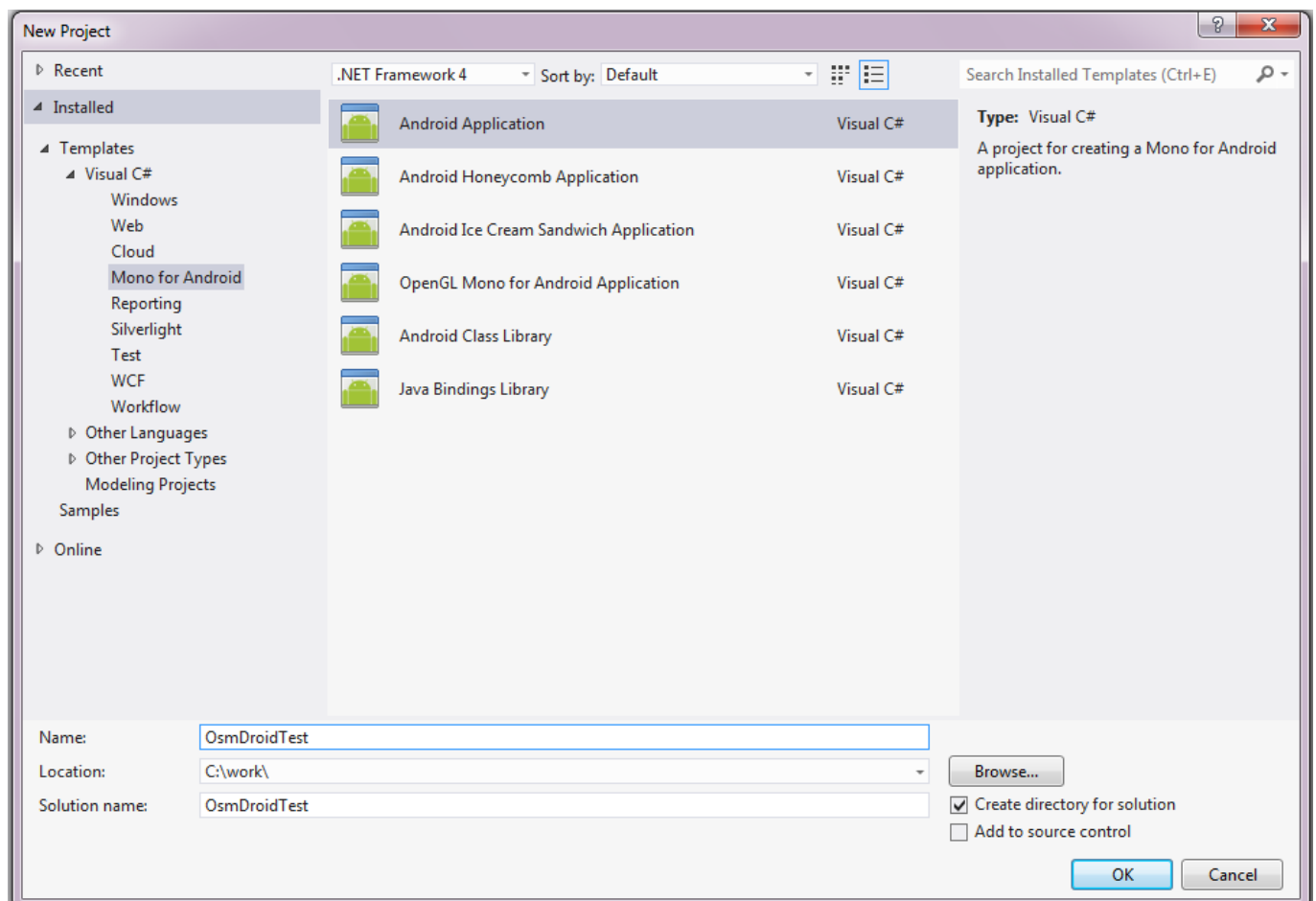
Using the Bound Library

Once the Java library has been bound, we can use the binding in a Xamarin.Android project just like any other .NET assembly. The only additional step we need to take is to include the original *jar* as well. A future version of Xamarin.Android will eliminate need to include the origin .jar file in the Xamarin.Android project.

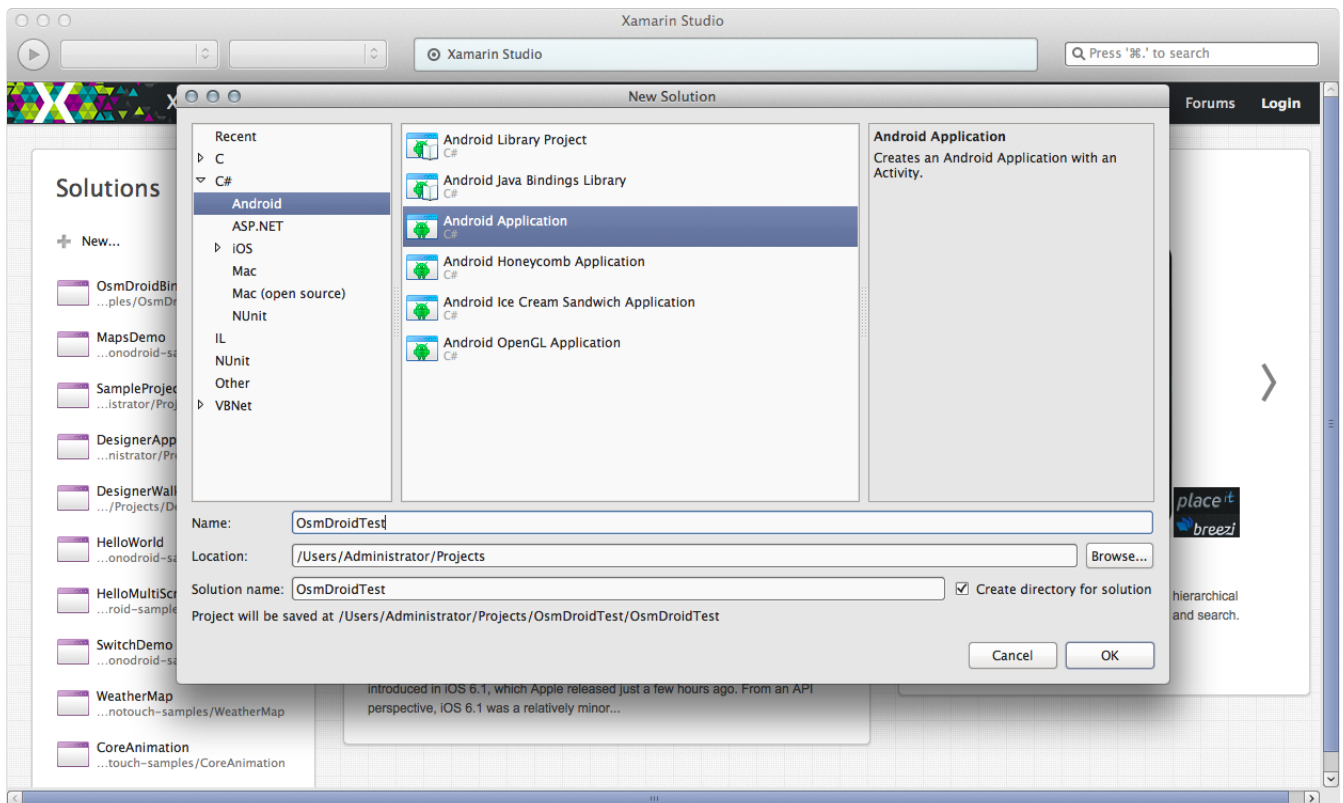
Add a Reference to the Binding Assembly

For example, to use the Google Maps binding from the previous section in another Xamarin.Android application, first add a reference to the dll for the binding:

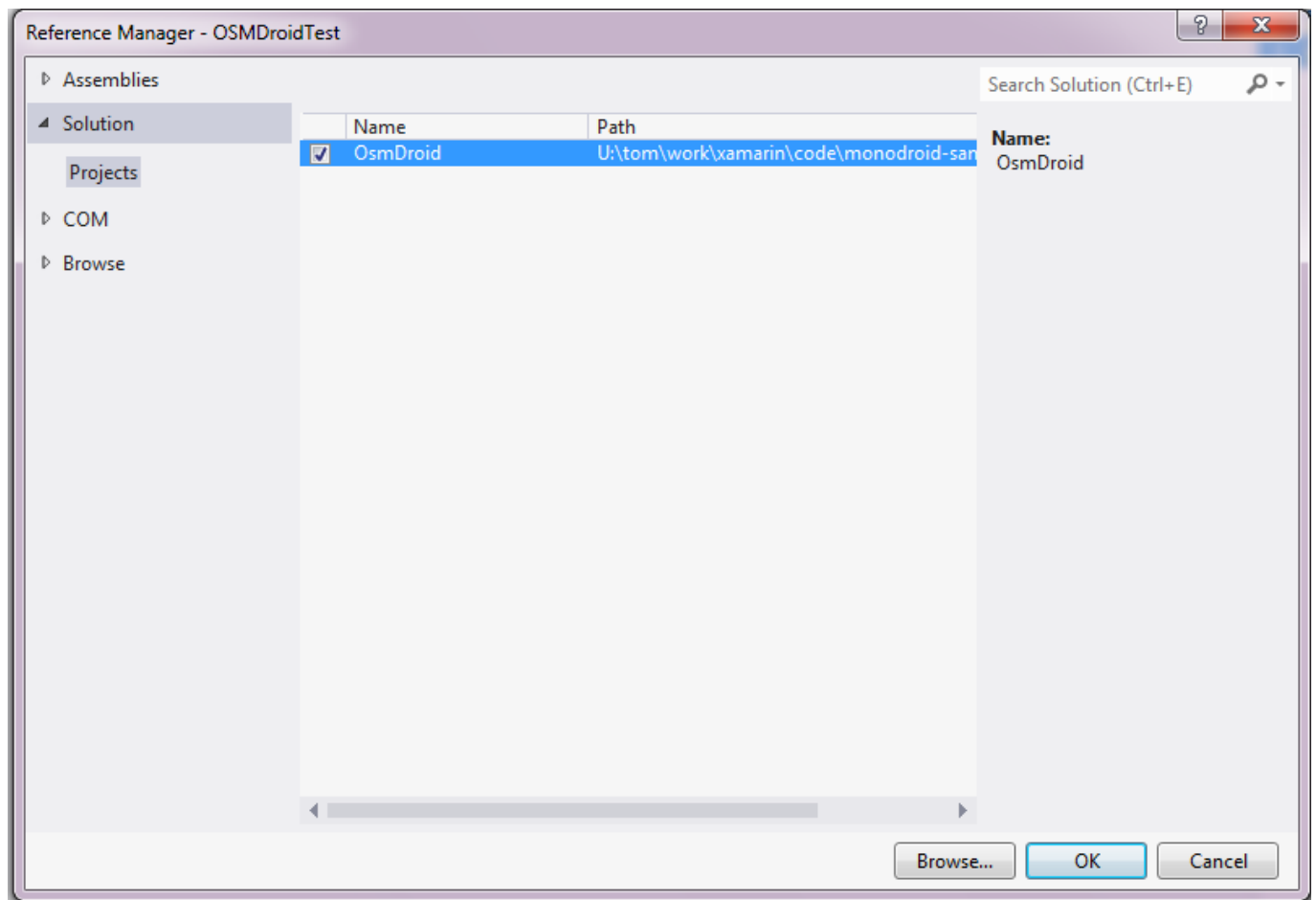
1. Right-click on the solution, and then select Add > New Project.
2. Select Mono for Android > Android Application, and then give it a name such as OsmDroidTest, as shown below in Visual Studio:



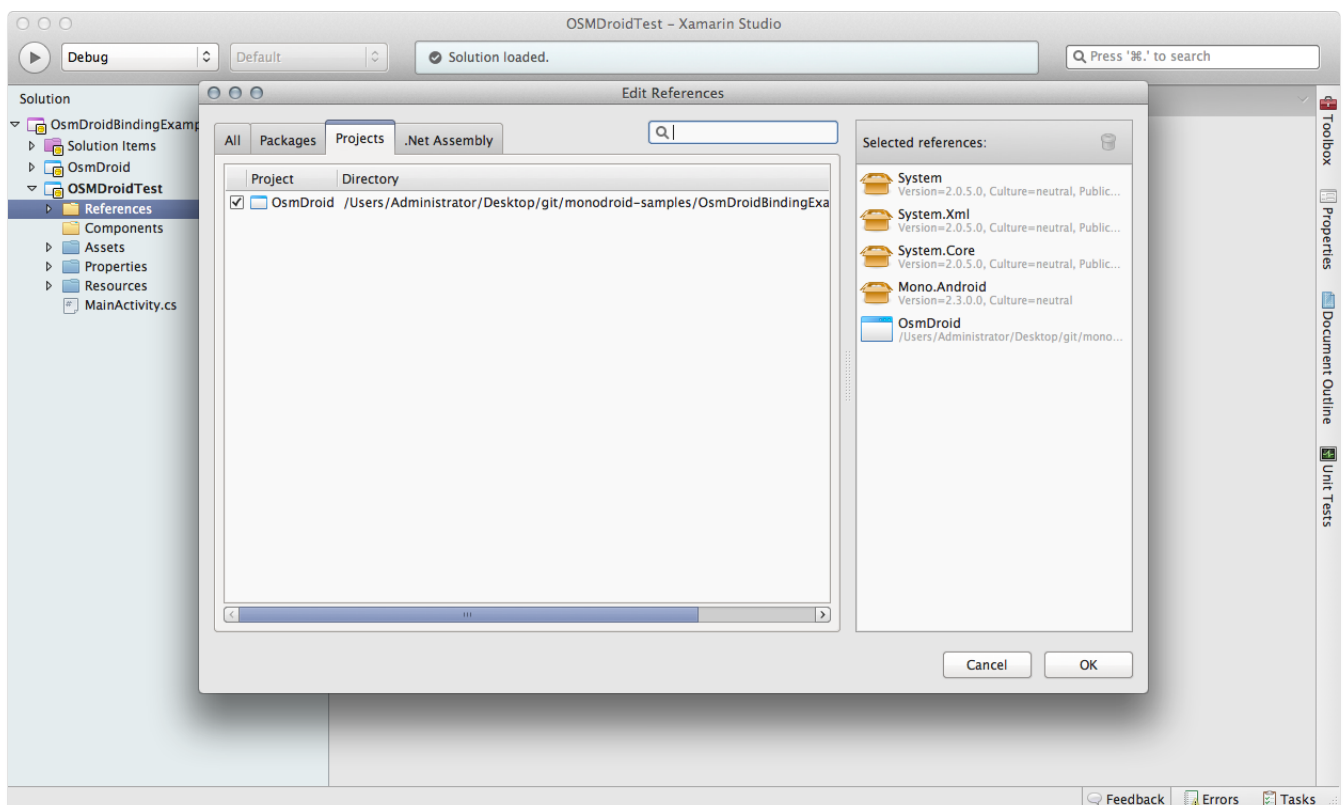
For Xamarin Studio, create the project by using the New Project dialog:



1. In Visual Studio, right-click on the new project in the Solution Explorer, and then select Add Reference. If using Xamarin Studio, right-click References, and then select Edit References.
2. Select the OsmDroid binding project, and then click OK to add the reference, as shown below in Visual Studio:



Or for Xamarin Studio, add the assembly in the Edit References dialog:



Note: Due to [bug 6695](#), when referencing the Java Binding project directly from a Xamarin.Android project, none of the namespaces, classes, or other members of the Java Binding project will be visible

in the IDE when working on the Xamarin.Android project. This bug will be fixed in a future version of Xamarin.Android.

Troubleshooting Bindings

The following table summaries several common errors that may occur when generating bindings, along with possible causes.

Problem	Possible Causes
You receive the error "at least one Java library is required," even though a jar has been added."	Make sure Build Action is set to EmbeddedJar. Since there are two build actions for *.jar files (EmbeddedJar and EmbeddedReferenceJar), the binding generator does not automatically guess which to use by default.
The binding library generator fails to load the jar library.	Some jar libraries that use code obfuscation (via tools such as Proguard) fail to load by the Java tools. Since our tool makes use of Java reflection and ASM byte code engineering library, those dependent tools may reject the obfuscated libraries while Dalvik tools may pass. The workaround for this is to hand-bind these libraries instead of using the binding generator.
The binding .dll builds but misses some Java types, or the generated C# source does not build due to an error stating there are missing types.	<p>This error may occur due to several reasons as listed below:</p> <ul style="list-style-type: none">· There was a bug in .NET 4.0 runtime that failed to load assemblies when it should have. This issue has been fixed in .NET 4.5 beta runtime. When there is .NET 4.5 beta installed, it does not happen. This issue is fixed in Xamarin.Android version 4.2.· The library being bound may reference a second java library. If the public API for the bound library uses types from the second library, you must reference a managed binding for the second library as well.· It's possible that a library was injected due to Java reflection, similar the reason for the library load error above, causing the unexpected loading of metadata. Xamarin.Android's tooling cannot currently resolve this situation. In such a case, the library must be manually bound.· Java allows deriving a public class from non-public class, which is unsupported in .NET. Since the binding generator does not generate bindings for non-public classes, derived classes such as these cannot be generated correctly.<ul style="list-style-type: none">o To fix this, either remove the metadata entry for those derived classes using the remove-node in metadata.xml, or fix the metadata making the non-public class public. For example, public</attr>o However, although the latter solution will create the binding so that the C# source will build, the non-public class should not be used. For information on using metadata.xml see the API Metadata Reference .
Is there any syntax documentation for metadata transform XML?	Please refer to: API Metadata Reference .
The generated C# source does not build. Overridden method's parameter types do not match.	Xamarin.Android includes a variety of Java fields that are mapped to enums in the C# bindings. These can cause type incompatibilities in the generated bindings. To resolve this, the method signatures created from the binding generator need to be modified to use the enums. For Please see the Correcting Enums section.
java.lang.NoClassDefFoundError is thrown in the packaging step.	<p>The most likely reason for this error is a mandatory Java library needs to be added to the application project (csproj). Jars are not automatically resolved. A Java library binding is not always generated against a user assembly that does not exist in the target device or emulator (such as Google Maps maps.jar).</p> <ul style="list-style-type: none">· This is not the case for Android Library project support, as the library jar is embedded in the library dll. <p>For example: https://bugzilla.xamarin.com/show_bug.cgi?id=4288</p>
Build fails due to duplicate custom EventArgs types. Some error like this occurs: error CS0102: The type 'Com.Google.Ads.Mediation.DismissScreenEventArgs' already contains a definition for 'p0'	<p>This is because there is some conflict between event types that come from more than one interface "listener" type that shares methods having identical names. For example, if there are two Java interfaces like below, the generator creates XEventArgs for both A and B, resulting in the error.</p> <pre>public interface AListener { void onX (C arg); } public interface BListener { void onX (C arg); }</pre> <p>This is by design to avoid lengthy names on event argument types. To avoid these conflicts, some metadata transformation is required. On metadata for either of the conflicting interface methods, there has to be an argsTypeadditional attribute.</p>

<p>I get an error saying that that a generated class does implement a method that is required for an interface which the generated class implements. However, when I look at the generated code, I can see that the method is implemented.</p> <p>Here is an example of the error:</p> <pre>obj\Debug\generated\src\Oauth.Signpost

 .Basic.HttpURLConnectionRequestAdapter.cs(8,23):

 error CS0738: 'Oauth.Signpost.Basic

 .HttpURLConnectionRequestAdapter'

 does not implement interface member 'Oauth.Signpost.Http

 .IHttpRequest.Unwrap()'. 'Oauth.Signpost.Basic

 .HttpURLConnectionRequestAdapter.Unwrap()' cannot implement 'Oauth.Signpost.Http

 .IHttpRequest.Unwrap()''

 because it does not have the

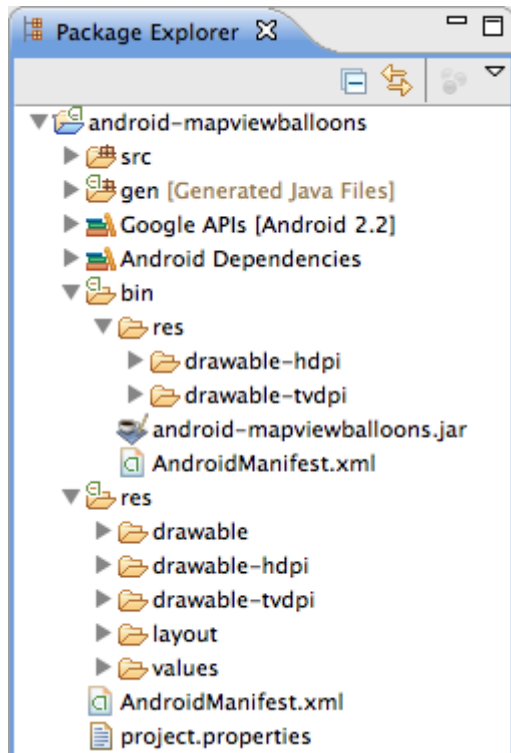
 matching return type of 'Java.Lang.Object'</pre>	<p>This is a problem that occurs with binding Java methods with covariant return types. In this example, the method <code>Oauth.Signpost.Http.IHttpRequest.UnWrap()</code> needs to return <code>Java.Lang.Object</code>. However, the method <code>Oauth.Signpost.Basic.HttpURLConnectionRequestAdapter.UnWrap()</code> has a return type of <code>HttpURLConnection</code>. There are two ways to fix this issue:</p> <ol style="list-style-type: none"> 1. Add a partial class declaration for <code>HttpURLConnectionRequestAdapter</code> and explicitly implement <code>IHttpRequest.Unwrap()</code>: <pre>namespace Oauth.Signpost.Basic { partial class HttpURLConnectionRequestAdapter { Java.Lang.Object OauthSignpost.Http.IHttpRequest.Unwrap() { return Unwrap(); } } }</pre> 2. Remove the covariance from the generated C# code. This involves adding the following transform to <code>Transforms\Metadata.xml</code> which will cause the generated C# code to have a return type of <code>Java.Lang.Object</code>: <pre><attr path= "/api/package[@name='oauth.signpost.basic']/class[@name='HttpURLConnectionRequestAdapter']/method[@name='unwrap']" name="managedReturn"> Java.Lang.Object</attr></pre>
--	--

Android Library Projects

Android library projects are special Android projects that contain shareable code and resources that can be referenced by Android application projects. Android library projects are different from regular Android projects in that they are not compiled into an .apk and are not, on their own, deployable to a device.

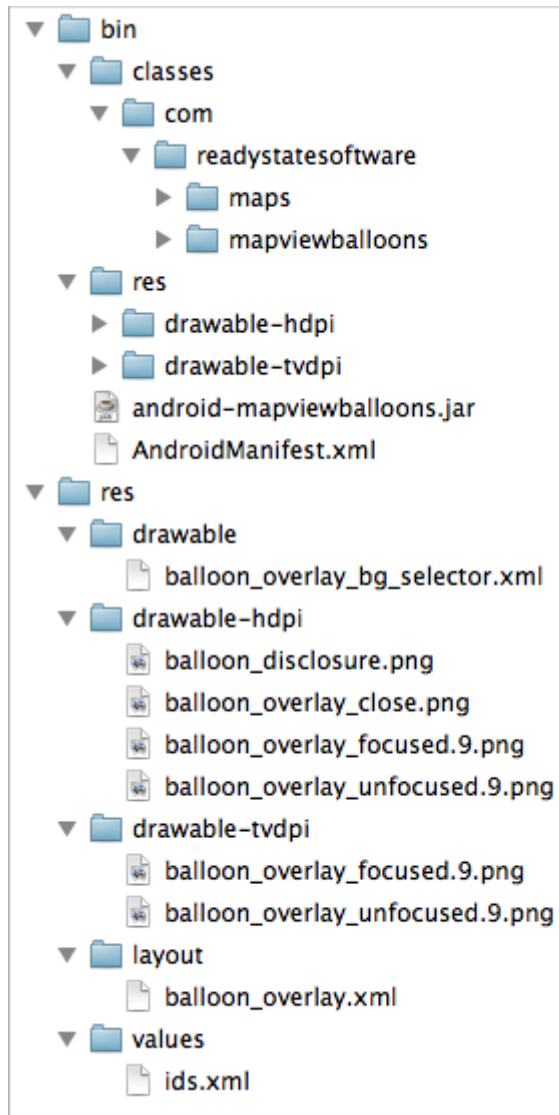
Instead, an Android library project is meant to be referenced by an Android application project. When an Android application project is built, the Android library project is compiled first. The Android application project will then absorb the compiled Android library project and include the code and resources into the APK for distribution. Because of this difference, creating a binding for an Android library project is slightly different than creating a binding for a Java .jar file.

To use an Android library project in a Xamarin.Android Java Binding project it is first necessary to build the Android library project in Eclipse. The following screenshot shows an example of one Android library project after compilation:

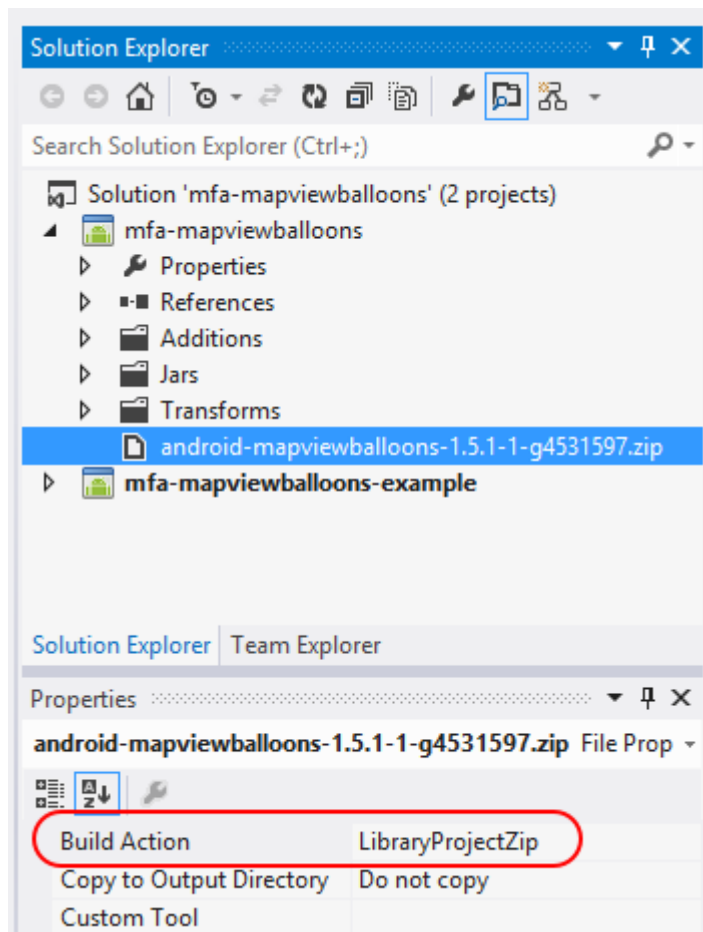


Notice that the source code from the Android library project has been compiled to a temporary .jar file named android-mapviewballoons.jar, and that the resources have been copied to the folder bin/res/.

Once the Android library project has been compiled in Eclipse, it can then be bound using a Xamarin.Android Java Binding project. First a .zip file must be created which contains the bin and res folders of the Android library project. The following screenshot shows the contents of one such .zip file:

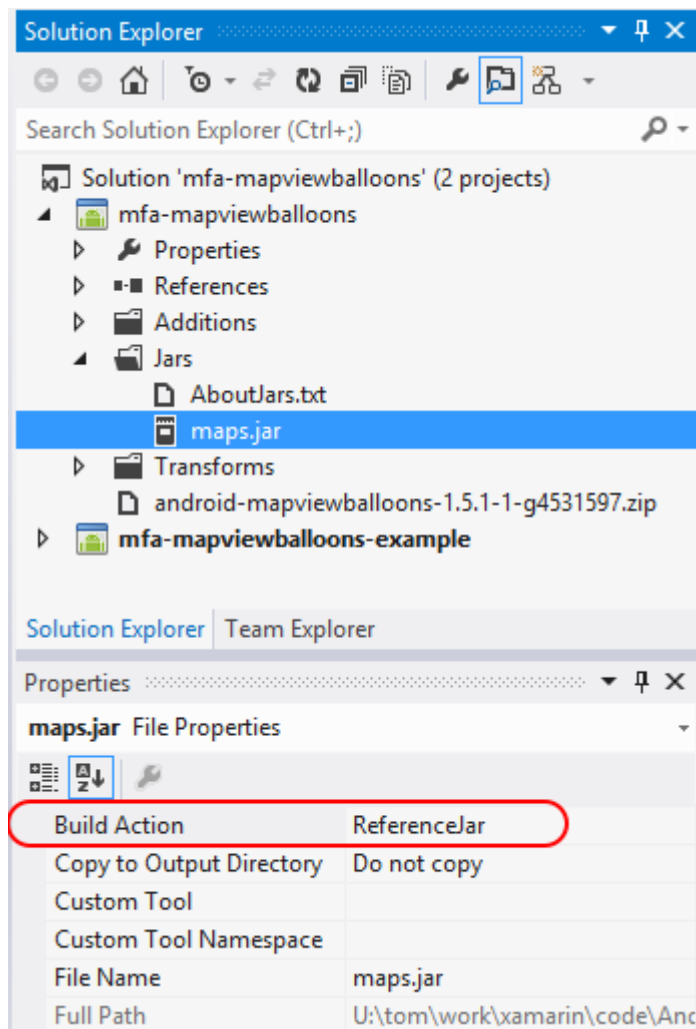


This .zip file is then added to Xamarin.Android Java Binding project, as shown in the following screenshot:



Notice that the Build Action of the .zip file has been automatically set to LibraryProjectZip.

If there are any .jar files that are required by the Android library project, they should be added to the Jars folder of the Java Binding Library project and the Build Action set to ReferenceJar. An example of this can be seen in the screenshot below:



Once these steps are complete, the Xamarin.Android Java Binding project can be used as described earlier on in this document.

Note: Compiling the Android library projects in other IDEs is not supported at this time. Other IDEs may not create the same directory structure or files in the bin folder as Eclipse.

Summary

In this article, we walked through how to use the new *Java Bindings Library* project template to create a binding to a Java library. Using the Google Maps library as an example, we showed how to add a *jar* to the project and how to generate that binding. We then examined how to resolve any errors that resulted from incompatibilities between the Java library and Xamarin.Android. We also examined various trouble shooting issues and how to resolve them. Finally, we discussed techniques that showed us how to shape an API exposed by a binding to more closely follow .NET design guidelines.

Source URL:

http://docs.xamarin.com/guides/android/advanced_topics/java_integration_overview/binding_a_java_library_%28.jar%28