Android Callable Wrappers

Android callable wrappers are needed any time the Android runtime needs to invoke managed code, and are required because there is no way to register classes with Dalvik at runtime. (Specifically, the JNI DefineClass() function is not supported by Dalvik. Android callable wrappers thus make up for the lack of runtime type registration support.

Every time Android code needs to execute a virtual or interface method which is overridden or implemented in managed code, Xamarin.Android needs to provide a Java proxy so that the method gets dispatched to the appropriate managed type. These Java proxy types are Java code which have the "same" base class and Java interface list as the managed type, implementing the same constructors and declaring any overridden base class and interface methods.

Android callable wrappers are generated by the monodroid.exe program during the build process , and are generated for all types that (directly or indirectly) inherit Java.Lang.Object .

Implementing Interfaces

There are times when you may need to implement an Android interface, such as Android.Content.IComponentCallbacks . Since all Android classes and interface extend the Android.Runtime.IJavaObject interface, the question arises: how do we implement IJavaObject?

The question was answered above: the reason all Android types need to implement IJavaObject is so that Xamarin.Android has an Android callable wrapper to provide to Android, i.e. a Java proxy for the given type. Since monodroid.exe only looks for Java.Lang.Object subclasses, and Java.Lang.Object implements IJavaObject, the answer is obvious: subclass Java.Lang.Object:

```
class MyComponentCallbacks : Java.Lang.Object, Android.Content.IComponentCallbacks {

    public void OnConfigurationChanged (Android.Content.Res.Configuration newConfig)
    {
        // implementation goes here...
    }

    public void OnLowMemory ()
    {
        // implementation goes here...
    }
}
```

Implementation Details

The remainder of this page provides implementation details subject to change without notice (and presented here only because developers will be curious about what's going on).

For example, given the following C# source:

```
using System;
using Android.App;
using Android.OS;
```

```
namespace Mono.Samples.HelloWorld
{
        public class HelloAndroid : Activity
        {
                protected override void OnCreate (Bundle savedInstanceState)
                {
                        base.OnCreate (savedInstanceState);
                        SetContentView (R.layout.main);
                }
        }
}
```

The mandroid.exe program will generate the following Android Callable Wrapper:

```
package mono.samples.helloWorld;

public class HelloAndroid
        extends android.app.Activity
{
        static final String __md_methods;
        static {
                __md_methods = 
                        "n_onCreate:(Landroid/os/Bundle;)V:GetOnCreate_Landroid_os_Bundle_Handler\n" +
                        "";
                mono.android.Runtime.register ("Mono.Samples.HelloWorld.HelloAndroid, HelloWorld, Version=1.0.0.0, 
Culture=neutral, PublicKeyToken=null", HelloAndroid.class, __md_methods);
        }

        public HelloAndroid ()
        {
                super ();
                if (getClass () == HelloAndroid.class)
                        mono.android.TypeManager.Activate ("Mono.Samples.HelloWorld.HelloAndroid, HelloWorld, 
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null", "", this, new java.lang.Object[] {  });
        }

        @Override
        public void onCreate (android.os.Bundle p0)
        {
                n_onCreate (p0);
        }

        private native void n_onCreate (android.os.Bundle p0);
}
```

Notice that the base class is preserved, and native method declarations are provided for each method that is overridden within managed code.

Source URL:
http://docs.xamarin.com/guides/android/advanced_topics/java_integration_overview/android_callable_wrappers