Writing interfaces that can be implemented by both Native and Managed classes

This article explains how to specify an interface that can be implemented by native and managed classes on Window Phone 8. The interface is then used in an Inversion of Control pattern (IoC).

Introduction

Interfaces are fundamental in object-oriented systems. An interface is a contract, that specifies the complete set of requests that can be sent to an object, typically its methods, properties and fields. On Windows Phone 8, it is very easy to define interfaces in managed code (C#, Visual Basic), and on the native side (C++), an interface is defined using abstract classes.

What is not so obvious is how to define an interface that can be implemented by both native and managed classes. Windows Phone Runtime framework offers a mechanism that enables this use case - after creating a concrete implementation of the interface, you can use the class without knowing whether it was implemented using a native or a managed code. The mechanism also makes it possible to call .NET code from C++.

This wiki article presents how such an interface is defined, and proposes scenarios where this pattern is interesting.

Interface definitions in C# and C++

Every object oriented programming language has the concept of an interface.

In C#, an interface is defined by using the `interface` keyword:

```
interface ICameraEffect
{
    Windows.Foundation.Size OutputBufferSize { get; set; }       // A property.
    Windows.Foundation.IAsyncAction GetNewFrameAndApplyEffect(); // A method
}
```

In C++, you would define an interface by having a class with only *pure virtual methods*.

```
class CameraEffectInterface
{
public:
  virtual void OutputBufferSize(Size size) = 0; // Setter
  virtual Size OutputBufferSize() = 0;     // Getter

  virtual IAsyncAction * GetNewFrameAndApplyEffect() = 0; // A method
};
```

Interfaces in Windows Phone Runtime

The Windows Phone Runtime is a great new feature of the Windows Phone 8. The runtime allows developers to freely pass objects that obey the Windows Phone Runtime requirements between the Native (C++) and the managed side (C#/Visual Basic). A C# application can use an object defined in C++ and vice-versa. Since most .NET classes do not obey these rules, .NET classes can't be passed between Native and Managed code.

The framework makes it possible to define an interface that can be implemented either by native (C++) or a managed (C#) components. Behind the scenes, our interface will be compiled to a **.winmd** file, the file that contains descriptions of the public Windows Runtime types, which include classes, structs, enumerations, interfaces, parameterized interfaces, and delegates. This is a key file: the universal language that is understood by both the C++ and the C# projects.

To define the interface, two options are possible:

- Define the interface in Interface Design Language (IDL).
- Define the interface in C++, using the Windows Phone Runtime extensions (WinPRT)

(c) Copyright Nokia 2014. All rights reserved.

There is one big problem hidden in that short list: you can't define an interface using either approach in managed code. That comes from the limitation that one can't create WinPRT components in managed code on Windows Phone 8 (it is possible to create such components on the Windows 8). As we are about to see it is fairly easy to work around this limitation.

Interface Design Language (IDL)

The IDL is the good old way of defining interfaces across the different Windows frameworks - as its name implies its a language dedicated to define interfaces. In the Component Model (COM), interfaces would be defined using IDL; and although COM has been kept out of developer reach in Windows Phone 8, it is still an important part of the platform and the IDL will work in WP8 (see this excellent example).

The IDL path seems promising at first. However, it is limited by a tooling problem. If the interface uses any other types than the ones from **Windows.Foundation**, the MIDL compiler fails to create a valid **.winmd** file and the linker will complains about missing an assembly:

```
    c:\MainPage.xaml.cs(20,26,20,34): error CS0012: The type
'Windows.Phone.Media.Capture.CameraCaptureFrame' is defined in an assembly that is not
referenced. You must add a reference to assembly 'Windows.Phone.Media.Capture,
Version=255.255.255.255, Culture=neutral, PublicKeyToken=null,
ContentType=WindowsRuntime'.
    c:\MainPage.xaml.cs(20,26,20,34): error CS0535: 'PhoneApp15.MainPage' does not
implement interface member
'WRLCompV1.IPerson.Name2(Windows.Phone.Media.Capture.CameraCaptureFrame)'
```

Although it could be used for our purpose, the IDL is not that relevant anymore as it is superseded by the newest Windows Runtime and the Windows Phone Runtime frameworks.

Using C++ for the WinPRT interface definition

The Windows Phone RunTime framework comes with the Microsoft Visual C++ component extensions (C++/CX) that are used to define WinPRT interfaces. The extensions introduce keywords that are foreign to the vanilla C++ language but familiar to .NET programmers, like `interface` and `property`.

In WinPRT, an interface defined in C++ (typically in a header file) looks like this:

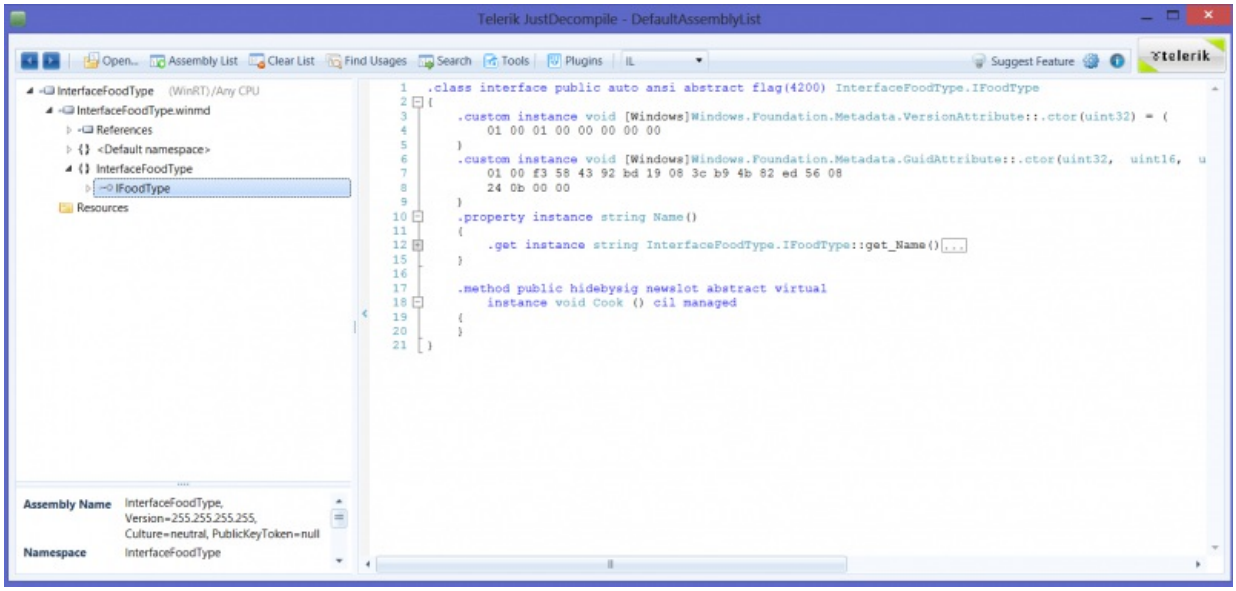
```
namespace InterfaceFoodType
{
   public interface class IFoodType
      {
  // Define a property with a public getter
         property Platform::String^ Name
         {
               Platform::String^ get();
         }

  // A method
         void Cook();
      };
}
```

From this declaration, the compiler will create a **.winmd** file, the all important file that documents the interface in a form understood by WinPRT.

The .winmd datafile

That file contains descriptions of the public Windows Runtime types, which include classes, structs, enumerations, interfaces, parameterized interfaces, and delegates. This is a key file: the universal language that is understood by both the C++ and the C# projects. It is a generated file from either the IDL or from the Microsoft Visual C++ component extensions (C++/CX). It is interesting to take a look at what it contains. A wonderful (and free!) tool for inspecting the content is Telerik JustDecompile.

(c) Copyright Nokia 2014. All rights reserved.

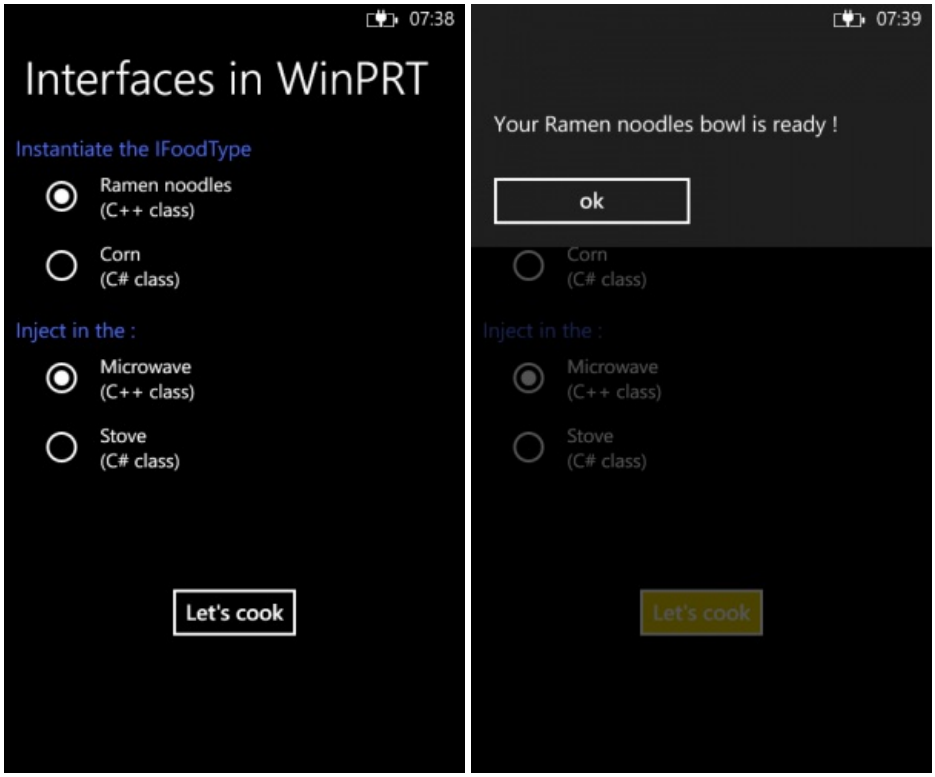


Implementing an example

Here is a very simple application that demonstrates the interface concepts. The application allows the user to select between two implementations of the `IFoodType` interface: some grains of corn and a bowl of ramen noodles.

The user can then select to cook the food on a stove or in the microwave.

The UI allows mix and matching food types on any cooking appliance:



At the code level, the following will be defined:

- An Interface (`IFoodType`), representing the food. That interface offers a `Name` property as well as a `Cook()` method.
- One concrete implementation of the `IFoodType` interface in C++: `RamenNoodles`.
- One concrete implementation of the `IFoodType` interface in C#: `PopCorn`.
- One C++ class that heats food : `Microwave`.
- One C# class that heats food : `Stove`.

The `IFoodType` interface is injected into the heating class following the IoC pattern.

The source code for this application : Media:InterfacesInWinPRT.zip

(c) Copyright Nokia 2014. All rights reserved.

Defining the interface

We will declare the IFoodType interface in C++. We have decided to define the interface a standalone project, `InterfaceFoodType`. The interface is defined in one header file, **IFoodtype.h**.

```
namespace InterfaceFoodType
{
   public interface class IFoodType
      {
 // Define a property with a public getter
         property Platform::String^ Name
         {
               Platform::String^ get();
         }

 // A method
         void Cook();
      };
}
```

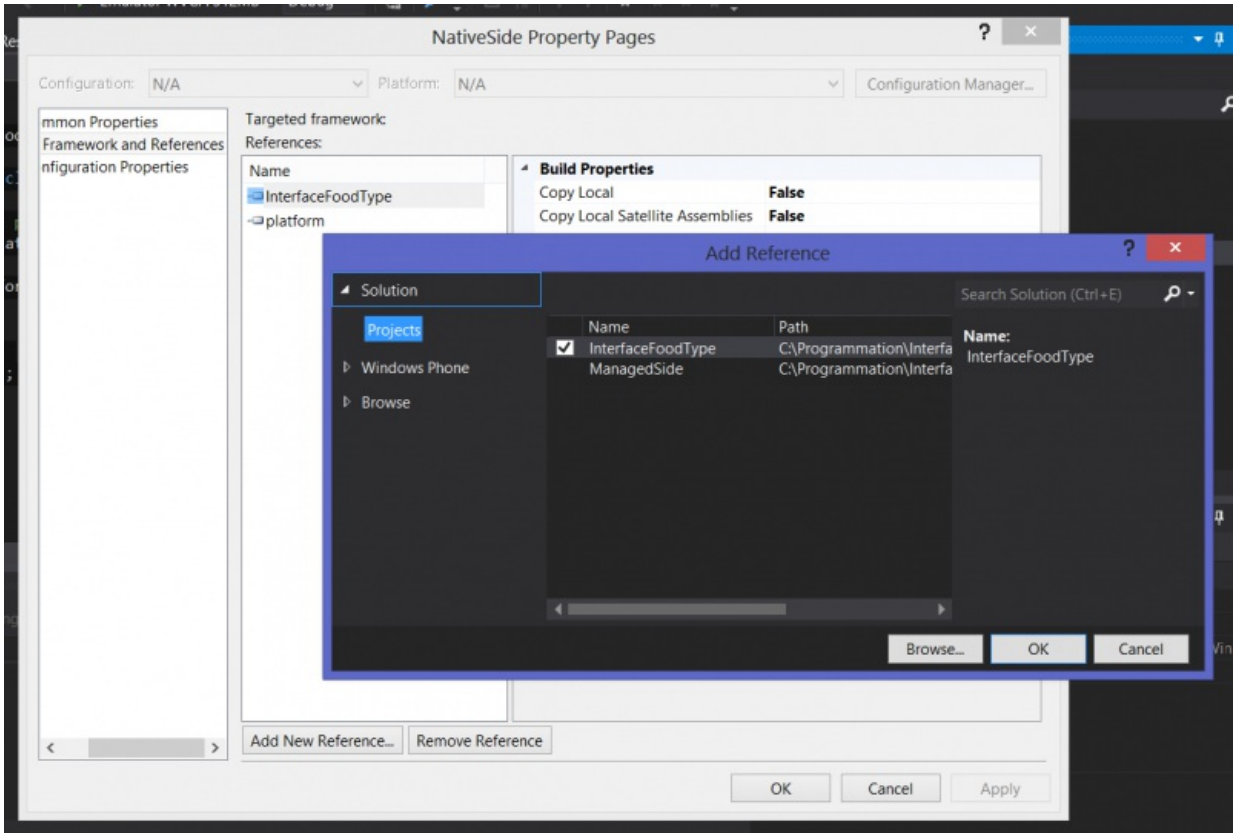
If you compile this project, a **.winmd** file will be created. If you were compiling for x86 target (the emulator), the file will show up in the directory *projectRoot/debug/InterfaceFoodType*. When compiling for an ARM target, the file ends up in the directory (*projectRoot/ARM/debug/InterfaceFoodType*).

Implementing the interface in C++

In this example, the C++ class `RamenNoodles` implements the `IFoodType` interface.

To implement the interface in a C++ class, one needs first to add the reference to the project where the interface is defined.

1. Select the C++ project properties, choose "Framework and References".
2. Click **Add New Reference**
3. On the left hand side, Select **Solution | Projects**.
4. Finally, select the library "InterfaceFoodType"



Then define your class, inheriting from the interface (`IFoodType` in this case).

(c) Copyright Nokia 2014. All rights reserved.

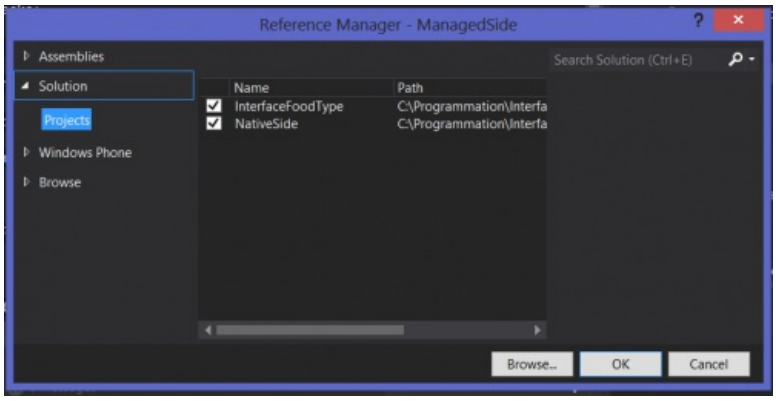
```
using namespace InterfaceFoodType;

namespace NativeSide
{
 public ref class RamenNoodles sealed : IFoodType
 {
 public:
  RamenNoodles(void);

  virtual property Platform::String^ Name
  {
   Platform::String^ get();
  }

  virtual void Cook();
 };
}
```

Note that the compiler is clever enough and will understand your interface (`IFoodType` in this case) from the **.winmd** file you provided in the step above. No need to `#include` any files where `IFoodType` effect is defined!

Implementing the interface in C#

In this example, the C# class Popcorn implements the `IFoodType` interface. Things are very similar in C# than in C++. To implement the interface in a C# class, you will first have to add a reference. The dialog is slightly different:

1. Select **Add Reference...**
2. On the left hand side, select **Solution | Projects**
3. The select the `InterfaceFoodType` project.



Then define your class, inheriting from the interface (`IFoodType` in this case):

```
using InterfaceFoodType;

namespace InterfacesInWinPRT
{
    class Popcorn : IFoodType
    {
        public String Name
        {
            get
            {
                return "Popcorn";
            }
        }
```


(c) Copyright Nokia 2014. All rights reserved.

```
        public void Cook()
        {
            MessageBox.Show("Pop! Pop! Pop!");
        }
    }
}
```

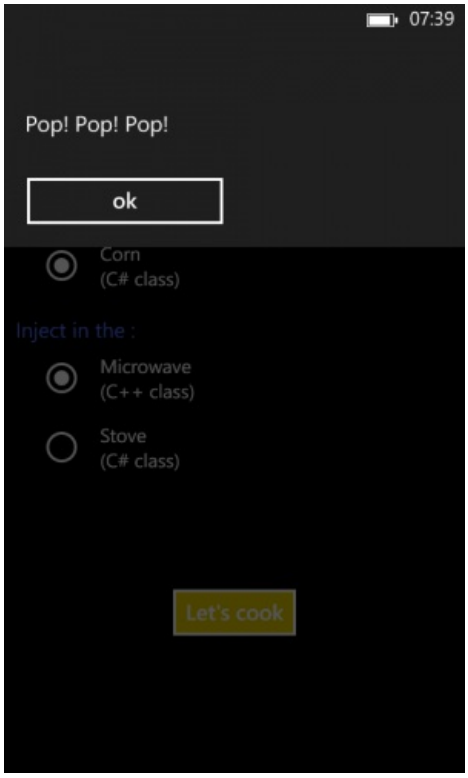
Using the classes

Now that you provided one or several concrete implementation to the interface, you can use the classes completely without knowing if they were implemented by a native or a managed class.

```
IFoodType food= null;
if (ManagedFood.IsChecked.GetValueOrDefault())
     {
         food = new Popcorn();
     }
     else
      {
          food = new RamenNoodles();
       }
 // Do something with the food object...
```

Calling .NET classes from C++

In addition to allow nice and clean architecture, using the interface as described in this article enables one cool trick. It makes it possible to call .NET classes from your C++ code. If in the sample application you selected `Corn` (C#) and `Microwave` (C++), you might have noticed that the popcorn pops:



The .NET message box "Pop! Pop! Pop!" is displayed from within a C++ class (Microwave).

In order to accomplish this feat, the managed code injects one concrete implementation of a WinPRT interface into a native class. In this case, the `Popcorn` class (C#) is injected into the `Microwave` class (C++). Once the object on the native side has a grip on the interface, it can call any of the interface methods. And if the the concrete implementation of the interface has been implemented in managed side, those methods may contain calls to any .NET classes.

Although it is not mandatory, the injection is typically done at construction time. Here the microwave class accept `IFoodType`. Look

(c) Copyright Nokia 2014. All rights reserved.

for the call to `m_food->cook()` which executes .NET code when the concrete implementation of `IFoodType` was implemented in C#.

```
Microwave::Microwave(IFoodType^ food)
{
 m_food = food;
}

Platform::String^ Microwave::StartTheOven()
{
 m_food->Cook();
 auto message = "Your " + m_food->Name + " is ready !";
 return message;
}
```

The source code

The source code for the sample application : File:InterfacesInWinPRT.zip

(c) Copyright Nokia 2014. All rights reserved.