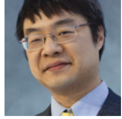
On static methods in the Windows Runtime and C++/WinRT

devblogs.microsoft.com/oldnewthing/20210524-00

May 24, 2021



Raymond Chen

The Windows Runtime supports static members, which are members that apply to a class as a whole, rather than to particular instances of a class. The expectation is that these members are exposed via the language projection as static members of some language-specific representation of the class.

Under the covers, though, there are no static members. That's because static members have to belong to a class, but the Windows Runtime uses COM as its low-level interface, and all COM members belong to COM interfaces, and COM interfaces are implemented by *objects*.

For illustration purposes, say we have this Windows Runtime class:

```
runtimeclass Widget
{
    Widget(); // default constructor
    Widget(String name); // nondefault constructor

    void InstanceMethod();

    static void StaticMethod();
}
```

Accessing the `InstanceMethod()` from a `Widget` object is easy: The object implements `IWidget`, and you call the `IWidget::InstanceMethod()` method on it.

But it's less obvious how you get to the constructors and static methods. Because those start from nothing; there is no object in your hand from which to call the methods.

The solution is to fabricate another object, known as the *activation factory*. This object contains all the operations that are not dependent upon an existing instance. You can think of this object as representing *the class itself*.

from nothing

↓RoGetActivationFactory(“Widget”)
Widget factory
`IActivationFactory`
`ActivateInstance()`
`IWidgetFactory`
`CreateInstance(name)`
`IWidgetStatics`
`StaticMethod()`

Every activation factory implements `IActivationFactory` at a minimum. This interface provides the default constructor, known as `IActivationFactory::ActivateInstance()`. Even if an object doesn't have a default constructor, the `IActivationFactory` interface will still be there; its `ActivateInstance` method will just return `E_NOTIMPL`.

If a class has nondefault constructors, they exist on a separate `IWidgetFactory` method. By convention, these nondefault constructor methods are named `CreateInstance` or some variation thereof.

And if a class has static members, then they exist on an `IWidgetStatics` method.

For example, under the covers, calling a static method works like this:

```
IWidgetStatics* statics;
RoGetActivationFactory(L"Widget", IID_PPV_ARGS(&statics));
statics->StaticMethod();
```

Okay, so how does C++/WinRT represent static members?

At the projection level, they look like static class members.

```
winrt::Widget::StaticMethod();
```

C++/WinRT does the under-the-covers thing we described above, though with the bonus feature of caching the activation factory object for better performance.

At the implementation level, what happens depends on what version of C++/WinRT you're using.

In C++/WinRT version 1 (or C++/WinRT without the `/optimize` option), the implementation mirrors the under-the-covers behavior:

```
namespace winrt::factory_implementation::Widget
{
    struct Widget : WidgetT<Widget>
    {
        // instance method on factory object
        void SomeMethod() { ... }
    }
}
```

C++/WinRT autogenerates the `IActivationFactory` and `IWidgetFactory` by having the corresponding methods construct an instance via a corresponding public constructor of the `implementation::Widget` type. But the static members are up to you to implement, and they are members of the factory object.

When the Widget object is consumed by the projection, it goes through the factory:

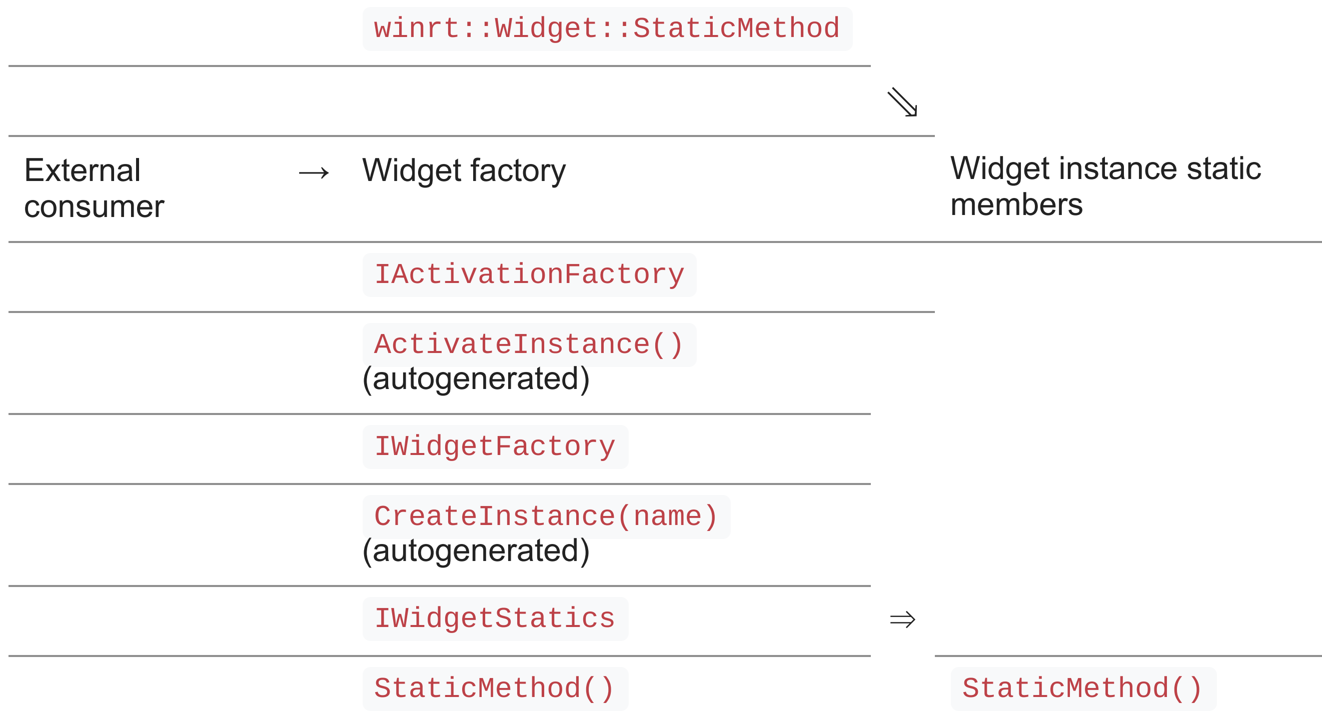
(I put the static method call in a dotted box to emphasize that there is no object involved here. It's a free function.)

In the case where your static methods are stateless, this creates an inefficiency in the projection when used from within the same module: They still go through the formality of obtaining a factory and calling the nominally static method as a member method of the `IWidgetStatics` interface. But if the static method is stateless, then it has no use for the factory object. We went through the effort of locating it, and then making a virtual method call on it, when we could have just gone straight to the implementation.

C++/WinRT version 2 with the `/optimize` option fixes this. Calls to static methods are forwarded to the corresponding static method on the *implementation* class, rather than to the instance method on the factory implementation class.

```
namespace winrt::implementation::Widget
{
    struct Widget : WidgetT<Widget>
    {
        // static method on instance object
        static void SomeMethod() { ... }
    }
}
```

Furthermore, the factory implementation also forwards its instance members (corresponding to Windows Runtime static members) to the static members of the implementation type.



		`winrt::Widget::StaticMethod`		
			⇘	
External consumer	→	Widget factory		Widget instance static members
		`IActivationFactory`		
		`ActivateInstance()` (autogenerated)		
		`IWidgetFactory`		
		`CreateInstance(name)` (autogenerated)		
		`IWidgetStatics`	⇒	
		`StaticMethod()`		`StaticMethod()`

Sending the projection's static method straight to the instance static method avoids the hassle of obtaining the widget factory object, which we never use anyway. It avoids the virtual call through the factory's COM interfaces, and thereby opens inlining opportunities for very simple static methods.

But what if your static methods are stateful? Well, you could just keep that state in global variables, but that's a problem if some of the state involves COM objects, because you now have a COM object in a global variable: The global variable will destruct when the DLL unloads, which is likely to be *after* COM has shut down.

This is where the COM static store comes to the rescue. You can ask C++/WinRT to put the class factory in the COM static store by adding `static_lifetime` to the template parameter list:

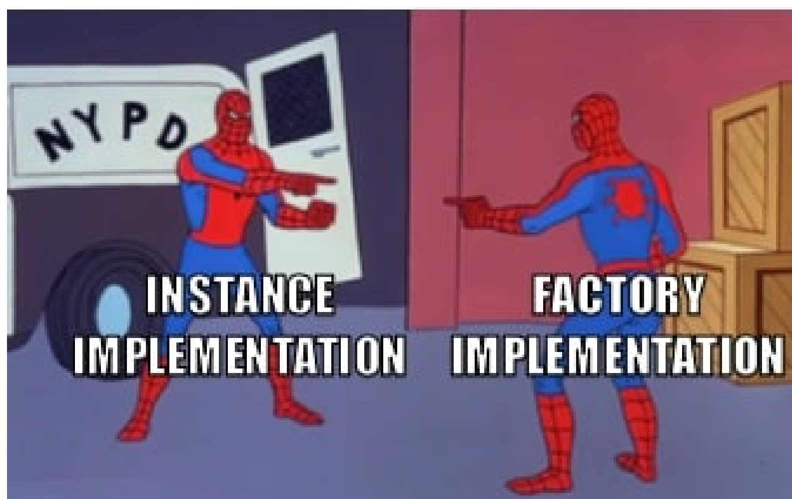
```
namespace winrt::factory_implementation::Widget
{
    struct Widget : WidgetT<Widget, static_lifetime>
    {
        ...
    }
}
```

Now you can put your state in the factory object, and it will be destructed when COM tears down.

But how do you access the factory object from the static method in the instance object? You'll just have to get it manually.

```
namespace winrt::implementation::Widget
{
    struct Widget : WidgetT<Widget>
    {
        // static method on instance object
        // forward to factory object
        static void SomeMethod() {
            get_activation_factory<winrt::Widget, IWidgetStatics>()
                ->StaticMethod();
        }
    }
}
```

At this point, you start to get a little dizzy because there's this game of "Where's the static method?" being played. The instance implementation is forwarding to the factory implementation, but the factory implementation is forwarding to the instance implementation:

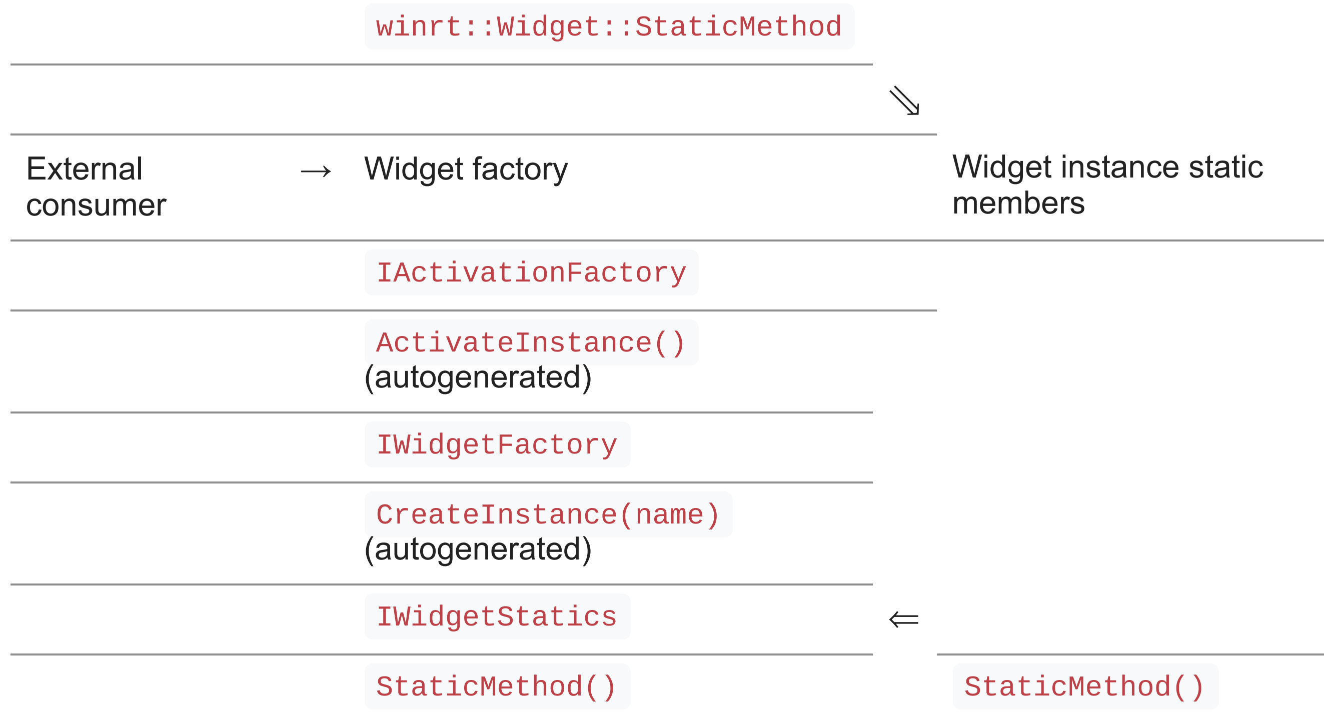


You break the infinite loop by implementing the method in the factory implementation, just like you did back in C++/WinRT version 1:

```
namespace winrt::factory_implementation::Widget
{
    struct Widget : WidgetT<Widget>
    {
        // instance method on factory object
        void SomeMethod() { ... }
    }

    // state variables go here
    int32_t m_state;
}
```

An explicit implementation in the factory implementation object overrides the default implementation, thereby breaking the cycle.



Bonus chatter: Even if you never call the static method yourself, you still have to include a declaration for it, so that the projection short-circuit (the diagonal arrow) can call it. You don't have to implement it, though.

Raymond Chen

Follow