Why does my overridable Windows Runtime method turn into a protected method, and how can I work around it?

devblogs.microsoft.com/oldnewthing/20220509-00

May 9, 2022

Raymond Chen

If you declare a Windows Runtime class with a method marked `overridable`, the method is automatically made `protected`, even if you wanted it to be public.

```
unsealed runtimeclass Base
{
    overridable void DoSomething();
}
```

If you look at the generated metadata in ILSpy, you'll see that it was secretly marked as `protected`:

```
public class Base : IBase, IBaseOverrides
{
    [MethodImpl(MethodImplOptions.InternalCall)]
    protected virtual extern string DoSomething();
}
```

What's going on?

In the Windows Runtime type system, overridable methods may only be called from within the class itself. so they are automatically marked as `protected`. The reason for this is lost to the mists of time, but I have some theories.

One contributing factor is probably that the way you call an overridable method is different from how you call a regular method. Overridable methods are implement via COM aggregation, so calling an overridable method requires querying the controlling unknown rather than the object in hand, so that the controlling unknown can choose to override the interface. This extra dance isn't something we want to bother all of the clients with, since they don't even know or typically care about how the class internally organizes its methods.

Another factor is that different languages provide different levels of support for accessing overridden methods. For example, in C++, you can explicitly qualify the method with the base class to bypass the derived class:

```
struct Base
{
    virtual void Method();
};

struct Derived
{
    void Method() override;
}

void f(Derived& d)
{
    // This bypasses Derived::Method and calls
    // Base::Method directly.
    d.Base::Method();
}
```

In JavaScript, you can bypass the most derived class by walking the prototype chain.

```
class Base {
    constructor() { }
    method() { console.log("Base!"); }
}

class Derived extends Base {
    constructor() { super(); }
    method() { console.log("Derived!"); }
}

var d = new Derived();
// This bypasses Derived.method and calls
// Base.method directly.
d.__proto__.__proto__.method.call(d);
```

On the other hand, C# doesn't have the ability to bypass the derived class from outside the class. (From inside the class, you can use the `base` keyword to access the overridden method.)

To avoid introducing concepts that cannot be expressed in some languages, the Windows Runtime just removes the problem. Overridable methods are always protected.

So what do you do if you want to let people call an overridable method?

The answer is to provide two methods, one overridable and one not. People outside the class call the non-overridable method, and the base class implementation calls back out to the overridable one.

```
unsealed runtimeclass Base
{
    void DoSomething();
    overridable void DoSomethingOverride();
}

void Base::DoSomething()
{
    // This calls the most derived class
    this->DoSomethingOverride();
}
```

Raymond Chen

Follow