Windows Runtime objects are represented by their default interface, so choose your default interface wisely

devblogs.microsoft.com/oldnewthing/20201028-00

October 28, 2020

Raymond Chen

As I noted some time ago, in the Windows Runtime, objects are represented at the ABI by a pointer to their default interface. The choice of default interface is usually obvious, but on occasion, the non-obvious choice may be better.

If your runtime object supports only one interface, then you have no choice but to make that interface the default interface. But if your runtime object supports multiple interfaces, then you have a choice. For example:

```
runtimeclass AudioTrack : IMediaTrack
{
    event Windows.Foundation.TypedEventHandler<AudioTrack, AudioTrackOpenedEventArgs>
OpenFailed;

    AudioEncodingProperties GetEncodingProperties();
    MediaPlaybackItem PlaybackItem { get; };
    String Name { get; };
    AudioTrackSupportInfo SupportInfo { get; };
};
```

As written, the MIDL compiler does the following:

- Autogenerates an interface called `IAudioTrack` to contain the members declared in the class definition.
- Defines the `AudioTrack` class as implementing the `IAudioTrack` and `IMediaTrack` interfaces.
- Marks the `IAudioTrack` class as the default interface.

As noted above, the default interface is used to represent the object. Method calls on the default interface will be faster than method calls on non-default interfaces, because methods on non-default interfaces require a `QueryInterface` to obtain the interface. Therefore, you should choose your default interface to be one that holds the methods that you anticipate will be used the most. If that interface is not the autogenerated interface, you can specify a custom default interface by putting the word `[default]` in front of the interface you want to be the default interface:

```
runtimeclass AudioTrack : [default] IMediaTrack
{
    event Windows.Foundation.TypedEventHandler<AudioTrack, AudioTrackOpenedEventArgs>
OpenFailed;

    AudioEncodingProperties GetEncodingProperties();
    MediaPlaybackItem PlaybackItem { get; };
    String Name { get; };
    AudioTrackSupportInfo SupportInfo { get; };
};
```

In this case, the object will be used primarily as a media track, and it is the methods on `IMediaTrack` that will get the most exercise. Registering for the `OpenFailed` event will probably happen only once, and the support info might never be used at all. It would be preferable to make the `IMediaTrack` the default interface, so that the commonly-used methods are readily available.

Another scenario where you may want to override the MIDL compiler's choice of default interface is if your class implements a collection, possibly with an extra method or two. The object will almost certainly be used as a collection, so you should choose the collection as your default interface:

```
runtimeclass PlayerCollection : [default] IVector<Page>,
{
  void MoveToIndex(Player player, Int32 newIndex);
}
```

This hypothetical `PlayerCollection` class implements `IVector`, so you can do all the normal vector things with it. But `IVector` doesn't support reordering items. That's why the `PlayCollection` has a bonus `MoveToIndex` method that lets you take an item in the collection and move it to another position. The class may offer this so that it can provide a more suitable animation: You could get the same effect by removing the player and then reinserting it at the desired new index. However, that would result in a delete animation followed by an insertion animation, rather than a reordering animation.

Raymond Chen

Follow