Windows Runtime delegates and object lifetime in C# and other GC languages

devblogs.microsoft.com/oldnewthing/20190522-00

May 22, 2019

Raymond Chen

In C# and other GC languages such as JavaScript, delegates (most typically used as event handlers) capture strong references to objects in their closures. This means that you can create reference cycles that are beyond the ability of the GC to collect.

```
using Windows.Devices.Enumeration;

class Circular
{
  DeviceWatcher watcher;

  public Circular()
  {
    watcher = DeviceInformation.CreateWatcher();
    watcher.Added += OnDeviceAdded;
  }

  void OnDeviceAdded(DeviceWatcher sender, DeviceInformation info)
  {
    ...
  }
}
```

The `Circular` class contains a reference to a `DeviceWatcher`, which in turn contains a reference (via the delegate) back to the `Circular`. This circular reference will never be collected because one of the participants is a `DeviceWatcher`, which is beyond the knowledge of the garbage collector.

From the garbage collector's point of view, the system looks like this:

? → delegate → Circular → DeviceWatcher

The garbage collector has full knowledge of the green boxes “delegate” and “Circular” because they are CLR objects. The garbage collector does not know about the dotted-line boxes because they are external objects beyond the scope of the CLR.

What the garbage collector knows is that there is an outstanding reference to the delegate from some unknown external source, and it knows that that delegate has a reference to the `Circular` object, and it knows that the `Circular` object has a reference to some external object that goes by the name of `DeviceWatcher`. but it has no insight into what the `DeviceWatcher` object may have references to, because the `DeviceWatcher` is not a CLR object. It has no idea that the `DeviceWatcher` was in fact the question mark the whole time.[1]

To avoid a memory leak, you will have to break this circular reference. Ideally, there is some natural place to do this cleanup. For example, if you are a `Page`, you can clean up in your `OnNavigatedFrom` method, or in response to the `Unloaded` event. Less ideally, you could add a cleanup method, possibly codified in the `IDisposable` pattern.

There is a special case: The XAML framework has a secret deal with the CLR, whereby XAML shares more detailed information about the references it holds. This information makes it possible for the CLR to break certain categories of circular references that are commonly-encounted in XAML code. For example, this circular reference can be detected by the CLR with the assistance of information provided by the XAML framework:

```
<!-- XAML -->
<Page x:Name="AwesomePage" ...>
  ...
  <Button x:Name="SomeNamedButton" ... >
  ...
</Page>

// C# code-behind
partial class AwesomePage : Page
{
  AwesomePage()
  {
    InitializeComponent();
    SomeNamedButton.Click += SomeNamedButton_Click;
  }

  void SomeNamedButton_Click(object sender, RoutedEventArgs e)
  {
    ...
  }
}
```

There is a circular reference here between the `AwesomePage` and the `SomeNamedButton`, but the extra information provided by the XAML framework gives the CLR enough information to recognize the cycle and collect it when it becomes garbage.

[1] "It was the question mark all along" sounds like the spoiler to a bad M. Night Shyamalan movie.

Raymond Chen

Follow