
Windows Presentation Foundation (WPF)

Event handlers

Event handlers are a common source of memory leaks in WPF applications and, if not handled properly, they can cause problems in WinForms as well. The problem starts when a method from one class is wired up as the event handler for an event from another class. Let's call these the “listener” and the “source;” the problem happens when the source has a longer lifetime than the listener because, as long as the source is alive and has a reference to the listener, the listener cannot be garbage collected.

This will often happen with related windows such as a master/detail or a parent/child window. It can be a subtle problem with a big impact, and it's easy to overlook until you start having memory issues. Once you start seeing the problem, chances are you will have a substantial amount of cleanup work to catch up on. Your best bet is not to worry about when it can lead to problems, and instead always code defensively to avoid the problem in the first place.

A common UI design uses a master/detail relationship. You may have a grid with a list of master records, and when you click on a record from the master grid, a new window opens with the details associated with that master record. If the master window wires up events in the detail window, then the detail window cannot be garbage collected until the event handlers in the master window release their references to the detail window. In this case, the detail window will be the listener and the master window is the source. In most cases, the master window will outlive the details window, but if the event handlers are not handled properly, then the detail window cannot be garbage collected as long as the master window is alive.

```
Detail.SomeEvent += new EventHandler(Master.SomeEvent_Handler);
```

Listing 5.12: Registering an event handler.

In this example, `Detail` will not be collected as long as `Master` is still wired as an Event Handler. The easiest way to avoid this problem is to explicitly remove the event handlers when you are through with the object.

```
Detail.SomeEvent -= new EventHandler(Master.SomeEvent_Handler);
```

Listing 5.13: Removing an event handler.

Removing the event handlers is easy enough. *Remembering* to do this is the problem.

Another potential pitfall is wiring up events to methods defined in a static class. The problem is that the static class will always be in memory, so the class with the events wired up to the static method will not be garbage collected until the reference is explicitly removed. Again, the fix is to explicitly remove the event handlers.

In general, be wary of static event handlers; even though there would only ever be a single instance of one, it is *always* reachable, as is any class that it is wired up to.

With some of the new language constructs in C# 3.0, a common trend has been to replace the event handler with a lambda expression. Depending on your sense of style, this makes the code easier to read and keeps your class from being cluttered with simple one-line methods that are only used for event handling. Some people may mistakenly believe that this provides a solution to unwiring an event handler, since only a lambda expression is used. However, consider the example in Listing 5.14.

```
public Form()
{
   InitializeComponent();
   button.Click += (sender, e) => MessageBox.Show(“Under the Hood”);
}
```

Listing 5.14: Event handler implemented in a lambda expression.

Looking at this code in .NET Reflector without any optimizations, we see that the compiler generates a delegate to be wired up as the event handler (Listing 5.15).

```
[CompilerGenerated]
private static void <.ctor>b__0(object sender, EventArgs e)
{
   MessageBox.Show("Under the Hood");
}
public Form1()
{
   this.components = null;
   this.InitializeComponent();
   this.button.Click +=
     ((CS$<>9__CachedAnonymousMethodDelegate1 != null)
     ? CS$<>9__CachedAnonymousMethodDelegate1
     : (CS$<>9__CachedAnonymousMethodDelegate1 =
     new EventHandler(Form1.<.ctor>b__0)));
}
[CompilerGenerated]
private static EventHandler CS$<>9__CachedAnonymousMethodDelegate1;
```

Listing 5.15: What the compiler produces to deal with a lambda expression as an event handler.

This approach will actually backfire. Not only do you still get the delegate but, because it is anonymous and compiler generated, you don't have a reference to unwire it, which is still a necessary process. Therefore, if you want to use lambda expressions, make sure that you save a reference to the lambda first, so that you can explicitly remove the event handler as needed!

```
private readonly EventHandler _buttonClickHandler;
public Form1()
{
   InitializeComponent();
   _buttonClickHandler = (sender, e) =>
     MessageBox.Show("under the hood");
   button.Click += _buttonClickHandler;
}
```

Listing 5.16: Saving the lambda expression so that we can later unregister it.

Now you have a handy reference to the lambda that can be removed as needed.

```
button.Click -= _buttonClickHandler;
```

Listing 5.17: Unregistering the saved lambda expression.

Weak event pattern

As we previously discussed, it is important to always remove event handlers when they are no longer needed. Not doing so will lead to objects not being eligible for garbage collection for longer than they should be, which will then cause your application to use more memory than it should. So far, all of this should be obvious. Unfortunately, it is *not* always obvious when it is safe to remove the event handler, and this is where weak events come into play.

The weak event pattern is a new design pattern introduced with WPF, designed specifically to solve this problem. This pattern will be used primarily by control designers, particularly when users of the control need to wire up to an event but either cannot easily remove the wired-up event handler, or don't know when to do so.

Weak events work by separating the source from the listener and, in the middle, we place a `WeakEventManager`. In general, you will have a `WeakEventManager` for every "weak" event that you want to expose.

When you want to employ this pattern, you start by deriving a class from `WeakEventManager`, which requires you to complete six steps.

1. Provide a static `AddListener` method.
2. Provide a static `RemoveListener` method.
3. Override the `StartListening` method.
4. Override the `StopListening` method.
5. Implement the `DeliverEvent`.
6. Implement the `CurrentManager` property.

This might look like a lot, but it really isn't. That said, the downside is that you will generally need a `WeakEventManager` for every event that you want a weak event reference to. With *that* said, the upside is that you can have such a relationship with any class, even if it was not designed with weak events in mind.

A typical implementation can be as simple as the one shown in Listing 5.18.

```
protected override void StartListening(object source)
{
   var button = source as Button ;
   if (button != null)
   {
     button.Click += ButtonClick;
   }
}
protected override void StopListening(object source)
{
   var button = source as Button;
   if (button != null)
   {
     button.Click -= ButtonClick;
   }
}
```

Listing 5.18: Basics for implementing the `StartListening` and `StopListening` methods.

The `StartListening` and `StopListening` methods look like a generalized wiring and unwiring of the event handlers, and that's exactly all that is going on. The next thing we have to worry about is the event handler that the manager will provide (Listing 5.19).

```
void ButtonClick(object sender, EventArgs e)
{
   DeliverEvent(sender, e);
}
```

Listing 5.19: The humble `ButtonClick` method.

Most of the time, this event handler will look the same; we are simply forwarding the event to everyone who expressed an interest in it. The only real difference from one implementation to the next will be the method signature, as it *must* match the signature of the method being managed.

Now we can move on to the mechanics of allowing relevant listeners to express their interest in these events (Listing 5.20).

```
public static void AddListener(Button source,
      IWeakEventListener listener)
{
   CurrentManager.ProtectedAddListener(source, listener);
}
public static void RemoveListener(Button source,
      IWeakEventListener listener)
{
   CurrentManager.ProtectedRemoveListener(source, listener);
}
```

Listing 5.20: The basics of implementing the `AddListener` and `RemoveListener`.

You may notice that both methods are expecting the object that will be raising the events *and* an object implementing the `IWeakEventListener` interface. We will deal with this interface shortly but, for now, just know that this is a simple interface that any object listening for weak events needs to implement. These two methods will probably look very similar to this in most implementations, with only the data type of the first parameter changing.

You'll also notice that both of these methods refer to a `CurrentManager`, which is a simple static property that you are responsible for implementing. The implementation shown in Listing 5.21 ensures that only a single instance is used, and handles initializing this instance if it is not already initialized.

```
private static ButtonClickEventManager CurrentManager
{
   get
   {
     var managerType = typeof(ButtonClickEventManager);
     var manager = GetCurrentManager(managerType)
       as ButtonClickEventManager;
     if (manager == null)
     {
       manager = new ButtonClickEventManager();
       SetCurrentManager(managerType, manager);
     }
     return manager;
   }
}
```

Listing 5.21: Manager's implementation of the `CurrentManager` method.

With this plumbing out of the way, we are ready to turn our attention to the `IWeakListener` interface that we saw earlier.

Any object that wants to receive weak events needs to implement the `IWeakListener` interface and then be passed to the `WeakEventManager`'s `AddListener` method. The interface has a very simple definition; it just requires a

single method `ReceiveWeakEvent`.

A common implementation may look similar to the one in Listing 5.22.

```
public Boolean ReceiveWeakEvent(Type managerType,
     Object sender, EventArgs e)
{
   if (managerType == typeof(ButtonClickEventManager))
   {
     OnButtonClick(sender, e );
     return true;
   }
   else
   {
     return false;
   }
}
```

Listing 5.22: A simple implementation of the `ReceiveWeakEvent` method.

The type parameter will tell you which event manager delivered the event, which will tell you which event was raised. Return `true` if you were able to respond to the event, or `false` if you were not expecting the event.

If multiple buttons are being listened to, you can use the sender parameter to tell you which one raised the event. Alternatively, if multiple events are being monitored, you can use the `managerType` parameter to identify which event was raised. Even if you are interested in multiple events from multiple objects, you need only a single `ReceiveWeakEvent` method.

This has been a fairly lengthy discussion (in relation to the rest of this chapter), but hopefully you can see that, once properly laid out, the weak event pattern is fairly straightforward to implement, and does effectively eliminate the problem of the listener staying alive longer than expected.

Command bindings

Command bindings were introduced with Silverlight 4. They create strong references and may have the same issues as event handlers.

A command is a piece of logic or, more precisely, an object implementing the `ICommand` interface. This interface requires two methods and an event handler. The methods are `CanExecute` and `Execute`, and the event is `CanExecuteChanged`. The `CommandBinding` is the object that maps the command logic to the command, and it also hosts the event handlers. The `Command` object is responsible for determining whether or not a command is valid.

Consider the example in Listing 5.23.

```
private void commandButton_Click(object sender, RoutedEventArgs e)
{
   var child = new ChildWindow();
   var command = new RoutedCommand();
   command.InputGestures.Add(new KeyGesture(Key.Escape));
   child.CommandBindings.Add(
      new CommandBinding(command, CloseWindow));
   child.Show();
}
private void CloseWindow(object sender, ExecutedRoutedEventArgs e)
```

```
{
    var window = sender as Window;
    if (window != null)
    {
      window.Close();
    }
}
```

Listing 5.23: Properly releasing an event handler after it is no longer needed.

In this example, a reference will be held in each child window, and the main window cannot be collected until all child windows with this command are ready for collection or have removed the command. Just as with event handlers, you need to explicitly remove the command.

A good practice is to clear all of the binding information when you are through with the child. From within the context of the child window, you can easily clear this binding information (Listing 5.24).

```
BindingOperations.ClearAllBindings(this);
```

Listing 5.24: Clearing all bindings including the command bindings.

This simple strategy will ensure that command bindings do not prevent your objects from being eligible for collection as soon as possible.

Data binding

The data binding system in WPF was built so that it is easy to bind to any kind of object, convert values to an appropriate representation, and update the binding in case the object is changed. It does all this while trying to ensure that references are not maintained, or are at least weak, so that garbage collection can be performed at the appropriate time. There is a great deal of work done behind the scenes to simplify data binding and make type conversions straightforward.

WPF also sets the lofty goal of minimizing the likelihood of data binding creating memory leaks. Among other things, WPF wants to make it simple and straightforward to bind data to the UI.

```
control.ItemsSource = collection;
```

Listing 5.25: Simple data binding.

If the control in question is an `ItemControl` and the collection implements `INotifyCollectionChanged`, then the item control will automatically wire up to the collection's `CollectionChanged` event. This is both powerful and dangerous, as "easy" does not always also mean "sensible."

For all that WPF takes a lot of complexity away from tired eyes, I hope you can see why it's worth being aware that that complexity still exists, and still needs to be factored into your code.