

Re: asynchronous delegates and events question

Source:

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.csharp/2004-12/2112.html>

From: Philip Coveney (*kokopeli_at_sbcglobal.net*)

Date: 12/09/04

Date: Thu, 09 Dec 2004 00:28:06 GMT

Natalia,

>>*because essentially my worker thread updates the UI, which is BAD.*

Agreed. I have an application that collects data in background threads, which the UI is to display. FYI, I can tell you that when I updated the UI directly from the background thread, I experienced frequent crashes. Here's how I solved that.

I'm going to show you an example routine in one of my windows, which logs a message into a RichTextBox. I expect you will be able to generalize it easily to whatever the specifics of your UI.

1. First, I defined an interface with a single method, which logs the message. This is not strictly necessary, but removed dependency between the objects that wish to update the UI and the thing with the actual UI.

```
public interface IStatusLogger
{
    void LogWrite (string aMessage);
}
```

My objects that wish to log have an IStatusLogger property

```
private m_Logger = null;
public IStatusLogger Logger
{
    get {return m_Logger}; set {m_Logger = value;}
}
```

and update the UI by doing the following

```
if (m_Logger != null)
    m_Logger.LogWrite("Some message");
```

The window where I wish the log messages to appear implements this IStatusLogger interface

```
public class frmLogWindow: System.Windows.Forms.Form,
IStatusLogger
    (more on the implementation in a moment)
```

Finally, some high-level object (the main window in my case, but there may be a better choice for you) sets this property on the objects that wish to log

```
DataCollector1.Logger = LogWindow;  
SomeOtherObject.Logger = LogWindow;
```

Now all the objects that wish to log have reference to a method they can call to cause logging to happen, without being dependent on the specific implementation. Furthermore, as long I provide a thread-safe implementation of LogWrite, these objects can call it without worrying about whether they're in the foreground thread or a background.thread.

2. Now for the implementation of the part that updates the UI, LogWrite.

We start with a delegate definition

```
private delegate void UpdateLogDelegate (string aMessage);
```

Then a method that does the UI updating that's unsafe to do from a background thread which matches the delegate signature. In my case, the UI element being updated is a RichTextBox named rtbLog.

```
private void UpdateTheLog (string a Message)  
{  
    if (rtbLog != null)  
    {  
        rtbLog.AppendText(aMessage);  
  
        // force the log to scroll to show the nex text  
        rtbLog.Select(rtbLog.Length-1, 0);  
        rtbLog.ScrollToCaret();  
        ActiveControl = rtbLog;  
    }  
}
```

Finally, the thread-safe implementation of LogWrite

```
public void LogWrite (string a Message)  
{  
    if (InvokeRequired)  
    {  
        if (rtbLog != null)  
        {  
            UpdateLogDelegate updateLog = new  
UpdateLogDelegate(UpdateTheLog);  
            rtbLog.BeginInvoke(updateLog, new object[] { aMessage +  
Environment.NewLine } );  
        }  
    }  
    else  
        UpdateTheLog(aMessage);  
}
```

This is a little funky. InvokeRequired is a property built into all (as

far as I know) Windows.Forms.Controls, and evaluates to true if it being called from a background thread at the time. So, that outer if{ } helps me determine if I need to do anything fancy because I'm in a background thread. If not (the else case here), I just call the method that does the UI updating, since it's safe to do from the foreground. However, if InvokeRequired is true, I instantiate a delegate, passing the name of the update–UI routine. Then I use the BeginInvoke method built into the RichTextBox (again, available to all Windows.Forms.Controls, as far as I know, including the form itself), passing the delegate I just instantiated, and an array of parameters (in this case, just the string I'm trying to log, with a CRLF appended.) The syntax for that anonymous object array in which parameters are passed is a little crazy, I know—just make sure that you pass the right parameters in the right order, because strong type–checking won't help you at compile time in this case.

As the .NET docs note, Begin Invoke...

"Executes the specified delegate asynchronously with the specified arguments, on the thread that the control's underlying handle was created on."

Variations on this little mechanism have been working reliably for me for a little more than a year now, so I consider this reliable.

Good luck,

PC

"Natalia DeBow" <natalia.debow@unisys.com> wrote in message news:cp823c\$ojp\$1@si05.rsvl.unisys.com...

> *Hi,*

>

> *I am working on a Windows–based client–server application. I am involved*

> *in*

> *the development of the remote client modules. I am using asynchronous*

> *delegates to obtain information from remote server and display this info*

> *on*

> *the UI.*

>

> *From doing some research, I know that the way my implementation works*

> *today*

> *is not thread–safe, because essentially my worker thread updates the UI,*

> *which is BAD. So, here I am trying to figure out how to make my code to*

> *be*

> *thread safe.*

>

> *Another area where I would like some input is exception handling with*

> *asynchronous delegates. As a general rule, I've read that if an exception*

> *has been raised by the worker thread, that this exception would caught*

> *when*

> *a call to EndInvoke(...) is being made. However, the complication comes*

> in
> with a possibility of having innerexception to be passed in, which I am
> not
> sure how they should be handled.
>
> Here is a brief explanation of what the code is supposed to accomplish.
> 1. Retrieve a hash table of text strings and display them for all of the
> UI
> controls the require text to be displayed to the user.
> Here I have a question with updating UI technique, exception handling
> and also thread synchronization since I need to wait until this async call
> completes, before I make the next call to Initialize UI. The next call
> depends on the hash table data structure which needs to be fully populated
> before the next call proceeds.
> 2. Initialize UI controls with certain values. (For example, a ListView
> control needs to be populated with values obtained from the remote
> server.)
> Here gain, I have a question with thread safe UI updating technique and
> exception handling.
> 3. When the user requests to perform some action, say by clicking on a
> button on the UI, perform this action via an asynchronous delegate call
> and
> display the results of this action on the UI.
> Here I am using asynchronous delegates and events. When an event is
> triggered on the server side, the client should be able to handle this
> event
> and display the result on the UI. The action is triggered by the user
> initiating an asynchronous call, however, I am not sure where to call
> EndInvoke() and how to do exception handling here.
>
> Thanks so much for your assistance.
> Natalia
>
> ***** Interface shared b/w client and remote server *****
>
> public interface ISomeInterface
> {
> //...
> Hashtable GetText(int[] textIds);
>
> void InitializeComponent(out ComponentInfo[] ComponentInfoArray);
>
> void DoSomeAction(Action action, params object[] parameters);
>
> //...
>
> event UpdateEventHandler SomeItemCompleted;
>
> }
>
> ***** Some client side helper implementation *****

```
> public class SomeHelperSample
>
> {
>
> //...
>
> public delegate Hashtable GetTextDelegate(int[] textIds);
>
> public delegate void InitializeComponentDelegate(out ComponentInfo[]
> components);
>
> public delegate void SomeActionDelegate(Action action, params object[]
> parameters);
>
> //...
>
> // ISomeInterface someInterface = (ISomeInterface)
> Activator.GetObject(...);
>
> public GetTextDelegate getText = new
> GetTextDelegate(someInteface.GetText);
>
> public InitializeComponentDelegate initializeComponent = new
> InitializeComponentDelegate(someInterface.InitializeComponent);
>
> public SomeActionDelegate doSomeAction = new
> SomeActionDelegate(someInteface.DoSomeAction);
>
> //...
>
> public event SomeInterface.UpdateEventHandler SomeItemCompleted
>
> {
>
> //...
>
> }
>
>
> ***** Client side implementation *****
>
> public class ClientSample : System.Windows.Forms.Form
>
> {
>
> public SomeHelperSample helper = new SomeHelperSample();
>
> // ...
>
> // obtain text to be displayed on the UI controls
```

```
>
> private void InitializeText()
>
> {
>
> int[] textIds = new int[] {0,1,2,3,4};
>
> try
>
> {
>
> // asynchronously call GetText method
>
> AsyncCallback callback = new AsyncCallback(DisplayTextCallback);
>
> helper.GetText.BeginInvoke(textIds, callback, null);
>
> }
>
> catch (Exception e)
>
> {
>
> MessageBox.Show("Error obtaining text. ", e.Message);
>
> e = e.InnerException;
>
> while(e != null)
>
> {
>
> e = e.InnerException;
>
> }
>
> }
>
> // callback function to display text on UI controls
>
> private void DisplayTextCallback(IAsyncResult result)
>
> {
>
> try
>
> {
>
> // extract the delegate from the AsyncResult
>
> GetTextDelegate getText =
>
>
```

```
> (GetTextDelegate) ((AsyncResult)result).AsyncDelegate;
>
>
> textMessages = getText.EndInvoke(result);
>
>
> // access the key in the hash table to retrieve text strings and update
> the
> UI
>
> if (textMessages.ContainsKey(0))
>
> {
>
> this.tabPage.Text = textMessages[0].ToString(); // updates UI components
> with retrieved text
>
> }
>
> //...
>
> InitializeUIControls(); /// Need to have the HashTable populated before
> this method can be called, since it requires some of the HashTable values.
>
> }
>
> catch (Exception e)
>
> {
>
> // TODO: Exception handling code.
>
> }
>
> }
>
> // Need to have the HashTable populated before this method can be called,
> since it depends on the HashTable values.
>
> private void InitializeUIControls()
>
> {
>
> try
>
> {
>
> // asynchronously call InitializeComponent method
>
> AsyncCallback callback = new AsyncCallback(DisplayInfoCallback);
>
>
```

```
> helper.InitializeComponent.BeginInvoke(out controls, callback, null);
>
> }
>
> catch (Exception e)
>
> {
>
>     MessageBox.Show("Error initializing component. ", e.Message);
>
>     // TO DO: Add a message to the messages panel.
>
> }
>
> }
>
> private void DisplayInfoCallback(IAsyncResult result)
>
> {
>
>     try
>
>     {
>
>         // extract the delegate from the AsyncResult
>
>         InitializeComponentDelegate initializeComponent =
>
>         (InitializeComponentDelegate)((AsyncResult)result).AsyncDelegate;
>
>
>         // obtain the list of user accounts
>
>         initializeComponent.EndInvoke(out controls, result);
>
>         // display user account name on the UI
>
>         foreach (ControlInfo controlInfo in controls)
>
>         {
>
>             AddInfoToDisplay(controlInfo);
>
>         }
>
>     }
>
>     catch (Exception e)
>
>     {
>
>     }
```

```
> // TODO: Exception handling code goes here.
>
> }
>
> }
>
> //...
>
>
>
> private void buttonAction_Click(object sender, System.EventArgs e)
>
> {
>
> //.....
>
> try
>
> {
>
> // subscribe to the update event
>
> helper.SomeItemCompleted += new UpdateEventHandler(OnItemCompleted);
>
> // asynchronously call DoSomeAction method
>
> helper.DoSomeAction.BeginInvoke(actionCode, componentInfoArray, null,
> null);
>
> }
>
> catch(Exception e)
>
> {
>
> MessageBox.Show("Error. ", re.Message);
>
> }
>
> }
>
> private void OnItemCompleted(object sender, UpdateEventArgs e) // custom
> UpdateEventArgs class derived from EventArgs
>
> {
>
> // updates UI directly. Where would I put the EndInvoke corresponding to
> the above BeginInvoke?
>
> }
>
```

microsoft.public.dotnet.languages.csharp: Re: asynchronous delegates and events question

> /
>
>