Articles » Desktop Development » Shell and IE programming » Shell Programming

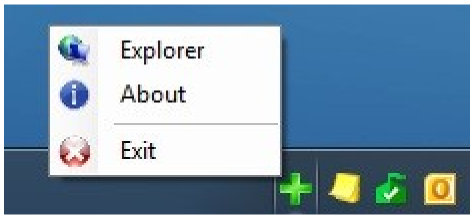
Formless System Tray Application

R. Giskard Reventlov

25 Nov 2011 CPOL

Create a formless system tray application.

Download demo/source project - 139 KB



Introduction

This article is designed to show you how to create a formless system tray application. If you were to use a form, you could attach a `NotifyIcon` and `ContextMenu` object from the toolbox but you may have found that you would need to create a formless version or may have just wondered how it could be accomplished. This article aims to illustrate how to do it.

Getting Started

In the normal course of events, you would fire up Visual Studio, create a new WinForms project, drop some controls on *Form1.cs*, compile, and run. However, you may find that you want to know how to create a system tray application in code only and with no forms (other than, for instance, an About box called from a menu).

Initial Steps

1. Create a new Windows Forms Application.

2. Delete *Form1.cs* from the project.
3. Open *Program.cs* - remove the line that reads `Application.Run(new Form1());`.
4. You would now add in the classes and objects as below - a fully working sample application is included for you to dissect.

If you want to see all of the code and run the application, download the sample - there is an executable in *bin/debug* that can be run.

However, the important bits are the following:

The Nuts and Bolts

First, we need a class to create our system tray icons. If you were starting from scratch, you would create that now: I've called mine *ProcessIcon.cs*. I also want to inherit from `IDisposable`: this gives us the opportunity to handle our own cleanup and make sure that when the program closes, it does so neatly and tidily.

To that end, we need to create a `NotifyIcon` object in our code. Since we will need to access the object in more than one place in the class, we declare it as a private member so that we can instantiate and use it later on in the code.

This is as simple as adding this in the class:

```
NotifyIcon ni;
```

We instantiate the object in the class constructor as:

```
ni = new NotifyIcon();
```

We now have a `NotifyIcon` object that we can work with.

The first reason I've done it like this is because I want to handle the `Dispose` method myself to ensure that the icon is removed immediately from the system tray when the application closes. You don't need to do this but the icon will hang around in the system tray until you pass the mouse cursor over it if you don't.

The other reason for doing this is so that we can take advantage of the `using` pattern so that you would replace the code in *Program.cs* `Main` with this:

```
Application.EnableVisualStyles();
Application.SetCompatibleTextRenderingDefault(false);

// Show the system tray icon.
using (ProcessIcon pi = new ProcessIcon())
{
    pi.Display();

    // Make sure the application runs!
    Application.Run();
}
```

This ensures that the application will start and exit gracefully: If you haven't seen the `using` pattern before, take a look here for more information: using statement (C# Reference).

As you can see, I have also used `pi.Display();`. This is the method that takes care of displaying the icon on the system tray and adding the menu from which you can make or insert appropriate options (I'll leave that to your specific requirements).

One of the most important parts of this is to provide a handler that will take care of mouse clicks on the icon. For the purpose of this demonstration, I am only looking at the left mouse button click but you can intercept the right and middle buttons as well. Note, however, that the `ContextMenuStrip` is usually called on a right mouse button click so you don't normally have any need to do anything with that one.

It is also important to note that the icon you intend to use is embedded. You do this by opening *Resources.resx* (found in the *Properties* folder of the project) and drag/drop your selected icon (having included it into the project) onto the file. It will automatically set this to an Icon resource. I have provided an icon taken from the VS 2010 icon set - feel free to substitute your own.

Now we need to add a class to handle the context menus we're going to need to give the application the desired functionality. Note that you may not need any of that and that your application has some other purpose that does not require any interaction. On the basis that this one does, let's see how to create a context menu in code and attach it to the application.

You'll notice that the following line in `ProcessIcon.Display` is used to attach the menu to the `NotifyIcon` object:

```
ni.ContextMenuStrip = new ContextMenus().Create();
```

The menus are constructed within *ContextMenus.cs*. The basic process is to create either a `ToolStripMenuItem` or a `ToolStripSeparator` and then attach that to the `ContextMenuStrip`. As you can see from the code, there are a few things that you need to add to the item to make it do anything meaningful.

It should have a display name and an event handler (to respond to the mouse click). The image is optional though I have added one to each option (and, no, I don't know where I got the images supplied in the sample). Same rules as the icon - drag and drop onto *Resources.resx* and use as:

```
item.Image = Resources.About;
```

Again, each item requires an event handler (if you need to respond to a mouse click or carry out other actions). These can be added as:

```
item.Click += new EventHandler(Explorer_Click);
```

A quick way to do this is type `item.Click +=` and hit the tab button twice. VS will create an appropriate handler for you.

In the example, I have simply added each item in turn. You can place them specifically by using:

```
menu.Items.Insert(index, item);
```

where `index` is an integer indicating the position in the list that the item should be located at.

The final piece of the puzzle is the event handler for each of the menu options. As you can see by the code, each option carries out a specific task such as exiting the application (very important option!) or displaying a form - in this case a generic About box. Note also the construct that instantiates and displays the About box. The reason for the shenanigans is that you can't bind the About box to a parent form (as there isn't one) so you could just keep clicking on the menu option and fill the screen with About boxes! Not very helpful! This code overcomes that issue.

```
if (!isAboutLoaded)
{
    isAboutLoaded = true;
    new AboutBox().ShowDialog();
    isAboutLoaded = false;
}
```

All this does is to say if the About form isn't loaded, set a variable (`boolean`) to `true`, and load the form. When it closes (and returns control to this method), unset the variable and start again.

You wouldn't need this if you were displaying external applications (like Notepad or Windows Explorer or FireFox) but should do so for any form that belongs to the application; i.e., where a form is part of the application, this functionality ensures that it can only be loaded and displayed once.

Note that you would require a boolean variable for each form or only the first one you click on will open!

The rest of the code in the sample should be fairly self explanatory - feel free to dissect, discard, or steal. I'll try and respond to any queries, etc., as I get a moment.

Conclusion

The code contains examples of opening a form (the About box), starting and displaying Windows Explorer, and exiting the application. The code used to open Windows Explorer is very simple to use and can be called to open virtually any other application. The example is very simplistic - read this article to get further insight into this very useful functionality: `Process` class.

And that's it: you now have an application that will load an icon into the system tray area and allow users to select menu options and provide a default left mouse button action.

Warning!

Note that the code, as presented, is as basic as possible for clarity. You should ensure that you provide appropriate exception handling and test, test, and test again to ensure that it does what you expect it to do.

This application has been created using Visual Studio 2010 and .NET 4.0 in Windows 7. It has not been tested in any other environment and, whilst it should work in Windows XP and Vista, it is your responsibility to test it thoroughly before use.

History

This is the first and last version of this code unless anyone provides changes that must be put in as they drastically improve - you will get thanked and cited.

License

About the Author

R.

Giskard Reventlov

United States

No Biography provided

Comments and Discussions

52 messages have been posted for this article Visit **https://www.codeproject.com/Articles/290013/Formless-System-Tray-Application** to post and view comments on this article, or click **here** to get a print view with messages.