C# How-to 11: Use the API Browse for Folders function

Introduction

The .NET framework provides a simple way to use the various common dialogs, such as the Open or Save As boxes. These are very useful if you want the user to choose a file, but not so useful if you want a folder (directory) to be chosen. There is no way unfortunately to specify that only folders should be shown in the dialog. This is a hindrance since sometimes this is exactly what is required.

The Windows API does contain a function to show a dialog box which allows the user only to choose a folder. However, this dialog is not available from the .NET framework, but only by calling the API directly. This how-to will show how this can be done and provides a complete class which can be downloaded and incorporated into a .NET program.

Steps to take

1. Declare the API calls required

We need three API functions, two from shell32.dll and one from ole32.dll. These are:

API function	Purpose
SHBrowseForFolder	Displays a dialog box allowing the user to select a folder from the local file system
SHGetPathFromIDList	Takes the return value from SHBrowseForFolder and produces a fully-qualified path name
CoTaskMemFree	Frees memory allocated by the shell during the above calls

In fact, SHBrowseForFolder can do rather more than just folder browsing, but for the purpose of this page we will ignore the other possibilities.

We also need an API structure called BROWSEINFO and we need to fill in the various elements of this before calling SHBrowseForFolder (for more details, see the Windows SDK). Before we can go any further, the API calls all require a pointer of some kind. C# doesn't do pointers unless you force it to, so we must mark the class we develop with the 'unsafe' keyword to indicate that this is unmanaged code. We will also require the 'fixed' keyword at various points to stop the garbage collector moving blocks of memory we are pointing to (which would be disastrous!).

Our class looks like this so far:

```
using System;
using System.Windows.Forms;
using System.Runtime.InteropServices;
using System.Text;

namespace Microbion.SHBrowseFolders {
unsafe public class SHBrowser
{
```

```
// declarations for API functions
[DllImport("shell32.dll")]
static extern int SHBrowseForFolder(browseInfo* br);
[DllImport("shell32.dll")]
static extern bool SHGetPathFromIDList(int pidl, byte* pszPath);
[DllImport("ole32.dll")]
static extern void CoTaskMemFree(void* pv);

// declares an API structure - element names are from the Windows SDK
private struct browseInfo {
public int hwndOwner;
public int pidlRoot;
public byte* pszDisplayName;
public byte* lpszTitle;
public uint ulFlags;
public int lpfn;
public int lparam;
public int imImage;
}

// class-level fields
private browseInfo brinfo;
private string brtitle;
```

2. The class constructor

This is simple; all we do is initialise the various elements of the browseInfo structure to zero. The only exceptions are the two pointers, which can be initialised like this:

```
brinfo.pszDisplayName = (byte*)0;
brinfo.lpszTitle = (byte*)0;
```

3. Browsing for folders

This is done through a public method, DoBrowse(). I elected to provide four overloads for this method to allow some future flexibility. The first overload is the simplest and looks like this:

```
public string DoBrowse() { // no parameters, so provide defaults
brtitle = "Choose a folder:";
return BrowseIt();
}
```

This just sets the title of the dialog to "Choose a folder:" and leaves other settings as they are. The other three overloads all allow the user to set the dialog title, set various flags which affect its behaviour, and set the owning window of the dialog. The class works perfectly well with the default settings, and the most likely variation is probably the dialog title. In this version I have ignored the flags setting but it would be relatively simple to modify the class to allow these to be set, which would increase the functionality. The overloaded methods are there if required.

The actual work is all done in the private function BrowseIt(), which is covered in the next page of this how-to.

C# How-to 11: Use the API Browse for Folders function (part 2)

In the first part of this how-to, we set up the class which will call the API functions required to browse for a folder. The remaining function to cover is where the actual work is done and the dialog displayed.

Steps to take

1. The BrowseIt() function

This function will return a string which is either a fully-qualified path name or, if the user clicked the Cancel button in the dialog or in the event of some kind of failure, an empty string (""). An empty string will also be returned if the user selected one of the shell namespaces, such as Desktop or My Computer.

The function requires several variables including three buffers which are just byte arrays. These will hold the title of the dialog, the returned folder name (but not its path, which is what we want) and the fully-qualified path. The title of the dialog we will obtain by converting the title string object into a byte array using the ASCIIEncoding class in the framework, like so:

```
ASCIIEncoding enc = new ASCIIEncoding(); // converts Unicode to 8-bit ASCII and vice-versa
byte[] dialogTitle; // buffer for the dialog title
dialogTitle = enc.GetBytes(brtitle); // convert .NET string to API buffer
```

In the browseInfo structure, we have to supply pointers to the dialog title and to a buffer to hold the folder name. Getting a pointer is easy; we can do:

```
byte* pTitle = dialogTitle;
```

Except that the compiler won't let you do this because the array dialogTitle[] is managed by the runtime and there is the possibility that the garbage collector could move it in memory after we have obtained a pointer to it. The required syntax is therefore:

```
fixed (byte* pTitle = dialogTitle) {
// do whatever we need with the pointer here
} // end the fixed block here and allow the garbage collector to move the array if required
```

Essentially then, while the pointer is still required it must be used within the fixed block. If more than one pointer is required, the fixed blocks can be nested.

Once we have pointers to the buffers, we can finish initialising the browseInfo structure. We then need to do the same thing to get a pointer to the browseInfo structure and call the API function:

```
fixed (browseInfo* brp = &brinfo) // get a pointer to the browseInfo
pIDL = SHBrowseForFolder(brp); // call the shell and return a PIDL
```

Now, SHBrowseForFolder returns another API structure called an ITEMIDLIST. Fortunately, we don't need to know anything about this structure because we just pass it to SHGetPathFromIDLIst to get the fully qualified path into a buffer (for which we have to get a pointer, naturally):

```
fixed (byte* pbuffer = pathBuffer) { // get a pointer to the buffer for the path
if (pIDL != 0) {
bSuccess = SHGetPathFromIDList(pIDL, pbuffer); // get the path name from the ID list
CoTaskMemFree((void*)pIDL); // free memory used by the shell
showPath = enc.GetString(pathBuffer); // convert to a string from the byte array
}
}
```

Assuming a valid ITEMIDLIST was returned from ShBrowseForFolder, we get the path name in the buffer, convert it to a .NET string using the ASCIIEncoding class, and free the memory allocated by the shell for the ITEMIDLIST. We also check the success of the SHGetPathFromIDList call which returns a boolean value. BrowseIt() can then return a .NET string object with the path name, or an empty string if the path name could not be obtained.

Finally, the only other thing we need to do is catch any exception which may occur so in the finished version the code is surrounded in a try block and displays an error message if an exception is caught.

That is basically it. If you want to try it out, the complete and documented class is available from the software download page. Just add it to any project where you want to browse for folders, but remember that you will need to set the /unsafe flag in the compiler options.

Go to the download page for the class file