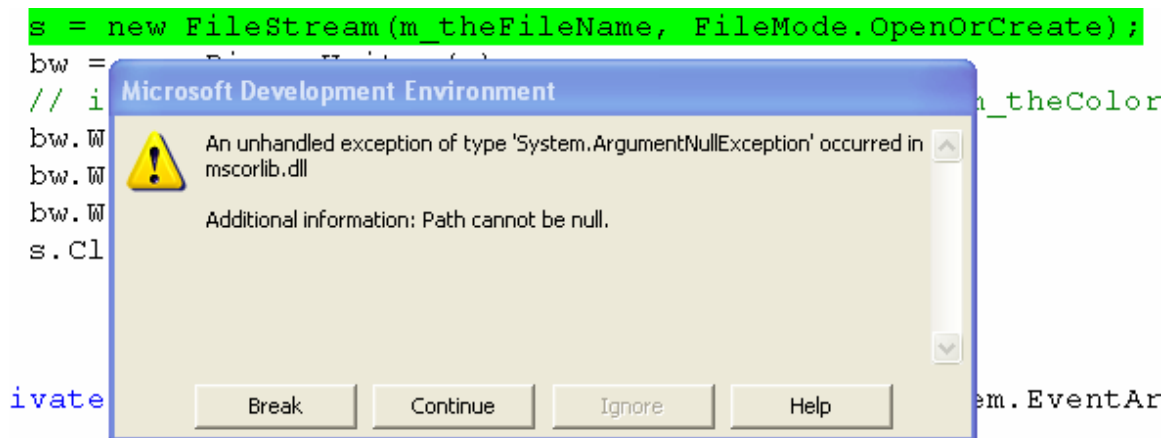


The SaveFile and OpenFileDialog Common Dialogs in Detail

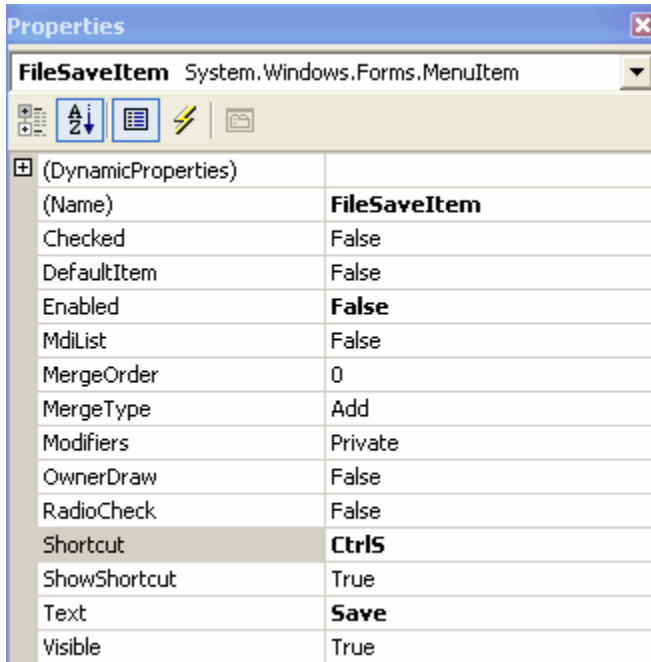
In this lecture, we will study the details of the *SaveFile* and *OpenFile* dialogs, and correct a number of flaws in the example program from the last lecture.

First: what happens if you try to use *Save* before you have used *Save As*? If we do that with the debugger on here's what happens:



Well, what did you expect? *M_theFileName* has no default value and is not initialized until *Save As* is first used.

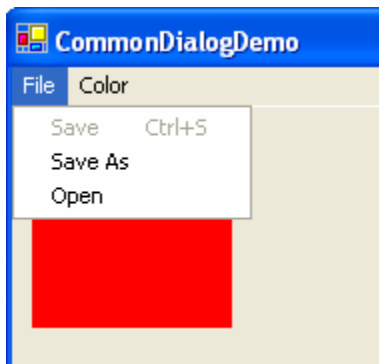
We could try to catch that error earlier, but the best thing would be if the *Save* menu item were simply disabled until *Save As* has been used successfully. We can arrange that by setting the *Save* menu item to be disabled in its property sheet, and then explicitly enabling it in the *Save As* handler. While we have the property sheet open, we might as well give the *Save* item the common shortcut key, *Control-S*.



Then enable the item in the `FileSaveItem_Click` handler as shown here:

```
bw.Write(m_theColor.B);  
s.Close();  
FileSaveItem.Enabled = true; // add this line
```

Now when you first open the menu you will see this:



After once saving the file, the *Save* item will be enabled again. OK, that's one bug fixed.

Remark: Another common solution to the problem of using *Save* before *SaveAs* is this: instead of disabling *Save*, just have *Save* do the same thing

as *SaveAs* if it is called before *SaveAs* has established a filename. Personally, I prefer to disable *Save*, but both solutions work.

File Name Extensions

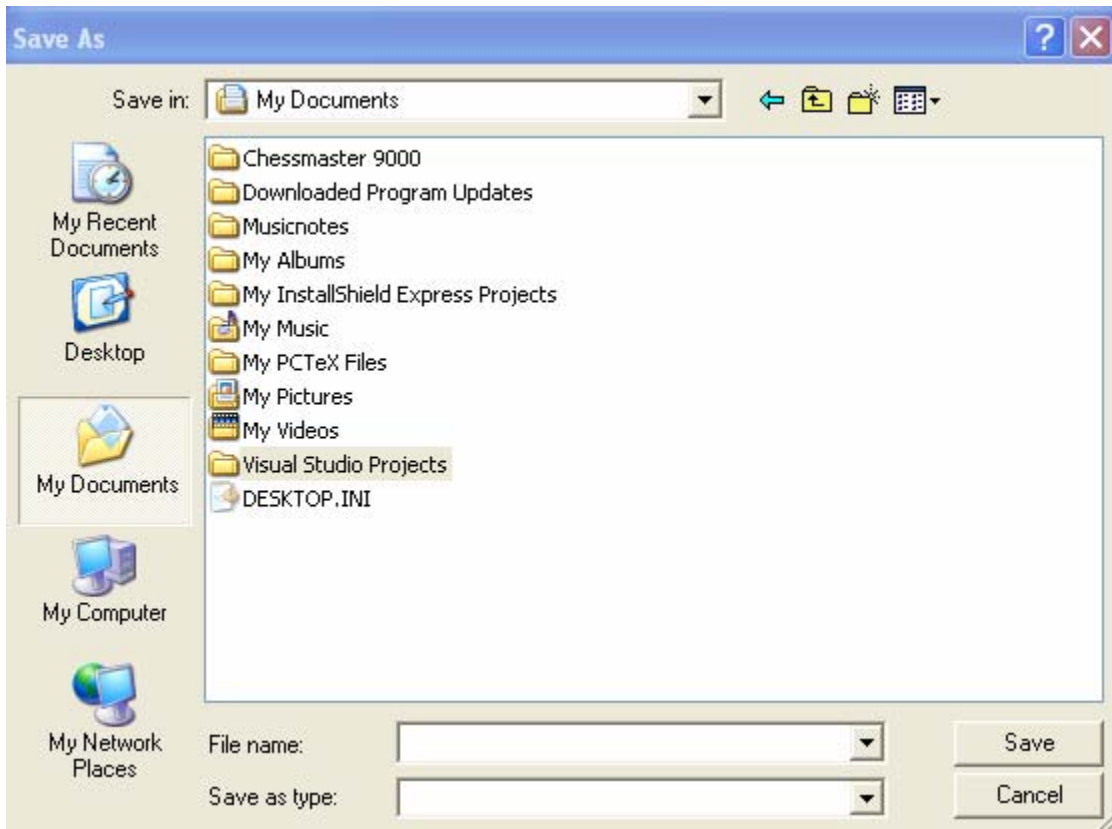
This section reviews basic computer literacy about file name extensions. That phrase refers to the conventional three characters after a period at the end of a filename. Normally each type of Windows file should have its own default extension, as in *.doc* for a Word document, *.cpp* for a C++ file, *.cs* for a C# file, *.exe* for an executable file, etc. When you write a Windows program with a *Save As* option, you should provide a default file extension, and *strongly* encourage (perhaps even force) your users to save their work in files with that extension.

File name extensions are important in Windows because that is the mechanism Windows uses to determine what program should be used to open a file when you double-click the file in Windows Explorer (or type the file name to a command prompt). Windows maintains a table of associations between file name extensions and programs. You can view and edit that table, in Windows Explorer (*Tools / Folder Options / File Types*).

Incidentally, Windows is set up by default to “hide common file types”. That means that the filename extension *.exe* will not be shown, for example. If a virus writer creates a virus in file *prettygirl.jpg.exe*, and attaches it to an email, it may appear as *prettygirl.jpg*. An unsuspecting user might think it was only a picture! You can change this default and instruct Windows to show all file name extensions, in Windows Explorer (*Tools / Folder Options / View* and uncheck *Hide Extensions for Known File Types*).

File Name Extensions and the Common Dialogs

Let’s have another look at the *SaveFile* dialog:



There are several things about this dialog that can be controlled by the programmer:

- The initial folder, shown at the top of the dialog.
- The default file name, which can be shown in the *File name* box.
- The file type. A list of possible file types and their descriptions can be shown in the *Save as type* box.
- The files that are shown in the large list box. In the screen shot shown, there is only one file *DESKTOP.ini*, but it doesn't make sense to show files of type *.ini* as candidates for *Save As* in this program.

We will pick *.clr* as the default file extension for our program. We then want to program the *SaveFileDialog* so that

- The *Save as type* combo box displays *Color Documents (*.clr)*
- Only files with extension *.clr* are displayed in the large list box

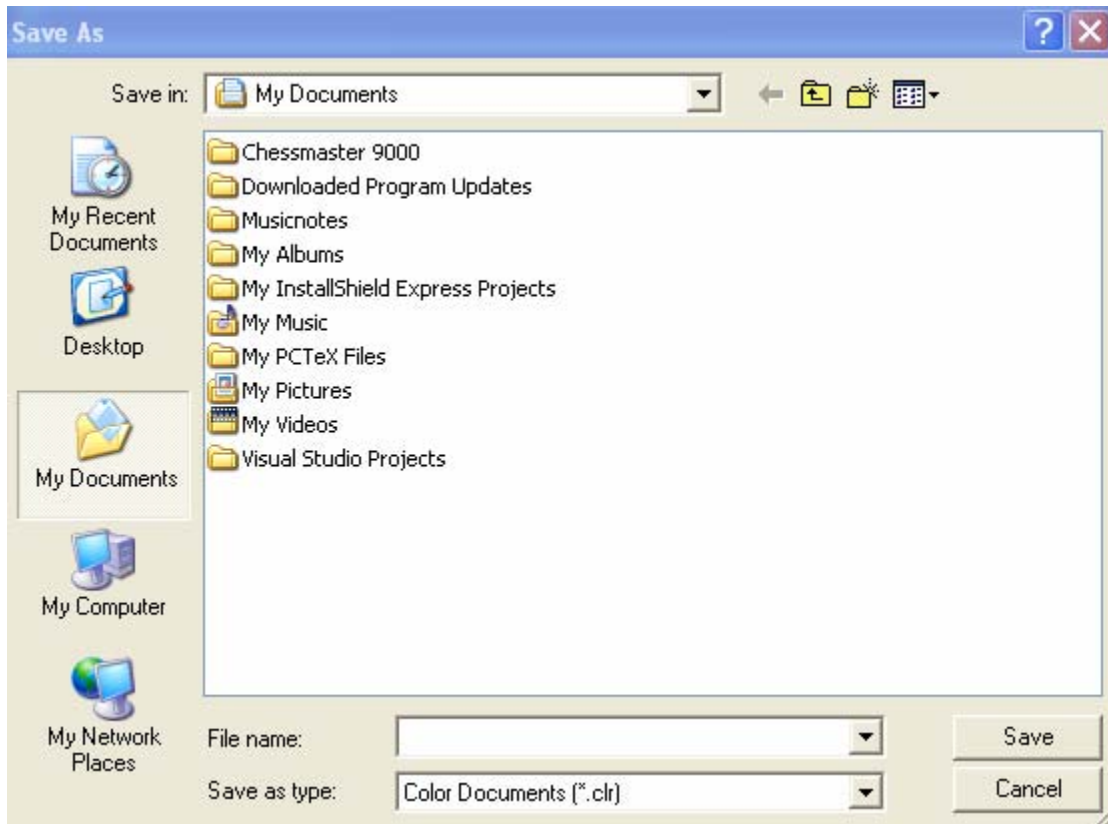
Both these aims are accomplished by setting the *Filter* member of the *SaveFileDialog* object correctly before showing the dialog. Before we discuss how to do this, you should understand the following points:

- The *Save Type* combo box can display several different strings, one on each line. For example, when saving files in Word, you can choose between **.doc* (*Word document*) and **.html*, **.htm* (*HTML files*) and several other choices, with each choice getting one line in the combo box.
- What string or strings are displayed in that combo box does not have any influence over what files are displayed in the large list box above, which are the files the user can actually select. Usually, of course, they do correspond, but that is because the programmer so specified. You *could* say one thing in the combo box and show completely different files as possible selections.

Nevertheless, for historical reasons these things are all specified in one single string, and even the FCL has not improved this matter, since all they did was wrap the classical common dialogs in a C# class. Therefore you, like so many Windows programmers before you, will have to learn how to encode all this information in a single string. In the example at hand, the desired line of code is this:

```
saveFileDialog1.Filter = "Color Documents (*.clr)|*.clr";
```

Put this line just before showing the dialog. The vertical bar in this “filter string” serves to separate the two parts. The first part specifies a line in the *Save as type* combo box. The part *after* the vertical bar specifies that only **.clr* files will be shown in the list box.



The file *DESKTOP.ini* is no longer displayed, since it doesn't end in *.clr*. There are no *.clr* files in that folder yet, so none are displayed. Now save a file, entering the name *test1*, but don't type any extension. The program now automatically appends *.clr*, so that the next time we open this dialog we'll see *test1.clr* displayed. This behavior is controlled by the *AddExtension* property of the *FileSave* dialog; its default value is true.

This behavior can sometimes confuse a user: consider the following scenario. We use Notepad to edit a *.cs* source code file. We then save the file under a new name *foo.cs*. But we forget to select *All files (*.*)* in the combo box, and we leave *Text files (*.txt)* selected. So then we get *foo.cs.txt* instead of *foo.cs*. When we go to open our file in Visual Studio, with a filter **.cs* in effect, we don't see our file and we wonder what happened to it! Therefore, you might consider setting this property to *false* when the program can deal with more than one file extension. However, that might *also* confuse a user.

You should also set the filter on the *OpenFileDialog* before showing it.

In a more complicated program, you might be able to save or open more than one kind of file. In that case, you might have more than one line in the *Save as type* combo box. But you still have to pack all the information into a single filter string. Here's how you would do that:

```
saveFileDialog1.Filter = "Color Documents (*.clr)|*.clr|" +  
    "All Files|*.*";
```

Of course the use of + to concatenate the two strings is just a device to make the code more readable, by making lines in the source file correspond to desired lines of the combo box. In reality, what happens is that the string will be split into pieces at the vertical bars and pieces 0,2,4,... will be displayed as lines in the combo box, while pieces 1,3,5,... will determine what files are actually available to be selected. This is just a small sample of the arcane programming required in the Win32 API. The FCL mostly insulates you from this sort of thing, but here the ugliness of the Win32 API hasn't been completely papered over!

Here is one final detail: If the second member of a filter pair (the part that controls the displayed files) lists more than one file type, semicolons separate them (and no spaces may occur). You use wild-cards. For example,

```
"Web pages (*.htm, *.html)|*.htm;*.html"
```

The initial folder, pathnames, and the current directory.

Next we take up the question of the initial folder. *My Documents* is the default, and you probably shouldn't change that, but many people have their own folders, organized to their taste, and after they have navigated to the desired folder, say *C:\home\cs130\examples\CommonDialogDemo*, to save a file, it would be nice if the next *OpenFile* or *SaveFile* dialog came up in that same folder. You may have experienced the frustration of having to save a series of similar files in a poorly-written program that required you to navigate from *My Documents* to the destination directory for each save.

Does our demo program have this flaw? Experiment shows that actually, the FCL *FileDialog* authors have done more than just wrap the Win32 file dialog in a C# class. Our demo program works ideally without any further effort: if we save a file in a certain folder, next time, both the *OpenFile* and the *SaveFile* dialogs come up in that folder. What is even more amazing, if we close the program and restart it, they *still* remember what folder we were last using! To do this, the *FileDialog* must be using the Windows Registry to store the path to this folder. This is not a feature that happens effortlessly in the Windows API.

Nevertheless, for some reason we might want the default directory to be something other than *MyDocuments*, or we might want to initialize to the same path at startup each time. The *SaveFileDialog* and *OpenFileDialog* classes each have a field *InitialDirectory*. All we have to do is set that field before we open the dialog.

File Errors

Traditionally, opening and saving files is a difficult programming task because there are many possible errors:

- an open drive door
- a full disk
- write protection tab on
- a bad disk
- a corrupt file
- a read-only file
- network permission problems
- a flash memory stick or removable disk has been removed

SaveFileDialog by default tests whether the file can be opened or created, then closes it again. That's the only certain way to be sure that none of these errors occur. Therefore, you don't have to check for these errors before proceeding to open the file.

However, you do have to make some decisions about how you want your file dialogs to behave. Here are some examples:

By default, if you try to open a file that does not exist, the file dialog puts up another dialog that asks you whether you really want to create a new file. You can disable that behavior by an appropriate initialization of the *OpenFileDialog*.

Similarly, if you try to save to a file that already exists, you will by default get a dialog asking if you really want to overwrite the existing file. This behavior can also be disabled.

As mentioned above, the default behavior checks for open disk doors, corrupt files, etc., by actually opening and then closing the file. It leaves it closed, so the statement at the beginning of the last lecture that the file dialog only returns a file name, and does not leave any files open, is accurate. However, there is one circumstance in which this behavior leads to trouble: there is such a thing as a "create non-modify network share", and if you must allow users to work with such network permissions, this default behavior must be disabled, or else the trial opening will create the file and then the user won't be able to modify it. This behavior occurs on the department network at San José State University, for example.

After your file dialog closes, and you open your file, assuming you have not disabled the default behavior, there should not be a problem opening the file. However, there still might be a problem *reading* the file, or even writing to it (wrong contents, or full disk, for example). In that case the *FileStream* constructor, or later, the *BinaryWriter* object will throw an exception. When you are producing a commercial-quality program, you should therefore enclose those calls in a try-catch block, and recover gracefully from the error. To keep the code simple, that has not been done in this demo.