Introduction to the Common Dialogs

The Win32 API provides five "common dialogs", which have been wrapped in suitable classes in the Foundation Class Library of .NET. You use these built-in dialog classes when you need to allow the user to

- print
- open or save a file
- select a color
- select a font
- find-and-replace in a text file

In practice, unless you are writing a word processor, you will not use the last two; this lecture will focus on the FCL classes *ColorDialog* and *FileDialog*. While these classes take care of much of the busy work that a Win32 programmer has to do to use these dialogs, there are still some options available that require a conscious choice by the programmer. You need to understand those options and how to use them.

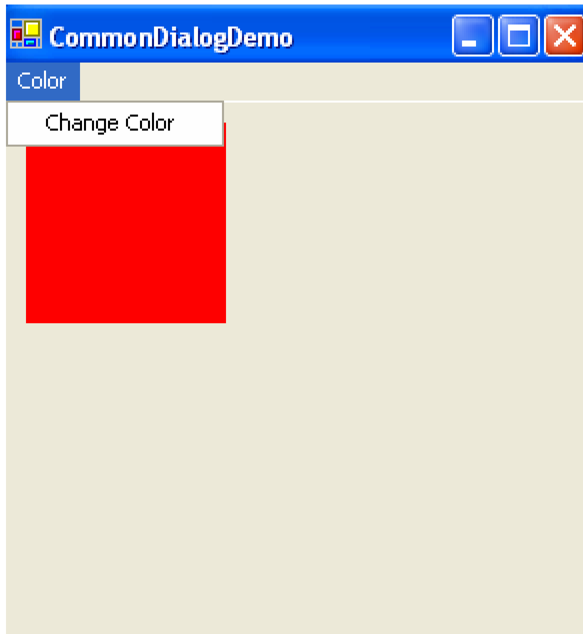
One basic principle governs the color and file dialogs: They don't actually *do anything*. That is, the color dialog doesn't change any colors; the file dialog doesn't produce an open file or save anything into a file. The color dialog allows the user to select a color; the file dialog allows the user to select a filename. That's it. What you do with the resulting color or filename is up to you. The same is almost true of the print dialog: it doesn't do any printing. However, it *does* produce a *Graphics* object that you can use to write to the printer.

Using a Color Dialog

The color dialog is simpler than the file dialog, so we'll start with it. Create a program *CommonDialogDemo*. Add member variables

```
public Color m_theColor = Color.Red;
public Rectangle m_theRect = new Rectangle(10,10,100,100);
```

In *Form1_Paint*, put code to fill *m_theRect* with *m_theColor*. Add a menu as shown in the following screen shot:



The plan is to have the *Change Color* menu item bring up a color dialog that allows the user to select a new color for the rectangle. The FCL provides a class *ColorDialog*, and we need one of those. We could either write two lines of code ourselves, or we could drop-and-drag a *ColorDialog* from near the bottom of the toolbox onto our form. Either way we wind up with two lines of code in our source file, but in slightly different places, as shown:

If you do this by drop-and-drag, you get

```
private System.Windows.Forms.ColorDialog colorDialog1;
```

in your *Form1* class and

```
this.colorDialog1 = new System.Windows.Forms.ColorDialog();
```

in *InitializeComponent*, the place where automatically written code is kept.

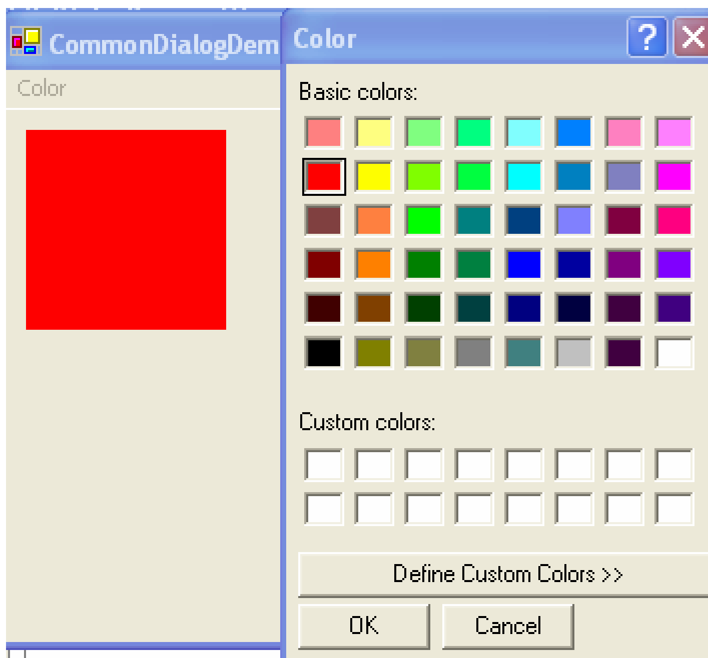
If you do it yourself, one line will do:

```
private System.Windows.Forms.ColorDialog colorDialog1 =
        new System.Windows.Forms.ColorDialog();
```

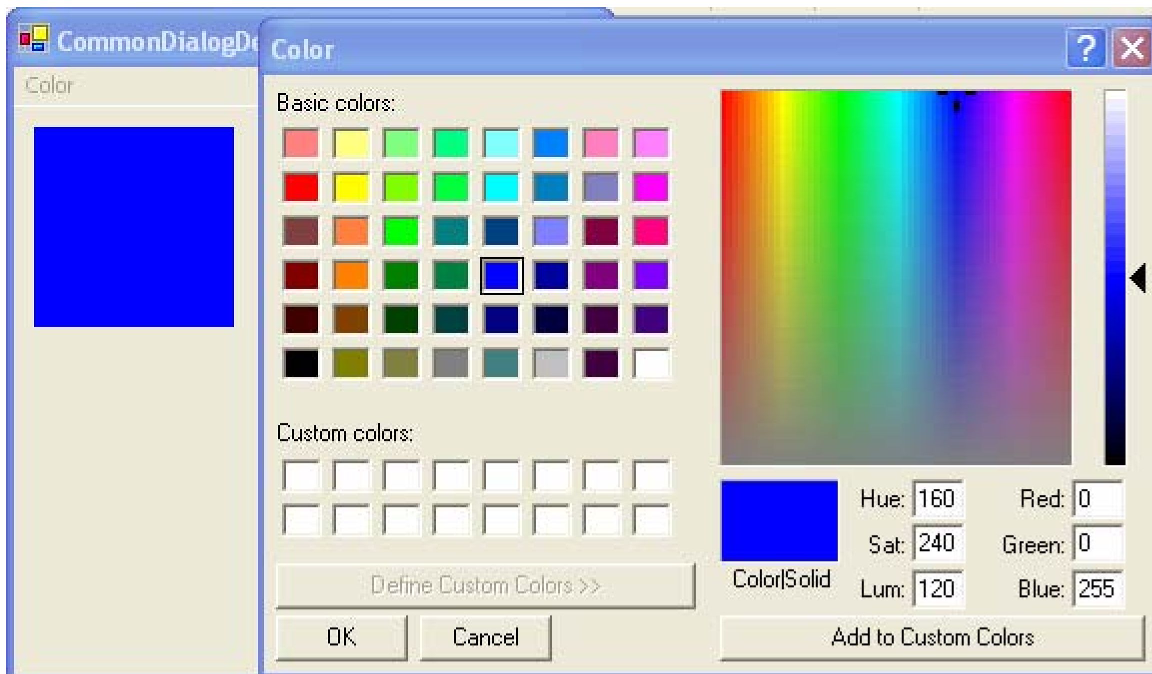
However you accomplish the creation of *colorDialog1*, the next task is to bring up the dialog when the menu item is chosen. Here's a menu handler that will work:

```
private void ChangeColorItem_Click(object sender,
System.EventArgs e)
        {
           colorDialog1.Color = m_theColor;
           if(colorDialog1.ShowDialog() == DialogResult.OK)
              { m_theColor = colorDialog1.Color;
                Invalidate();
              }
        }
```

Here's what you get when you run the program and choose *Change Color*:



You can test this program out. First, pick one of the visible colors and verify that the rectangle changes color when you click OK. Next, try *Define Custom Colors*:



See those funny black dots at the top of the blue strip? Drag them around with the cursor. You will see the sample of the "custom color" change accordingly, and the text fields that present the color's definition also change. A color can be defined in two different ways: either by its RGB values (red, green, blue), as we have studied in a previous lecture, or by its *hue, saturation,* and *luminosity*, a system sometimes used by graphic artists. We will not study that system in this course. Try the *Add to Custom Colors* button.

Notice that the *ColorDialog* itself remembers the custom colors that you defined on a previous invocation. That functionality deserves some discussion. Compare to Petzold page 764, where he says,

"Suppose a user invokes the color dialog box…and carefully defines 16 custom colors. The user then selects one of them and presses OK. Then the user invokes the dialog box again and…The custom colors are gone!"

Then Petzold instructs you on how to save the custom colors and set them before the next invocation of the dialog box. But in our demo, we don't have that problem. Why not? Because our *colorDialog1* is a member variable in *Form1*, so it isn't destroyed until the program ends. In Petzold's program, the color dialog object is created new every time it is

shown. Clearly it's better to have the color dialog be a member variable, and let *ColorDialog* do the work of remembering the custom colors.

As the programmer, you may or may not wish to allow your users the entire functionality of the *ColorDialog*. If you just want to let them choose a basic color, and not use or define custom colors, then you have to initialize *colorDialog1* appropriately:

```
colorDialog1.Color = m_theColor;
colorDialog1.AllowFullOpen = false;
if(colorDialog1.ShowDialog() == DialogResult.OK)
   { m_theColor = colorDialog1.Color;
     Invalidate();
   }
```

Of course, if you do this, then there's no longer a good reason to have the color dialog be a member variable—you might as well create it new each time you use it. On the other hand, there's no good reason to do that either. It really wouldn't matter which way you did it. You could argue about whether it's worse to use memory for the dialog when it's not in use, or to create many dialog objects that then have to be garbage-collected, but the amount of memory involved is completely insignificant.

Here is a complete list of the properties of *ColorDialog* that can be initialized, along with their default values:

Property	Initial Value
AllowFullOpen	**true**
AnyColor	**false**
Color	**Color.Black**
CustomColors	A null reference (**Nothing** in Visual Basic)
FullOpen	**false**
ShowHelp	**false**
SolidColorOnly	**false**

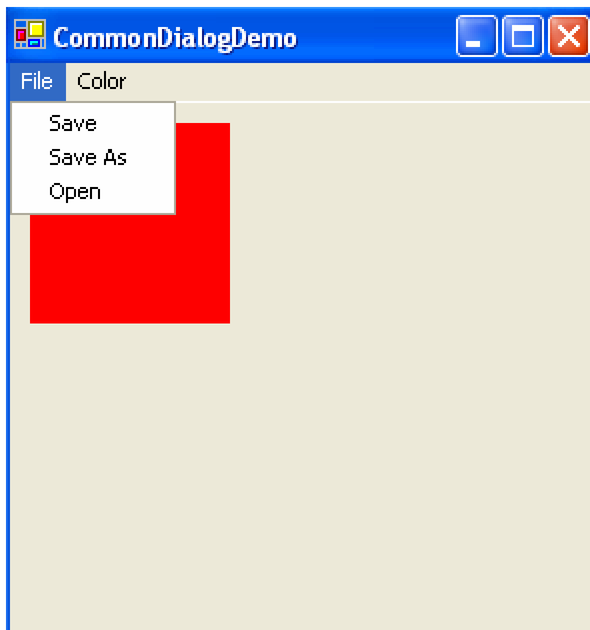
If you set *FullOpen* to true, then the dialog will come up with the custom color selection area already showing. If you set *SolidColorOnly* to true, then the dialog will not show any "hatched" colors. You can use the *CustomColors* property to set a list of pre-defined custom colors that will

appear in the custom colors box. These custom colors are saved as an array of sixteen 32-bit integers. Each integer stores RGB information, but not alpha-channel information. The order of the bytes is not the same as the order used in the *ColorFromArgb* method, so you can't use these stored floors directly to create colors. The array returned by this property should only be used to initialize another color dialog. If for some reason you want to initialize custom colors from known RGB values (which might happen, say if you wanted to provide some colors that occur in an image in your window), you would have to put R in the least significant byte, G in the next, B in the next, and zero in the most significant byte.

Note that the *ColorDialog* object just wraps the Win32 color dialog in a class, and since Win32 did not have partially-transparent colors, the *ColorDialog* cannot handle them either. Thus the user can't choose a color with anything but 255 in the alpha byte.

File Open Dialog

Now we turn to the *FileDialog* class. Add a *File* item to the menu bar and then drag it to the left end of the menu bar in the form editor. Make it look like this:



Now add an *OpenFileDialog* and a *SaveFileDialog* object to your form, just as you did for the *ColorDialog*. Again Visual Studio will add them as member variables to your *Form1* class. Again you could also write the necessary two lines of code yourself just as easily.

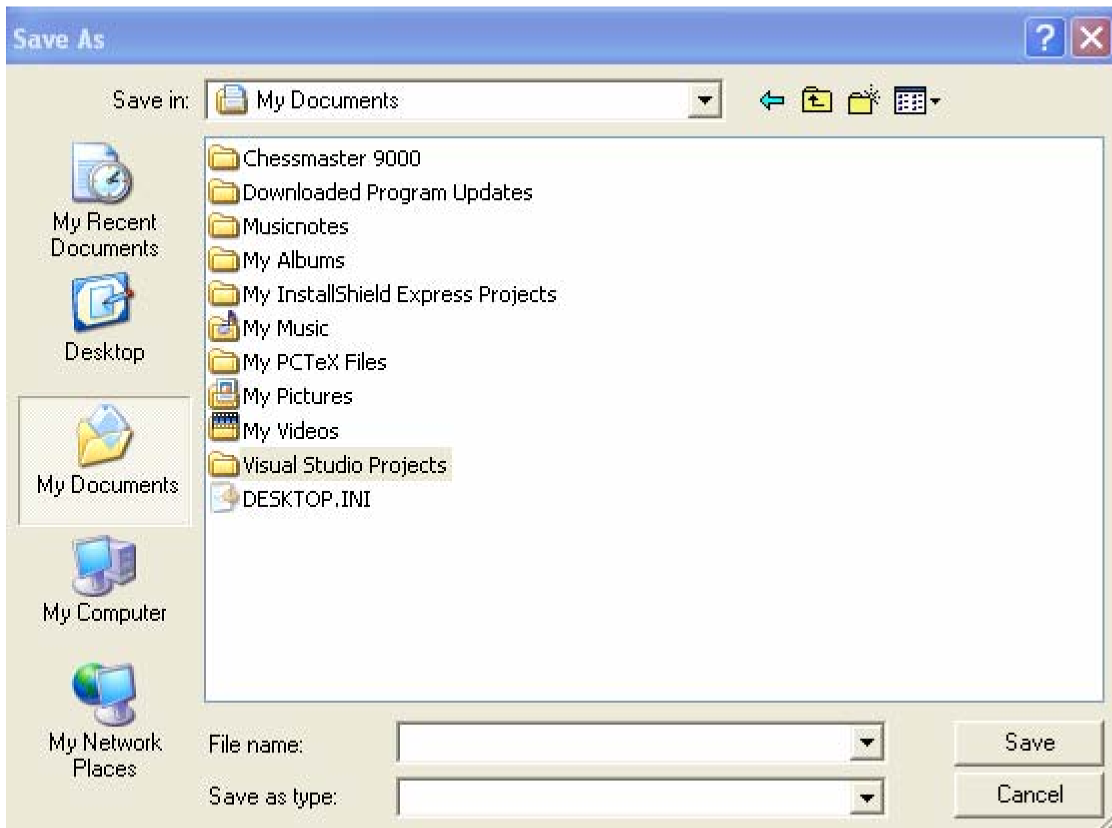
To make these menu items work is a two-step process:

- Use a common dialog to let the user select or enter a filename.
- Open the file; read or write data; then close the file.

In our case, the data to be saved is just one *Color* object, *m_theColor*. The *OpenFileDialog* and *SaveFileDialog* objects can be made to work (in their default modes) with just one line of code. This is somewhat deceptive as there are still a lot of things to learn about their correct use, but to begin with we'll just use their default behavior to get the program running. Add a member variable *m_theFileName* of type *String*, which we will need to make the *Save* menu item work. When the user chooses *Save*, no file dialog is involved—the data should be immediately and silently saved—so the program has to know the filename. The filename is stored when the data is saved for the first time, using *Save As*. The following code shows how to save the data. The sequence is: first get a filename, then use that to create a *FileStream*, then use that to create a *BinaryWriter*, then use that to write data to the stream. Don't forget to close the stream.

```
private void FileSaveAsItem_Click(object sender,
                                  System.EventArgs e)
{ BinaryWriter bw;
  FileStream s;
  if(saveFileDialog1.ShowDialog() == DialogResult.OK)
    { m_theFileName = saveFileDialog1.FileName;
      s = new FileStream(m_theFileName,
                         FileMode.OpenOrCreate);
      bw = new BinaryWriter(s);
      bw.Write(m_theColor.R);
      bw.Write(m_theColor.G);
      bw.Write(m_theColor.B);
      s.Close();
    }
}
```

Here's a screen shot showing the *SaveFile* dialog:



The appearance of this dialog can vary widely. Even on the same computer, the user can choose to have "classic" dialogs or "Explorer-style" dialogs. This is an Explorer-style dialog.

File input/output is similar to what you already know from other languages. In particular the distinction between *file* and *stream* should already be familiar. A *file* is data stored in a particular format on some storage medium, such as a hard disk. A *stream* is a data structure from which data can be read or written. There are thus at least three varieties of stream: read-only, write-only, and read-write. When a file is *opened* the result is a stream, but a stream can also be created for passing data over a network, with no file storage associated to it, and a stream can also be created in memory, associated to some data object (such as a string) without involving either a file or a network connection. You can see in the above code that when the *FileStream* constructor is used, the second argument tells what kind of stream to construct.

There are many possible errors that can occur at run-time when working with files. That subject will be taken up in the next lecture; I mention it

here only to warn you that the code in this example is not production-quality.

The *BinaryWriter* class has a *Write* method with 18 overloaded versions, capable of writing all the value types built into C#, such as byte, int, unsigned int, char, signed char, etc. It can also write an array of characters or an array of bytes (remember that characters in C# use two bytes, to support Unicode). But it does not have overloaded versions to give direct support for writing common FCL objects such as *Rectangle, String*, and *Color.* Therefore, we had to write the bytes of *m_theColor* separately. As an old Win32 API programmer, my instinct was to store the color as an integer, either by

```
 bw.Write(Convert.ToInt32(m_theColor));  // bad code
//compiles but produces run time exception
```

or by doing the conversion myself:

```
int c =
m_theColor.R | (m_theColor.G << 8) | m_theColor.B << 16;
```

This works fine, but it requires familiarity with bit-shifting and bitwise or, and besides, it will use four bytes per color instead of only three. The code exhibited above, writing one byte at a time, is better.

The code for implementing *Save* is similar, but we just use the stored filename instead of a *SaveFileDialog*:

```
private void FileSaveItem_Click(object sender,
                                System.EventArgs e)
 { BinaryWriter bw;
   FileStream s;
   s = new FileStream(m_theFileName,FileMode.OpenOrCreate);
   bw = new BinaryWriter(s);
   bw.Write(m_theColor.R);
   bw.Write(m_theColor.G);
   bw.Write(m_theColor.B);
   s.Close();
 }
```

Of course, we can't test this code until we also implement *Open*:

```
private void FileOpenItem_Click(object sender,
```

```
                                System.EventArgs e)
{ BinaryReader br;
  FileStream s;
  if(openFileDialog1.ShowDialog() == DialogResult.OK)
     { s = new FileStream(openFileDialog1.FileName,
                          FileMode.Open);
       br = new BinaryReader(s);
       byte r,g,b;
       r = br.ReadByte();
       g = br.ReadByte();
       b = br.ReadByte();
       m_theColor = Color.FromArgb(r,g,b);
       s.Close();
       Invalidate();
     }
 }
```

To test the code, open the program, change the color, save a file, then change the color again, then open the file. Also, one should test whether it works when you close the program and then start it again, and then open a saved file. (It does.)

You might think we are done now, but there are still some important things to study that we only now can begin to study. Those will be taken up in the next lecture.