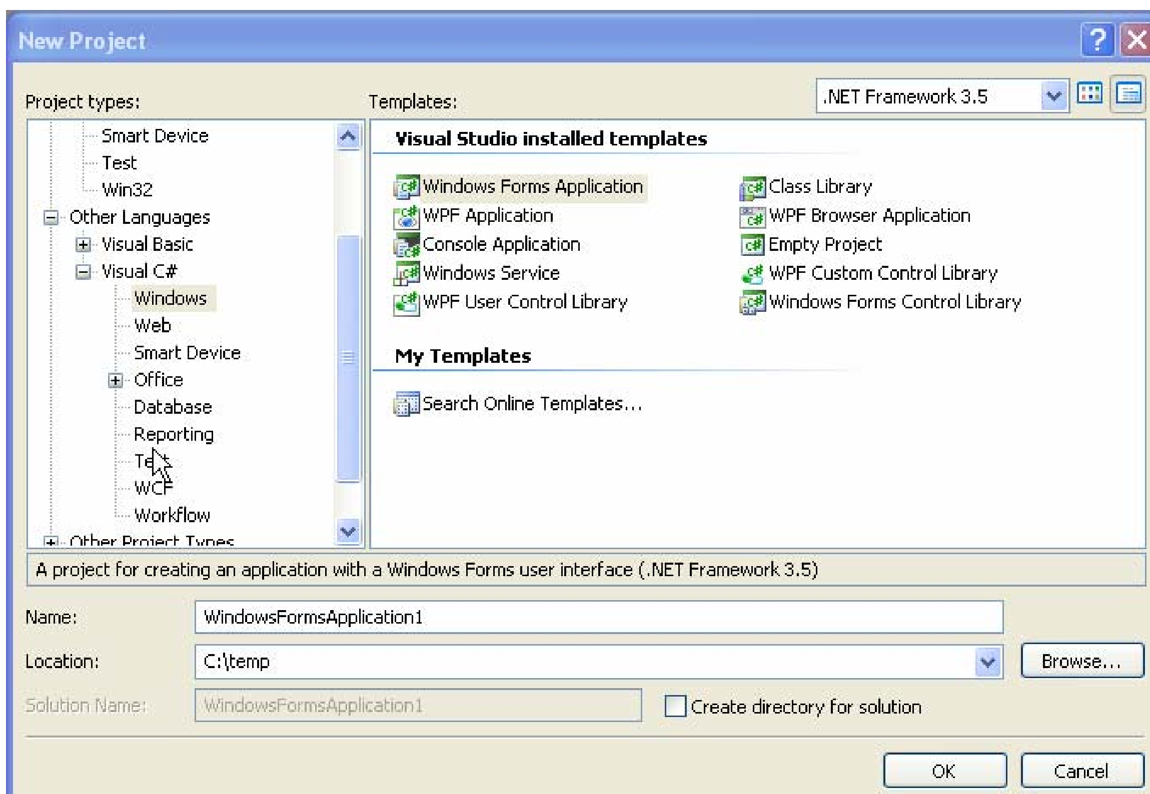
Using Visual Studio

"Solutions" and Projects

A "solution" contains one or several related "projects". Formerly, the word "workspace" was used instead of "solution", and it was a more descriptive word. For example, the solution for a large program would contain one project for the .exe file and several projects for associated.dll files, and perhaps a database project.

Your solutions in this course will usually contain only one project, at least for the first half of the course. Later in the course we will have solutions containing more than one project, so you should be aware of the difference.

To get started, open *Microsoft Visual Studio 2008.* If this is the first time you have opened it, it may ask you what kinds of projects you will mostly be using (in order to customize Help). You will be using C# to build Windows Forms Applications. There are MANY different kinds of projects that can be made with Visual Studio Once you open Visual C++ and choose File | New Project, you see:

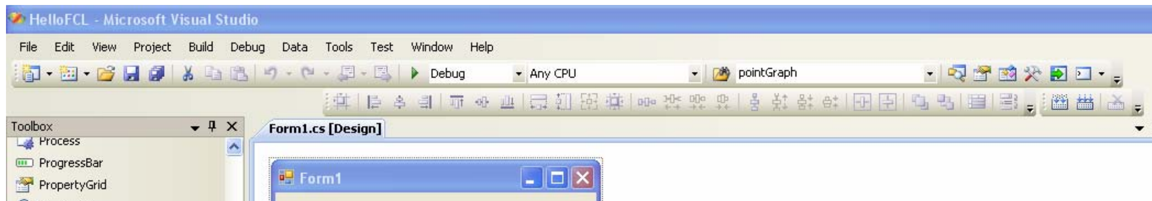


Choose *Visual C# | Windows* on the left and *Windows Forms Application* on the right, as shown in the picture. Now you need to choose a name and location. Today, choose *HelloFCL* for the name. If you check the *Create directory for solution* box, Visual Studio will do just that. So you could, for example, create a *CS130* folder (directory) and keep all your projects for this course in subfolders created by checking this box, and putting C:\CS130 in the *Location* box.

Note for those working on the lab's desktops: In the lab, you will work in C:\temp and in subfolders you create there. DO NOT WORK on the J: drive!! First of all, you have a limited amount of space available there, and using it up can cause Visual C++ to crash. You might lose your work. Second, the network connection from MacQuarrie Hall (where your J: drive is) to the lab is very slow. Therefore enter *C:\temp* in the location field.

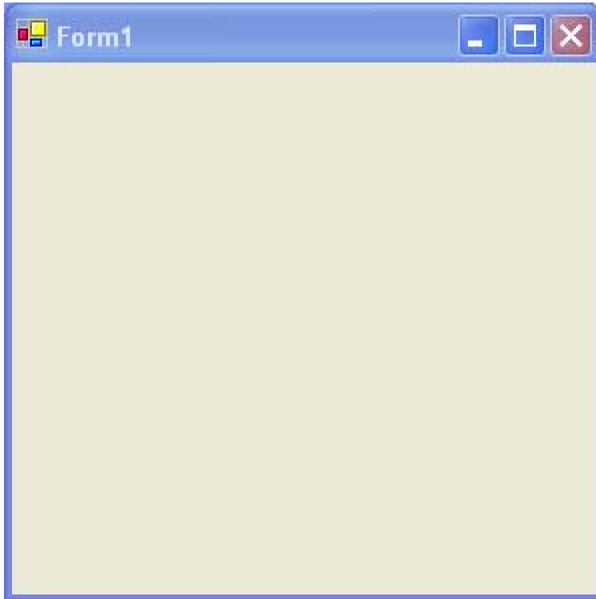
When you click OK, Visual Studio will create your new project, *and* a new solution to contain that project. Later on, when we want to create a new project and add it to an existing solution, you'll learn how to do that.

Click OK. Now, what you see next can vary, because the Visual Studio environment is very customizable, and it remembers how you had it set up. Even your toolbar, near the top of the screen, is composed of many smaller "dockable" toolbars. Therefore it may look quite different from this screen shot:



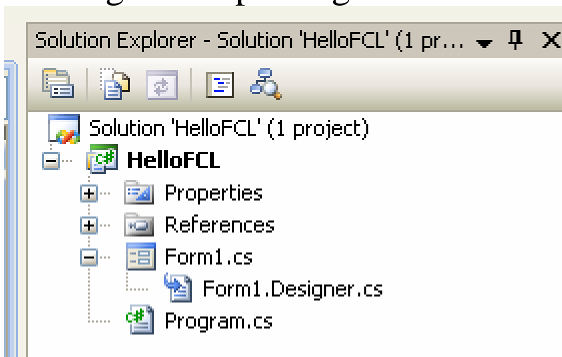
Whatever it looks like on your screen, pass the mouse over the various icons and observe the "Tool Tips" that pop up, describing the functions of the toolbar buttons in words. Identify those two icons shown in this screen shot on the lower row, second and third from the right. One of them is "Build Solution", and the other is "Build helloFCL" (or whatever you named your project). Since you only have one project in your solution, these two buttons do the same thing. Click one of those buttons. Visual Studio will report to you that it has successfully built your project.

Now, look for the green triangle in the toolbar just to the left of the window that says “Debug”. Click that triangle to run your program. You should see something like this:

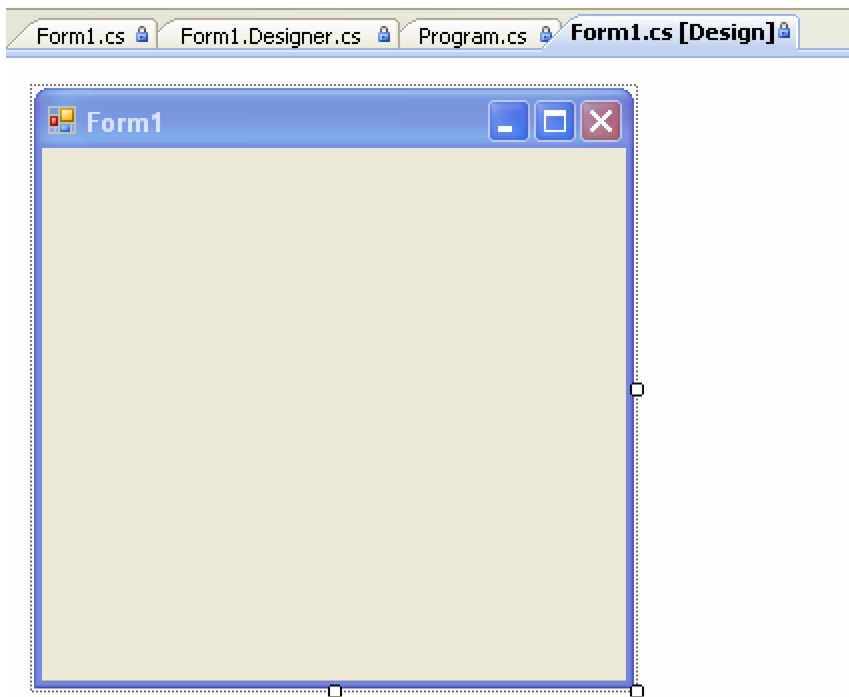


Yes, you’ve already produced a running Windows program, without writing a line of code! Of course, it’s not a very impressive program, but it does have some features. It can be minimized, maximized, moved (by dragging the title bar, or by using the “system menu” accessible from the upper left corner), and resized. Close your running program.

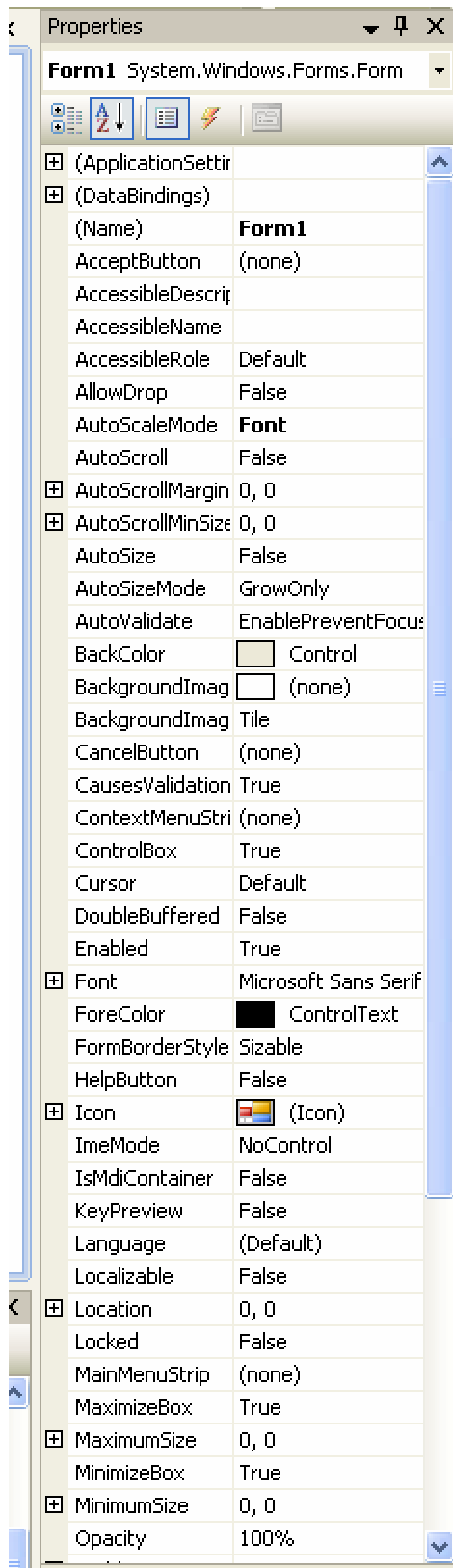
To make the program do something more than exhibit this generic Windows behavior, we will have to write code. All the code for your program is contained in .cs files. Somewhere on your screen, you should have something like the next picture. To see exactly what is illustrated here, you may have to select the *Class View* tab and then expand the nodes by clicking on the plus signs.



This shows the C# files (.cs files) that Visual Studio has created for your project. If you can't find this windows, try *View | Solution Explorer*. We will not use the list of files much; we will access code through Class View or through the "Form Editor", which we will now explore. Look for this window on your screen:



This is the "form editor". Visual Studio is called "Visual" because of this form editor—it lets you drag and drop components such as menus, buttons, edit boxes, etc. and arrange them as you want them to appear. Right now we're not going to do that yet. Instead, right-click anywhere on the form, and select *Properties* to open your form's property sheet:

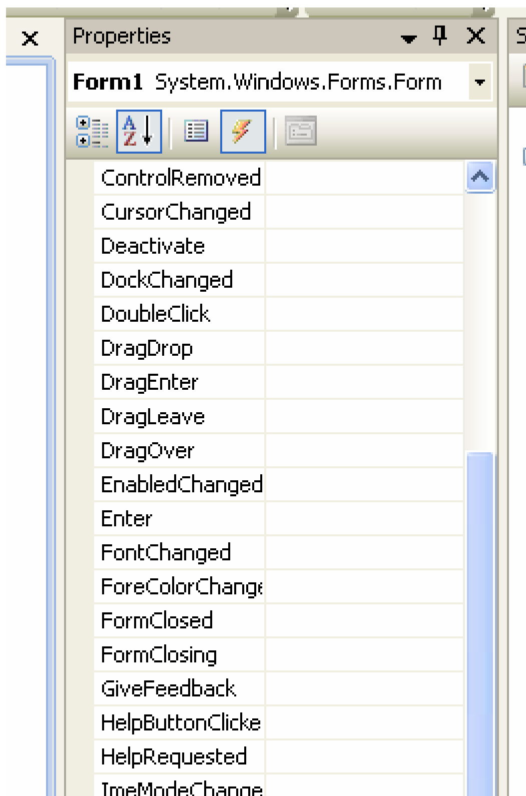


Properties
Form1 System.Windows.Forms.Form
(ApplicationSettir
(DataBindings)
(Name) Form1
AcceptButton (none)
AccessibleDescrip
AccessibleName
AccessibleRole Default
AllowDrop False
AutoScaleMode Font
AutoScroll False
AutoScrollMargin 0, 0
AutoScrollMinSize 0, 0
AutoSize False
AutoSizeMode GrowOnly
AutoValidate EnablePreventFocus
BackColor Control
BackgroundImag (none)
BackgroundImag Tile
CancelButton (none)
CausesValidation True
ContextMenuStri (none)
ControlBox True
Cursor Default
DoubleBuffered False
Enabled True
Font Microsoft Sans Serif
ForeColor ControlText
FormBorderStyle Sizable
HelpButton False
Icon (Icon)
ImeMode NoControl
IsMdiContainer False
KeyPreview False
Language (Default)
Localizable False
Location 0, 0
Locked False
MainMenuStrip (none)
MaximizeBox True
MaximumSize 0, 0
MinimizeBox True
MinimumSize 0, 0
Opacity 100%

What you see here is a list of "design-time properties". Each one of these corresponds to a line of initialization code that has been automatically written for you, and can be changed through this interface (or directly in a text-editor window).

Change the *Text* property to *First Program* (and press *Enter*). That will affect the text in the title bar.

Click the lightning-bolt icon near the top of the property sheet. That will show you a list of "events" to which your program could possibly respond:



(the actual list is longer than shown here)

Double-click on *Paint*. You will be taken to a text-editor window with a skeleton event-handler already written:

```
private void Form1_Paint(object sender, PaintEventArgs e)
        {
        }
```

Our immediate plan is to write code in this function to get some text drawn on the screen. You might hope this would take only one line of code, but it is a bit more complicated. It's worthwhile to explain why.

.NET graphics is (almost) a *stateless* graphics system. By contrast, previous Windows graphics libraries were *stateful*. (There are actually 12 read-write properties of the *Graphics* class, see Petzold p. 181, but people call .NET graphics stateless anyway.) In object-oriented terms, you can think of the state as specified by some static variables in the *Graphics* class. These variables would remember the current font, the current location of the cursor, the current direction to draw text, etc. Having state makes it easier to write code, but it creates problems when more than one application (or more than one thread of the same application) is using the library, since changes made to the static variables by one thread can affect another thread. Since programs that run on Web servers often open one thread for each user, programming with threads is much more common nowadays. Therefore .NET has been designed to be stateless. That means that every graphics call must pass a lot of parameters. Instead of simply passing the string to be drawn and the starting coordinates to *DrawString*, we have to pass a font, a brush, and a *StringFormat* object as well.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{ // Create string to draw.
  String drawString =
        "Damn the torpedoes, full speed ahead!";
  // Create font and brush.
  Font drawFont = new Font("Arial", 16);
  SolidBrush drawBrush = new SolidBrush(Color.Black);
  // Create point for upper-left corner of drawing.
  float x = 50.0F;     // x=50 will work too but not x=50.0
  float y = 70.0F;
  StringFormat drawFormat = new StringFormat();
  e.Graphics.DrawString(drawString, drawFont,
                         drawBrush, x, y, drawFormat);
}
```

The *PaintEventArgs* class belongs to *System.Windows.Forms*. There is a *using* command for this class at the top of the file, so *PaintEventArgs e* is equivalent to *System.Windows.Forms.PaintEventArgs e*.

Instead of

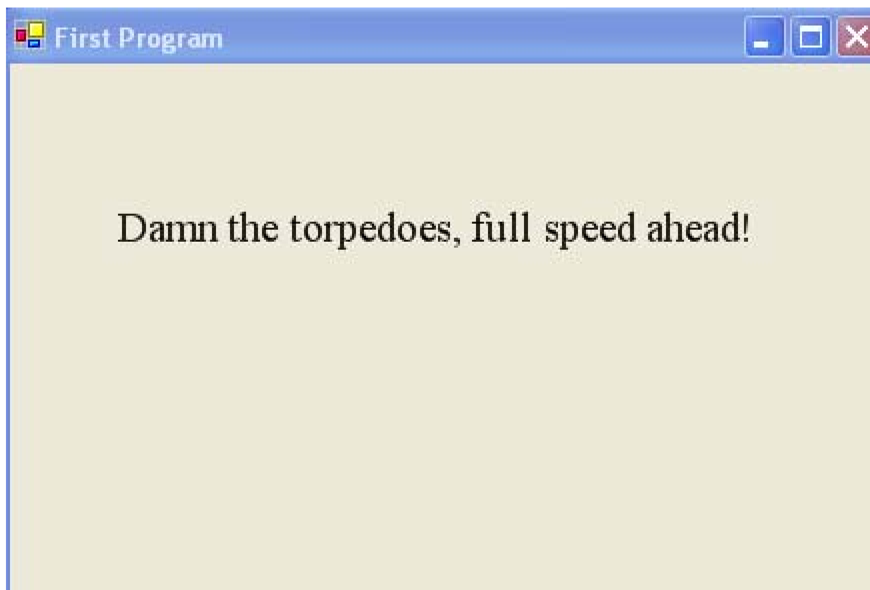
```
  SolidBrush drawBrush = new SolidBrush(Color.Black);
```

you can also use

```
Brush drawBrush = Brushes.Black;
```

The *Brushes* class contains some static members in pre-defined colors. Usually you can get by with these pre-defined colors, but sometimes you might need a custom color (for example, to match a color in an image), and in that case you would use *new SolidBrush* as above. Note that *SolidBrush* is derived from *Brush*—sometimes you can get strange error messages if you use *SolidBrush* where you need *Brush* or vice-versa. Since C# has garbage collection and computers are fast, the choice whether to use static brushes or dynamic ones is not a performance issue, but one of programmer convenience.

Build and run this program. Here's the output:



Since C# has garbage collection, we don't have to worry about deleting the font, brush, and format objects.

The coordinate system used for *x* and *y* is called "client coordinates". The origin is at the upper left corner of the client area, i.e. just below the title bar at the left. The *y* coordinate increases downwards, not upwards as in mathematics.

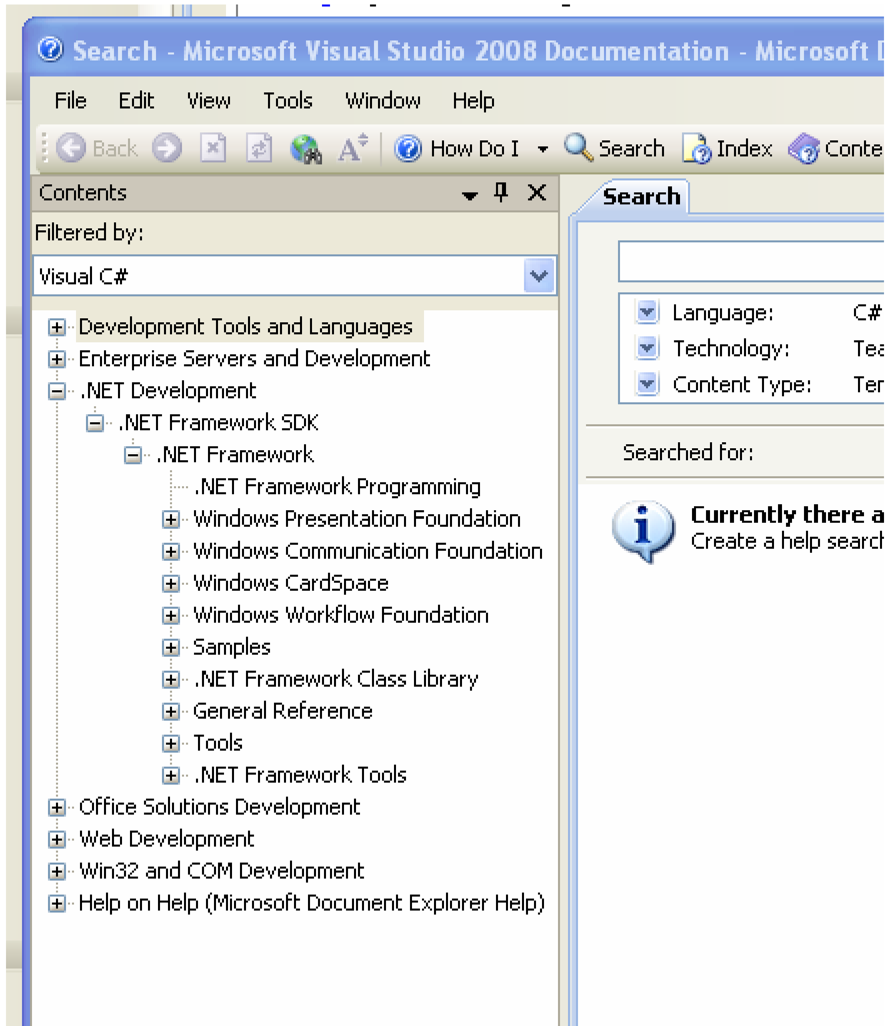
Try changing the font size. Try changing the font family name from “Arial” to “Times New Roman”. Try changing the brush color from black to red or green. Try inserting the line

```
drawFormat.FormatFlags=StringFormatFlags.DirectionVertical;
```

after “new StringFormat();”. That will give you some idea what the *StringFormat* parameter is for.

Historical note. The famous words quoted in this code were spoken by Admiral David Glasgow Farragut in 1864, during the American Civil War. The torpedoes in question would today be called tethered mines. (Look up the whole fascinating story in Wikipedia if you like, later on at home.) Here’s a painting of the battle of Mobile Bay:

You are going to need to study the documentation of the Foundation Class Libraries that you will use to write programs in .NET. Here is how to find that documentation in Visual Studio. Choose *Help | Contents* from the menu and then open up *.NET Develpment | .NET Framework*



and then scroll down through the long list of namespaces. Note the existence of the *System.Drawing* and *System.Windows.Forms* namespaces. We will be using the classes in those namespaces extensively.

As you can see, there are thousands of classes and methods in .NET. Nobody knows them all. Even after this semester is over, you won't know more than a small fraction of them. Therefore, being able to use the documentation and the tools provided to access it is an extremely important skill.

Let's say, for instance, that we want to know what colors have predefined brushes in the *Brushes* class.

In that example, we don't need to browse the documentation directly at all. Just type *Brushes.* in your code, and a window will pop up showing you the possible completions. This feature is called *Intellisense*. You will find it incredibly useful. In particular, if you want to know what method to use to accomplish some specific task, often you can find it this way, even if you don't know its name or even if such a function exists.

Once you do know the exact name of the class or function you want to look up, just type it in a text file, put the cursor on it, and press F1. Instantly (well, after looking at the hourglass briefly) you are taken to the correct manual page. This too is wonderful.

If you do NOT know the exact name of the class or function, then F1 won't work, and usually *Search* won't work either. You will get hundreds of useless "hits".

When you are looking for an unknown function to accomplish a specific task (which is often the case when you want help), and you can't find it using Intellisense and F1, then the best way is to use *Help Contents* and look for the right class, and then see all the methods of that class, looking for a useful one for your present purpose. (And then, if you don't find it, go *up* the class hierarchy to the parent class.)