Chapter 10 – Working with Files

# Objectives

* Understanding Components and References
* Working with .NET Directories and Files
* Working with Sequential Access Files
* Handling Random Access Files
* Serialization
* The FileSystemWatcher Class

Many of the previous chapters up to this point have provided building blocks of the VB.NET language. We have covered a lot of topics that should have been familiar to you if you have ever programmed in any other language.

This chapter will start moving into more of the details of .NET. We will be covering components and references, which are ways that we can greatly extend the capabilities of .NET with little effort. We will also examine interacting with the operating system in order to manipulate files and directories (folders).

Finally, this chapter will finish up with a discussion on how to work with both sequential and random access files. When you are done with this chapter you ought to be able to read and write information from various file types, allowing you to greatly expand your programming capabilities through the ability to work with external data.

# Understanding Components and References

One of the things that makes Visual Basic so powerful is its ability to reuse existing pieces of software that someone else has already created – in fact that was what initially rocketed VB as a development language. You are already very well versed in this capability since you have been dragging and dropping controls onto forms when you create graphic user interfaces for your programs. Each control does a specific task and you can imagine how much work it would have taken for you to build that functionality if it didn't exist for you to use. Think back on trying to implement something similar to a textbox in good old command line Java.

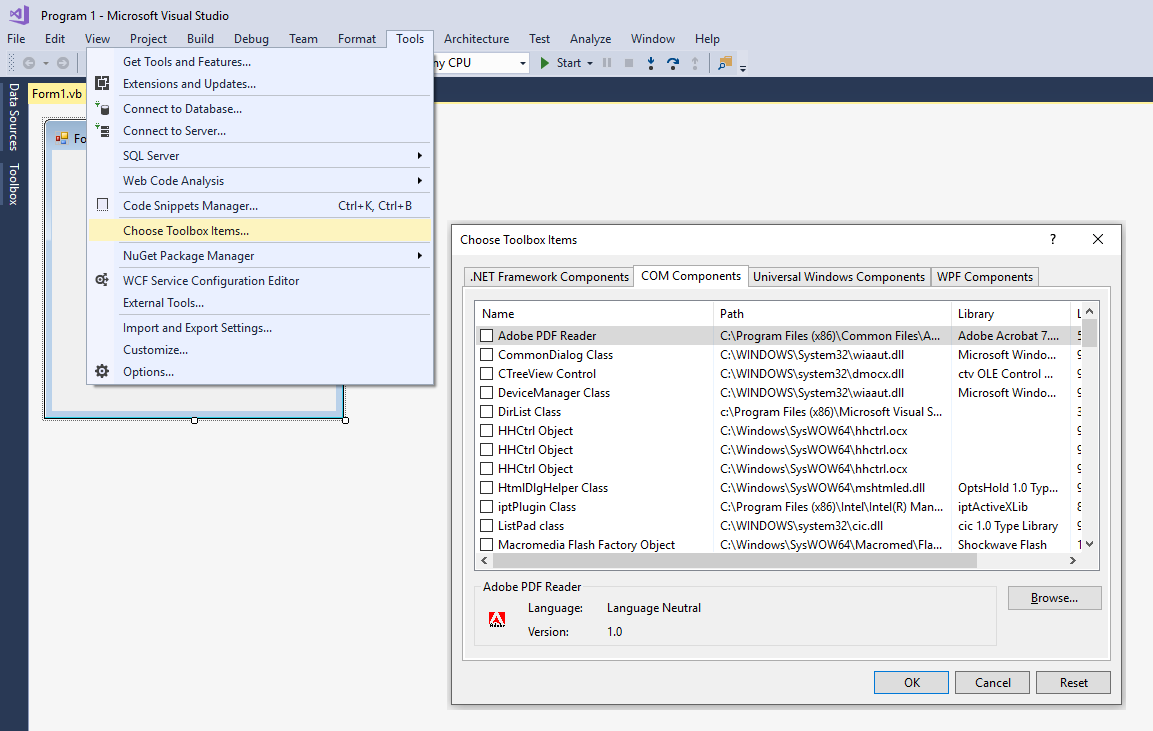
VB provides this reuse capability through two different flavors: components and references. Components typically refer to controls, which are self-contained (usually visual) controls that can be embedded in any container that can host controls including VB, VC++, C#, Delphi, IE, as well as third party applications. The TextBox and Command Button are two examples of visual controls; they show up visually during the execution of a program. The ToolTip and Timer controls are still examples of controls, but they are invisible in that they don’t render as a visual object.

Versions of VB prior to .NET used what were called COM (or ActiveX) controls. The COM (Component Object Model) technology was a binary standard that Microsoft established which laid out how controls could be created and utilized across multiple languages. At some point, Microsoft decided lightweight controls could be used in the company's Internet Explorer browser and started calling them ActiveX controls. ActiveX and COM controls are essentially the same thing. There are many free controls that you can find. Some companies still make their entire revenue by selling controls to increase the feature set of .NET development.

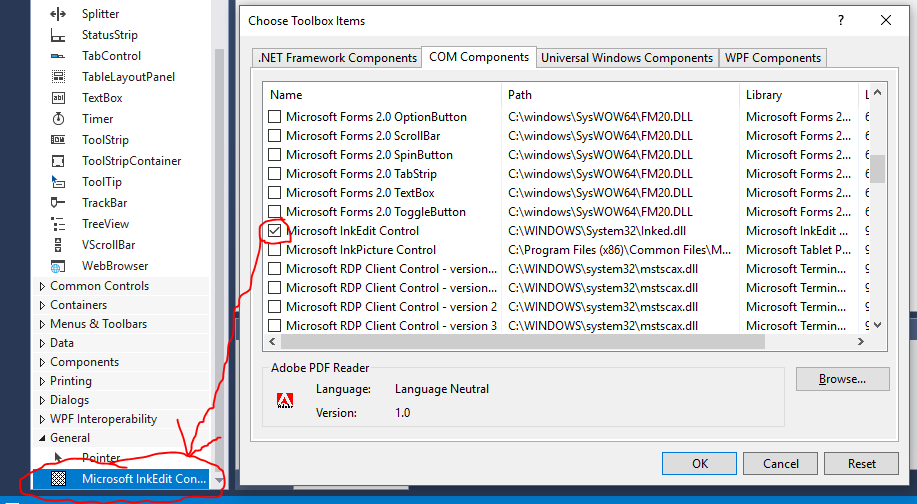
You can use many COM controls in .NET, and most will work with no problem. Microsoft has radically changed the architecture in .NET, however, so there is now a compatibility layer built-in that lets you use COM – put another way, it's not really the way Microsoft wants you to work in .NET. There is no guarantee that a particular COM control will work, therefore you should carefully test COM controls before blindly trying to use them in a project.

The new control model that Microsoft prefers to be used with .NET is called, for lack of a better name, .NET components. The .NET controls cannot be used in pre-VB.NET versions. In fact, .NET components can currently only be used by Visual Studio .NET languages and some are highly framework dependent. Therefore, this is a factor to keep in mind if you decide to write your own controls, which we will be doing in a couple of chapters.

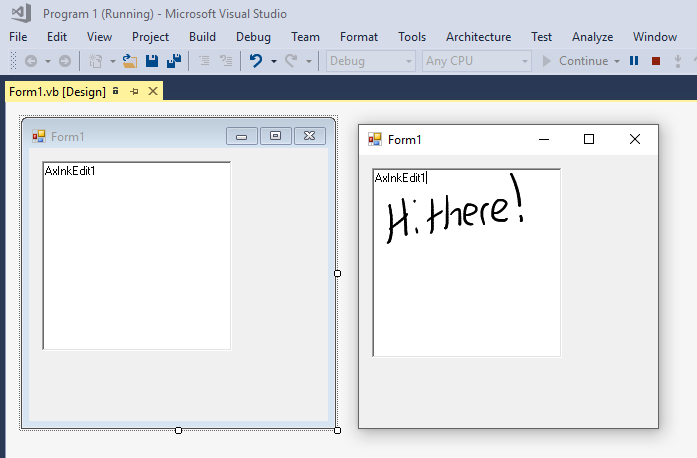
To use a graphical component, you must add it to your toolbox. To do this, simply click on Tools 🡪 Choose Toolbox Items. You will be presented with a tabbed dialog box that lists the .COM and .NET components that are available on your system. You’ll also see that there are WPF, Windows Phone, Silverlight and Activity controls as well – many of these are beyond the scope of this class to discuss. You can download and add-in new components as you find them.



Whatever items you select from the component type lists will then appear in your Toolbox for use in your project.



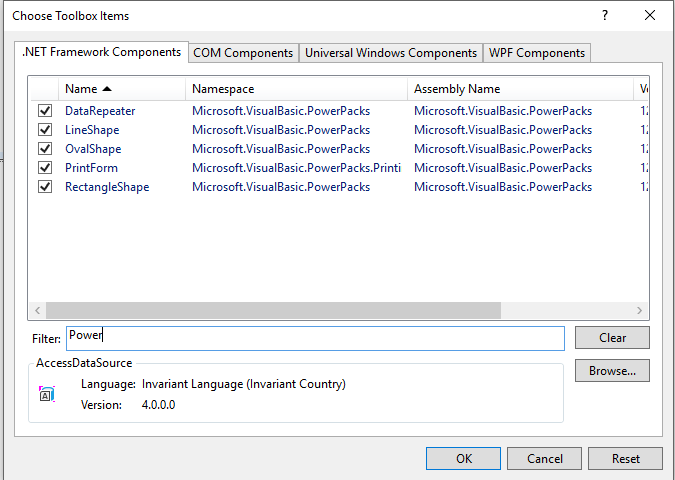
You can see from the previous figure that I added the Microsoft InkEdit control to my project since it appears checked in the list of COM components. In addition, if you examine the Toolbox once you press OK to save the checked changes, you will see that the control appears for use in my form under the “General” section of the Toolbox. I could now drag and drop that control on a form and I have new added functionality instantly available in my program:



As you can see from the screenshot, the new control allows me to write using my pen onto the control’s surface. Even though you don’t see it, once I’m done with the pen, the control will then convert my handwriting into text. Pretty awesome functionality to give to an end-user and it cost me nothing other than the time to import the control!

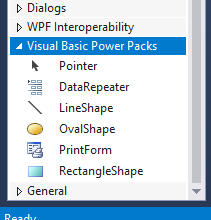
Microsoft has actually provided some older components certified for use in .NET from the pre-.NET collection of controls. One handy set to pick up is the Visual Basic Power Packs, which can be downloaded from <https://docs.microsoft.com/en-us/dotnet/visual-basic/developing-apps/windows-forms/power-packs-controls>. As with other things that Microsoft has broken, the Power Packs no longer completely install the new tools into Visual Studio – they are there, just not available for use. It’s really not hard to finish this installation up following these steps:

1. Right click in the empty canvas on the Toolbox and choose Add Tab from the context menu. Name the new tab Visual Basic Power Packs.
2. Left click on the Power Packs tab in the Toolbox, and it will expand showing you that nothing is in there.
3. Add the controls by right clicking in the Power Packs expanded area and click on Choose Items from the context menu that appears.
4. When the components dialog appears, make sure that the .NET Components tab is highlighted and then type Power in the Filter box. You’ll now see the five controls. Click on each on so that it becomes checked:



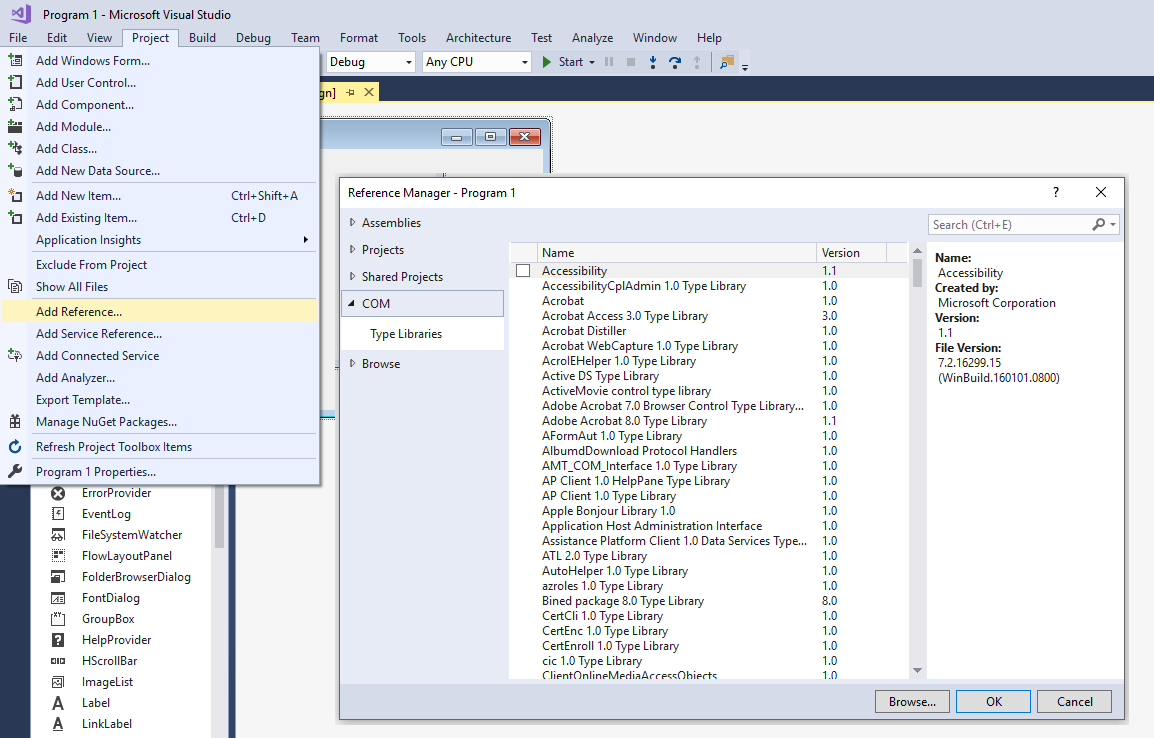
Click on the OK button once you’ve selected all the controls.

1. Check your Toolbox and you should see that the Visual Basic Power Packs section now has the controls added to it for your use:



The truth is that many VB6-era components will work fine with .NET (the Winsock control is one exception!). You might want to look around when you decide to start building controls just to get a sense of what’s available (both COM and .NET) -- there are some really nice controls out there that can save you a lot of time and effort!

Now that we've covered the basics of what components are, let's examine references. References, in contrast to controls, are best thought of as code libraries (usually implemented as dynamic link libraries or Visual Basic type libraries) that contain definitions and interfaces for various functions. You select references through the IDE menu system, but in a different location than the one you used to pick components. The menu location is Project🡪Add Reference. References come in multiple flavors too. The most common reference types obviously are .NET and COM references. The .NET references are available under the Assemblies tab on the left side of the Reference Manager. The next snapshot shows of some of the COM references that I have available on my computer, which I accessed via the COM tab on the left side of the Reference Manager:



From here on out, many of the projects and techniques that we will be discussing will require that you add components and/or references to your project. That's the reason that we spent some time looking at the difference between the two and how to add in either of the entity types. It’s always better off to know how to do something before you actually have to do so.

# Working with .NET Directories and Files

The next topic that we need to discuss includes the mechanisms for working with directories and folders. While the O/S handles the gory details of the physical work, most programming languages provide a programming interface to allow developers to interact with the O/S. VB.NET provides us with several classes to handle this type of work. We'll slowly move though the basic classes for manipulating directories and files and then eventually move into examples of reading/writing files. Most of these classes live within the System.IO namespace.

## The Directory Class

The Directory class provides a convenient mechanism for working with directories and drives. Microsoft now uses the term "folders" to refer to directories underneath Windows. UNIX and MS-DOS refer to these entities as directories. I don't know why Microsoft chose the folder name and then called this class Directory instead of something like “Folder.”

Here are a list of the common methods and their purposes. We'll look at using some of these methods in a bit:

Directory class methods:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| CreateDirectory | Creates the specified directory |
| Delete | Deletes a directory and possibly anything contained within the directory |
| EnumerateDirectories | Returns an enumerable collection of directory names in a specified path |
| EnumerateFiles | Returns an enumerable collection of file names in a specified path |
| EnumerateFileSystemEntries | Returns an enumerable collection of file system entries in a specific path |
| Exists | Determines if the specified directory exists |
| GetAccessControl | Gets a DirectorySecurity object that holds the access control list (ACL) list for a specified directory |
| GetCreationTime | Gets the date and time the directory was created |
| GetCreationTimeUtc | Gets the UTC date and time the directory was created |
| GetCurrentDirectory | Gets the application's current directory |
| GetDirectories | Gets a collection with the names of subdirectories within a directory |
| GetDirectoryRoot | Gets the volume information (disk letter) and root directory for a directory |
| GetFiles | Gets a collection with the names of all files within a directory |
| GetFileSystemEntries | Gets a collection that contains the names of all files and subdirectories in the specified directory |
| GetLastAccessTime | Gets the date and time the file or directory was last accessed |
| GetLastAccessTimeUtc | Gets the UTC date and time the file or directory was last accessed |
| GetLastWriteTime | Gets the date and time the file or directory was last written to |
| GetLastWriteTimeUtc | Gets the UTC date and time the file or directory was last written to |
| GetLogicalDrives | Gets a collection that contains the names of all logical drives |
| GetParent | Gets the parent directory of the specified directory |
| Move | Moves a file or directory |
| SetAccessControl | Applies access control list (ACL) entries described by a DirectorySecurity object to the specified directory |
| SetCreationTime | Sets the creation date and time for the specified file or directory |
| SetCreationTimeUtc | Sets the creation UTC date and time for the specified file or directory |
| SetCurrentDirectory | Sets the application's current directory |
| SetLastAccessTime | Sets the date and time the specified file or directory was last accessed |
| SetLastAccessTimeUtc | Sets the UTC date and time the specified file or directory was last accessed |
| SetLastWriteTime | Sets the date and time a file or directory was last written to |
| SetLastWriteTimeUtc | Sets the UTC date and time a file or directory was last written to |

## The File Class

Just as we saw with directories, there is a class that allows us to manipulate files: the File class. Since you will be using this class quite often, you should familiarize yourself with the methods that are available within the class. Again, there will be a sample program in a bit that shows the use of some of these methods.

Here are our File methods:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| AppendAllLines | Appends lines to a file and then closes the file. This will create a file if it doesn’t exist |
| AppendAllText | Opens a file, appends the specified text string and then closes the file. This will create a file if it doesn’t exist |
| AppendText | Creates a StreamWriter object that a program can use to append text to an existing file or to a new file if the specified file does not exist |
| Copy | Copies the specified file to a new file |
| Create | Creates or overwrites the specified file within the directory given in a path |
| CreateText | Creates or opens a new file for text output, returning a StreamWriter object that the program can use to write to the file |
| Decrypt | Decrypts a file that was encrypted by the current account using the Encrypt method |
| Delete | Deletes the specified file |
| Encrypt | Encrypts a file so that only the account used to encrypt the file can decrypt it |
| Exists | Determines if the specified file exists |
| GetAccessControl | Gets a FileSecurity object that holds the access control list (ACL) for the specified file |
| GetAttributes | Returns a FileAttributes object that contains the attributes for the specified file |
| GetCreationTime/Utc | Gets the date and time the specified file was created |
| GetLastAccessTime/Utc | Gets the date and time the file or directory was last accessed |
| GetLastWriteTime/Utc | Gets the date and time the file or directory was last written to |
| Move | Moves a file or directory |
| Open | Opens the specified file, returning a FileStream object that can be used to read or write the file |
| OpenRead | Opens the specified file, returning a FileStream object than can be used to read the file |
| OpenText | Opens the specified file, returning a StreamWriter object that can be used to read the file |
| OpenWrite | Opens the specified file, returning a FileStream object that can be used to write to the file |
| ReadAllBytes | Opens a binary file, reads the contents of the file into a byte array and then closes the file |
| ReadAllLines | Opens a text file, reads all lines of the file and then closes the file |
| ReadAllText | Opens a text file, reads all lines of the file and then closes the file (this assumes UTF-8 [ASCII] text file encoding) |
| ReadLines | Reads the lines of a file |
| Replace | Replaces the contents of the specified file with the contents of another file, deleting the original file and creating a backup of the replaced file |
| SetAccessControl | Applies access control list (ACL) entries described by a FileSecurity object to the specified file |
| SetAttributes | Sets the attributes specified in a FileAttributes object for a given file |
| SetCreationTime/Utc | Sets the creation date and time for the specified file or directory |
| SetLastAccessTime/Utc | Sets the date and time the specified file or directory was last accessed |
| SetLastWriteTime/Utc | Sets the date and time a file or directory was last written to |
| WriteAllBytes | Creates a new file, writes the specified byte array to the file and closes the file. If the target file exists, it will be overwritten |
| WriteAllLines | Creates a new file, writes the specified string array to the file and then closes the file. If the target file exists, it will be overwritten |
| WriteAllText | Creates a new file, write the specified string to the file and then closes the file. If the target file exists, it will be overwritten |

Let's write a program that will accept the name of a file or directory and determine whether the item is a file or directory. If it is a directory, the contents of the directory will be displayed. In either case, the attribute information of the item will be shown.

Here's the code:

Imports System.IO

'Chapter 10 - Program 1

Module Module1

Sub Main()

Dim strFileName As String

Dim strDirList As String()

Dim intLoop As Integer

Dim strDetails As String

strFileName = InputBox("Enter a directory or filename: ")

'Is it a file...

If File.Exists(strFileName) Then

'...if it was, then print out the details...

strDetails = "File: " & strFileName & " Exists" & vbCrLf & vbCrLf

strDetails &= "Created: " & File.GetCreationTime(strFileName) & vbCrLf

strDetails &= "Last Modified: " & File.GetLastWriteTime(strFileName) &

vbCrLf

strDetails &= "Last Accessed: " & File.GetLastAccessTime(strFileName)

& vbCrLf & vbCrLf

Else

'...otherwise was it a directory...

If Directory.Exists(strFileName) Then

'...print out the details...

strDetails = "Directory: " & strFileName & " Exists" & vbCrLf &

vbCrLf

strDetails &= "Created: " & Directory.GetCreationTime(strFileName)

& vbCrLf

strDetails &= "Last Modified: " &

Directory.GetLastWriteTime(strFileName) & vbCrLf

strDetails &= "Last Accessed: " &

Directory.GetLastAccessTime(strFileName) & vbCrLf &

vbCrLf

'Get all directory name in this directory

strDirList = Directory.GetDirectories(strFileName)

For intLoop = 0 To strDirList.Length - 1

strDetails &= strDirList(intLoop) & vbCrLf

Next

Else

'...must have been a bad file or directory name!

strDetails = "The entity " & strFileName & " does not exist"

End If

End If

Debug.WriteLine(strDetails)

End Sub

End Module

Here’s the output from the program when run on C:\Windows:

Directory: C:\Windows Exists

Created: 9/29/2017 4:45:11 AM

Last Modified: 12/17/2017 7:35:16 AM

Last Accessed: 12/17/2017 7:35:16 AM

C:\Windows\addins

C:\Windows\appcompat

C:\Windows\apppatch

C:\Windows\AppReadiness

C:\Windows\assembly

C:\Windows\bcastdvr

C:\Windows\Boot

C:\Windows\Branding

C:\Windows\CbsTemp

C:\Windows\Cursors

C:\Windows\debug

C:\Windows\DeliveryOptimization

C:\Windows\diagnostics

C:\Windows\DigitalLocker

C:\Windows\Downloaded Program Files

C:\Windows\ELAMBKUP

C:\Windows\en

C:\Windows\en-US

C:\Windows\Fonts

C:\Windows\GameBarPresenceWriter

C:\Windows\Globalization

C:\Windows\Help

C:\Windows\Hewlett-Packard

C:\Windows\IME

C:\Windows\ImmersiveControlPanel

C:\Windows\INF

C:\Windows\InfusedApps

C:\Windows\InputMethod

C:\Windows\Installer

C:\Windows\L2Schemas

.

.

.

.

C:\Windows\SysWOW64

C:\Windows\TAPI

C:\Windows\Tasks

C:\Windows\Temp

C:\Windows\TextInput

C:\Windows\ToastData

C:\Windows\tracing

C:\Windows\twain\_32

C:\Windows\UpdateAssistantV2

C:\Windows\vpnplugins

C:\Windows\Vss

C:\Windows\Web

C:\Windows\WinSxS

## The Path Class

The Path class is another very handy utility class. A path is the full name to a particular file or directory on a system. An example path would be C:\Windows\Notepad.Exe, which is the full path from the drive letter, through all directories, down to where the Windows Notepad text editor program is stored.

The Path class allows you to examine and manipulate path information such as whether a drive letter, a directory name, a filename and/or a filename extension was entered in a path.

Here are the Path class methods:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| ChangeExtension | Changes the extension of a path string |
| Combine | Combines an array of strings into a path |
| GetDirectoryName | Returns a path's directory name |
| GetExtension | Returns a path filename's extension |
| GetFileName | Returns a path's filename and extension |
| GetFileNameWithoutExtension | As above, but without the extension |
| GetFullPath | Returns an absolute path |
| GetInvalidFileNameChars | Returns an array of characters not allowed in a filename |
| GetInvalidPathChars | Returns an array of characters not allowed in a path |
| GetPathRoot | Returns a path’s absolute root |
| GetRandomFileName | Returns a random folder name or file name |
| GetTempFileName | Creates a unique 0-byte file in the directory |
| GetTempPath | Returns the path to the O/S temporary directory |
| HasExtension | Returns a Boolean that specifies if a path includes a filename extension |
| IsPathRooted | Returns a Boolean value that indicates whether the path is absolute (rooted) or relative |

Path class properties:

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| AltDirectorySeparatorChar | Gets a secondary O/S-specific alternate character that separates directory levels in a path |
| DirectorySeparatorChar | Gets the primary character used to separate directories in an O/S |
| PathSeparator | Gets the separator character that separates path strings in environment variables |
| VolumeSeparatorChar | Gets an O/S-specific character that separates the volume from the path |

Let's write a sample program that showcases some of the Path class properties and methods.

Imports System.IO

'Chapter 10 – Program 2

Module Module1

Sub Main()

Dim strResults As String

'Point at a fully pathed valid filename

Dim aPath As String = "C:\windows\winhlp32.exe"

'Exercise the methods and properties

strResults &= "Started with " & aPath & vbCrLf

strResults &= "Dir name: " & Path.GetDirectoryName(aPath) & vbCrLf

strResults &= "Extension: " & Path.GetExtension(aPath) & vbCrLf

strResults &= "Filename: " & Path.GetFileName(aPath) & vbCrLf

strResults &= "Full Path: " & Path.GetFullPath(aPath) & vbCrLf

strResults &= "Root: " & Path.GetPathRoot(aPath) & vbCrLf & vbCrLf

strResults &= "Primary Directory Separator: " & \_

Path.DirectorySeparatorChar & vbCrLf

strResults &= "Secondary Directory Separator: " & \_

Path.AltDirectorySeparatorChar & vbCrLf

strResults &= "Invalid Path Chars: "

strResults &= vbCrLf

strResults &= "Path separator: " & Path.PathSeparator & vbCrLf

strResults &= "Volume separator: " & Path.VolumeSeparatorChar & vbCrLf &

vbCrLf

For Each aBadChar In Path.GetInvalidPathChars

strResults &= aBadChar & " "

Next

strResults &= vbCrLf

Debug.WriteLine(strResults)

End Sub

End Module

Here’s the output from that program:

Started with C:\windows\winhlp32.exe

Dir name: C:\windows

Extension: .exe

Filename: winhlp32.exe

Full Path: C:\windows\winhlp32.exe

Root: C:\

Primary Directory Separator: \

Secondary Directory Separator: /

Invalid Path Chars:

Path separator: ;

Volume separator: :

" < > |

## DriveInfo Class

This class can be used to work with the drives, both physical and logically mapped, that are available on the computer. Here is a list of the various properties that are available:

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| [AvailableFreeSpace](http://msdn.microsoft.com/en-us/library/system.io.driveinfo.availablefreespace.aspx) | Indicates the amount of available free space on a drive |
| [DriveFormat](http://msdn.microsoft.com/en-us/library/system.io.driveinfo.driveformat.aspx) | Gets the name of the file system, such as NTFS or FAT32 |
| [DriveType](http://msdn.microsoft.com/en-us/library/system.io.driveinfo.drivetype.aspx) | Gets the drive type |
| [IsReady](http://msdn.microsoft.com/en-us/library/system.io.driveinfo.isready.aspx) | Gets a value indicating whether a drive is ready |
| [Name](http://msdn.microsoft.com/en-us/library/system.io.driveinfo.name.aspx) | Gets the name of a drive |
| [RootDirectory](http://msdn.microsoft.com/en-us/library/system.io.driveinfo.rootdirectory.aspx) | Gets the root directory of a drive |
| [TotalFreeSpace](http://msdn.microsoft.com/en-us/library/system.io.driveinfo.totalfreespace.aspx) | Gets the total amount of free space available on a drive |
| [TotalSize](http://msdn.microsoft.com/en-us/library/system.io.driveinfo.totalsize.aspx) | Gets the total size of storage space on a drive |
| [VolumeLabel](http://msdn.microsoft.com/en-us/library/system.io.driveinfo.volumelabel.aspx) | Gets or sets the volume label of a drive |

These are the methods:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| [GetDrives](http://msdn.microsoft.com/en-us/library/system.io.driveinfo.volumelabel.aspx) | Returns the drive name of all drives on a computer |

Here’s a simple program to exercise the Drive class:

Imports System.IO

'Chapter 10 - Program 3

Module Module1

Sub Main()

Dim AllDrives() As DriveInfo = DriveInfo.GetDrives()

Dim aDrive As DriveInfo

For Each aDrive In AllDrives

Console.WriteLine("Drive {0}", aDrive.Name)

Console.WriteLine("File type: {0}", aDrive.DriveType)

If aDrive.IsReady = True Then

Console.WriteLine("File system: {0}", aDrive.DriveFormat)

Console.WriteLine("Volume label: {0}", aDrive.VolumeLabel)

Console.WriteLine("Available space to current user:{0, 15} bytes",

aDrive.AvailableFreeSpace)

Console.WriteLine("Total available space: {0, 15} bytes",

aDrive.TotalFreeSpace)

Console.WriteLine("Total size of drive: {0, 15} bytes",

aDrive.TotalSize)

Console.WriteLine(" ")

End If

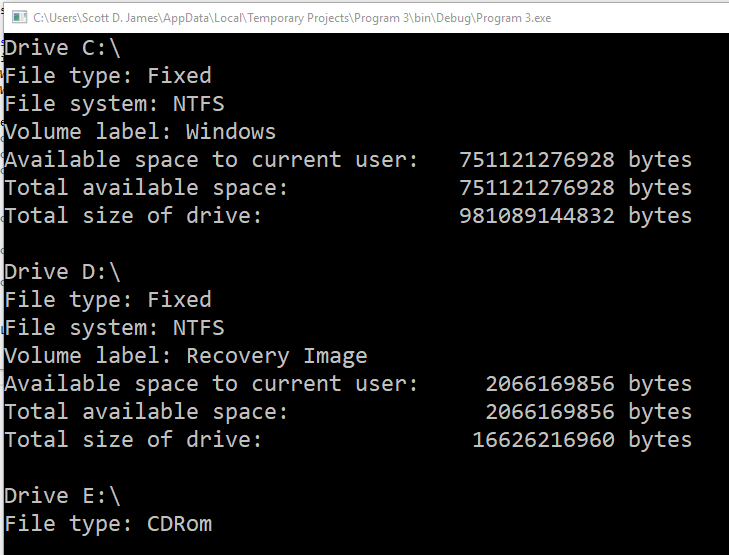
Next

Console.ReadLine()

End Sub

End Module

Here’s the output from DriveInfo program:



This section is by no means an exhaustive list of all the various I/O classes that are available in VB.NET. For example, there are DirectoryInfo, FileInfo and FileSystemInfo classes which are convenient for manipulating collections of directories and files. There are also classes that can be used specifically with file attributes and so forth. However, if you understand the DriveInfo, Directory, File and Path classes, you will be able to perform the majority of the O/S I/O related work that you need and you can consult the MSDN documentation for information on the other classes that were not detailed here.

The last comment I want to make regarding the filesystem related System.IO classes is that there is a FileSystemWatcher class. We usually drag and drop one of these as a control from our Toolbox if we are building a GUI, however the class is always available even in a non-graphical form. It allows us to raise events should something change on the filesystem, like a file/directory gets changed, created, deleted or renamed. Kind of handy, especially if you are building a Windows service that needs to watch for something and respond to filesystem activity.

# Working with Sequential Access Files

If you have worked with text files in any programming language, you essentially viewed a text file as a stream of characters that was read from a disk into your program, or as a stream of characters sent from your program to be written out to a disk. There is no true concept of “records” in a text file.

To work with text files in VB.NET, we will need to use a combination of the File class discussed above and the StreamReader and StreamWriter classes. StreamReader and StreamWriter are classes specifically designed to handle text streams. This is everything that will be required to read and write common text files. The StreamReader/StreamWriter classes are based off the TextReader/TextWriter classes which are also found in the System.IO namespace.

The next four tables will outline the methods and properties of the StreamWriter and StreamReader classes.

StreamWriter class methods:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| Close | Closes the current StreamWriter object, flushing data buffers and updating the system settings |
| Flush | Forces the contents of a stream buffer out to its related file and then empties the buffer |
| FlushAsync | Clears all buffers for this stream asynchronously, writing any buffered data out to the underlying stream |
| Write | Writes data to the stream |
| WriteAsync | Writes data to the stream asynchronously |
| WriteLine | Writes data to the stream followed by a line terminator |
| WriteLineAsync | Writes data to the stream followed by a line terminator asynchronously |

StreamWriter class properties:

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| AutoFlush | Get or set a Boolean value indicating if the StreamWriter flushes its buffer after each call to the .Write or .WriteLine methods |
| BaseStream | Gets a stream object that provides the StreamWriter with a backing store (file) |
| Encoding | Gets an Encoding object that corresponds with the stream's underlying encoding |
| FormatProvider | Gets an object that control formatting |
| NewLine | Gets or sets the line terminator string used by the current StreamWriter (good for creating Unix files!) |

StreamReader class methods:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| Close | Closes the StreamReader object and releases system resources |
| DiscardBufferedData | Allows a StreamReader to discard data that is beyond the end of the stream |
| Peek | Returns the next character in the stream but does not consume the character |
| Read | Reads the next character (or specified number of chars) from the stream |
| ReadAsync | Reads the next character (or specified number of chars) from the stream asynchronously |
| ReadBlock | Reads a specified maximum number of characters from the current stream into a buffer |
| ReadBlockAsync | Reads a specified maximum number of characters from the current stream into a buffer asynchronously |
| ReadLine | Reads characters from the stream up to the end of line |
| ReadLineAsync | Reads characters from the stream up to the end of line asynchronously |
| ReadToEnd | Reads the characters from the stream through the end of the stream |
| ReadToEndAsync | Reads the characters from the stream through the end of the stream asynchronously |

StreamReader class properties:

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| BaseStream | Gets a stream object that provides the StreamWriter with a backing store |
| CurrentEncoding | Gets an encoding object that corresponds to the stream's underlying encoding |
| EndOfStream | Gets a value that indicates whether the current stream position is at the end of the stream |

The easiest way to explain how to use the File, StreamWriter and StreamReader classes is through a program. Here’s an example application which shows how to create a directory and then a sequential file, write some text out to the file and then read the text back in. The program cleans up before terminating by deleting the file and directory that were created. We will use a space as a delimiter between each item on a line and terminate each line with the default Windows CR/LF character. Here’s the program:

Imports System.IO

'Chapter 10 - Program 4

Module Module1

Sub Main()

Dim txtName As String

Dim txtID As String

Dim txtRate As String

Dim aLine As String

Dim strValues As String()

'Create a temporary directory off of the root -- we will get

'an error if we try to write directly to the root.

Directory.CreateDirectory("C:\VBTemp\")

'Create a StreamWriter and have it create/overwrite a file

Dim myOutput As New StreamWriter("C:\VBTemp\testit.txt")

'NOTE: If I hadn't included a path in the filename, it would

'have been written into the VB project's .\Bin\Debug\ folder.

'We'll need this StreamReader later

Dim myInput As StreamReader

Debug.WriteLine("Created File")

'Let's write some lines of text out to the File

Debug.WriteLine("Writing Record 1")

myOutput.WriteLine("Scott" & " " & "101" & " " & "15.00")

Debug.WriteLine("Writing Record 2")

myOutput.WriteLine("Sue" & " " & "304" & " " & "12.75")

Debug.WriteLine("Writing Record 3")

myOutput.WriteLine("Bill" & " " & "407" & " " & "10.55")

'Let's close the file down

Debug.WriteLine("Closing File")

myOutput.Close()

'Now prepare the StreamReader for reading by creating

'a new instantiation which opens the text file we

'just created above

myInput = New StreamReader("C:\VBTemp\testit.txt")

Debug.WriteLine(" ")

Debug.WriteLine("Opening File for Reading")

'Process all of the lines using a loop

Do

'Try to read in a line of text

aLine = myInput.ReadLine()

'If we were able to read in a line,

'then use Split to tear the line back

'into the three component values we wrote

If (aLine <> Nothing) Then

strValues = Split(aLine)

txtName = strValues(0)

txtID = strValues(1)

txtRate = strValues(2)

'Show the three pieces of data from the text line

Debug.WriteLine("Name: " & txtName & vbCrLf & "ID: " & txtID &

vbCrLf & "Rate: " & txtRate)

End If

Loop While (aLine <> Nothing)

'Close the input file

Debug.WriteLine("Closing File")

myInput.Close()

'Delete the input file

File.Delete("C:\VBTemp\testit.txt")

'Delete the temporary directory

Directory.Delete("C:\VBTemp\")

End Sub

End Module

Let's take a look at the program in operation via its output:

Created File

Writing Record 1

Writing Record 2

Writing Record 3

Closing File

Opening File for Reading

Name: Scott

ID: 101

Rate: 15.00

Name: Sue

ID: 304

Rate: 12.75

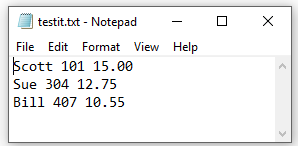
Name: Bill

ID: 407

Rate: 10.55

Closing File

It’s important to note that we could have could have also written one item out per line and then read the whole line in at one time to get the information instead of splitting – File I/O doesn’t get much easier than this! Also, if you comment out the File.Delete and Directory.Delete statements at the end of the program, you could then take a look at the file in Notepad or some other file editing/viewing program:



# Handling Random Access Files

Random access files have the advantage of allowing record-type structures to be recalled/stored with each read/write operation. This means that for a sequential access file, the most atomic element is a character, whereas for a random access file, the most atomic element is an entire logical record. Since we can read or write so much more with each read/write operation, random access files are simply far more efficient to process than sequential files.

Each record in a random access file is exactly the same size as any other record. The file is typically in a binary format however, and not text – this is another advantage. Integers, floating point numbers and other types of information are stored directly in the computer’s native format. Due to this fact, we do not have to read information from a random access file in as a string and then convert it into some other format before it becomes usable. The only real downside to a binary file is that it is not human readable. If you have ever accidentally (or maybe on purpose) opened up a graphic image, sound file or program file in Notepad or Microsoft Word and saw a bunch of junk on the screen, you know what trying to read a binary file in a text editor looks like.

The final advantage that random access files provide us over sequential access files is that random access files allow any record to be directly accessed by "jumping" to that record. Sequential files force us to resort to a line-by-line sequential access type of reading. This is obviously the underpinnings of how a database works and why it’s able to return results so quickly.

Random access files are not supported by the StreamReader and StreamWriter classes; those classes are intended specifically for text streams. Instead, when we want to work with random access files, we must use the BinaryReader and BinaryWriter classes.

The next two tables detail the BinaryWriter and BinaryReader class methods.

BinaryWriter class methods:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| Close | Closes the current BinaryWriter object, flushing data buffers and updating system settings |
| Flush | Forces the contents of a stream's buffers to the file and empties the buffer |
| Seek | Moves the stream pointer to the specified offset |
| Write | Writes data to the stream (see the various forms in the BinaryReader Read methods) |

BinaryReader class methods:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| Close | Closes the BinaryReader and releases system resources |
| FillBuffer | Fills the internal buffer with the specified number of bytes read from the stream |
| PeekChar | Returns the next character in the stream, but does not consume the character |
| Read | Reads the next character or specified number of characters from the stream |
| ReadBoolean | Reads a Boolean value from the stream, advancing the stream pointer one byte |
| ReadBytes | Reads a Byte value from the stream, advancing the stream pointer one byte |
| ReadBytes | Reads the specified number of bytes from the stream into an array, advancing the stream pointer by the same number of bytes |
| ReadChar | Reads a Char value from the stream, advancing the stream pointer one character (the size will be based on the character’s encoding) |
| ReadChars | Reads the specified number of chars from the stream into an array, advancing the stream pointer by the same number of character bytes |
| ReadDecimal | Reads a decimal value from the stream, advancing the stream pointer by 16 bytes |
| ReadDouble | Reads a floating point number from the stream, advancing by 8 bytes |
| ReadInt16 | Reads a signed integer from the stream, advancing by 2 bytes |
| ReadInt32 | Reads a signed integer from the stream, advancing by 4 bytes |
| ReadInt64 | Reads a signed integer from the stream, advancing by 8 bytes |
| ReadSByte | Reads a signed byte from the stream, advancing by one byte |
| ReadSingle | Reads a floating point number from the stream, advancing by 4 bytes |
| ReadString | Reads a String from the stream, advancing the stream pointer based on the string's encoded length |
| ReadUInt16 | Reads an unsigned integer from the stream, advancing by 2 bytes |
| ReadUInt32 | Reads an unsigned integer from the stream, advancing by 4 bytes |
| ReadUInt64 | Reads an unsigned integer from the stream, advancing by 8 bytes |

Here's one more aside that you need to know before using these new classes: you can't just send a filename through to a BinaryReader or BinaryWriter objecy. You will have to create a FileStream (guess what, StreamReader and StreamWriter are based on it).

The FileStream does two things for you: (1) you can create and open files and (2) you can move the file pointer to any location you want – this is a requirement if we are going to read a file randomly! Let's examine an application that creates a random access file, writes some records out to it and then randomly reads the records back in. Here’s the code:

Imports System.IO

'Chapter 10 - Program 5

Module Module1

Sub Main()

'We can't just send a filename string through to BinaryWriter

' -- we have to manually create a stream and then send the

'stream to BinaryWriter. Notice the FileMode.Create to actually

'create the file. Also notice no pathing on the filename, which

'means it will be created in the same directory as the executable.

Dim myStream As New FileStream("testit2.bin",

FileMode.Create)

'Attach the BinaryWriter to the stream we just created

Dim myOutput As New BinaryWriter(myStream)

'We will need the BinaryReader later, so declare it now

Dim myInput As BinaryReader

'A file record will consist of a 5 character long string,

'an integer and a single...

Dim txtName As String

Dim intID As Integer

Dim sngRate As Single

Debug.WriteLine("Created File")

Debug.WriteLine("")

'Write some records out to the file

Debug.WriteLine("Writing Record 1")

myOutput.Write("Scott")

myOutput.Write(Convert.ToInt32(101))

myOutput.Write(Convert.ToSingle(15.0))

'Notice the names are all padded to be 5 characters long

'since random access works by ensuring all records are

'exactly the same length...

Debug.WriteLine("Writing Record 2")

myOutput.Write("Sue ")

myOutput.Write(Convert.ToInt32(304))

myOutput.Write(Convert.ToSingle(12.75))

Debug.WriteLine("Writing Record 3")

myOutput.Write("Bill ") 'only one space pad needed here

myOutput.Write(Convert.ToInt32(407))

myOutput.Write(Convert.ToSingle(10.55))

'Close the file up

myOutput.Close()

Debug.WriteLine("Closing File")

Debug.WriteLine("")

'Now let's open the file back up by resetting the FileStream

'and using FileMode.Open

myStream = Nothing

myStream = New FileStream("testit2.bin", FileMode.Open)

myInput = New BinaryReader(myStream)

'let's figure out how long a record is -- notice all strings

'were forced to be the same length (5 chars + 1 byte for size)

' + an int32 (4 bytes) + a single (4 bytes)

'So a record is 14 bytes long -- let's print the file out in

'alphabetic order. This means we need to visit records in the

'order of record 3, then 1 and finally 2 -- so we will use

'seek to take us to the correct record and print what's there

'Record 1 is at offset 0 in the file – so any record we want

'can simply be found by taking the (record number – 1) \*

'(the size of a record)

'Prove we can read randomly, by jumping around

'Read in the last record we wrote out

myStream.Seek(2 \* 14, SeekOrigin.Begin)

txtName = myInput.ReadString()

intID = myInput.ReadInt32()

sngRate = myInput.ReadSingle()

Debug.WriteLine(txtName & " " & intID & " " & sngRate)

'Read in the first record we wrote out

myStream.Seek(0 \* 14, SeekOrigin.Begin)

txtName = myInput.ReadString()

intID = myInput.ReadInt32()

sngRate = myInput.ReadSingle()

Debug.WriteLine(txtName & " " & intID & " " & sngRate)

'Read in the middle record we wrote out

myStream.Seek(1 \* 14, SeekOrigin.Begin)

txtName = myInput.ReadString()

intID = myInput.ReadInt32()

sngRate = myInput.ReadSingle()

Debug.WriteLine(txtName & " " & intID & " " & sngRate)

'Close the file up

Debug.WriteLine("Closing file")

myInput.Close()

'Delete the file

File.Delete("testit2.bin")

End Sub

End Module

Here's the program output:

Created File

Writing Record 1

Writing Record 2

Writing Record 3

Closing File

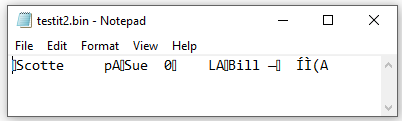
Bill 407 10.55

Scott 101 15

Sue 304 12.75

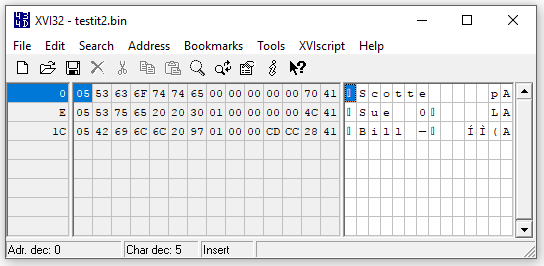
Closing file

You can see from the output that we successfully wrote and read the three records. In case you have never seen a binary file before, here's a look inside of the file we just created:



As you can see, the only thing human readable in the file are the strings containing each person's name. The Integer and Single values are just garbled bytes that mean nothing to us. The computer on the other hand, knows exactly what these values mean since they are stored in the computer’s internal format and do not have to be converted.

Using a hex editor to look at the file provides us with a bit more information. We can see that every fourteen bytes, starting from byte 0, we have a record. We can also see the marker placed in front of each string showing the string’s length followed by the string characters themselves. Unfortunately, the Integer and Single values are still just as cryptic to us:



# Serialization

Now if you didn’t know it, .NET doesn’t let you write an entire structure to a BinaryWriter file. It would have been a whole lot more convenient in the last program to just say “Hey VB: write out a whole structure” and then define a structure with the three pieces of data that we worked with. You’ll notice that I had to write out each value in turn, just ensuring the sizes were the same so that I could seek to a particular record later on. This is kind of a bummer because many languages will allow you to write entire structures at a time. However, VB does provide a workaround: serialization.

Serialization is the concept of having the programming language prepare some structure to be written to a file in one shot. I am not going to go into the gory details of all that can be done with serialization, but I will show you enough to make it possible to read and write an entire structure at a time.

The following code example basically reworks the previous program but is now using structures instead. As promised, we will read and write entire structures at a time. You’ll notice that we had to bring in a few new namespaces to allow the serialization to take place and to mark up our structure with a decorator:

Imports System.IO

Imports System.Runtime.Serialization.Formatters.Binary

Imports System.Runtime.Serialization

'Chapter 10 - Program 6

Module Module1

'This next line puts a special decorator on the structure

'marking it as "serializable," meaning that it can be written

'out to (or read in from) the file as a single item

<Serializable()> Structure Person

Dim strName As String

Dim intID As Integer

Dim sngRate As Single

End Structure

Sub Main()

'BinaryWriters won’t work here – we need to basically be able to

'send entire binary chunks at a time. We’ll chain into the FileStream

'class and use the BinaryFormatter class to Serialize and Deserialize

'the structures for us.

Dim myStream As New FileStream("testit3.bin",

FileMode.Create)

'Used to serialize the Person structure into a writable element

Dim myFormatter As New BinaryFormatter

'Create an instance of our Person structure that we will fill

'and write to the file

Dim myPerson As Person

'We’ll also need to know how big one of our serialized structure

'is when we want to read the structures back in...

Dim intSerializedStructureSize As Integer

Debug.WriteLine("Created File")

Debug.WriteLine("")

'Write some records out to the file

Debug.WriteLine("Writing Record 1")

With myPerson

.strName = "Scott"

.intID = 101

.sngRate = 15.0

End With

'Convert the myPerson structure over into a binary format so

'that it can be written to the file.

myFormatter.Serialize(myStream, myPerson)

'How big is the serialized structure? We'll need to know later on...

intSerializedStructureSize = myStream.Position

'Notice the names are again padded to be 5 characters long

Debug.WriteLine("Writing Record 2")

With myPerson

.strName = "Sue "

.intID = 304

.sngRate = 12.75

End With

myFormatter.Serialize(myStream, myPerson)

Debug.WriteLine("Writing Record 3")

With myPerson

.strName = "Bill "

.intID = 407

.sngRate = 10.55

End With

myFormatter.Serialize(myStream, myPerson)

'Close the file up

Debug.WriteLine("Closing File")

Debug.WriteLine("")

myStream.Close()

'Now let's open the file back up by resetting the FileStream

'and using FileMode.Open

myStream = Nothing

myStream = New FileStream("testit3.bin", FileMode.Open)

'We can loop through the entire file, reading structure

'after structure until we hit the end. Notice there is no

'EndOfStream operator, so we have to get smart about how

'to detect it. Basically, if there are more bytes still in

'the file than the size of a serialized structure, we haven't

'read everything in yet.

While (myStream.Position < myStream.Length)

'Deserialize a structure from the file

myPerson = myFormatter.Deserialize(myStream)

With myPerson

'Print out what we read in...

Debug.WriteLine(.strName & " " & .intID & " " & .sngRate)

End With

End While

'Show that we can still jump around, e.g. this is random access

'Go back to beginning of file and jump to a record

myStream.Position = 0

'Seek the second record

Debug.WriteLine("")

Debug.WriteLine("Seeking and printing just record #2")

myStream.Seek(intSerializedStructureSize \* 1, SeekOrigin.Current)

myPerson = myFormatter.Deserialize(myStream)

With myPerson

Debug.WriteLine(.strName & " " & .intID & " " & .sngRate)

End With

'Close the file up

Debug.WriteLine("Closing file")

myStream.Close()

'Delete the file

File.Delete("testit3.bin")

End Sub

End Module

Here’s the output:

Created File

Writing Record 1

Writing Record 2

Writing Record 3

Closing File

Scott 101 15

Sue 304 12.75

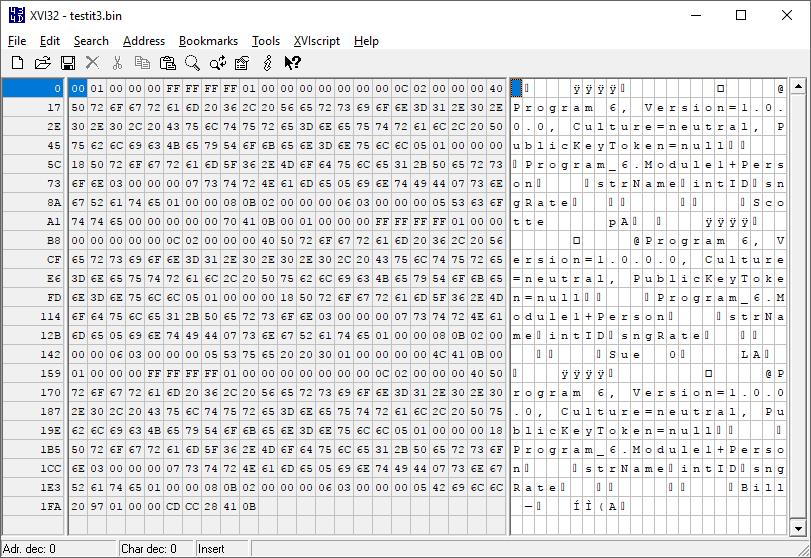
Bill 407 10.55

Seeking and printing just record #2

Sue 304 12.75

Closing file

If we take a look at the binary file, we can see that it’s got a whole lot more stuff in it than the previous version did. This is due to the serializer’s role in making things work:



Serialization is not something available in just VB, all of the .NET languages support it as do other modern programming languages like Java. The beauty about serialization is that we can serialize just about anything we want: Classes, Arrays, Lists – you name it. The penalty obviously is that we’re pretty limited with the interactions we have with the file. Nothing is really there to tell us how big our serialized objects are, and if we seek wrong, we will get an exception and gibberish. This is why most developers still persist their data to databases, which we will be examining in a future chapter. And, there is also a way to automatically get VB to write database persistence for object models that you create – more on that later…

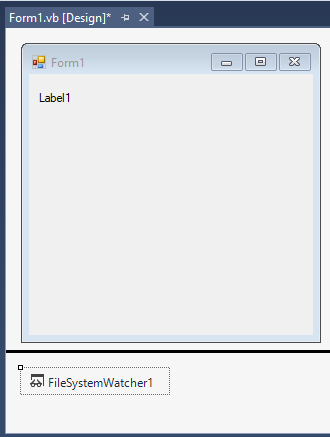
I need to mention a couple more things before finishing this section up. Just as with the Directory and File classes, there are many more classes related to stream processing available in the System.IO namespace. There are BufferedStream, MemoryStream, StringReader, StringWriter & UnmanagedMemoryStream classes. From the names, you ought to be able to guess the general purpose of each of those classes. This is on top of all the things that are possible to do with the FileStream class. The purpose of this section wasn't to make you a pro at VB.NET I/O, but rather to be able to perform the basic file processing that most applications require.

Finally, there are still many pre-.NET file manipulation techniques available in Visual Basic today. I would recommend shying away from them (I didn’t cover any of them) to stay as forward compatible as possible with the code you write. You also have access to making API (Application Programmer Interface) calls below the scenes from the O/S to handle anything you can possibly imagine, but I would really question what you are trying to do that can’t be done in VB natively.

# The FileSystemWatcher Class

This class is useful for notifying you when something changes on the filesystem. For example, you may want to know when a new file appears or if a directory is deleted – that’s the purpose behind the FileSystemWatcher.

The FileSystemWatcher is an invisible control, so if you drag and drop it to your form, it will show up in the control tray under your form:



I want the program to watch the C:\VBTemp directory for any new .TXT files that show up. If one gets created, the label on the form will be updated to show that it occurred. Here’s the code:

Public Class Form1

'Chapter 10 - Program 7

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles Me.Load

'Watch the C:\VBTemp directory

FileSystemWatcher1.Path = "C:\VBTemp"

'Watch for creationtime changes on files only (think of the Or as a +)

FileSystemWatcher1.NotifyFilter = (System.IO.NotifyFilters.CreationTime Or

System.IO.NotifyFilters.FileName)

'Only watch text (.TXT) files.

FileSystemWatcher1.Filter = "\*.txt"

End Sub

Private Sub FileSystemWatcher1\_Created(sender As Object, e As

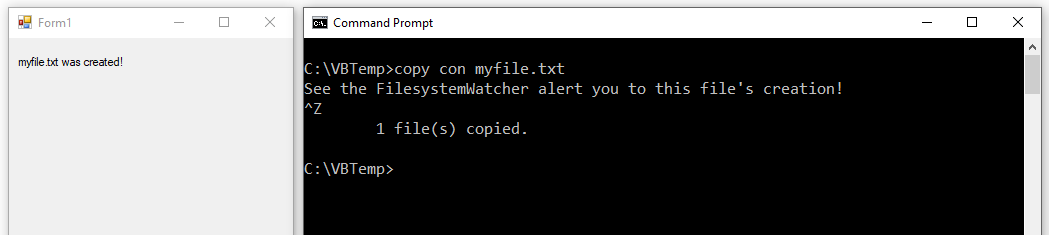
IO.FileSystemEventArgs) Handles FileSystemWatcher1.Created

Label1.Text = e.Name & " was created!"

End Sub

End Class

I started up a command session and created a new file – notice the FileSystemWatcher caught it:



These are the FileSystemWatcher properties we are interested in:

|  |  |
| --- | --- |
| *Name* | *Description* |
| [EnableRaisingEvents](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.enableraisingevents.aspx) | Gets or sets a value indicating whether the component is enabled. |
| [Filter](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.filter.aspx) | Gets or sets the filter string used to determine what files are monitored in a directory. |
| [IncludeSubdirectories](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.includesubdirectories.aspx) | Gets or sets a value indicating whether subdirectories within the specified path should be monitored. |
| [NotifyFilter](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.notifyfilter.aspx) | Gets or sets the type of changes to watch for. |
| [Path](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.path.aspx) | Gets or sets the path of the directory to watch. |
| [SynchronizingObject](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.synchronizingobject.aspx) | Gets or sets the object used to marshal the event handler calls issued as a result of a directory change. |

These are the methods:

|  |  |
| --- | --- |
| *Name* | *Description* |
| [BeginInit](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.begininit.aspx) | Begins the initialization of a FileSystemWatcher used on a form or used by another component. The initialization occurs at run time. |
| [EndInit](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.endinit.aspx) | Ends the initialization of a FileSystemWatcher used on a form or used by another component. The initialization occurs at run time. |
| [WaitForChanged(WatcherChangeTypes)](http://msdn.microsoft.com/en-us/library/67220zhk.aspx) | A synchronous method that returns a structure that contains specific information on the change that occurred, given the type of change you want to monitor. |
| [WaitForChanged(WatcherChangeTypes, Int32)](http://msdn.microsoft.com/en-us/library/s3tf46w9.aspx) | A synchronous method that returns a structure that contains specific information on the change that occurred, given the type of change you want to monitor and the time (in milliseconds) to wait before timing out. |

Finally, here are the events we can subscribe to:

|  |  |
| --- | --- |
| *Name* | *Description* |
| [Changed](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.changed.aspx) | Occurs when a file or directory in the specified [Path](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.path.aspx) is changed. |
| [Created](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.created.aspx) | Occurs when a file or directory in the specified [Path](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.path.aspx) is created. |
| [Deleted](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.deleted.aspx) | Occurs when a file or directory in the specified [Path](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.path.aspx) is deleted. |
| [Error](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.error.aspx) | Occurs when the instance of FileSystemWatcher is unable to continue monitoring changes or when the internal buffer overflows. |
| [Renamed](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.renamed.aspx) | Occurs when a file or directory in the specified [Path](http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.path.aspx) is renamed. |