Working with the File System

Note : Before you practically begin with this lecture you should include the following code to your program

```
using System.IO;
```

Applications often need to store data to the disk in order to save data between sessions, log data for troubleshooting or auditing, or communicate with other applications. This lecture describes how to examine and manage the file system and read and write files.

The .NET Framework includes classes for performing basic file management tasks, including browsing drives, managing files and folders, and responding to changes to the file system. Here we'll describe the most useful classes for managing the file system.

Enumerating Drives

To list all the drives connected to a computer, use the static *DriveInfo.GetDrives()* method (in the *System.IO* namespace) to retrieve a collection of *DriveInfo* objects. For example, the following loop outputs a list of all drives to the console:

```
private void button1_Click(object sender, EventArgs e)
        {
            foreach (DriveInfo di in DriveInfo.GetDrives())
                listBox1.Items.Add(di.DriveType + " " + di.Name);

        }
```

***DriveInfo* has the following properties:**

- ***AvailableFreeSpace*** Indicates the amount of available free space on a drive
- ***DriveFormat*** Gets the name of the file system, such as NTFS or FAT32
- ***DriveType*** Gets the drive type

- *IsReady* Indicates whether a drive is ready
- *Name* Gets the name of a drive
- *RootDirectory* Gets the root directory of a drive
- *TotalFreeSpace* Gets the total amount of free space available on a drive
- *TotalSize* Gets the total size of storage space on a drive
- *VolumeLabel* Gets or sets the volume label of a drive

Managing Files and Folders

The .NET Framework provides classes that you can use to browse files and folders, create new folders, and manage files. The following sections describe how to use these classes.

Browsing Folders

You can use the *DirectoryInfo* class to browse folders and files. First, create an instance of *DirectoryInfo* by specifying the folder to browse. Then, call the *DirectoryInfo.GetDirectories* or *DirectoryInfo.GetFiles* method. The following example displays the files and folders in the C:\Windows\folder.

```
private void button2_Click(object sender, EventArgs e)
        {
            DirectoryInfo dir = new DirectoryInfo(@"C:\Windows");

            foreach (DirectoryInfo dirInfo in dir.GetDirectories())
                listBox1.Items.Add(dirInfo.Name);
            foreach (FileInfo fi in dir.GetFiles())
                listBox2.Items.Add(fi.Name);

        }
```

Creating Folders

To create folders, create an instance of *DirectoryInfo* and then call the *DirectoryInfo.Create* method. You can check the boolean *DirectoryInfo.Exists* property to determine if a folder already exists. The following sample checks for the existence of a folder and creates it if it doesn't already exist, although the Common

Language Runtime (CLR) does not throw an exception if you attempt to create a folder that already exists.

```
private void button3_Click(object sender, EventArgs e)
        {
            DirectoryInfo newDir = new DirectoryInfo(@"C:\deleteme");
            if  (newDir.Exists)
                MessageBox.Show("The folder already exists");
            else
                newDir.Create();
        }
```

Creating, Copying, Moving, and Deleting Files

To create, copy, move, and delete files, you can use the static *File.Create*, *File.CreateText*, *File.Copy*, *File.Move*, and *File.Delete* methods. The following sample creates a file, copies it, and then moves/renames it:

```
private void button4_Click(object sender, EventArgs e)
        {
            File.CreateText(@"D:\mynewfile.txt");
            File.Copy("mynewfile.txt", @"D:\newfile2.txt");
            File.Move(@"D:\newfile2.txt", @"D:\newfile3.txt");
        }
```

To delete such file :

```
private void button5_Click(object sender, EventArgs e)
        {
            if (File.Exists(@"D:\newfile3.txt"))
                File.Delete(@"D:\newfile3.txt");
            else
                MessageBox.Show("No Such File");
        }
```

Alternatively, you can create an instance of the *FileInfo* class representing the file and call the *Create*, *CreateText*, *CopyTo*, *MoveTo*, and *Delete* methods. The following code performs the same functions as the previous sample:

```
FileInfo fi = new FileInfo("mynewfile.txt");
fi.CreateText();
fi.CopyTo("newfile2.txt");
```

Another example:

```
FileInfo fi2 = new FileInfo("newfile2.txt");
fi2.MoveTo("newfile3.txt");
```

To delete a file, create an instance of the *FileInfo* class and then call *FileInfo.Delete*.

```
Fi.Detele();
```

Monitoring the File System

You can use the *FileSystemWatcher* class (part of the *System.IO* namespace) to respond to updated files, creating new files, renamed files, and other updates to the file system. First, In design Time Drag an instance of *FileSystemWatcher* control and provide the path to be monitored using *Path* property . Then, configure properties of the *FileSystemWatcher* to control whether to monitor subdirectories by making *IncludeSubdirectories* true , and which types of changes to monitor by configuring *NotifyFilter* property . Next, add a method as an event handler.

On the Events of this control you can add the appropriate event to monitoring changes these events are : (*Changed* , *Deleted* , *Created* , and *Renamed*)

```
private void fileSystemWatcher1_Renamed(object sender, RenamedEventArgs e)
        {
            listBox3.Items.Add(e.ChangeType.ToString()+" " + e.Name);
        }

private void fileSystemWatcher1_Deleted(object sender, FileSystemEventArgs e)
        {
            listBox3.Items.Add(e.ChangeType.ToString()+" " + e.Name);
        }
```

Configuring *FileSystemWatcher* Properties

You can configure the following properties of the *FileSystemWatcher* class to control which types of updates cause the CLR to throw the *Changed* event:

- ***Filter*** Used to configure the filenames that trigger events. To watch for changes in all files, set the *Filter* property to an empty string ("") or use wildcards ("*.*"). To watch a specific file, set the *Filter* property to the filename. For example, to watch for changes in the file MyDoc.txt, set the *Filter* property to "MyDoc.txt". You can also watch for changes in a certain type of file. For example, to watch for changes in text files, set the *Filter* property to "*.txt".
- ***NotifyFilter*** Configure the types of changes for which to throw events by setting *NotifyFilter* to one or more of these values:
    - *FileName*
    - *DirectoryName*
    - *Attributes*
    - *Size*
    - *LastWrite*
    - *LastAccess*
    - *CreationTime*
    - *Security*
- ***Path*** Used to define the folder to be monitored. You can define the path using the *FileSystemWatcher* constructor.

You can watch for the renaming, deletion, or creation of files or directories. For example, to watch for the renaming of text files, set the *Filter* property to "*.txt" and call the *WaitForChanged* method with a *Renamed* value specified for its parameter.

Reading and Writing Files and Streams

You can use different *Stream* classes to read and write files. The .NET Framework provides different classes for text files and binary files, and specialized classes to allow you to compress data or store data in memory. You can also use streams to store files in isolated storage, a private file system managed by the .NET Framework.

Reading and Writing Text Files

To read a text file, create an instance of TextReader or StreamReader. Then, call one of the class's methods to read as much or as little of the file as you want. For example, the following sample code displays a text file to a *TextBox*.

```
private void button6_Click(object sender, EventArgs e)
      {
          TextReader tr = File.OpenText(@"C:\windows\win.ini");
          textBox1.Text =tr.ReadToEnd();
          tr.Close();

      }
```

The previous example uses the *TextReader* class, but it works equally well with the StreamReader class (which counterintuitively derives from TextReader). When using TextReader, you typically create an instance using the static File.OpenText method (as demonstrated in the previous example). When using StreamReader, you can use File.OpenText or the StreamReader constructor. The following example displays the same text file using an instance of StreamReader and a while loop:

```
private void button7_Click(object sender, EventArgs e)
      {
          StreamReader sr = new StreamReader(@"C:\windows\win.ini");
          string input;
          while ((input = sr.ReadLine()) != null)
              textBox1.Text += input + '\r'+'\n' ;
          sr.Close();
      }
```

When reading text files, you typically use *ReadLine* or *ReadToEnd* methods.

To write a text file, create an instance of *TextWriter* or *StreamWriter*. After creating an instance of the class, writing to the file is done using *Write* method , call *Flush* method to output buffer immediately. (although you need to call the *Close* method after you finish writing to the file) . See the following code example:

```
private void button8_Click(object sender, EventArgs e)
        {
            TextWriter tw = File.CreateText(@"D:\output.txt");
            tw.Write(textBox1.Text);
            tw.Flush();
            tw.Close();
        }
```

Reading and Writing Binary Files

You can use binary files and the *BinaryWriter* and *BinaryReader* classes to store and retrieve non-text values. The following code shows how to read and write a series of integers:

```
private void button9_Click(object sender, EventArgs e)
        {
            FileStream fs = new FileStream(@"D:\data.bin", FileMode.Create);
            BinaryWriter w = new BinaryWriter(fs);
            for (int i = 0; i < 11; i++)
                 w.Write((int)i);
            w.Close();
            fs.Close();
        }
private void button10_Click(object sender, EventArgs e)
        {
           FileStream fs = new FileStream(@"D:\data.bin",
                                        FileMode.Open,  FileAccess.Read);
            BinaryReader r = new BinaryReader(fs);
            for (int i = 0; i < 11; i++)
                textBox1.Text += r.ReadInt32().ToString() + '\r' + '\n';
            r.Close();
            fs.Close();
        }
```