Unit 13

16.0 C# I/O System Files and Streams (Text Book Chapter 15)

➤16.0 C# I/O System (Files and Streams)

16.0 C# I/O System – Files and Streams

16.1 Introduction

The C# input/output system framework is designed to be flexible and easily configurable. All C# I/O classes are defined in the **System.IO** namespace. The classes and interfaces defined in the **System.IO** namespace is shown below.

Directory	FileAccess	FileShare	Path	StreamReader	TextReader
DirectoryNotFoundException	FileLoadException	FileStream	PathTooLongException	StreamWriter	TextWriter
DirectoryInfo	FileMode	IOException	SeekOrigin	StringReader	
EndOfStreamException	File	FileNotFoundException	MemoryStream	Stream	StringWriter

Figure 16.1-1 C# I/O System Namespaces

16.2 C# Stream Types

C# supports two types of input/output (I/O) – Stream-IO and Radom Access-IO.

Stream I/O : A stream is a sequence of bytes. Stream-based I/O supports reading or writing data sequentially, that is, reading or writing data successively in one direction. A stream may be opened for reading or writing, but not reading and writing.

Random Access I/O : Random access I/O supports reading and writing data at any position of a stream. A random access stream may be opened for both reading and writing.

16.2.1 Stream IO

There are **two kinds of streams. Byte streams** and **character streams**. Byte streams support reading data and writing data of any type, including strings, in the binary format. Character streams support reading and writing of text, using locale-dependent character encodings.

16.2.1.1 Byte Streams

The most primitive I/O capabilities are declared in two classes: **StreamReader** and **StreamWriter**. They support reading and writing of a single byte and characters.

➢16.0 C# I/O System (Files and Streams)

StreamReader : The **StreamReader** class is designed for character input in a particular Encoding, whereas subclasses of Stream are designed for byte input and output. [Note: StreamReader defaults to UTF-8 encoding unless specified otherwise, instead of defaulting to the ANSI code page for the current system. UTF-8 handles Unicode characters correctly and provides consistent results on localized versions of the operating system.

When reading from a Stream, it is more efficient to use a buffer that is the same size as the internal buffer of the stream. StreamReader hierarchy is shown in Figure 16.2-1

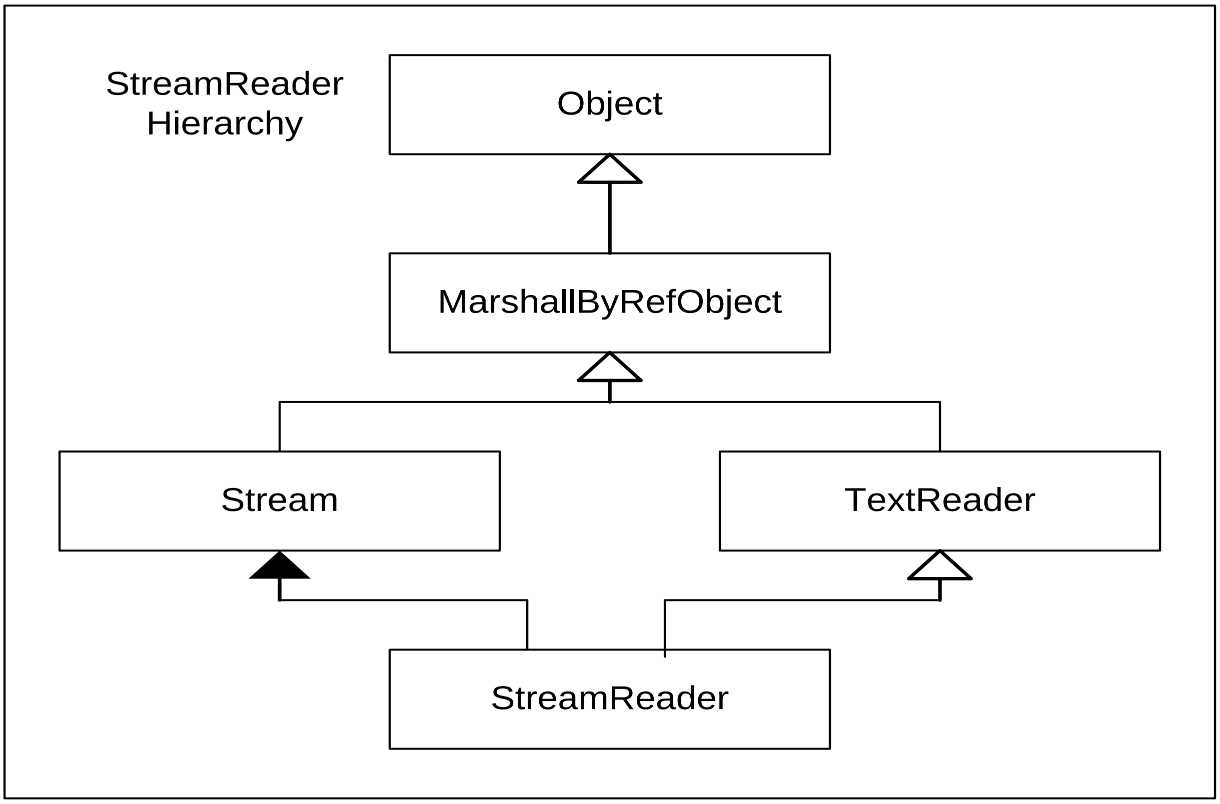


Figure 16.2-2 aStreamReader class hierarchy

Some of the common methods in the **StreamReader** Class is class is shown in Figure 16.2-2. The rest can be viewed here (StreamReader class).

➢16.0 C# I/O System (Files and Streams)

Method Name	Description
Close	Overridden. Closes the StreamReader object and the underlying stream, and releases any system resources associated with the reader.
CreateObjRef	Creates an object that contains all the relevant information required to generate a proxy used to communicate with a remote object. (Inherited from MarshalByRefObject.)
DiscardBufferedData	Allows a StreamReader object to discard its current data.
Dispose	Overloaded.
Equals	Overloaded. Determines whether two Object instances are equal. (Inherited from Object.)
GetType	Gets the Type of the current instance. (Inherited from Object.)
Peek	Overridden. Returns the next available character but does not consume it.
Read	Overloaded. Overridden. Reads the next character or next set of characters from the input stream.
ReadBlock	Reads a maximum of count characters from the current stream and writes the data to buffer, beginning at index. (Inherited from TextReader.)
ReadLine	Overridden. Reads a line of characters from the current stream and returns the data as a string.
ReadToEnd	Overridden. Reads the stream from the current position to the end of the stream.
Synchronized	Creates a thread-safe wrapper around the specified TextReader. (Inherited from TextReader.)
ToString	Returns a String that represents the current Object. (Inherited from Object.)

Figure 16.2-2b StreamReader class methods

StreamWriter: The StreamWriter class is designed for character output in a particular Encoding, whereas subclasses of Stream are designed for byte input and output. StreamWriter defaults to using an instance of UTF8Encoding unless specified otherwise. This instance of UTF8Encoding is constructed such that the System.Text.Encoding.GetPreamble method returns the Unicode byte order mark written in UTF-8. The preamble of the encoding is added to a stream when you are not appending to an existing stream. This means any text file you create with StreamWriter has three byte order marks at its beginning. UTF-8 handles all Unicode characters correctly and gives consistent results on localized versions of the operating system.
StreamWriter hierarchy is shown in Figure 16.2-3a

➢16.0 C# I/O System (Files and Streams)

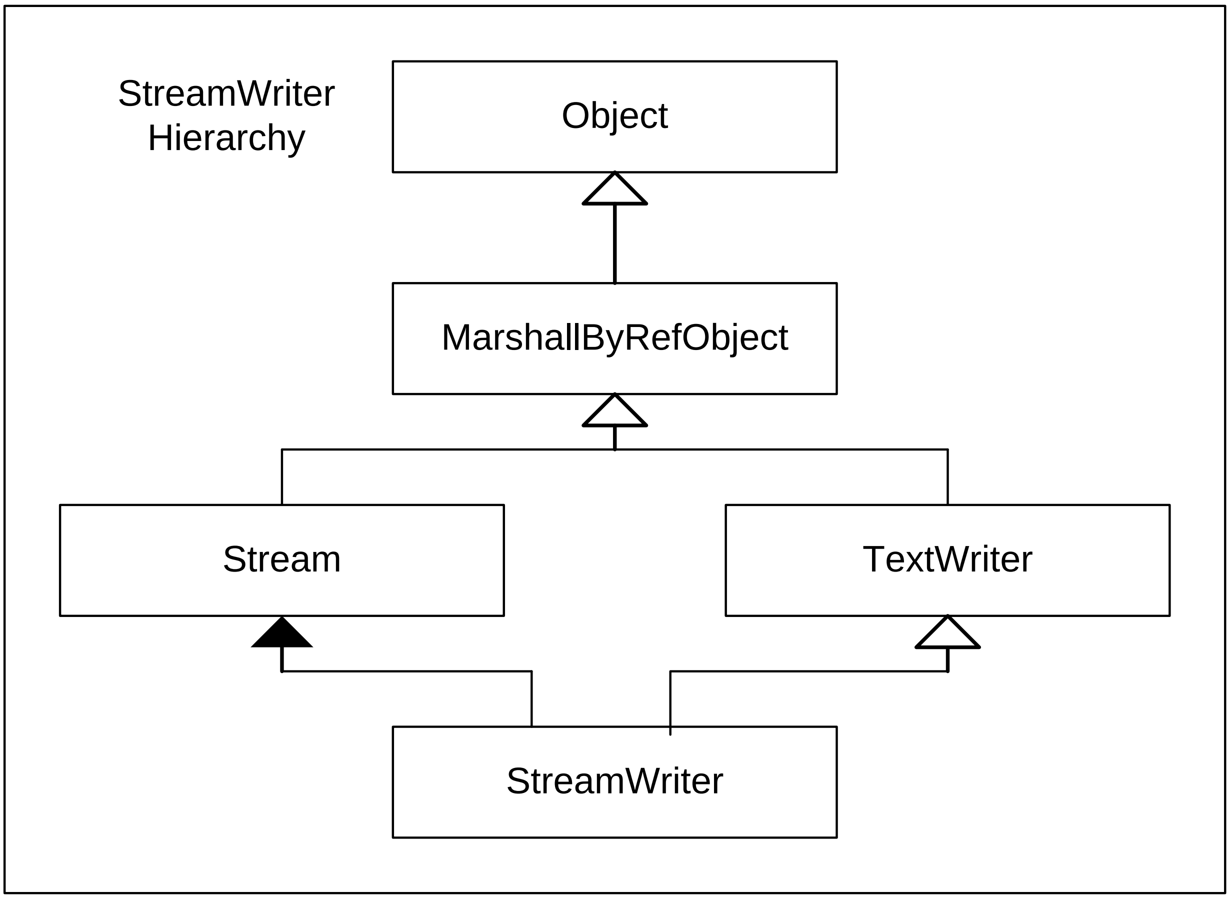


Figure 16.2-3aStreamWriter class Hierarchy

StreamWriter is a large class. It is not worth writing down the method in this class note. Some of the public method of the StreamWriter class is listed in next page Figure 16.2-3b. However, class's constructors, methods etc. can be viewed here (StreamWriter Class) .

The example 16.2.1 shows how both StreamWriter and StreamReader classes can be used along with the classes available for Directory information access. The result of the program is shown on Figure 16.2.4

➢16.0 C# I/O System (Files and Streams)

Method Name	Description
Close	Overridden. Closes the current StreamWriter and the underlying stream.
CreateObjRef(inherited from MarshalByRefObject)	Creates an object that contains all the relevant information required to generate a proxy used to communicate with a remote object.
Equals (inherited from Object)	Supported by the .NET Compact Framework. Overloaded. Determines whether two Object instances are equal.
Flush	Supported by the .NET Compact Framework. Overridden. Clears all buffers for the current writer and causes any buffered data to be written to the underlying stream.
GetHashCode (inherited from Object)	Supported by the .NET Compact Framework. Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetLifetimeService(inherited from MarshalByRefObject)	Retrieves the current lifetime service object that controls the lifetime policy for this instance.
GetType (inherited from Object)	Supported by the .NET Compact Framework. Gets the Type of the current instance.
InitializeLifetimeService (inherited from MarshalByRefObject)	Obtains a lifetime service object to control the lifetime policy for this instance.
ToString (inherited from Object)	Supported by the .NET Compact Framework. Returns a String that represents the current Object.
Write	Supported by the .NET Compact Framework. Overloaded. Overridden. Writes to the stream.
WriteLine(inherited from TextWriter)	Supported by the .NET Compact Framework. Overloaded. Writes some data as specified by the overloaded parameters, followed by a line terminator.

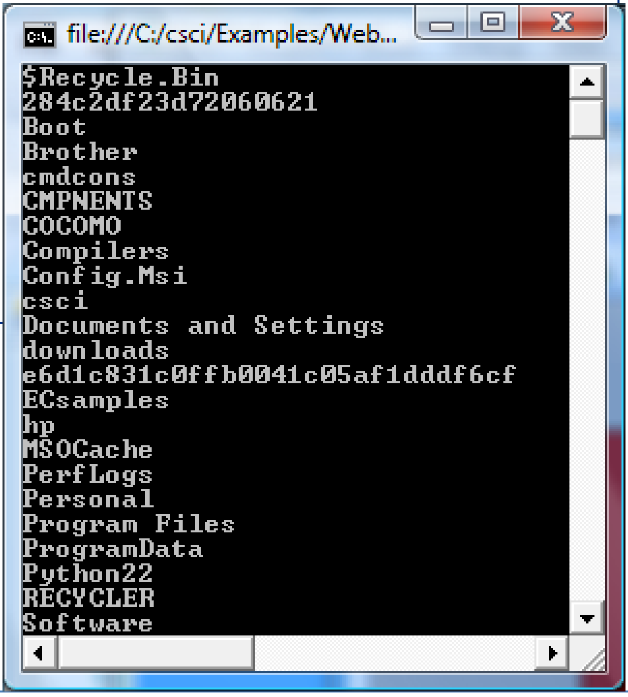
Figure 16.2-3b Public method of the StreamWriter class

The example 16.2.1 shows how both StreamWriter and StreamReader classes can be used along with the classes available for Directory information access. The result of the program is shown on Figure 16.2.4

➢16.0 C# I/O System (Files and Streams)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace StreamReadWrite
{
  class Program
  {
    static void Main(string[] args)
    {// Get the directories currently on the C drive.
     DirectoryInfo[] cDirs = new DirectoryInfo(@"c:\").GetDirectories();
     string fileName =
        @"C:\csci\Examples\WebApplicationDevelopment\CDriveDirs.txt";

     // Write each directory name to a file.
     using (StreamWriter sw = new StreamWriter(fileName ))
     {
       foreach (DirectoryInfo dir in cDirs)
       { sw.WriteLine(dir.Name);   }
     }

      // Read and show each line from the file.
     string line = "";
     using (StreamReader sr = new StreamReader("CDriveDirs.txt"))
     {
       while ((line = sr.ReadLine()) != null)
       {  Console.WriteLine(line); }
     }
     Console.ReadLine();
    }
  }
}
```

Example 16.2-1StreamReader and StreamWriter usage



file:///C:/csci/Examples/Web...

```
$Recycle.Bin
284c2df23d72060621
Boot
Brother
cmdcons
CMPNENTS
COCOMO
Compilers
Config.Msi
csci
Documents and Settings
downloads
e6d1c831c0ffb0041c05af1dddf6cf
ECsamples
hp
MSOCache
PerfLogs
Personal
Program Files
ProgramData
Python22
RECYCLER
Software
```

Figure 16.2-4

➢16.0 C# I/O System (Files and Streams)

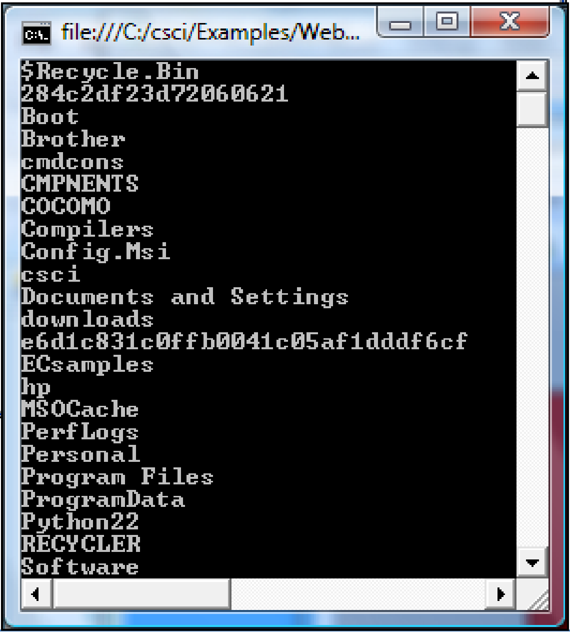
16.2.1.1.1 The File Class

File Class' methods can be used for shortcuts for Reading and Writing.

The File class provides several nice and quick methods to make life much easier. The File class provides two methods WriteAllLines and ReadAllLines that let you write an entire array of strings into a file and then read the entire file back into a string array. The Example 16.2-2 below shows the previous example (Example 16.2-1) now written in a different way but resulting in same. The result of the program is shown on Figure 16.2-2

```
namespace TestFileClass
{
  class Program
  {
    static void Main(string[] args)
    {// Get the directories currently on the C drive.
     string fileName =
        @"C:\csci\Examples\WebApplicationDevelopment\CDriveDirs.txt";

     DirectoryInfo[] cDirs = new DirectoryInfo(@"c:\").GetDirectories();
     string[] dirs = new string[cDirs.Length];
     for (int nextDir = 0; nextDir < dirs.Length; nextDir++)
      {  dirs[nextDir] = cDirs[nextDir].ToString();  }

     // Now write the whole array in one shot.
     File.WriteAllLines(fileName, dirs);

     // Now get written dir list back from file in one shot.
     string [] getBackDirs = File.ReadAllLines(fileName);

     foreach(string dir in getBackDirs)
            Console.WriteLine(dir);

     Console.ReadLine();
  }
}
```

Example 16.2-2StreamReader and StreamWriter usage with File class



```
$Recycle.Bin
284c2df23d72060621
Boot
Brother
cmdcons
CMPNENTS
COCOMO
Compilers
Config.Msi
csci
Documents and Settings
downloads
e6d1c831c0ffb0041c05af1dddf6cf
ECsamples
hp
MSOCache
PerfLogs
Personal
Program Files
ProgramData
Python22
RECYCLER
Software
```

Figure 16.2-5

➢16.0 C# I/O System (Files and Streams)

Some of the other useful methods of the File class is shown in the chart below.

Method Name	Description
ReadAllText(String)	Opens a text file, reads all lines of the file, and then closes the file.
ReadAllText(String, Encoding)	Opens a file, reads all lines of the file with the specified encoding, and then closes the file.
WriteAllLines(String, array<String>[]()[])	Creates a new file, write the specified string array to the file, and then closes the file. If the target file already exists, it is overwritten.
WriteAllLines(String, array<String>[]()[], Encoding)	Creates a new file, writes the specified string array to the file using the specified encoding, and then closes the file. If the target file already exists, it is overwritten.
ReadAllLines(String)	Opens a text file, reads all lines of the file, and then closes the file.
ReadAllLines(String, Encoding)	Opens a file, reads all lines of the file with the specified encoding, and then closes the file.
static byte[] ReadAllBytes(string path)	Opens a binary file, reads the contents of the file into a byte array, and then closes the file.

Figure 16.2-6 File class methods

➢16.0 C# I/O System (Files and Streams)

16.2.1.2 Binary Streams (Binary Encoded Streams)

The most primitive I/O capabilities are declared in two classes: **StreamReader** and **StreamWriter**. They support reading and writing of a single byte and characters.

For files with known internal structures, for example, Binary Files, the BinaryReader and BinaryWriter classes offer streaming functionality that's oriented towards particular data types. The good thing about writing binary files are you cannot read the files by just opening them in the notepad also binary files can be of very large size.

Binary Writing:

Binary writing is provided by the **BinaryWriter** class. The list of methods and their descriptions found in this class is listed in Figure 16.2-7.

Method Name	Description
Close	Closes the current BinaryWriter and the underlying stream.
Dispose	Releases the unmanaged resources used by the BinaryWriter and optionally releases the managed resources.
Equals	Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
Finalize	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection. (Inherited from Object.)
Flush	Clears all buffers for the current writer and causes any buffered data to be written to the underlying device.
GetHashCode	Serves as a hash function for a particular type. (Inherited from Object.)
GetType	Gets the Type of the current instance. (Inherited from Object.)
MemberwiseClone	Creates a shallow copy of the current Object. (Inherited from Object.)
Seek	Sets the position within the current stream.
ToString	Returns a String that represents the current Object. (Inherited from Object.)
Write	Overloaded. Writes a value to the current stream.
Write7BitEncodedInt	Writes a 32-bit integer in a compressed format.

Figure 16.2-7 BinaryWriter class methods

➢16.0 C# I/O System (Files and Streams)

Binary Reading:

Binary reading is provided by the **BinaryReader** class. The list of methods and their descriptions found in this class is listed in Figure 16.2-8

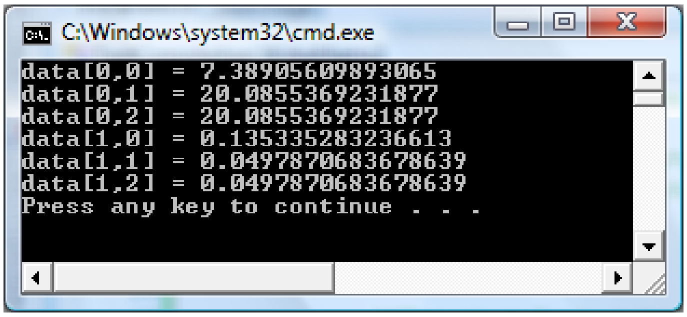
Method	**Description**
Close	Closes the current reader and the underlying stream.
Equals	Determines whether two Object instances are equal.
GetHashCode	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType	Gets the Type of the current instance.
PeekChar	Returns the next available character and does not advance the byte or character position.
Read	Reads characters from the underlying stream and advances the current position of the stream.
ReadBoolean	Reads a Boolean value from the current stream and advances the current position of the stream by one byte.
ReadByte	Reads the next byte from the current stream and advances the current position of the stream by one byte.
ReadBytes	Reads count bytes from the current stream into a byte array and advances the current position by count bytes.
ReadChar	Reads the next character from the current stream and advances the current position of the stream in accordance with the Encoding used and the specific character being read from the stream.
ReadChars	Reads count characters from the current stream, returns the data in a character array, and advances the current position in accordance with the Encoding used and the specific character being read from the stream.
ReadDecimal	Reads a decimal value from the current stream and advances the current position of the stream by sixteen bytes.
ReadDouble	Reads an 8-byte floating point value from the current stream and advances the current position of the stream by eight bytes.
ReadInt16	Reads a 2-byte signed integer from the current stream and advances the current position of the stream by two bytes.
ReadInt32	Reads a 4-byte signed integer from the current stream and advances the current position of the stream by four bytes.
ReadInt64	Reads an 8-byte signed integer from the current stream and advances the current position of the stream by four bytes.
ReadSByte	Reads a signed byte from this stream and advances the current position of the stream by one byte.
ReadSingle	Reads a 4-byte floating point value from the current stream and advances the current position of the stream by four bytes.
ReadString	Reads a string from the current stream. The string is prefixed with the length, encoded as an integer seven bits at a time.
ReadUInt16	Reads a 2-byte unsigned integer from the current stream using little endian encoding and advances the position of the stream by two bytes.
ReadUInt32	Reads a 4-byte unsigned integer from the current stream and advances the position of the stream by four bytes.
ReadUInt64	Reads an 8-byte unsigned integer from the current stream and advances the position of the stream by eight bytes.
ToString	Returns a String that represents the current Object.

Figure 16.2-8 BinaryReader class methods

➢16.0 C# I/O System (Files and Streams)

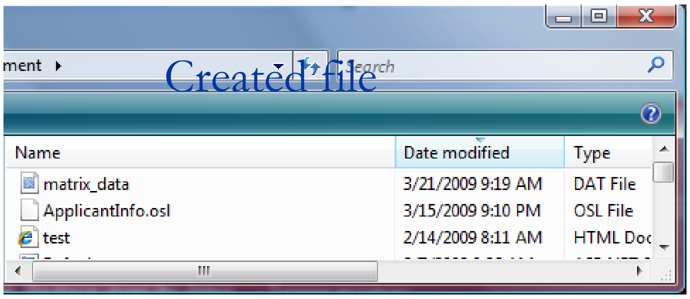
In the Example 16.2-3, a table consisting of 2 rows and 3 columns is created, Then it is written into a binary file called matrix_data.dat. The Figure 16.2-9 shows the debug statements emits from the program. Snap shot of the directory is show in Figure 16.2-10

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace MatrixWriter
{
  public class WriteMatrix
  {
   static void Main(String [] args)
   {
    string fileName = @"C:\csci\Examples\WebApplicationDevelopment\matrix_data.dat";
    double [,] data = {{Math.Exp(2.0), Math.Exp(3.0),Math.Exp(3.0)},
                                                    {Math.Exp(-2.0), Math.Exp(-3.0),Math.Exp(-3.0)}};

    int row = data.GetUpperBound(0)+1;
    int col = data.GetUpperBound(1)+1;

    Console.WriteLine("row :  " + row);
    Console.WriteLine("col : " + col);

    int i, j;

    for (i = 0; i < row; i++)
     for (j = 0; j < col; j++)
      Console.WriteLine("data[" + i + "," + j + "] = " + data[i, j]);


    if (fileName.Length > 0)
    try {
        FileStream fileStream = File.Create(fileName);
        BinaryWriter binaryWriter = new BinaryWriter(fileStream);

         binaryWriter .Write(row);
         binaryWriter .Write(col);

         for (i = 0; i < row; i++)
           for (j = 0; j < col; j++)
               binaryWriter .Write(data[i, j]);

         binaryWriter .Close();
     }
    catch (IOException e){
     Console.WriteLine("Exception caught..");
     }
   }
  }
}
```


C:\Windows\system32\cmd.exe

```
data[0,0] = 7.38905609893065
data[0,1] = 20.0855369231877
data[0,2] = 20.0855369231877
data[1,0] = 0.135335283236613
data[1,1] = 0.0497870683678639
data[1,2] = 0.0497870683678639
Press any key to continue . . .
```

Figure 16.2-9 Output


Created file

Name	Date modified	Type
matrix_data	3/21/2009 9:19 AM	DAT File
ApplicantInfo.osl	3/15/2009 9:10 PM	OSL File
test	2/14/2009 8:11 AM	HTML Doc

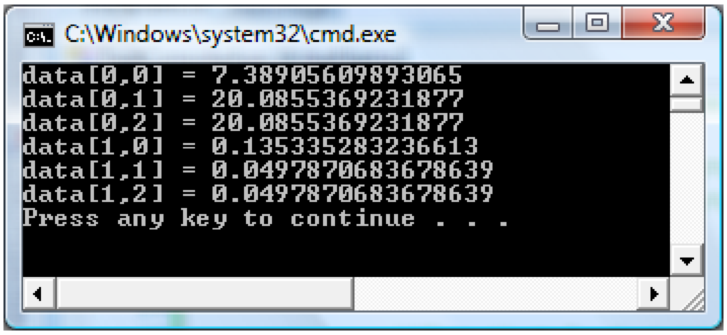
Example 16.2-3

Figure 16.2-10 Created File

➢16.0 C# I/O System (Files and Streams)

In this example you will see the BinaryReaader class is used to read back the table written in Example 16.2-4 and print it back. The program opens the matrix_data.dat, first reads the first two 32-bit locations to read the number of rows and number of columns . These two values are used to re-create the table in the memory. Then reads continuously the next 6 64-bit locations and fills them into the table. At the end of the program, the table is printed on the console. Output is shown in Figure 16.2-11

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace MatrixReader
{
 public class ReadMatrix
 {
   static void Main(String[] args)
   {
    string fileName = @"C:\csci\Examples\WebApplicationDevelopment\matrix_data.dat";
    double[,] data;

    int row = 0;
    int col = 0;
    int i, j;

    if (fileName.Length > 0)
    try
    { FileStream fileStream = File.OpenRead(fileName);
      BinaryReader binaryReader = new BinaryReader(fileStream);

      row = binaryReader.ReadInt32();
      col = binaryReader.ReadInt32();

      data = new double[row, col];

      for (i = 0; i < row; i++)
        for (j = 0; j < col; j++)
            data[i, j] =  binaryReader.ReadDouble();

      binaryReader.Close();

      for (i = 0; i < row; i++)
        for (j = 0; j < col; j++)
           Console.WriteLine("data[" + i + "," + j + "] = " + data[i, j]);
    }
    catch (IOException e)
    {
     Console.WriteLine("Exception caught..");
    } }  } }
```



```
data[0,0] = 7.38905609893065
data[0,1] = 20.0855369231877
data[0,2] = 20.0855369231877
data[1,0] = 0.135335283236613
data[1,1] = 0.0497870683678639
data[1,2] = 0.0497870683678639
Press any key to continue . . .
```

Figure 16.2-11 Output

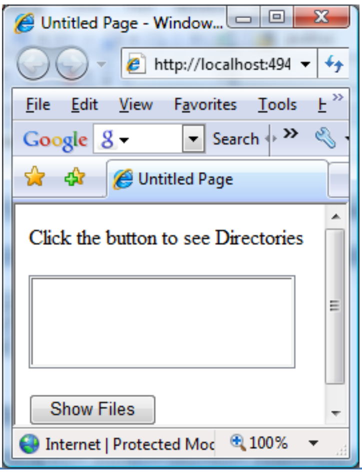
Example 16.2-4

16.3 File Usage in Web Applications

Most web applications use databases to store and retrieve data. But file handling can be used in any web application just as they are used in a non web application. Sometimes people build browser based application to be used locally within an enterprise. This may be a good example where you could use file systems and the backend to store data instead of a database. However the application cannot use the power of databases while reading and writing data. In file systems, the developer must write the code to achieve the specific needs for file I/O. The example listed below does the same you read in Example 16.2-1. But the result is displayed on a web browser instead of a console.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="ShowDirListing.aspx.cs"
Inherits="UsageOfListsAndFiles._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
    <asp:Label ID="aLabel" runat="server" Text="Click the button to see Directories"></asp:Label> <br/><br/>
    <asp:ListBox ID="FileList" runat="server" Width="200"></asp:ListBox><br/><br/>
    <asp:Button ID="PushButton" runat="server" Text="Show Files" OnClick="ShowDirectories"></asp:Button>
    </div>
    </form>
</body>
</html>
```

Example 16.3-1 a


Untitled Page - Window...
http://localhost:494
File Edit View Favorites Tools
Untitled Page
Click the button to see Directories
Show Files
Internet | Protected Moc
100%

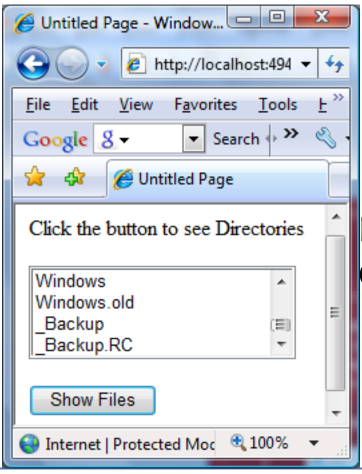


Figure 16.3-1 Output On browser.

➢16.0 C# I/O System (Files and Streams)

The code base for the example is shown in Example 16.3-1b. The output is the same as the example 16.2-1 but displayed on a browser (Figure 16.3-1)

```
using System;
using System.Collections;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;
using System.IO;

namespace UsageOfListsAndFiles
{  public partial class _Default : System.Web.UI.Page
   {
    protected void Page_Load(object sender, EventArgs e) {
       }
    protected void ShowDirectories(object sender, EventArgs e)
    {
      string fileName =
        @"C:\csci\Examples\WebApplicationDevelopment\CDriveDirs.txt";

      /** Now get written dir list back from file in one shot. **/
      string[] getBackDirs = File.ReadAllLines(fileName);

      /** Fill the ListBox component with the data **/
      foreach (string nextDir in getBackDirs)
          FileList.Items.Add(nextDir);
    }
  }
}
```

Example 16.3-1b

Take a look at the example on pages 590-594 in your test book.

16.4 Other Useful Classes

16.4.1 Directory class

Use the Directory class for typical operations such as copying, moving, renaming, creating, and deleting directories. You can also use the Directory class to get and set **DateTime** information related to the creation, access, and writing of a directory.

Because all Directory methods are static, it might be more efficient to use a Directory method rather than a corresponding **DirectoryInfo** instance method if you want to perform only one action. Most Directory methods require the path to the directory that you are manipulating.

The static methods of the Directory class Perform security checks on all methods. If you are going to reuse an object several times, consider using the corresponding instance method of **DirectoryInfo** instead, because the security check will not always be necessary. Methods and descriptions of the Directory class is shown in Figure 16.4-1.

Name	**Description**
ChangeExtension	Changes the extension of a path string.
Combine	Combines two path strings.
GetDirectoryName	Returns the directory information for the specified path string.
GetExtension	Returns the extension of the specified path string.
GetFileName	Returns the file name and extension of the specified path string.
GetFileNameWithoutExtension	Returns the file name of the specified path string without the extension.
GetFullPath	Returns the absolute path for the specified path string.
GetInvalidFileNameChars	Gets an array containing the characters that are not allowed in file names.
GetInvalidPathChars	Gets an array containing the characters that are not allowed in path names.
GetPathRoot	Gets the root directory information of the specified path.
GetRandomFileName	Returns a random folder name or file name.
GetTempFileName	Creates a uniquely named, zero-byte temporary file on disk and returns the full path of that file.
GetTempPath	Returns the path of the current system's temporary folder.
HasExtension	Determines whether a path includes a file name extension.
IsPathRooted	Gets a value indicating whether the specified path string contains absolute or relative path information.

Figure 16.4-1 Directory class methods

16.4.2 Path class

A path is a string that provides the location of a file or directory. A path does not necessarily point to a location on disk; for example, a path might map to a location in memory or on a device. The exact format of a path is determined by the current platform. For example, on some systems, a path can start with a drive or volume letter, while this element is not present in other systems. On some systems, file paths can contain extensions, which indicate the type of information stored in the file. The format of a file name extension is platform-dependent; for example, some systems limit extensions to three characters, and others do not. The current platform also determines the set of characters used to separate the elements of a path, and the set of characters that cannot be used when specifying paths. Because of these differences, the fields of the Path class as well as the exact behavior of some members of the Path class are platform-dependent. Methods and descriptions of the Path class is shown in Figure 16.4-2.

Name	**Description**
ChangeExtension	Changes the extension of a path string.
Combine	Combines two path strings.
GetDirectoryName	Returns the directory information for the specified path string.
GetExtension	Returns the extension of the specified path string.
GetFileName	Returns the file name and extension of the specified path string.
GetFileNameWithoutExtension	Returns the file name of the specified path string without the extension.
GetFullPath	Returns the absolute path for the specified path string.
GetInvalidFileNameChars	Gets an array containing the characters that are not allowed in file names.
GetInvalidPathChars	Gets an array containing the characters that are not allowed in path names.
GetPathRoot	Gets the root directory information of the specified path.
GetRandomFileName	Returns a random folder name or file name.
GetTempFileName	Creates a uniquely named, zero-byte temporary file on disk and returns the full path of that file.
GetTempPath	Returns the path of the current system's temporary folder.
HasExtension	Determines whether a path includes a file name extension.
IsPathRooted	Gets a value indicating whether the specified path string contains absolute or relative path information.

Figure 16.4-2 Path class methods

➢16.0 C# I/O System (Files and Streams)

16.4.3 DirectoryInfo class

Use the **DirectoryInfo** class for typical operations such as copying, moving, renaming, creating, and deleting directories. If you are going to reuse an object several times, consider using the instance method of **DirectoryInfo** instead of the corresponding static methods of the Directory class, because a security check will not always be necessary. Methods and descriptions of the **DirectoryInfor** class is shown in Figure 16.4.3.

Name	**Description**
Create	Overloaded. Creates a directory.
CreateObjRef	Creates an object that contains all the relevant information required to generate a proxy used to communicate with a remote object. (Inherited from MarshalByRefObject.)
CreateSubdirectory	Overloaded. Creates a subdirectory or subdirectories on the specified path. The specified path can be relative to this instance of the DirectoryInfo class.
Delete	Overloaded. Deletes a DirectoryInfo and its contents from a path.
Equals	Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
Finalize	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection. (Inherited from Object.)
GetAccessControl	Overloaded. Gets the access control list (ACL) entries for the current directory.
GetDirectories	Overloaded. Returns the subdirectories of the current directory.
GetFiles	Overloaded. Returns a file list from the current directory.
GetFileSystemInfos	Overloaded. Retrieves an array of strongly typed FileSystemInfo objects representing files and subdirectories of the current directory.
GetHashCode	Serves as a hash function for a particular type. (Inherited from Object.)
GetLifetimeService	Retrieves the current lifetime service object that controls the lifetime policy for this instance. (Inherited from MarshalByRefObject.)
GetObjectData	Sets the SerializationInfo object with the file name and additional exception information. (Inherited from FileSystemInfo.)
GetType	Gets the Type of the current instance. (Inherited from Object.)
InitializeLifetimeService	Obtains a lifetime service object to control the lifetime policy for this instance. (Inherited from MarshalByRefObject.)
MemberwiseClone	Overloaded.
MoveTo	Moves a DirectoryInfo instance and its contents to a new path.
Refresh	Refreshes the state of the object. (Inherited from FileSystemInfo.)
SetAccessControl	Applies access control list (ACL) entries described by a DirectorySecurity object to the directory described by the current DirectoryInfo object.
ToString	Returns the original path that was passed by the user. (Overrides Object..::.ToString()()().)

Figure 16.4-3 DirectoryInfor class methods

➢16.0 C# I/O System (Files and Streams)

16.4.4 FileInfo class

Use the **FileInfo** class for typical operations such as copying, moving, renaming, creating, opening, deleting, and appending to files.

Many of the **FileInfo** methods return other I/O types when you create or open files. You can use these other types to further manipulate a file. For more information, see specific **FileInfo** members such as Open, OpenRead, OpenText, CreateText, or Create.

If you are going to reuse an object several times, consider using the instance method of FileInfo instead of the corresponding static methods of the File class, because a security check will not always be necessary.

By default, full read/write access to new files is granted to all users.

The following table describes the enumerations that are used to customize the behavior of various FileInfo methods.

FileInfor class is shown in Figure 16.4.4.

Name	**Description**
AppendText	Creates a StreamWriter that appends text to the file represented by this instance of the FileInfo.
CopyTo	Overloaded. Copies an existing file to a new file.
Create	Creates a file.
CreateObjRef	Creates an object that contains all the relevant information required to generate a proxy used to communicate with a remote object. (Inherited from MarshalByRefObject.)
CreateText	Creates a StreamWriter that writes a new text file.
Decrypt	Decrypts a file that was encrypted by the current account using the Encrypt method.
Delete	Permanently deletes a file. (Overrides FileSystemInfo..::.Delete()()().)
Encrypt	Encrypts a file so that only the account used to encrypt the file can decrypt it.
Equals	Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
Finalize	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection. (Inherited from Object.)
GetAccessControl	Overloaded. Gets a FileSecurity object that encapsulates the access control list (ACL) entries for the file described by the current FileInfo object.
GetHashCode	Serves as a hash function for a particular type. (Inherited from Object.)
GetLifetimeService	Retrieves the current lifetime service object that controls the lifetime policy for this instance. (Inherited from MarshalByRefObject.)
GetObjectData	Sets the SerializationInfo object with the file name and additional exception information. (Inherited from FileSystemInfo.)
GetType	Gets the Type of the current instance. (Inherited from Object.)
InitializeLifetimeService	Obtains a lifetime service object to control the lifetime policy for this instance. (Inherited from MarshalByRefObject.)
MemberwiseClone	Overloaded.
MoveTo	Moves a specified file to a new location, providing the option to specify a new file name.
Open	Overloaded. Opens a file with various read/write and sharing privileges.
OpenRead	Creates a read-only FileStream.
OpenText	Creates a StreamReader with UTF8 encoding that reads from an existing text file.
OpenWrite	Creates a write-only FileStream.
Refresh	Refreshes the state of the object. (Inherited from FileSystemInfo.)
Replace	Overloaded. Replaces the contents of a specified file with the file described by the current FileInfo object, 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 file described by the current FileInfo object.
ToString	Returns the path as a string. (Overrides Object..::.ToString()()().)

Figure 16.4-4 FileInfor class methods

➢16.0 C# I/O System (Files and Streams)

16.4.5 DriveInfo class

This class models a drive and provides methods and properties to query for drive information. Use **DriveInfo** to determine what drives are available, and what type of drives they are. You can also query to determine the capacity and available free space on the drive. **DriveInfor** class is shown in Figure 16.4.5.

Method	**Description**
Equals	Determines whether the specified Object is equal to the current Object. (Inherited from Object.)
Finalize	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection. (Inherited from Object.)
GetDrives	Retrieves the drive names of all logical drives on a computer.
GetHashCode	Serves as a hash function for a particular type. (Inherited from Object.)
GetType	Gets the Type of the current instance. (Inherited from Object.)
MemberwiseClone	Creates a shallow copy of the current Object. (Inherited from Object.)
ToString	Returns a drive name as a string. (Overrides Object..::.ToString()()().)

Figure 16.4-5 DriveInfor class methods

➢16.0 C# I/O System (Files and Streams)

16.5 Object Serialization

Serialization is the process of writing the state of an object to a Byte Stream OR it is the process of reducing the objects instance into a format that can either be stored to disk or transported over a Network.

Serialization is useful when you want to save the state of your application to a persistence storage area. That storage area can be a file. At a later time, you may restore these objects by using the process of **deserialization.** In this section we will learn about serialization and why you need serialization in NET.

16.5.1 Serialization and Streams in .NET

.NET objects are serialized to a stream. Developers must use a .NET formatter class to control the serialization of the object to and from the stream. In addition to the serialized data, the serialization stream carries information about the object's type, including its assembly name, culture, and version.

16.5.2 The Binary Formatter

The Binary formatter provides binary encoding for compact serialization either for storage or for socket-based network streams. The **BinaryFormatter** class is generally not appropriate when data is meant to be passed through a firewall.

16.5.3 The SOAP Formatter

The **SOAP** (**Simple Object Access Protocol**) formatter provides formatting that can be used to enable objects to be serialized using the **SOAP** protocol. The Soap Formatter class is primarily used for serialization through firewalls or among diverse systems. SOAP is used to serialize object to XML files.

16.5.4 Other Formatters

The .NET framework also includes the abstract FORMATTERS class that can be used as a base class for custom formatters. This class inherits from the **IFormatter** interface.

➢16.0 C# I/O System (Files and Streams)

16.5.4 Requirements for Object Serialization
Serialization is done so that the object can be recreated with its current state at a later point in time or at a different location. The following are required to Serialize an object:

- The object that is to serialized itself
- A stream to contain the serialized object
- A Formatter used to serialize the object

System.Runtime.Serialization.Formatters.Binary is the namespace that contains the classes that are required to serialize an object.

To serialize using the **BinaryFormatter**, you use the Serialize() method of The **BinaryFormatter** while **Deserialize()** is used to **deserialize** an object from the **BinaryFormatter** class.

16.5.5 Making a Class Serializable
16.5.5.1 Using [Serializable] tag.
Every modern language contains a feature to allow you to serialize your objects. The good news is that with C# and the .NET Framework you can do this smart and easily. But an object cannot be serialized if teh class of that object is not made to create serialized objects. So the first thing is to make a class **serializable.** This is very simple in c#. All you have to do is to add the **[Serializable]** attribute to the class.

Here is an example…

```
[Serializable]
public class Store
{


}
```

➢16.0 C# I/O System (Files and Streams)

Example 16.5-1a shows the same Store class with more information.

The **[Serializable]** attribute 'stamps' the class ready To be serialized. Note that the Serialization applied ONLY to the data in the class and NOT for the methods. Example 16.5-1b shows the code you use to serialize an object of the **Store** class.

```
[Serializable]
public class Store
{
  private int sCount;

  public int stockCount
  {
    get
    {
      return sCount;
    }
    set
    {
      sCount = value;
    }
  }

  private int temp;
  private string storeName = "My local store";

  public Store()
  {
    stockCount = 0;
  }
   // Here may follow some code for this
   //class to function
 }
```

Example 16.5-1a Serializable Class

```
static void Main(string[] args)
{
Store myStore = new Store();
   myStore.stockCount = 50;

   FileStream flStream = new FileStream("MyStore.dat",
   FileMode.OpenOrCreate, FileAccess.Write);
   try
   {
      BinaryFormatter binFormatter = new BinaryFormatter();
      binFormatter.Serialize(flStream, myStore);
   }
   finally
   {
      flStream.Close();
      Console.WriteLine(Environment.NewLine + "Object
     'myStore' serialized!";
      Console.WriteLine(Environment.NewLine + "stockCount :
     =  " + myStore.stockCount ;
   }
}
```

Example 16.5-1b Serializing Process

➢16.0 C# I/O System (Files and Streams)

16.5.5.2 Using the [NonSerializable] attribute

At times there are non important or temporary data members you define in a class that are not really worth serializing. In fact sometimes there are even sensitive information that may not want to serialize. We can 'mark' those data members that we do not want to serialize in a class by adding the Marker **[NonSerializable] for each** data member.

```
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]
public class Store
{
  private int sCount;

  public int stockCount
  {
    get
    {
      return sCount;
    }
    set
    {
      sCount = value;
    }
  }
 [NonSerialized] private int temp;

  private string storeName = "My local store";

  public Store()
  {
    stockCount = 0;
  }
   // Here may follow some code for this
   //class to function
}
```

Example 16.5-2a Serializable Class

```
static void Main(string[] args)
{
   Store myStore = new Store();
   myStore.stockCount = 50;

   FileStream flStream = new FileStream("MyStore.dat",
   FileMode.OpenOrCreate, FileAccess.Write);
   try
   {
      BinaryFormatter binFormatter = new BinaryFormatter();
      binFormatter.Serialize(flStream, myStore);
   }
   finally
   {
      flStream.Close();
      Console.WriteLine(Environment.NewLine + "Object
     'myStore' serialized!";
      Console.WriteLine(Environment.NewLine + "stockCount
     = 50";
   }
}
```

Example 16.5-2b Serializing Process

In this example, the temp data member will not be serialized. Instead of the actual value of the data member, it will just store the default value of the data type. In this case for temp, it will store **0** as temp is an int type.

➢16.0 C# I/O System (Files and Streams)

16.5.6 DeSerializing

16.5.6.1 Using BinaryFormatter

Deserializing is re-constructing the object back into the memory. You can use the **Deserialize()** method in the **BinaryFormatter** class. In the Example 16.5-3 you see a object reference Store is first created. Really the object is not created at tht time. But the method **Deserialize()** brings a Object type object which is then casted to a Store type and linked to the readStore reference.

```
using System.Runtime.Serialization.Formatters.Binary
static void Main(string[] args)
{
  Store readStore = null;

   flStream = new FileStream("MyStore.dat", FileMode.Open, FileAccess.Read);
   try {
           BinaryFormatter binFormatter = new BinaryFormatter();
            readStore = (Store)binFormatter.Deserialize(flStream);
         }
   finally{
              flStream.Close();
              Console.WriteLine(Environment.NewLine + "Object 'readStore' deserialized!");
              Console.WriteLine(Environment.NewLine + "stockCount = " +
                          System.Convert.ToString(readStore.stockCount));
            }
}
```

Example 16.5-3 DeSerializing using BinaryFormatter

We create an instance of **FileStream** to read the file. We create a **BinaryFormatter** object again. This time we use its 'Deserialize' method. It returns the deserialized object which we cast to our own class 'Store'. and in the end we close the stream and output the 'stockCount' value to check if everything is OK.

➢16.0 C# I/O System (Files and Streams)

16.5.6.2 Using SOAP (Simple Object Access Protocol) Formatter

We have seen how binary formatter is used in Serialization. The fact that it is separated in a detached namespace may be already gave you a hint that there are other types of formatters.

The SOAP formatter serializes your object to an XML file obeying special rules. The SOAP is the standardized format which all the web services use. It is a simple XML-based protocol to let applications exchange information over HTTP.

You can replace the **BinaryFormatter** with the **SoapFormatter** and it will work with no other changes. The only thing you have to do is to add a reference to the **System.Runtime.Serialization.Formatters.Soa**p (not only a uses clause but a reference too!).

The SOAP format being an XML is readable by humans unlike the binary one. See Example 16.5-4.

```
static void Main(string [] args)
{
Store myStore = new Store();

MemoryStream memStream = new MemoryStream();
   try {
      myStore.stockCount = 50;
      SoapFormatter soapFormatter = new SoapFormatter();
      soapFormatter.Serialize(memStream, myStore);

      byte[] buff = memStream.GetBuffer();

      string soapOutput = "";

      foreach(byte b in buff)
         soapOutput += (char)b;

      tbOutput.Text += Environment.NewLine +  "Object 'myStore' serialized to SOAP!";
      tbOutput.Text += soapOutput;
   }
   finally {
      memStream.Close();
   }
}
```

Example 16.5-4a DeSerializing using SOAP

➢16.0 C# I/O System (Files and Streams)

Here the **MemoryStream** class is used instead of the **FileStream** we used previously. This way we save ourselves the use of file.

After we store the data to the stream we get its contents with **GetBuffer().** Note that the **MemoryStream** is not closed right after the write as did with the **FileStream**. This is because we loose the stored data once we close it.

Look at the code:

```
foreach(byte b in buff)
    soapOutput += (char)b;
```

It is used to convert the byte array to string. Do you notice something wrong? If not, you have to study more about the strings. In C# the strings can not be changed once their value is set. This way we create a new string every time we add a character. More appropriate is the use of **StringBuffer**. In our case the buffer is small and the very serious speed losses are not visible.

The output we get by the SOAP formatter is shown in Figure 16.5-5

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
      <a1:Store id="ref-1" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/
SerializeTest/SerializeTest%2C%20Version%3D1.0.938.186%2C
%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
         <sCount>50</sCount>
         <storeName id="ref-3">My local store</storeName>
      </a1:Store>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 16.5-4b Output using SOAP Formatter