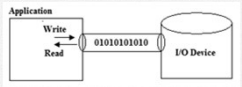
# I/O Classes

File Handling in C#

## System.IO

- The System.IO namespaces contain types that support input and output, including the ability to read and write data to streams either synchronously or asynchronously, to compress data in streams, to create and use isolated stores, to map files to an application's logical address space, to store multiple data objects in a single container, to communicate using anonymous or named pipes, to implement custom logging, and to handle the flow of data to and from serial ports.




## Exploring the System.IO Namespace

| Non-abstract I/O Class | Type Meaning in Life |
|---|---|
| **BinaryReader BinaryWriter** | These classes allow you to store and retrieve primitive data types (integers, Booleans, strings, and whatnot) as a binary value. |
| **BufferedStream** | This class provides temporary storage for a stream of bytes that you can commit to storage at a later time. |
| **Directory DirectoryInfo** | You use these classes to manipulate a machine's directory structure. The Directory type exposes functionality using static members, while the DirectoryInfo type exposes similar functionality from a valid object reference. |
| **DriveInfo** | This class provides detailed information regarding the drives that a given machine uses. |
| **File FileInfo** | You use these classes to manipulate a machine's set of files. The File type exposes functionality using static members, while the FileInfo type exposes similar functionality from a valid object reference. |

## Exploring the System.IO Namespace

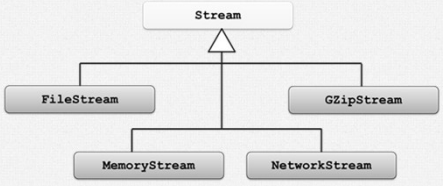
| Non-abstract I/O Class | Type Meaning in Life |
|---|---|
| **FileStream** | This class gives you random file access (e.g., seeking capabilities) with data represented as a stream of bytes. |
| **FileSystemWatcher** | This class allows you to monitor the modification of external files in a specified directory. |
| **MemoryStream** | This class provides random access to streamed data stored in memory rather than a physical file. |
| **Path** | This class performs operations on System.String types that contain file or directory path information in a platform-neutral manner. |
| **StreamWriter StreamReader** | You use these classes to store (and retrieve) textual information to (or from) a file. These types do not support random file access. |
| **StringWriter StringReader** | Like the StreamReader/StreamWriter classes, these classes also work with textual information. However, the underlying storage is a string buffer rather than a physical file. |

## IO in C#

- C# does I/O via Stream classes
- Streams are abstract flow of data from source to sink
- C# IO classes are found in **System.IO**
- Two types of streams are used
  - byte-oriented
  - character-oriented
- Streams are used to read and write data from file, memory, network
- They are also be used to move the file pointer to seek a particular location.

## Standard IO

- C# programs automatically open three text streams
  - *Standard input:* connected to the keyboard **(Console.In)**
  - *Standard output:* connected to the monitor **(Console.Out)**
  - *Standard error:* connected to the monitor **(Console.Error)**




Presented by

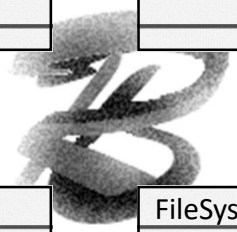
Ranjan Bhatnagar

## Stream

- Stream is the abstract base class of all streams.
- It provides method for reading, writing and seeking.
- The classes inheriting form Stream may provide some of the capabilities depending on the underlying data source or repository. Application can query a stream for its capabilities by using the **CanRead, CanWrite**, and **CanSeek** properties.

## File Handling

- With respect to files two types information becomes necessary:
  - Information about the file system, directories, creating, moving, renaming and deleting etc.
    - Classes inherited from **FileSystemInfo** provide this information
      - **DirectoryInfo**
      - **FileInfo**
  - Reading and writing files
    - **FileStream**



## FileSystemInfo

- Provides the base class for both FileInfo and DirectoryInfo objects.
  - **DirectoryInfo**
    - Exposes instance methods for creating, moving, and enumerating through directories and subdirectories. This class cannot be inherited.
  - **FileInfo**
    - Provides properties and instance methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of FileStream objects. This class cannot be inherited.

## FileSystemInfo Properties

| Property | Meaning in Life |
|---|---|
| **Attributes** | Gets or sets the attributes associated with the current file that are represented by the FileAttributes enumeration (e.g., is the file or directory read-only, encrypted, hidden, or compressed?). |
| **CreationTime** | Gets or sets the time of creation for the current file or directory. |
| **Exists** | You can use this to determine whether a given file or directory exists. |
| **Extension** | Retrieves a file's extension. |
| **FullName** | Gets the full path of the directory or file. |
| **LastAccessTime** | Gets or sets the time the current file or directory was last accessed. |
| **LastWriteTime** | Gets or sets the time when the current file or directory was last written to. |
| **Name** | Obtains the name of the current file or directory. |

## Working with the DirectoryInfo

- This class contains a set of members used for creating, moving, deleting, and enumerating over directories and subdirectories.
- In addition to the functionality provided by its base class, DirectoryInfo offers the key members listed below:

| Member | Meaning in Life |
|---|---|
| **Create(), CreateSubdirectory()** | Create a directory (or set of subdirectories) when given a path name. |
| **Delete()** | Deletes a directory and all its contents. |
| **GetDirectories()** | Returns an array of DirectoryInfo objects that represent all subdirectories in the current directory. |
| **GetFiles()** | Retrieves an array of FileInfo objects that represent a set of files in the given directory. |
| **MoveTo()** | Moves a directory and its contents to a new path. |
| **Parent** | Retrieves the parent directory of this directory. |
| **Root** | Gets the root portion of a path. |

## Few Example

- How to bind to the current working directory.

```
DirectoryInfo dir1 = new DirectoryInfo(".");
```

- Bind to the path passed into the constructor (E:\Test) already exists on the physical machine.

```
DirectoryInfo dir2 = new DirectoryInfo(@"E:\Test");
```

- Bind to a non existent directory, then create it.

```
DirectoryInfo dir3 = new DirectoryInfo(@"C:\Test\MyCode");
dir3.Create();
```

Presented by

Ranjan Bhatnagar

## Using DirectoryInfo Object

- Once you create a DirectoryInfo object, you can investigate the underlying directory contents using any of the properties inherited from FileSystemInfo.

```
using System;
using System.IO;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Directory Info *****\n");
        DirectoryInfo dir = new DirectoryInfo(@"C:\Windows");
        Console.WriteLine("FullName: {0}", dir.FullName);
        Console.WriteLine("Name: {0}", dir.Name);
        Console.WriteLine("Parent: {0}", dir.Parent);
        Console.WriteLine("Creation: {0}", dir.CreationTime);
        Console.WriteLine("Attributes: {0}", dir.Attributes);
        Console.WriteLine("Root: {0}", dir.Root);
        Console.WriteLine("**************************\n");
    }
}
```

## Extend a Directory Structure

- You can programmatically extend a directory structure using the DirectoryInfo.CreateSubdirectory() method.
- Create \MyFolder off application directory.

```
DirectoryInfo dir = new DirectoryInfo(@"E:\");
dir.CreateSubdirectory("MyFolder");
```

- Create \MyFolder2\Data off application directory.

```
dir.CreateSubdirectory(@"MyFolder2\Data");
```

## Working with the Directory Type

- List all drives on current computer and Delete folders created in the previous example.

```
string[] drives = Directory.GetLogicalDrives();
Console.WriteLine("Here are your drives:");
foreach (string s in drives)
    Console.WriteLine("--> {0} ", s);
Console.WriteLine("Press Enter to delete directories");
Console.ReadLine();
try
{
    Directory.Delete(@"E:\MyFolder");
    Directory.Delete(@"E:\MyFolder2", true);
}
catch (IOException e)
{
    Console.WriteLine(e.Message);
}
```

## Path Exceptions

- If the path provided in the **DirectoryInfo** or **FileInfo** contains invalid characters such as ", <, >, or | **ArgumentException** is thrown.
- The specified path, file name, or both exceed the system-defined maximum length, **PathTooLongException** is thrown. For Windows, paths must be less than 248 characters, and file names must be less than 260 characters.
- If path is null, **ArgumentNullException** is thrown

## FileInfo Core Members

| Member | Meaning in Life |
|---|---|
| **AppendText()** | Creates a StreamWriter object that appends text to a file. |
| **CopyTo()** | Copies an existing file to a new file. |
| **Create()** | Creates a new file and returns a FileStream object to interact with the newly created file. |
| **CreateText()** | Creates a StreamWriter object that writes a new text file. |
| **Delete()** | Deletes the file to which a FileInfo instance is bound. |
| **Directory** | Gets an instance of the parent directory. |
| **DirectoryName** | Gets the full path to the parent directory. Length Gets the size of the current file. |
| **MoveTo()** | Moves a specified file to a new location, providing the option to specify a new file name. Name Gets the name of the file. |
| **Open()** | Opens a file with various read/write and sharing privileges. |
| **OpenRead()** | Creates a read-only FileStream object. |
| **OpenText()** | Creates a StreamReader object, reads from an existing text file. |
| **OpenWrite()** | Creates a write-only FileStream object. |

## The FileInfo.Create() Method

- You can create a file handle to use the FileInfo.Create() method.

```
using System;
using System.IO;
class Program
{
    static void Main(string[] args)
    {
        // Make a new file on the E drive.
        FileInfo f = new FileInfo(@"E:\Test.dat");
        FileStream fs = f.Create();
        // Use the FileStream object...
        // Close down file stream.
        fs.Close();
    }
}
```

Presented by

Ranjan Bhatnagar

## The FileInfo.Open() Method

- You can use the FileInfo.Open() method to open existing files, as well as to create new files.

```
using System;
using System.IO;
class Program
{
    static void Main(string[] args)
    {
        // Make a new file via FileInfo.Open().
        FileInfo f2 = new FileInfo(@"E:\Test.dat");
        using (FileStream fs2 = f2.Open(FileMode.OpenOrCreate,
                                FileAccess.ReadWrite, FileShare.None))
        {
            // Use the FileStream object...
        }
    }
}
```

## Example: File and Directory info

- To list all the files whose names begin with 'win'.

```
using System;
using System.IO;
public class WinList
{
    public static void Main()
    {
        String searchName = "win";
        DirectoryInfo dir = new DirectoryInfo(@"c:\Program Files");
        SearchDirectories(dir, searchName);
    }
```

## SearchDirectories()

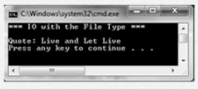
```
public static void SearchDirectories(DirectoryInfo dir,
                                                    String target)
{
        FileInfo[] files = dir.GetFiles();
        foreach (FileInfo file in files)
        {
            if (file.Name.StartsWith(target))
            {
                Console.WriteLine(file.Name);
            }
        }
        DirectoryInfo[] dirs = dir.GetDirectories();
        foreach (DirectoryInfo subDir in dirs)
        {
            SearchDirectories(subDir, target);
        }
    }
}
```

## Additional File-centric Members

| Method | Meaning in Life |
|---|---|
| **ReadAllBytes()** | Opens the specified file, returns the binary data as an array of bytes, and then closes the file. |
| **ReadAllLines()** | Opens a specified file, returns the character data as an array of strings, and then closes the file. |
| **ReadAllText()** | Opens a specified file, returns the character data as a System.String, and then closes the file. |
| **WriteAllBytes()** | Opens the specified file, writes out the byte array, and then closes the file. |
| **WriteAllLines()** | Opens a specified file, writes out an array of strings, and then closes the file. |
| **WriteAllText()** | Opens a specified file, writes the character data from a specified string, and then closes the file. |

## Example : Read and Write

```
using System;
using System.IO;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("*** IO with the File Type ***\n");
        // Write out Text to file on E drive.
        File.WriteAllText("E:\\quote.txt", "Live and Let Live");
        // Read it all back and print.
        string quote = File.ReadAllText(@"E:\quote.txt");
        Console.WriteLine("Quote: {0}", quote);
    }
}
```


## Working with the DriveInfo Class Type

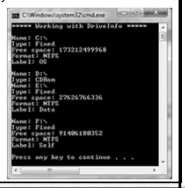
- The System.IO namespace provides a class named DriveInfo.
- Like Directory.GetLogicalDrives(), the static DriveInfo.GetDrives() method allows you to discover the names of a machine's drives.
- Unlike Directory.GetLogicalDrives(), however, DriveInfo provides numerous other details (e.g., the drive type, available free space, and volume label).
- Consider the Program in next slide

Presented by

Ranjan Bhatnagar

## Example

```
using System;
using System.IO;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Working with DriveInfo *****\n");
        DriveInfo[] myDrives = DriveInfo.GetDrives();
        foreach (DriveInfo d in myDrives)
        {
            Console.WriteLine("Name: {0}", d.Name);
            Console.WriteLine("Type: {0}", d.DriveType);
            if (d.IsReady)
            {
                Console.WriteLine("Free space: {0}",
                                  d.TotalFreeSpace);
                Console.WriteLine("Format: {0}",
                                  d.DriveFormat);
                Console.WriteLine("Label: {0}",
                                  d.VolumeLabel);
                Console.WriteLine();
            }
        }
    }
}
```



## File and Directory class

- **File** and **Directory** static classes are very similar to **FileInfo** and **DirectoryInfo** class except that they provide static methods instead of the instance methods to deal with file and directories.
- Since **File** and **Directory** have static methods, the methods in these take an extra argument representing file/directory name.
- There are slight differences in the methods names and parameters in these classes. For instance,

## File and Directory class

- **CreateDirectory**() method in **Directory** class is same as that of the **Create()** method in DirectoryInfo class.
- If only one operation on a file or directory is required **File** or **Directory** class is handy.
- For many operations on the same file or directory, **FileInfo** or **DirectoryInfo** is better because if there is any security checking done, with this class the check is done only once for that instance.

## Example: Writing and reading using File

```
using System;
using System.IO;
class MyClass
{
    static void Main()
    {
        // Write a string array to a file.
        string[] stringArray = new string[]
                               { "C#", "ASP.NET", "VB.NET" };
        File.WriteAllLines("e:\\file.txt", stringArray);
        // Read in lines from file.
        foreach (string line in File.ReadAllLines("e:\\file.txt"))
        {
            Console.WriteLine("{0}", line);
        }
    }

}
```

## The Abstract Stream Class

- In the world of I/O manipulation, a stream represents a chunk of data flowing between a source and a destination.
- Streams provide a common way to interact with a sequence of bytes, regardless of what kind of device stores or displays the bytes.
- The abstract System.IO.Stream class defines several members that provide support for synchronous and asynchronous interactions with the storage medium

## Abstract Stream Members

| Method | Meaning in Life |
|---|---|
| **CanRead**<br>**CanWrite**<br>**CanSeek** | Determines whether the current stream supports reading, seeking, and/or writing. |
| **Close()** | Closes the current stream and releases any resources (such as sockets and file handles) associated with the current stream.<br>Internally, this method is aliased to the Dispose() method; therefore closing a stream is functionally equivalent to disposing a stream. |
| **Flush()** | Updates the underlying data source or repository with the current state of the buffer and then clears the buffer. If a stream does not implement a buffer, this method does nothing. |

## Abstract Stream Members Cont.

| Method | Meaning in Life |
|---|---|
| **Length** | Returns the length of the stream in bytes. |
| **Position** | Determines the position in the current stream. |
| **Read()**<br>**ReadByte()** | Reads a sequence of bytes (or a single byte) from the current stream and advances the current position in the stream by the number of bytes read. |
| **Seek()** | Sets the position in the current stream. |
| **SetLength()** | Sets the length of the current stream. |
| **Write()**<br>**WriteByte()** | Writes a sequence of bytes (or a single byte) to the current stream and advances the current position in this stream by the number of bytes written. |

## FileStream

- This stream supports synchronous and asynchronous read and write operations.
- It also supports support random access to files using the **Seek** method
- The constructor generally used is

**FileStream(String, FileMode, [FileAccess, FileShare])**

- FileMode
  - CreateNew ,Create, Open, Truncate, OpenOrCreate, Append
- FileAccess
  - Read, Write, ReadWrite
- FileShare
  - None , Read , Write, ReadWrite, Delete, Inheritable

## Example: FileStream

- This example first writes into a file and then reads from that file.
- Note that since it is a byte stream, to write text we need to use conversion functions that will convert byte array to string and vice versa
- System.Text.UTF8Encoding class has methods to do this:
  - byte[] GetBytes( string s )
  - string GetString( byte[] bytes )

## Example: FileStream

```
using System;
using System.IO;
using System.Text;
class FileReadWrite
{
    static string file = @"E:\test.txt";
    static void Main()
    {
        Write();
        Read();
    }
static void Write()
{
       FileStream fileStream = new FileStream(file,
       FileMode.Create, FileAccess.Write);
       String content = @"What is this life if, full of care,
                          We have no time to stand and stare";
       byte[] info = new UTF8Encoding(true).GetBytes(content);
       fileStream.Write(info, 0, content.Length);
       fileStream.Close();
}
```

## Example: FileStream

```
static void Read()
{
       FileStream fileStream = new FileStream(file,
                               FileMode.Open, FileAccess.Read);
        byte[] b = new byte[1024];
        UTF8Encoding data = new UTF8Encoding(true);
        while (fileStream.Read(b, 0, b.Length) > 0)
        {
            Console.WriteLine(data.GetString(b));
        }
    }
}
```

## Reading characters from stream

- Instead of reading byte and converting them, C# also provides way to read characters directly from the stream,
- In this case there are different classes for reading and writing.
- The 2 abstract classes listed below for the base class for the character stream.
  - **TextReader**
  - **TextWriter**
- **StreamReader** and **StreamWriter** implements the above classes to read and write reads characters from a byte stream

Presented by Ranjan Bhatnagar

## StreamReader

- StreamReader(Stream s)
- StreamReader(String filename)
- int Read()
- string ReadLine()
- int Read( char[] buffer, int index, int count )
- int Peek()

## StreamWriter

- **StreamWriter(Stream s)**
- StreamWriter(String filename): If the file exists, it is overwritten; otherwise, a new file is created.
- Write(XXX value) where XXX represents String, Char, Char[], Int32, Int64, Boolean, Decimal, Double, Object
- void WriteLine()
- void WriteLine(XXX value) where XXX represents String, Char, Char[], Int32, Int64, Boolean, Decimal, Double, Object.

## Example: StreamWriter, StreamReader

```
using System;
using System.Text;
using System.IO;
class WriterAndReader
{
    static string file = @"E:\poem.txt";
    static void Main()
    {
        Write();
        Read();
    }
    static void Write()
    {
        StreamWriter fileStream = new StreamWriter(file);
        fileStream.WriteLine("What is this life if, full of care,");
        fileStream.WriteLine("We have no time to stand and stare");
        fileStream.Close();
    }
```

## Example: StreamWriter, StreamReader

```
 static void Read()
    {
        StreamReader fileStream = new StreamReader(file);
        string s = null;
        while ((s = fileStream.ReadLine()) != null)
        {
            Console.WriteLine(s);
        }
        fileStream.Close();
    }
}
```

## Working with StringWriters and StringReaders

```
using System;
using System.IO;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Working with StringWriter / StringReader\n");
        // Create a StringWriter and emit character data to memory.
        using (StringWriter strWriter = new StringWriter())
        {
            strWriter.WriteLine("Don't forget Mother's Day this year...");
            //Get a copy of the contents from string to Console.
            Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);
        }
    }
}
```


## Make a Note

- StringWriter and StreamWriter both derive from the same base class (TextWriter), so the writing logic is more or less identical. However, given the nature of StringWriter, you should also be aware that this class allows you to use the GetStringBuilder() method to extract a System.Text.StringBuilder object.
- And when you wish to read from a stream of character data, you can use the corresponding StringReader type, which functions identically to the related StreamReader class. In fact, the StringReader class does nothing more than override the inherited members to read from a block of character data, rather than from a file.

## Extract System.Text.StringBuilder Object

```
using System;
using System.IO;
using System.Text;
class Program
{
    static void Main(string[] args)
    {
        using (StringWriter strWriter = new StringWriter())
        {
            strWriter.WriteLine("Don't forget Mother's Day this year...");
            Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);
            // Get the internal StringBuilder.
            StringBuilder sb = strWriter.GetStringBuilder();
            sb.Insert(0, "Hey!! ");
            Console.WriteLine("-> {0}", sb.ToString());
            sb.Remove(0, "Hey!! ".Length);
            Console.WriteLine("-> {0}", sb.ToString());
        }
    }
}
```

## Read data from the StringWriter

```
using System;
using System.IO;
class Program
{
    static void Main(string[] args)
    {
        using (StringWriter strWriter = new StringWriter())
        {
            strWriter.WriteLine("Don't forget Mother's Day this year...");
            Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);
            // Read data from the StringWriter.
            using (StringReader strReader =
                                new StringReader(strWriter.ToString()))
            {
                string input = null;
                while ((input = strReader.ReadLine()) != null)
                {
                    Console.WriteLine(input);
                }
            }
        }
    }
}
```

## Serialization

- Serialization is the process of saving the state of an object in the hard disk to make it persistent or transportable.
- The object state is converted to byte form and saved in the hard disk.
- The reverse of serialization is de-serialization that converts a byte stream into an object.
- NET Framework features two serializing technologies
  - Binary serialization
  - Data contract serialization
  - XmlSerializer

## Binary serialization

- Binary encoding is used to produce serialization for uses such as storage or socket-based network streams. It is not suitable for passing data through a firewall but provides better performance when storing data.
- Serialization process
  - Obtain the object that is Serializable.
  - Create a stream to contain the serialized object
  - Use System.Runtime.Serialization to serialize the object

## Deserialization

- Deserialization is reverse of Serialization.
- BinaryFormatter is used to serializes and de-serializes an object, or an entire graph of connected objects, in binary format.
- This class implements IFormatter interface that provides methods to serialize and de-serialize.
  - void Serialize( Stream sStream, Object graph )
  - Object Deserialize( Stream sStream )

## Example: using BinaryFormatter

- This example serializes and de-serializes the HashTable collection

```
using System;
using System.Collections;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
class Ser
{
    static string file = @"e:\email.dat";
    static void Main() { Write(); Read(); }
    static void Write()
    {
        Hashtable email = new Hashtable();
        email.Add("Mani", "mani@yahoo.com");
        email.Add("Fred", "red@gmail.com");
        email.Add("Kartik", "kar@rediffmail.com");
        FileStream fs = new FileStream(file, FileMode.Create);
```

Presented by

Ranjan Bhatnagar

## Example: using BinaryFormatter

```
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Serialize(fs, email);
        fs.Close();
    }
    static void Read()
    {
        Hashtable email = null;
        FileStream fs = new FileStream(file, FileMode.Open);

        BinaryFormatter formatter = new BinaryFormatter();
        email = (Hashtable)formatter.Deserialize(fs);

        fs.Close();
        foreach (DictionaryEntry de in email)
        {
            Console.WriteLine("{0}'s email is {1}.", de.Key, de.Value);
        }
    }
}
```

## SerializableAttribute

- All the objects are not Serializable.
- For an object to be **Serializable**, the class must have **[Serializable()]** or **[Serializable]** attribute above the class declaration. Note that even though the classes are in the same file every class must explicitly specify that it is Serializable.
- If this is not done, then an attempt to call **Serialize()** on that object would cause **System.Runtime.Serialization.SerializationException** to be thrown.
- All the public and private fields in a type that are marked by the **SerializableAttribute** are serialized by default,
- All the collection classes, string and the value wrapper classes are **Serializable**.

## NonSerialized

- We may not want all the fields of a class to be saved in hard disk. For instance, we may not want to save sensitive information like password or credit card number.
- To exclude such fields, tag it with the [**NonSerialized**] or [**NonSerialized**()] attribute must be added.

## Example: Serializable objects

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
[Serializable]
public class Student
{
    public int roll = 0; public string name;
    [NonSerialized]
    public string pwd;
    public Student(int r, string n, string p)
    {
        roll = r;
        name = n;
        pwd = p;
    }
    public override string ToString()
    {
        return name + "(" + roll + ")" + "pwd: " + pwd;
    }
}
```

## Example: Serializable objects

```
class Ser
{
    static string file = @"E:\stud.dat";
    static void Main() { Write(); Read(); }
    static void Write()
    {
        Student s1 = new Student(12292, "Rahul", "alliswell");
        FileStream fs = new FileStream(file, FileMode.Create);
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Serialize(fs, s1);
        fs.Close();
    }
    static void Read()
    {
        Student s1 = null;
        FileStream fs = new FileStream(file, FileMode.Open);
        BinaryFormatter formatter = new BinaryFormatter();
        s1 = (Student)formatter.Deserialize(fs);
        Console.WriteLine(s1);
        fs.Close();
    }
}
```

## Read from Keyboard & Output on Monitor

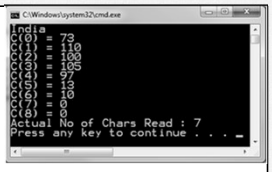
```
using System;
using System.IO;
class Echo
{
    public static void Main()
    {
        int data;
        while (true)
        {
            data = Console.In.Read();
            if (data == 32)
                break;
            Console.Out.Write((char)data);
        }
    }
}
```

Presented by

Ranjan Bhatnagar

## Program to Read a stream of characters

```
using System;
using System.IO;
class Echo
{
    public static void Main()
    {
        int n;
        char[] c = new char[10];
        n = Console.In.Read(c, 0, c.Length);
        for (int i = 0; i < c.Length - 1; i++)
            Console.Out.WriteLine("C({0}) = {1}", i, (int)c[i]);
        Console.WriteLine("Actual No of Chars Read : " + n);
    }
}
```




## Reading From File and Writing to Console

```
using System;
using System.IO;
class Program
{
    public static void Main()
    {
        FileStream fs = null;
        try {
            fs = new FileStream("e:\\demo.txt",
              FileMode.Open, FileAccess.Read, FileShare.None);
            while (true) {
                int data = fs.ReadByte();
                if (data == -1) break;
                Console.Out.Write ((char)data);
            }
            Console.WriteLine();
        }
        catch (FileNotFoundException ex) {
            Console.WriteLine(ex.Message);
        }
        finally { if (fs != null) fs.Close(); }
    }
}
```

## BinaryReader and BinaryWriter

```
using System; using System.IO;
class BinaryWriteReadProgram
{
    public static void Main()
    {
        int Id = 1; string Name = "Demo"; double Marks = 69.5;
        FileStream fs = new FileStream("e:\\student.dat",
                        FileMode.Create, FileAccess.ReadWrite);
        BinaryWriter bw = new BinaryWriter(fs,
                          System.Text.Encoding.Unicode);
        bw.Write(Id); // 4 bytes
        bw.Write(Name); // No of Chars * 2 + 1(EndOfString)
        bw.Write(Marks); // 8 byts
        fs.Flush();
        fs.Seek(0, SeekOrigin.Begin); //Move the Pointer to BOF
        BinaryReader br = new
               BinaryReader(fs, System.Text.Encoding.Unicode);
        Id = br.ReadInt32();
        Name = br.ReadString(); Marks =
            br.ReadDouble();
        Console.WriteLine(Id + " " + Name + " " + Marks); br.Close();
    }
}
```

## Serializing Objects Using the XmlSerializer

- In addition to the SOAP and binary formatters, the System.Xml.dll assembly provides a third formatter,
- System.Xml.Serialization.XmlSerializer. You can use this formatter to persist the public state of a given object as pure XML, as opposed to XML data wrapped within a SOAP message.
- Working with this type is a bit different from working with the SoapFormatter or BinaryFormatter type

## Example XmlSerializer

```
using System;
using System.IO;
using System.Xml.Serialization;
[Serializable]
public class Radio
{
    public bool hasTweeters;
    public bool hasSubWoofers;
    public double[] stationPresets;
    [NonSerialized]
    public string radioID
               = "XF-552RR6";
}
[Serializable]
public class Car
{
    public Radio theRadio
               = new Radio();
    public bool isHatchBack;
}
```

```
[Serializable]
public class JamesBondCar : Car
{
    public bool canFly;
    public bool canSubmerge;
}
```

## Example XmlSerializer

```
class Program
{
    static void Main(string[] args)
    {
        JamesBondCar jbc = new JamesBondCar();
        XmlSerializer xmlFormat =
               new XmlSerializer(typeof(JamesBondCar));
        using (Stream fStream =
               new FileStream(@"e:\cardata.xml",
                FileMode.Create, FileAccess.Write, FileShare.None))
        {
            xmlFormat.Serialize(fStream, jbc);
        }
        Console.WriteLine("=> Saved car in XML format!");
    }
}
```

Presented by

Ranjan Bhatnagar

## Serializing Objects Using the SoapFormatter

- Another choice of formatter is the SoapFormatter type, which serializes data in a proper SOAP envelope.
- The Simple Object Access Protocol (SOAP) defines a standard process in which you can invoke methods in a platform and OS-neutral manner.
- You have set a reference to the **System.Runtime.Serialization.Formatters.Soap.dll** assembly and import the **System.Runtime.Serialization.Formatters.Soap** namespace

## Simple Example SOAP Formatter

```
using System;
using System.IO;
using System.Collections;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;
class App
{
    [STAThread]
    static void Main()
    {
        Serialize();
        Deserialize();
    }
    static void Serialize()
    {
        // Create a hashtable of values that will eventually be serialized.
        Hashtable addresses = new Hashtable();
        addresses.Add("Ram", "123 Main Road, Ameerpet, Hyderabad");
        addresses.Add("Hari", "20-10/123 X-Block, Whitefields, Bangalore");
        addresses.Add("Lalit", "344-10-98 First Cross, Anna Nagar, Chennai");
        // To serialize the hashtable (and its key/value pairs),
        // you must first open a stream for writing.
        // Use a file stream here.
        FileStream fs = new FileStream(@"e:\DataFile.soap", FileMode.Create);
```

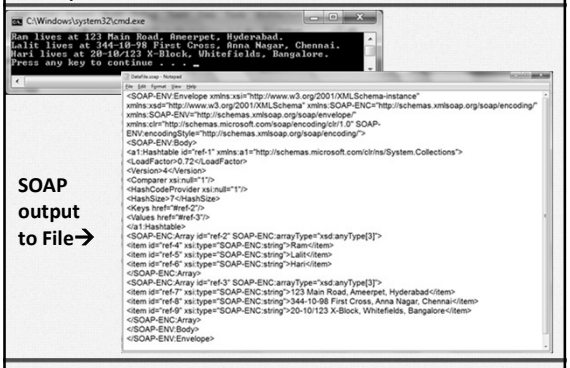
## Simple Example SOAP Formatter

```
// Construct a SoapFormatter and use it
        // to serialize the data to the stream.
        SoapFormatter formatter = new SoapFormatter();
        try
        {
            formatter.Serialize(fs, addresses);
        }
        catch (SerializationException e)
        {
            Console.WriteLine("Failed to serialize. Reason: " + e.Message);
            throw;
        }
        finally
        {
            fs.Close();
        }
    }
    static void Deserialize()
    {
        // Declare the hashtable reference.
        Hashtable addresses = null;

        // Open the file containing the data that you want to deserialize.
        FileStream fs = new FileStream(@"E:\DataFile.soap", FileMode.Open);
        try
        {
            SoapFormatter formatter = new SoapFormatter();
```

## Simple Example SOAP Formatter

```
// Deserialize the hashtable from the file and
            // assign the reference to the local variable.
            addresses = (Hashtable)formatter.Deserialize(fs);
        }
        catch (SerializationException e)
        {
            Console.WriteLine("Failed to deserialize. Reason: " + e.Message);
            throw;
        }
        finally
        {
            fs.Close();
        }

        // To prove that the table deserialized correctly,
        // display the key/value pairs to the console.
        foreach (DictionaryEntry de in addresses)
        {
            Console.WriteLine("{0} lives at {1}.", de.Key, de.Value);
        }
    }
}
```

## Output




## Data Contract Serialization

- Data contract serialization uses DataContractAttribute, DataMemberAttribute, DataContractSerializer, and NetDataContractSerializer attributes for serialization.
- DataContractSerializer is faster than XmlSerializer. WCF proxy classes use XmlSerializer when DataContractSerializer is not supported.

Presented by Ranjan Bhatnagar

## Example : Data Contract

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Xml;
[DataContract]
class Employee
{
    [DataMember(Name = "EmployeeName")]
    internal string Name;
    [DataMember(Name = "EmployeeAge")]
    internal int Age;
    public Employee(string newName, int newAge)
    {
        Name = newName;
        Age = newAge;
    }
}
```

## Example : Data Contract Example

```
class Program
{
    public static void Main()
    {
        Employee p1 = new Employee("Ram Kumar", 22);
        XmlDictionaryWriter writer = null;
        FileStream fs = null;
        DataContractSerializer ser = null;
        // Write the object to Employee.xml
        try
        {
            fs = new FileStream("E:\\Employee.xml", FileMode.Create);
            writer = XmlDictionaryWriter.CreateTextWriter(fs);
            ser = new DataContractSerializer(typeof(Employee));
            ser.WriteObject(writer, p1);
            Console.WriteLine("Completed writing Object");
        }
        catch (SerializationException se)
        {
            Console.WriteLine("Exception : " + se.Message);
            Console.WriteLine(se.Data);
        }
        finally
        {
            writer.Close();
            fs.Close();
        }
```

## Example : Data Contract Example

```
 // Read the Object from Employee.xml
        fs = null;
        ser = null;
        XmlDictionaryReader reader = null;
        try
        {
            fs = new FileStream("E:\\Employee.xml", FileMode.OpenOrCreate);
            reader = XmlDictionaryReader.CreateTextReader(fs,
                        new XmlDictionaryReaderQuotas());
            ser = new DataContractSerializer(typeof(Employee));
            Employee newPerson = (Employee)ser.ReadObject(reader);
            Console.WriteLine("Reading Employee object:" +
                        newPerson.Name + " " + newPerson.Age);
        }
        catch (SerializationException se)
        {
            Console.WriteLine("Exception : " + se.Message);
            Console.WriteLine(se.Data);
        }
        finally
        {
            reader.Close();
            fs.Close();
        }
    }
}
```

Thanks

Presented by

Ranjan Bhatnagar