msdn® training

Module 10: Data Streams and Files

Contents

Overview	1
Streams	2
Readers and Writers	5
Basic File I/O	8
Lab 10: Files	21
Review	26

Microsoft®

Instructor Notes

Presentation:
45 Minutes

Lab:
45 Minutes

After completing this module, students will be able to:

- Use **Stream** objects to read and write bytes to backing stores, such as strings and files.
- Use **BinaryReader** and **BinaryWriter** objects to read and write primitive types as binary values.
- Use **StreamReader** and **StreamWriter** objects to read and write characters to a stream.
- Use **StringReader** and **StringWriter** objects to read and write characters to strings.
- Use **Directory** and **DirectoryInfo** objects to create, move, and enumerate through directories and subdirectories.
- Use **FileSystemWatcher** objects to monitor and react to changes in the file system.
- Explain the key features of the Microsoft® .NET Framework isolated storage mechanism.

Materials and Preparation

This section provides the materials and preparation tasks that you need to teach this module.

Required Materials

To teach this module, you need the Microsoft PowerPoint® file 2349B_10.ppt.

Preparation Tasks

To prepare for this module, you should:

- Read all of the materials for this module.
- Complete the lab.

Module Strategy

Use the following strategy to present this module:

- Streams

  Briefly review fundamental stream operations and introduce the stream classes that are provided by **System.IO**. Point out that this module discusses synchronous operations only; asynchronous operations are beyond the scope of this course.

  Tell students that the **NetworkStream** class is covered in more detail in Module 11, "Internet Access," in Course 2349B, *Programming with the Microsoft .NET Framework (Microsoft Visual C#™.NET)*.

- Readers and Writers

  Cover the commonly used reader and writer classes that are used to input and output to streams and strings that use types other than bytes.

- Basic File I/O

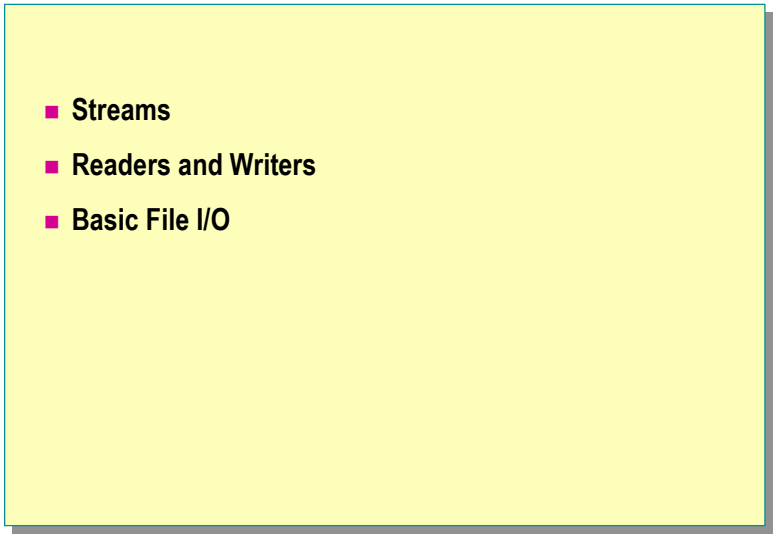
  Discuss in more detail the stream classes that are provided by **System.IO** for manipulating files and directories.

  Discuss the security issues that are associated with writing code that will be downloaded over the Internet.

Overview

Topic Objective
To provide an overview of the module topics and objectives.

Lead-in
In this module, you will learn about how to use types that allow reading from and writing to data streams and files.



The **System.IO** namespace contains types that allow synchronous and asynchronous reading from and writing to data streams and files. This module discusses synchronous operations only, because asynchronous operations are beyond the scope of this course.

After completing this module, you will be able to:

- Use **Stream** objects to read and write bytes to backing stores, such as strings and files.
- Use **BinaryReader** and **BinaryWriter** objects to read and write primitive types as binary values.
- Use **StreamReader** and **StreamWriter** objects to read and write characters to a stream.
- Use **StringReader** and **StringWriter** objects to read and write characters to strings.
- Use **Directory** and **DirectoryInfo** objects to create, move, and enumerate through directories and subdirectories.
- Use **FileSystemWatcher** objects to monitor and react to changes in the file system.
- Explain the key features of the Microsoft® .NET Framework isolated storage mechanism.

For Your Information
When you talk about a particular class, you may want to display the class information for **System.IO** from the .NET Framework Reference section in the .NET Framework SDK.

Streams

Topic Objective
To introduce the functions of the **Stream** class and its subclasses.

Lead-in
Streams provide a way to read and write bytes from and to a backing store. A backing store is a storage medium, such as a disk or memory.

- **A Way to Read and Write Bytes from and to a Backing Store**
  - Stream classes inherit from **System.IO.Stream**
- **Fundamental Stream Operations: Read, Write, and Seek**
  - **CanRead**, **CanWrite**, and **CanSeek** properties
- **Some Streams Support Buffering for Performance**
  - **Flush** method outputs and clears internal buffers
- **Close Method Frees Resources**
  - **Close** method performs an implicit **Flush** for buffered streams
- **Stream Classes Provided by the .NET Framework**
  - **NetworkStream**, **BufferedStream**, **MemoryStream**, **FileStream**, **CryptoStream**
- **Null Stream Instance Has No Backing Store**

Streams provide a way to read and write bytes from and to a backing store. A *backing store* is a storage medium, such as a disk or memory.

All classes that represent streams inherit from the **Stream** class. The **Stream** class and its subclasses provide a generic view of data sources and repositories, and isolate the programmer from the specific details of the operating system and underlying devices.

Fundamental Stream Operations

Streams allow you to perform three fundamental operations:

1. You can read from streams.

   Reading is the transfer of data from a stream into a data structure, such as an array of bytes.

2. You can write to streams.

   Writing is the transfer of data from a data structure into a stream.

3. Streams can support seeking.

   Seeking is the querying and modifying of the current position within a stream. Seek capability depends on the kind of backing store that a stream has. For example, network streams have no unified concept of a current position and therefore typically do not support seeking.

Depending on the underlying data source or repository, streams may support only some of these capabilities. An application can query a stream for its capabilities by using the **CanRead**, **CanWrite**, and **CanSeek** properties.

The **Read** and **Write** methods read and write byte data. For streams that support seeking, the **Seek** and **SetLength** methods and the **Position** and **Length** properties can be used to query and modify the current position and length of a stream.

Support for Buffering

Some stream implementations perform local buffering of the underlying data to improve performance. For such streams, you can use the **Flush** method to clear internal buffers and ensure that all data has been written to the underlying data source or repository.

Calling the **Close** method on a stream flushes any buffered data, essentially calling the **Flush** method for you. The **Close** method also releases operating system resources, such as file handles, network connections, or memory that is used for any internal buffering.

Stream Classes Provided by the .NET Framework

The .NET Framework contains several stream classes that derive from the **System.IO.Stream** class. The **System.Net.Sockets** namespace contains the **NetworkStream** class. **NetworkStream** provides the underlying stream of data for network access and will be discussed in more detail in Module 11, "Internet Access," in Course 2349B, *Programming with the Microsoft .NET Framework (Microsoft Visual C#™.NET).*

The **System.IO** namespace contains the **BufferedStream**, **MemoryStream**, and **FileStream** classes, which are derived from the **System.IO.Stream** class.

BufferedStream Class

The **BufferedStream** class is used to buffer reads and writes to another stream. A buffer is a block of bytes in memory that is used to cache data, thereby reducing the number of calls to the operating system. Buffers thus can be used to improve read and write performance. Another class cannot inherit from the **BufferedStream** class.

MemoryStream Class

The **MemoryStream** class provides a way to create streams that have memory as a backing store, instead of a disk or a network connection. The **MemoryStream** class creates a stream out of an array of bytes.

FileStream Class

The **FileStream** class is used for reading from and writing to files. By default, the **FileStream** class opens files synchronously, but it provides a constructor to open files asynchronously.

CryptoStream Class

The **CryptoStream** class defines a stream that links data streams to cryptographic transformations. The common language runtime uses a stream-oriented design for cryptography. The core of this design is **CryptoStream**. Any cryptographic objects that implement **CryptoStream** can be chained together with any objects that implement **Stream**, so the streamed output from one object can be fed into the input of another object. The intermediate result (the output from the first object) does not need to be stored separately. For further details about the **CryptoStream** class see the .NET Framework SDK.

Null Stream Instance

There are times when an application needs a stream that simply discards its output and returns no input. You can obtain such a stream that has no backing store and that will not consume any operating resources from the **Stream** class's public static field named **Null**.

For example, you may code an application to always write its output to the **FileStream** that is specified by the user. When the user does not want an output file, the application directs its output to the **Null** stream. When the **Write** methods of **Stream** are invoked on this **Null** stream, the call simply returns, and no data is written. When the **Read** methods are invoked, the **Null** stream returns zero without reading data.

Readers and Writers

Topic Objective
To show how reader and writer classes are used to input and output to streams and strings.

Lead-in
As previously mentioned, the **Stream** class is designed for byte input and output. You can use the reader and writer classes to input and output to streams and strings using other types.

- **Classes That Are Derived from System.IO.Stream Take Byte Input and Output**
- **Readers and Writers Take Other Types of Input and Output and Read and Write Them to Streams or Strings**
- **BinaryReader and BinaryWriter Read and Write Primitive Types to a Stream**
- **TextReader and TextWriter Are Abstract Classes That Implement Read Character and Write Character Methods**
- **TextReader and TextWriter Derived Classes Include:**
  - **StreamReader** and **StreamWriter**, which read and write to a stream
  - **StringReader** and **StringWriter**, which read and write to a string and **StringBuilder** respectively

As discussed in Streams in this module, the **Stream** class is designed for byte input and output. You can use the reader and writer classes to input and output to streams and strings that use other types.

The following table describes some commonly used reader and writer classes.

Class	Description
BinaryReader and **BinaryWriter**	These classes read and write primitive types as binary values in a specific encoding to and from a stream.
TextReader and **TextWriter**	The implementations of these classes are designed for character input and output.
StreamReader and **StreamWriter**	These classes are derived from the **TextReader** and **TextWriter** classes, and read and write their characters to a stream.
StringReader and **StringWriter**	Theses classes also derive from the **TextReader** and **TextWriter** classes, but read their characters from a string and write their characters to a **StringBuilder** class.

A reader or writer is attached to a stream so that the desired types can be read or written easily.

The following example shows how to write data of type **Integer** to and read from a new, empty file stream that is named Test.data. After creating the data file in the current directory, the **BinaryWriter** class is used to write the integers 0 through 10 to Test.data. Then the **BinaryReader** class reads the file and displays the file's content to the console.

```
using System;
using System.IO;

class MyStream {
   private const string FILE_NAME = "Test.data";
   public static void Main(String[] args) {
      // Create the new, empty data file.
      if (File.Exists(FILE_NAME)) {
         Console.WriteLine("{0} already exists!", FILE_NAME);
         return;
      }
      FileStream fs = new FileStream(FILE_NAME,
         FileMode.CreateNew);
      // Create the writer for data.
      BinaryWriter w = new BinaryWriter(fs);
      // Write data to Test.data.
      for (int i = 0; i < 11; i++) {
         w.Write( (int) i);
      }
      w.Close();
      fs.Close();
      // Create the reader for data.
      fs = new FileStream(FILE_NAME, FileMode.Open,
          FileAccess.Read);
      BinaryReader r = new BinaryReader(fs);
      // Read data from Test.data.
      for (int i = 0; i < 11; i++) {
         Console.WriteLine(r.ReadInt32());
         w.Close();
      }
   }
}
```

In the following example, the code defines a string and converts it to an array of characters, which can then be read as desired by using the appropriate **StringReader.Read** method:

```
using System;
using System.IO;

public class CharsFromStr
{
   public static void Main(String[] args) {
     // Create a string to read characters from.
     String str = "Some number of characters";
     // Size the array to hold all the characters of the
     // string, so that they are all accessible.
     char[] b = new char[24];
     // Create a StringReader and attach it to the string.
     StringReader sr = new StringReader(str);
     // Read 13 characters from the array that holds
     // the string, starting from the first array member.
     sr.Read(b, 0, 13);
     // Display the output.
     Console.WriteLine(b);
     // Close the StringReader.
     sr.Close();
   }
}
```

The preceding example produces the following output:

```
Some number o
```

System.Text.Encoding

Internally, the common language runtime represents all characters as Unicode. However, Unicode can be inefficient when transferring characters over a network or when persisting in a file. To improve efficiency, the .NET Framework class library provides several types that are derived from the **System.Text.Encoding** abstract base class. These classes know how to encode and decode Unicode characters to ASCII, UTF-7, UTF-8, Unicode, and other arbitrary code pages. When you construct a **BinaryReader**, **BinaryWriter**, **StreamReader**, or **StreamWriter**, you can choose any of these encodings. The default encoding is UTF-8.

◆ Basic File I/O

Topic Objective
To introduce the classes of the **System.IO** namespace, which are discussed in this section.

Lead-in
The .NET Framework's **System.IO** namespace provides a number of useful classes for manipulating files and directories.

- **FileStream Class**
- **File and FileInfo Class**
- **Reading Text Example**
- **Writing Text Example**
- **Directory and DirectoryInfo Class**
- **FileSystemWatcher**
- **Isolated Storage**

The .NET Framework's **System.IO** namespace provides a number of useful classes for manipulating files and directories.

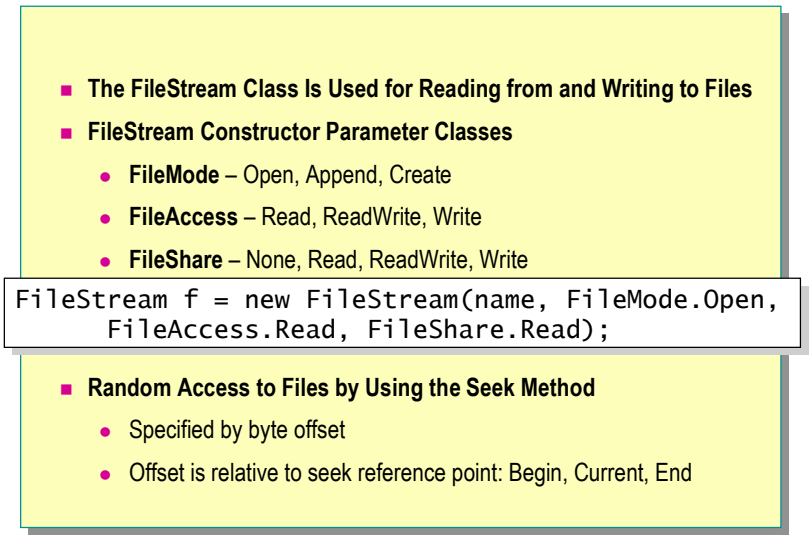
Important Default security policy for the Internet and intranets does not allow access to files. Therefore, do not use the regular, nonisolated storage IO classes if you are writing code that will be downloaded over the Internet. Use **Isolated Storage** instead.

Caution When a file or network stream is opened, a security check is performed only when the stream is constructed. Therefore, be careful when handing off these streams to less trusted code or application domains.

FileStream Class

Topic Objective
To define the **FileStream** class and the types that are used as parameters in some **FileStream** constructors.

Lead-in
The **FileStream** class is used for reading from and writing to files. The **FileMode**, **FileAccess**, and **FileShare** types are used as parameters in some **FileStream** constructors.



The **FileStream** class is used for reading from and writing to files. The **FileMode**, **FileAccess**, and **FileShare** types are used as parameters in some **FileStream** constructors.

FileMode Parameter

FileMode parameters control whether a file is overwritten, created, or opened, or any combination of those operations. The following table describes constants that are used with the **FileMode** parameter class.

Constant	Description
Open	This constant is used to open an existing file.
Append	This constant is used to append to a file.
Create	This constant is used to create a file if it does not exist.

FileAccess Enumeration

The **FileAccess** enumeration defines constants for read, write, or read/write access to a file. This enumeration has a **FlagsAttribute** that allows a bitwise combination of its member values. A **FileAccess** parameter is specified in many of the constructors for **File**, **FileInfo**, and **FileStream**, and in other class constructors where it is important to control the kind of access that users have to a file.

FileShare Enumeration

The **FileShare** enumeration contains constants for controlling the kind of access that other **FileStreams** can have to the same file. This enumeration has a **FlagsAttribute** that allows a bitwise combination of its member values.

The **FileShare** enumeration is typically used to define whether two processes can simultaneously read from the same file. For example, if a file is opened and **FileShare.Read** is specified, other users can open the file for reading but not for writing. **FileShare.Write** specifies that other users can simultaneously write to the same file. **FileShare.None** declines sharing of the file.

In the following example, a **FileStream** constructor opens an existing file for read access and allows other users to read the file simultaneously:

```
FileStream f = new FileStream(name, FileMode.Open,
  FileAccess.Read, FileShare.Read);
```

Using the Seek Method for Random Access to Files

FileStream objects support random access to files by using the **Seek** method. The **Seek** method allows the read/write position within the file stream to be moved to any position within the file. The read/write position can be moved by using byte offset reference point parameters.

The byte offset is relative to the seek reference point, as represented by the three properties of the **SeekOrigin** class, which are described in the following table.

Property Name	Description
Begin	The seek reference position of the beginning of a stream.
Current	The seek reference position of the current position within a stream.
End	The seek reference position of the end of a stream.

File and FileInfo Class

Topic Objective
To introduce the **File** and **FileInfo** classes and demonstrate how they are used to create a new object.

Lead-in
The **File** and **FileInfo** classes are utility classes with methods that are primarily used for the creation, copying, deletion, moving, and opening of files.

- **File Is a Utility Class with Static Methods Used to:**
  - Create, copy, delete, move, and open files
- **FileInfo Is a Utility Class with Instance Methods Used to:**
  - Create, copy, delete, move, and open files
  - Can eliminate some security checks when reusing an object.
- **Example:**
  - Assign to **aStream** a newly created file named foo.txt in the current directory

```
FileStream aStream = File.Create("foo.txt");
```


The **File** and **FileInfo** classes are utility classes with methods that are primarily used for the creation, copying, deletion, moving, and opening of files.

All methods of the **File** class are static and can therefore be called without having an instance of a file. The **FileInfo** class contains all instance methods. The static methods of the **File** class perform security checks on all methods. If you are going to reuse an object several times, consider using the corresponding instance method of **FileInfo** instead, because the security check will not always be necessary.

For example, to create a file named Foo.txt and return a **FileStream** object, use the following code:

```
FileStream aStream = File.Create("Foo.txt");
```

To create a file named Foo.txt and return a **StreamWriter** object, use the following code:

```
StreamWriter sw = File.CreateText("Foo.txt");
```

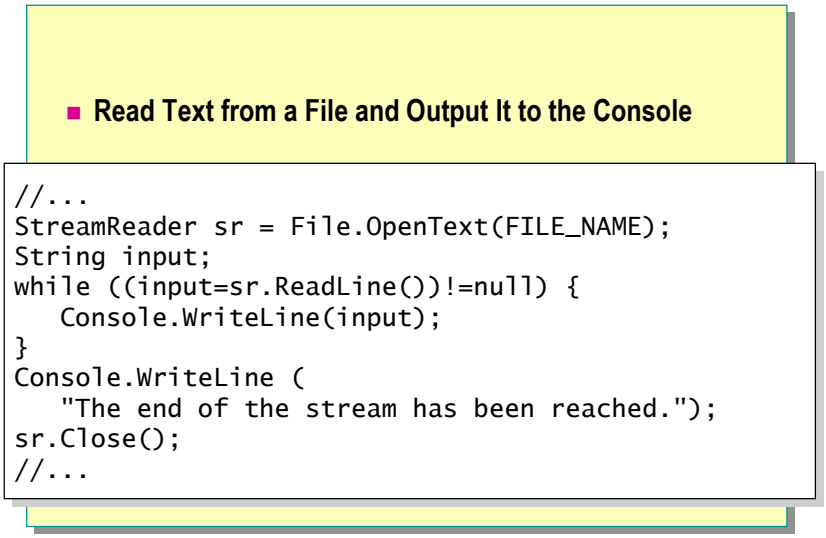
To open a file named Foo.txt and return a **StreamReader** object, use the following code:

```
StreamReader sr = File.OpenText("Foo.txt");
```

Reading Text Example

Topic Objective
To provide an example of reading.

Lead-in
In the following example, you read an entire file and are notified when the end of the file is detected.



In the following example of reading text, you read an entire file and are notified when the end of the file is detected.

```
using System;
using System.IO;
public class TextFromFile {
   private const string FILE_NAME = "MyFile.txt";
   public static void Main(String[] args) {
      if (!File.Exists(FILE_NAME)) {
         Console.WriteLine("{0} does not exist!", FILE_NAME);
         return;
      }
      StreamReader sr = File.OpenText(FILE_NAME);
      String input;
      while ((input=sr.ReadLine())!=null) {
         Console.WriteLine(input);
      }
      Console.WriteLine (
        "The end of the stream has been reached.");
      sr.Close();
   }
}
```

This code creates a **StreamReader** object that points to a file named MyFile.txt through a call to **File.OpenText**. **StreamReader.ReadLine** returns each line as a string. When there are no more characters to read, a message is displayed to that effect, and the stream is closed.

Writing Text Example

Topic Objective
To provide an example of writing text.

Lead-in
This example creates a new text file that is named MyFile.txt, writes a string, integer, and floating point number to it, and finally closes the file.

- Create a File
- Write a String, an Integer, and a Floating Point Number
- Close the File

```
//...
StreamWriter sw = File.CreateText("MyFile.txt");
sw.WriteLine ("This is my file");
sw.WriteLine (
   "I can write ints {0} or floats {1}", 1, 4.2);
sw.Close();
//...
```


The following example creates a new text file that is named MyFile.txt, writes a string, integer, and floating-point number to it, and finally closes the file.

```
using System;
using System.IO;

public class TextToFile {
   private const string FILE_NAME = "MyFile.txt";
   public static void Main(String[] args) {
      if (File.Exists(FILE_NAME)) {
         Console.WriteLine("{0} already exists!", FILE_NAME);
         return;
      }
      StreamWriter sw = File.CreateText(FILE_NAME);
      sw.WriteLine ("This is my file.");
      sw.WriteLine (
         "I can write ints {0} or floats {1}, and so on.",
         1, 4.2);
      sw.Close();
   }
}
```

Directory and DirectoryInfo Class

Topic Objective
To explain how the **Directory** and **DirectoryInfo** classes are used to create directory listings.

Lead-in
The **Directory** and **DirectoryInfo** classes expose routines for creating, moving, and enumerating through directories and subdirectories.

- **Directory Has Static Methods Used to:**
  - Create, move, and enumerate through directories and subdirectories
- **DirectoryInfo Has Instance Methods Used to:**
  - Create, move, and enumerate through directories and subdirectories
  - Can eliminate some security checks when reusing an object
- **Example:**
  - Enumerating through the current directory

```
DirectoryInfo dir = new DirectoryInfo(".");
foreach (FileInfo f in dir.GetFiles("*.cs")) {
  String name = f.FullName; }
```

- **Use Path Class Objects to Process Directory Strings**

The **Directory** and **DirectoryInfo** classes expose routines for creating, moving, and enumerating through directories and subdirectories. All methods of the **Directory** class are static and can therefore be called without having an instance of a directory. The **DirectoryInfo** class contains all instance methods. The static methods of the **Directory** class do a security check 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 then not always be necessary.

The following example shows how to use the **DirectoryInfo** class to create a listing of a directory:

```
using System;
using System.IO;

class DirectoryLister
{
   public static void Main(String[] args)
   {
      DirectoryInfo dir = new DirectoryInfo(".");
      foreach (FileInfo f in dir.GetFiles("*.cs"))
      {
         String name = f.FullName;
         long size = f.Length;
         DateTime creationTime = f.CreationTime;
         Console.WriteLine("{0,-12:N0} {1,-20:g} {2}", size,
            creationTime, name);
      }
   }
}
```

In the preceding example, the **DirectoryInfo** object is the current directory, denoted by (".."). The code lists the names of all of the files in the current directory that have a .cs extension, together with their file size and creation time.

Assuming that there are .cs files in the \Bin subdirectory of drive C, the output of this code appears as follows:

```
953          7/20/2000 10:42 AM   C:\Bin\paramatt.cs
664          7/27/2000 3:11 PM    C:\Bin\tst.cs
403          8/8/2000 10:25 AM    C:\Bin\dirlist.cs
```

If you want a list of files in another directory, such as C:\, remember to use the backslash (\) escape character, as in the following example:

```
"C:\\"
```

Or, use an @-quoted string literal in C#, as in the following example:

```
@"C:\"
```

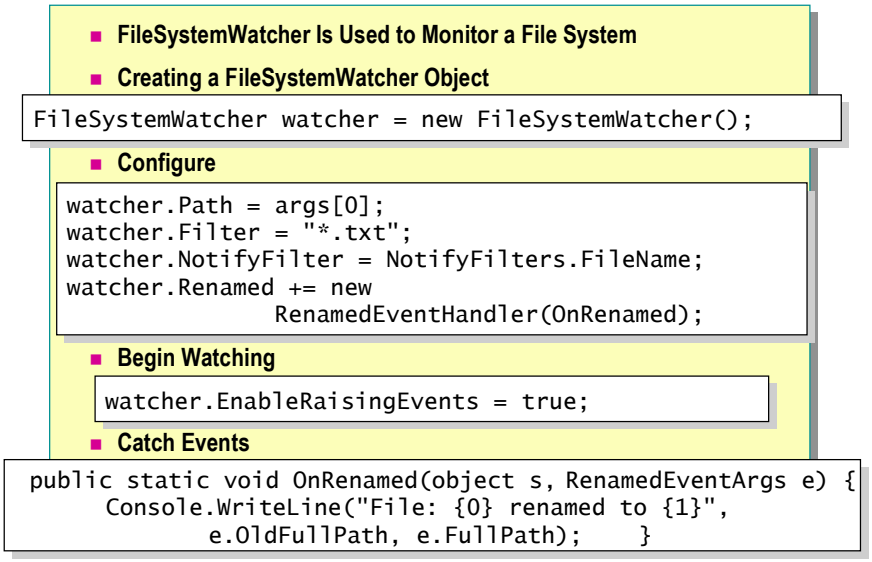
Paths

To processes directory strings in a cross-platform manner, use the **Path** class. The members of the **Path** class enable you to quickly and easily perform common operations, such as determining whether a file extension is part of a path, and combining two strings into one path name.

FileSystemWatcher

Topic Objective
To explain how the **FileSystemWatcher** component can be used to monitor and react to changes in a file system.

Lead-in
You use the **FileSystemWatcher** component to monitor a file system and react when changes to it occur.



You use the **FileSystemWatcher** component to monitor a file system and react when changes to it occur. By using the **FileSystemWatcher** component, you can quickly and easily launch business processes when specified files or directories are created, modified, or deleted.

For example, if a group of users is collaborating on a document that is stored in a shared directory on a server, you can use the **FileSystemWatcher** component to easily program your application to watch for changes to the shared directory. When a change is detected, the component can run processing that notifies each user through e-mail.

You can configure the component to watch an entire directory and its contents or a specific file or set of files within a specific directory. To watch for changes in all files, set the **Filter** property to an empty string (""). To watch a specific file, set the **Filter** property to the file name. 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".

Note Hidden files are not ignored.

There are several types of changes you can watch for in a directory or file. For example, you can watch for changes in **Attributes**, the **LastWrite** date and time, or the **Size** of files or directories. This is done by setting the **FileSystemWatcher.NotifyFilter** property to one of the **NotifyFilters** values. For more information on the type of changes you can watch, see **NotifyFilters** in the .NET Framework Software Development Kit (SDK).

You can watch for renaming, deletion, or creation of files or directories. For example, to watch for renaming of text files, set the **Filter** property to "*.txt" and call one of the **WaitForChanged** methods with the **WatcherChangeTypes** value **Renamed** provided.

Creating a FileSystemWatcher Component

The following example creates a **FileSystemWatcher** component to watch the directory that is specified at run time. The component is set to watch for changes in **LastWrite** and **LastAccess** times, and the creation, deletion, or renaming of text files in the directory. If a file is changed, created, or deleted, the path to the file prints to the console. When a file is renamed, the old and new paths print to the console.

```
using System;
using System.IO;

public class Watcher
{

    public static void Main(string[] args)
    {
        // If a directory is not specified, exit program.
        if(args.Length != 1)
        {
            // Display the proper way to call the program.
            Console.WriteLine(
              "Usage: Watcher.exe (directory)");
            return;
        }

        // Create a new FileSystemWatcher
        // and set its properties.
        FileSystemWatcher watcher = new FileSystemWatcher();
        watcher.Path = args[0];
        /* Watch for changes in LastAccess and LastWrite
           times, and the renaming of files or directories */
        watcher.NotifyFilter =
               NotifyFilters.LastAccess |
               NotifyFilters.LastWrite |
               NotifyFilters.FileName |
               NotifyFilters.DirectoryName;
        // Only watch text files.
        watcher.Filter = "*.txt";

        // Add event handlers.

        // The Changed event occurs when changes are made to
        // the size, system attributes, last write time, last
        // access time, or security permissions in a file or
        // directory in the specified Path of a
        // FileSystemWatcher.
        watcher.Changed += new
               FileSystemEventHandler(OnChanged);
```

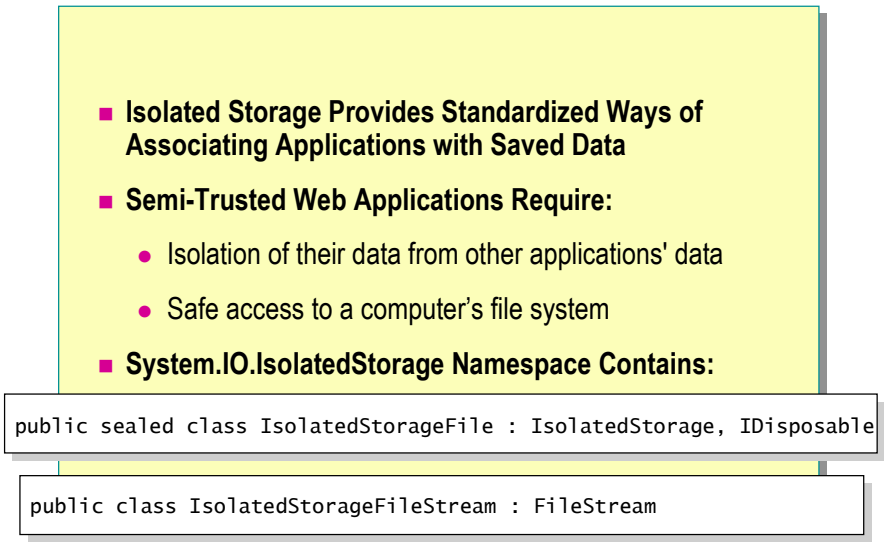
(*Code continued on the following page.*)

```
            // The Created event occurs when a file or directory
            // in the specified Path of a FileSystemWatcher is
            // created.
            watcher.Created += new
                   FileSystemEventHandler(OnChanged);

            // The Deleted event occurs when a file or directory
            // in the specified Path of a FileSystemWatcher is
            // deleted.
            watcher.Deleted += new
                   FileSystemEventHandler(OnChanged);

            // The Deleted event occurs when a file or directory
            // in the specified Path of a FileSystemWatcher is
            // deleted.

            watcher.Renamed += new
                   RenamedEventHandler(OnRenamed);

            // Begin watching.
            watcher.EnableRaisingEvents = true;

            // Wait for the user to quit the program.
            Console.WriteLine("Press \'q\' to quit the sample.");
            while(Console.Read()!='q');
        }
        // Define the event handlers.
        public static void OnChanged(
                    object source, FileSystemEventArgs e)
        {
            // Specify what is done when a file is changed,
            //  created, or deleted.
           Console.WriteLine("File: " +  e.FullPath + " " +
                   e.ChangeType);
        }

        public static void OnRenamed(
                    object source, RenamedEventArgs e)
        {
            // Specify what is done when a file is renamed.
            Console.WriteLine("File: {0} renamed to {1}",
                  e.OldFullPath, e.FullPath);
        }
    }
```

Isolated Storage

Topic Objective
To introduce isolated storage and its potential uses.

Lead-in
For some applications, such as downloaded Web applications and code that may come from untrusted sources, the basic file system does not provide the necessary isolation and safety.



Basic file I/O functionality, found in the **System.IO** root, provides the ability to access, store, and manipulate data that is stored in hierarchical file systems whose files are referenced by using unique paths. For some applications, such as downloaded Web applications and code that may come from un-trusted sources, the basic file system does not provide the necessary isolation and safety. *Isolated storage* is a data storage mechanism that provides isolation and safety by defining standardized ways of associating code with saved data.

Isolation

When an application stores data in a file, the file name and storage location must be carefully chosen to minimize the possibility that the storage location will be known to another application and, therefore, vulnerable to corruption. Isolated storage provides the means to manage downloaded Web application files to minimize storage conflicts.

Security Risks of Semi-Trusted Code

It is important to restrict semi-trusted code's access from a computer's file system. Allowing code that has been downloaded and run from the Internet to have access to I/O functions leaves a system vulnerable to viruses and unintentional damage.

The security risks associated with file access are sometimes addressed by using access control lists (ACLs), which restrict the access that users have to files. However, this approach is often not feasible with Web applications because it requires administrators to configure ACLs on all of the systems on which the application will run.

Safety Through Isolated Storage

Administrators can use tools that are designed to manipulate isolated storage to configure file storage space, set security policies, and delete unused data. With isolated storage, code no longer needs to invent unique paths to specify safe locations in the file system, while data is protected from unauthorized access. There is no need for hard coding of information that indicates where an application's storage area is located. With isolated storage, partially trusted applications can store data in a manner that is controlled by the computer's security policy. Security policies rarely grant permission to access the file system by using standard I/O mechanisms. However, by default, code that runs from a local computer, a local network, or the Internet is granted the right to use isolated storage. Web applications can also use isolated storage with roaming user profiles, thereby allowing a user's isolated stores to roam with their profile.

The namespace **System.IO.IsolatedStorage** contains the **IsolatedStorageFile** and **IsolatedStorageFileStream** classes, which applications can use to access the files and directory in their isolated storage area.

Further discussion of isolated storage is beyond the scope of this course.

Lab 10: Files

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will create an application that reads and writes characters to and from files, and create an application that can use **StringReader** and **StreamReader** objects to read character data from either files or strings.

Objectives

After completing this lab, you will be able to:

- Create an application that reads and writes characters to and from files.
- Create an application that can use **StringReader** and **StreamReader** objects to read character data from files or strings.

Lab Setup

Starter and solution files are associated with this lab. The starter files are in the folder <*install folder*>\Labs\Lab10\Starter, and the solution files are in the folder <*install folder*>\Labs\Lab10\Solution.

Scenario

In this lab, you are provided with a Microsoft Visual Studio® .NET console application as a starting point. The application, named Files, opens one or more files, which are specified on the command line, and counts each file's bytes, characters, words, and lines. The results from each file and the total of all files are displayed on the console.

The application supports an **–f** switch to allow the output display to be redirected to a file and a **–t** switch to run a special test mode of operation where input is obtained from a coded-in test string, instead of from user-specified files. The application is based on a slightly modified version of the .NET Framework SDK sample, Word Count.

Estimated time to complete this lab: 45 minutes

Exercise 1
Reading and Writing Files and Strings

In this exercise, you will modify the Files application to output to a specified file.

► **Examine the application**

1. In Visual Studio .NET, open the **Files** project, which is located in <*install folder*>\Labs\Lab10\Starter\Files.
2. Open the WordCount.cs file and examine the code. Pay attention to those methods that you will be modifying: The **Main** method of the **Application** class and the **CountStats** method of the **WordCounter** class.
3. Build the WordCount application.
4. Set the application's command line arguments by performing the following steps:
   a. If the Solution Explorer pane in the Visual Studio .NET window is not visible, on the **View** menu, click **Solution Explorer**.
   b. In the Solution Explorer pane, right-click **Files** to display the context-sensitive menu, and then click **Properties**.
   c. In the **Files Property Pages** dialog box, click the **Configuration Properties** folder, and then click **Debugging**.
   d. In the right pane under **Start Options**, set the command line arguments, as follows:

      ```
      -a -o -t -foutput.txt test.txt
      ```

5. View the WordCount.cs file, and locate the last line in **Application**'s **Main** method, which is:

   ```
   return 0;
   ```

6. Right-click this line, and select **Run To Cursor**. You should see the following output:

   ```
   Replace this Console.WriteLine in Application's Main➥
    method with code as per the lab
   Lines   Words   Chars   Bytes   Pathname
   Replace this Console.WriteLine in WordCounter's➥
    CountStats method
       0       0       0       0   Test String
   -----   -----   -----   -----   --------------------
       0       0       0       0   Total in all files
   Word usage sorted alphabetically (0 unique words)
   Word usage sorted by occurrence (0 unique words)
   ```

7. Stop debugging, and in the **Main** method code of the **Application** class, locate the following line of code:

   ```
   if (ap.OutputFile != null) {
   ```

   If ap.OutputFile does not contain **null**, then it contains the name of the output file that is specified in the **–f** command line switch.

8. Replace the following line's **Console.WriteLine** call with code that:
   a. Assigns **fsOut** to a **FileStream** object that creates a file with the specified name with Write access and no file sharing.
   b. Assigns **sw** to a **StreamWriter** object that is bound to the **FileStream** object created in step a.
   c. Redirects the console's output to the file by associating the **StreamWriter** object with the console by using the following command:

      ```
      Console.SetOut(sw);
      ```

9. Rebuild and run the application, and examine the output file that is specified in the command line switch options.

   The file should be located in the bin\Debug subdirectory. It should contain the following text:

```
Lines  Words   Chars   Bytes   Pathname
Replace this Console.WriteLine in WordCounter's➥
 CountStats method
    0      0       0       0   Test String
-----  -----   -----   -----   --------------------
    0      0       0       0   Total in all files
Word usage sorted alphabetically (0 unique words)
Word usage sorted by occurrence (0 unique words)
```

► **Add the test mode and normal mode CountStats processing**

1. Locate the following lines in the **CountStats** method code of the **WordCounter** class:

```
Boolean Ok = true;
numLines = numWords = numChars = numBytes = 0;
try {
```

   If the **–t** option has been set, the **CountStats** caller will have set the pathname parameter to the empty string, which is called *String.Empty*.

2. Replace the following line's **Console.WriteLine** call with code that:
   a. Declares a variable of type **TextReader** and names it **tr**.

      You use the type **TextReader** because it is the common base type for both **StringReader** and **StreamReader**. Polymorphism will allow code that uses a **TextReader** object to be provided with either a **StringReader** or **StreamReader** object.

   b. If the pathname is empty:
      i. Create a **StringReader** that is named **sr**, which is bound to a member of **WordCounter** that is named **testString**.
      ii. Assign the number of bytes in this string to **numBytes**.
      iii. Assign **sr** to **tr**.

         This assignment does an implicit cast of a **StringReader** to a **TextReader**.

c. If the pathname is not empty:

   i. Create a **FileStream** that is named **fsIn** by opening the file that is specified in the parameter pathname with read access and shared read access.

   ii. Assign the number of bytes in **fsIn** to **numBytes**.

   iii. Create a **StreamReader** that is named **sr** that is bound to this file.

   iv. Assign **sr** to **tr**.

   This assignment does an implicit cast of a **StreamReader** to a **TextReader**

d. In this module, uncomment out the **for** loop that follows the comment:

```
// Process every line in the file
```

e. Following the **for** loop, add code to close the **TextReader** object.

3. Rebuild and run the application, and examine the output file in the bin\Debug subdirectory. It should contain the following text:

```
Lines  Words   Chars   Bytes   Pathname
    2      3      16      17   Test String
-----  -----   -----   -----   --------------------
    2      3      16      17   Total in all files
Word usage sorted alphabetically (2 unique words)
    2: "hello"
    1: "world"
Word usage sorted by occurrence (2 unique words)
    1: world
    2: hello
```

4. Change the command line options that Visual Studio .NET will use to run the application to remove the test switch. It should be set to:

```
-a -o -foutput.txt test.txt
```

5. Run the application, and examine the output file in the bin\Debug subdirectory. It should contain the following text:

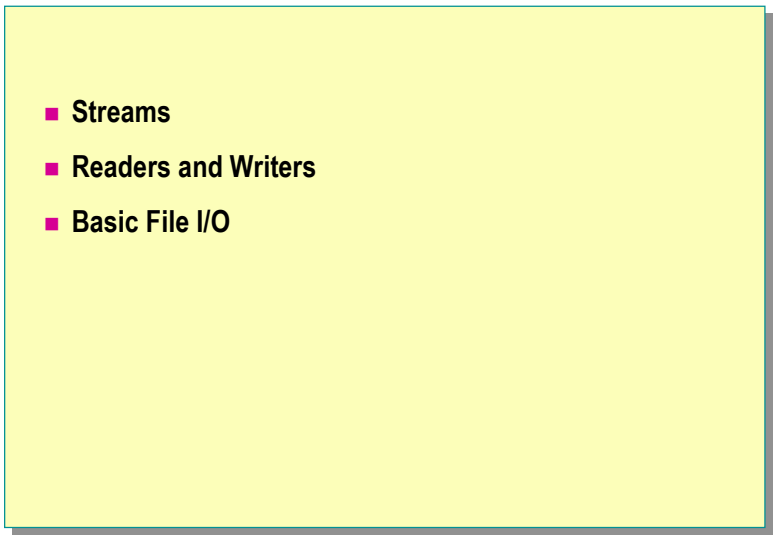
```
Lines  Words   Chars   Bytes   Pathname
    5     16      65      73   ...\test.txt
-----  -----   -----   -----   --------------------
    5     16      65      73   Total in all files
Word usage sorted alphabetically (14 unique words)
    1: "aid"
    1: "all"
    1: "come"
    1: "country"
    1: "for"
    1: "good"
    1: "is"
    1: "men"
    1: "now"
    1: "of"
    2: "the"
    1: "their"
    1: "time"
    2: "to"
Word usage sorted by occurrence (14 unique words)
    1: aid
    1: all
    1: come
    1: country
    1: for
    1: good
    1: is
    1: men
    1: now
    1: of
    1: their
    1: time
    2: the
    2: to
```

6. Examine the file **Test.txt** in the bin\Debug subdirectory, and verify that the output is what you expected.

Review

Topic Objective
To reinforce module objectives by reviewing key points.

Lead-in
The review questions cover some of the key concepts taught in the module.



1. Name at least three types of .NET Framework streams and how they differ.

   FileStream does reads and writes to a file.

   MemoryStream does reads and writes to memory.

   BufferedStream is used to buffer reads and writes to another stream.

   NetworkStream provides the underlying stream of data for network access.

2. Name the three basic stream operations.

   Read, Write, and Seek.

3. Name the classes that are used to read and write primitive types as binary values.

   BinaryReader and BinaryWriter.

4. Name the method used to provide random access to files.

   Seek.

5. Name the class that you would use to monitor changes to a file system.

   FileSystemWatcher.

6. Name the two important features that the .NET Framework's isolated storage provides for an application.

   Isolation and Safety.