File and Stream I/O

Article • 09/15/2021 • 7 minutes to read

File and stream I/O (input/output) refers to the transfer of data either to or from a storage medium. In .NET, the `System.IO` namespaces contain types that enable reading and writing, both synchronously and asynchronously, on data streams and files. These namespaces also contain types that perform compression and decompression on files, and types that enable communication through pipes and serial ports.

A file is an ordered and named collection of bytes that has persistent storage. When you work with files, you work with directory paths, disk storage, and file and directory names. In contrast, a stream is a sequence of bytes that you can use to read from and write to a backing store, which can be one of several storage mediums (for example, disks or memory). Just as there are several backing stores other than disks, there are several kinds of streams other than file streams, such as network, memory, and pipe streams.

Files and directories

You can use the types in the System.IO namespace to interact with files and directories. For example, you can get and set properties for files and directories, and retrieve collections of files and directories based on search criteria.

For path naming conventions and the ways to express a file path for Windows systems, including with the DOS device syntax supported in .NET Core 1.1 and later and .NET Framework 4.6.2 and later, see File path formats on Windows systems.

Here are some commonly used file and directory classes:

- File - provides static methods for creating, copying, deleting, moving, and opening files, and helps create a FileStream object.
- FileInfo - provides instance methods for creating, copying, deleting, moving, and opening files, and helps create a FileStream object.
- Directory - provides static methods for creating, moving, and enumerating through directories and subdirectories.
- DirectoryInfo - provides instance methods for creating, moving, and enumerating

through directories and subdirectories.

- Path - provides methods and properties for processing directory strings in a cross-platform manner.

You should always provide robust exception handling when calling filesystem methods. For more information, see Handling I/O errors.

In addition to using these classes, Visual Basic users can use the methods and properties provided by the Microsoft.VisualBasic.FileIO.FileSystem class for file I/O.

See How to: Copy Directories, How to: Create a Directory Listing, and How to: Enumerate Directories and Files.

Streams

The abstract base class Stream supports reading and writing bytes. All classes that represent streams inherit from the Stream class. The Stream class and its derived classes provide a common view of data sources and repositories, and isolate the programmer from the specific details of the operating system and underlying devices.

Streams involve three fundamental operations:

- Reading - transferring data from a stream into a data structure, such as an array of bytes.
- Writing - transferring data to a stream from a data source.
- Seeking - querying and modifying the current position within a stream.

Depending on the underlying data source or repository, a stream might support only some of these capabilities. For example, the PipeStream class does not support seeking. The CanRead, CanWrite, and CanSeek properties of a stream specify the operations that the stream supports.

Here are some commonly used stream classes:

- FileStream – for reading and writing to a file.
- IsolatedStorageFileStream – for reading and writing to a file in isolated storage.
- MemoryStream – for reading and writing to memory as the backing store.

- BufferedStream – for improving performance of read and write operations.
- NetworkStream – for reading and writing over network sockets.
- PipeStream – for reading and writing over anonymous and named pipes.
- CryptoStream – for linking data streams to cryptographic transformations.

For an example of working with streams asynchronously, see Asynchronous File I/O.

Readers and writers

The System.IO namespace also provides types for reading encoded characters from streams and writing them to streams. Typically, streams are designed for byte input and output. The reader and writer types handle the conversion of the encoded characters to and from bytes so the stream can complete the operation. Each reader and writer class is associated with a stream, which can be retrieved through the class's `BaseStream` property.

Here are some commonly used reader and writer classes:

- BinaryReader and BinaryWriter – for reading and writing primitive data types as binary values.
- StreamReader and StreamWriter – for reading and writing characters by using an encoding value to convert the characters to and from bytes.
- StringReader and StringWriter – for reading and writing characters to and from strings.
- TextReader and TextWriter – serve as the abstract base classes for other readers and writers that read and write characters and strings, but not binary data.

See How to: Read Text from a File, How to: Write Text to a File, How to: Read Characters from a String, and How to: Write Characters to a String.

Asynchronous I/O operations

Reading or writing a large amount of data can be resource-intensive. You should perform these tasks asynchronously if your application needs to remain responsive to

the user. With synchronous I/O operations, the UI thread is blocked until the resource-intensive operation has completed. Use asynchronous I/O operations when developing Windows 8.x Store apps to prevent creating the impression that your app has stopped working.

The asynchronous members contain `Async` in their names, such as the CopyToAsync, FlushAsync, ReadAsync, and WriteAsync methods. You use these methods with the `async` and `await` keywords.

For more information, see Asynchronous File I/O.

Compression

Compression refers to the process of reducing the size of a file for storage. Decompression is the process of extracting the contents of a compressed file so they are in a usable format. The System.IO.Compression namespace contains types for compressing and decompressing files and streams.

The following classes are frequently used when compressing and decompressing files and streams:

- ZipArchive – for creating and retrieving entries in the zip archive.
- ZipArchiveEntry – for representing a compressed file.
- ZipFile – for creating, extracting, and opening a compressed package.
- ZipFileExtensions – for creating and extracting entries in a compressed package.
- DeflateStream – for compressing and decompressing streams using the Deflate algorithm.
- GZipStream – for compressing and decompressing streams in gzip data format.

See How to: Compress and Extract Files.

Isolated storage

Isolated storage is a data storage mechanism that provides isolation and safety by defining standardized ways of associating code with saved data. The storage provides a

virtual file system that is isolated by user, assembly, and (optionally) domain. Isolated storage is particularly useful when your application does not have permission to access user files. You can save settings or files for your application in a manner that is controlled by the computer's security policy.

Isolated storage is not available for Windows 8.x Store apps; instead, use application data classes in the Windows.Storage namespace. For more information, see Application data.

The following classes are frequently used when implementing isolated storage:

- IsolatedStorage – provides the base class for isolated storage implementations.
- IsolatedStorageFile – provides an isolated storage area that contains files and directories.
- IsolatedStorageFileStream - exposes a file within isolated storage.

See Isolated Storage.

I/O operations in Windows Store apps

.NET for Windows 8.x Store apps contains many of the types for reading from and writing to streams; however, this set does not include all the .NET I/O types.

Some important differences to note when using I/O operations in Windows 8.x Store apps:

- Types specifically related to file operations, such as File, FileInfo, Directory and DirectoryInfo, are not included in the .NET for Windows 8.x Store apps. Instead, use the types in the Windows.Storage namespace of the Windows Runtime, such as StorageFile and StorageFolder.
- Isolated storage is not available; instead, use application data.
- Use asynchronous methods, such as ReadAsync and WriteAsync, to prevent blocking the UI thread.
- The path-based compression types ZipFile and ZipFileExtensions are not available. Instead, use the types in the Windows.Storage.Compression namespace.

You can convert between .NET Framework streams and Windows Runtime streams, if necessary. For more information, see How to: Convert Between .NET Framework Streams and Windows Runtime Streams or WindowsRuntimeStreamExtensions.

For more information about I/O operations in a Windows 8.x Store app, see Quickstart: Reading and writing files.

I/O and security

When you use the classes in the System.IO namespace, you must follow operating system security requirements such as access control lists (ACLs) to control access to files and directories. This requirement is in addition to any FileIOPermission requirements. You can manage ACLs programmatically. For more information, see How to: Add or Remove Access Control List Entries.

Default security policies prevent Internet or intranet applications from accessing files on the user's computer. Therefore, do not use the I/O classes that require a path to a physical file when writing code that will be downloaded over the internet or intranet. Instead, use isolated storage for .NET applications.

A security check is performed only when the stream is constructed. Therefore, do not open a stream and then pass it to less-trusted code or application domains.

Related topics

- Common I/O Tasks
  Provides a list of I/O tasks associated with files, directories, and streams, and links to relevant content and examples for each task.
- Asynchronous File I/O
  Describes the performance advantages and basic operation of asynchronous I/O.
- Isolated Storage
  Describes a data storage mechanism that provides isolation and safety by defining standardized ways of associating code with saved data.
- Pipes
  Describes anonymous and named pipe operations in .NET.

- Memory-Mapped Files

  Describes memory-mapped files, which contain the contents of files on disk in virtual memory. You can use memory-mapped files to edit very large files and to create shared memory for interprocess communication.