# Chapter 15. Streams and I/O

This chapter describes the fundamental types for input and output in .NET, with emphasis on the following topics:

- The .NET stream architecture and how it provides a consistent programming interface for reading and writing across a variety of I/O types
- Classes for manipulating files and directories on disk
- Specialized streams for compression, named pipes, and memory-mapped files

This chapter concentrates on the types in the `System.IO` namespace, the home of lower-level I/O functionality.

## Stream Architecture

The .NET stream architecture centers on three concepts: backing stores, decorators, and adapters, as shown in Figure 15-1.

A *backing store* is the endpoint that makes input and output useful, such as a file or network connection. Precisely, it is either or both of the following:

- A source from which bytes can be sequentially read
- A destination to which bytes can be sequentially written

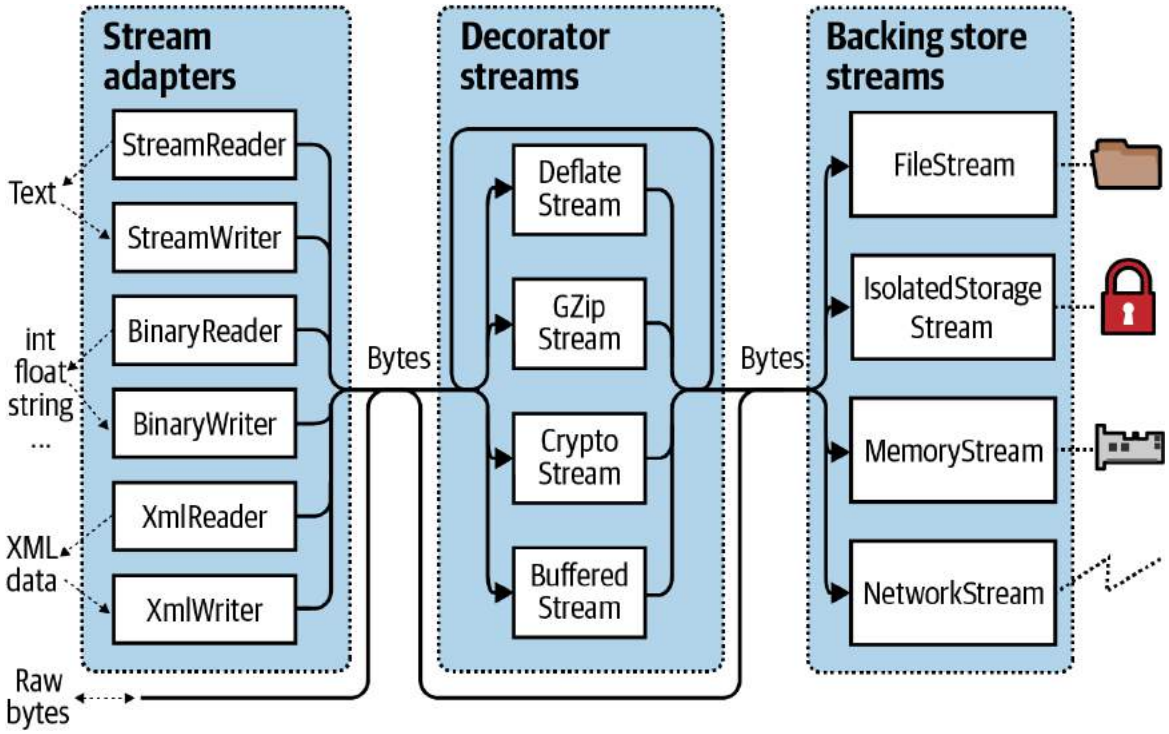


*Figure 15-1. Stream architecture*

A backing store is of no use, though, unless exposed to the programmer. A `Stream` is the standard .NET class for this purpose; it exposes a standard set of methods for reading, writing, and positioning. Unlike an array, for which all the backing data exists in memory at once, a stream deals with data serially—either one byte at a time or in blocks of a manageable size. Hence, a stream can use a small, fixed amount of memory regardless of the size of its backing store.

Streams fall into two categories:

*Backing store streams*

> These are hardwired to a particular type of backing store, such as `FileStream` or `NetworkStream`.

*Decorator streams*

> These feed off another stream, transforming the data in some way, such as `DeflateStream` or `CryptoStream`.

Decorator streams have the following architectural benefits:

- They liberate backing store streams from needing to implement such features as compression and encryption themselves.
- Streams don't suffer a change of interface when decorated.
- You connect decorators at runtime.
- You can chain decorators together (e.g., a compressor followed by an encryptor).

Both backing store and decorator streams deal exclusively in bytes. Although this is flexible and efficient, applications often work at higher levels such as text or XML. *Adapters* bridge this gap by wrapping a stream in a class with specialized methods typed to a particular format. For example, a text reader exposes a `ReadLine` method; an XML writer exposes a `WriteAttributes` method.

> **NOTE**
>
> An adapter wraps a stream, just like a decorator. Unlike a decorator, however, an adapter is not *itself* a stream; it typically hides the byte-oriented methods completely.

To summarize, backing store streams provide the raw data; decorator streams provide transparent binary transformations such as encryption; adapters offer typed methods for dealing in higher-level types such as strings and XML.

Figure 15-1 illustrates their associations. To compose a chain, you simply pass one object into another's constructor.

## Using Streams

The abstract `Stream` class is the base for all streams. It defines methods and properties for three fundamental operations: *reading*, *writing*, and *seeking*,

as well as for administrative tasks such as closing, flushing, and configuring timeouts (see Table 15-1).

*Table 15-1. Stream class members*

| Category | Members |
|---|---|
| Reading | `public abstract bool CanRead { get; }` |
| | `public abstract int Read (byte[] buffer, int offset, int count)` |
| | `public virtual int ReadByte();` |
| Writing | `public abstract bool CanWrite { get; }` |
| | `public abstract void Write (byte[] buffer, int offset, int count);` |
| | `public virtual void WriteByte (byte value);` |
| Seeking | `public abstract bool CanSeek { get; }` |
| | `public abstract long Position { get; set; }` |
| | `public abstract void SetLength (long value);` |
| | `public abstract long Length { get; }` |
| | `public abstract long Seek (long offset, SeekOrigin origin);` |
| Closing/flushing | `public virtual void Close();` |
| | `public void Dispose();` |
| | `public abstract void Flush();` |
| Timeouts | `public virtual bool CanTimeout { get; }` |

| Category | Members |
| --- | --- |
| | `public virtual int ReadTimeout { get; set; }` |
| | `public virtual int WriteTimeout { get; set; }` |
| Other | `public static readonly Stream Null; // "Null" stream` |
| | `public static Stream Synchronized (Stream stream);` |

There are also asynchronous versions of the `Read` and `Write` methods, both of which return `Task`s and optionally accept a cancellation token, and overloads that work with `Span<T>` and `Memory<T>` types that we describe in Chapter 23.

In the following example, we use a file stream to read, write, and seek:

```
using System;
using System.IO;

// Create a file called test.txt in the current directory:
using (Stream s = new FileStream ("test.txt", FileMode.Create))
{
  Console.WriteLine (s.CanRead);       // True
  Console.WriteLine (s.CanWrite);      // True
  Console.WriteLine (s.CanSeek);       // True

  s.WriteByte (101);
  s.WriteByte (102);
  byte[] block = { 1, 2, 3, 4, 5 };
  s.Write (block, 0, block.Length);     // Write block of 5 bytes

  Console.WriteLine (s.Length);         // 7
  Console.WriteLine (s.Position);       // 7
  s.Position = 0;                       // Move back to the start

  Console.WriteLine (s.ReadByte());     // 101
  Console.WriteLine (s.ReadByte());     // 102

  // Read from the stream back into the block array:
  Console.WriteLine (s.Read (block, 0, block.Length));   // 5
```

```
  // Assuming the last Read returned 5, we'll be at
  // the end of the file, so Read will now return 0:
  Console.WriteLine (s.Read (block, 0, block.Length));   // 0
}
```

Reading or writing asynchronously is simply a question of calling `ReadAsync`/`WriteAsync` instead of `Read`/`Write`, and `await`ing the expression (we must also add the `async` keyword to the calling method, as we described in Chapter 14):

```
async static void AsyncDemo()
{
  using (Stream s = new FileStream ("test.txt", FileMode.Create))
  {
    byte[] block = { 1, 2, 3, 4, 5 };
    await s.WriteAsync (block, 0, block.Length);    // Write asychronously

    s.Position = 0;                       // Move back to the start

    // Read from the stream back into the block array:
    Console.WriteLine (await s.ReadAsync (block, 0, block.Length));   // 5
  }
}
```

The asynchronous methods make it easy to write responsive and scalable applications that work with potentially slow streams (particularly network streams), without tying up a thread.

> **NOTE**
>
> For the sake of brevity, we'll continue to use synchronous methods for most of the examples in this chapter; however, we recommend the asynchronous `Read`/`Write` operations as preferable in most scenarios involving network I/O.

## Reading and Writing

A stream can support reading, writing, or both. If `CanWrite` returns `false`, the stream is read-only; if `CanRead` returns `false`, the stream is write-only.

`Read` receives a block of data from the stream into an array. It returns the number of bytes received, which is always either less than or equal to the `count` argument. If it's less than `count`, it means that either the end of the stream has been reached or the stream is giving you the data in smaller chunks (as is often the case with network streams). In either case, the balance of bytes in the array will remain unwritten, their previous values preserved.

**WARNING**

With `Read`, you can be certain you've reached the end of the stream only when the method returns `0`. So, if you have a 1,000-byte stream, the following code might fail to read it all into memory:

```
// Assuming s is a stream:
byte[] data = new byte [1000];
s.Read (data, 0, data.Length);
```

The `Read` method could read anywhere from 1 to 1,000 bytes, leaving the balance of the stream unread.

Here's the correct way to read a 1,000-byte stream via the `Read` method:

```
byte[] data = new byte [1000];

// bytesRead will always end up at 1000, unless the stream is
// itself smaller in length:

int bytesRead = 0;
int chunkSize = 1;
while (bytesRead < data.Length && chunkSize > 0)
  bytesRead +=
    chunkSize = s.Read (data, bytesRead, data.Length - bytesRead);
```

To make this easier, from .NET 7, the `Stream` class includes helper methods called `ReadExactly` and `ReadAtLeast` (and async versions of each). The following reads exactly 1,000 bytes from the stream (throwing an exception if the stream ends before then):

```
byte[] data = new byte [1000];
s.ReadExactly (data);   // Reads exactly 1000 bytes
```

The last line is equivalent to:

```
s.ReadExactly (data, offset:0, count:1000);
```

> **NOTE**
>
> The `BinaryReader` type provides another solution:
>
> ```
> byte[] data = new BinaryReader (s).ReadBytes (1000);
> ```
>
> If the stream is less than 1,000 bytes long, the byte array returned reflects the actual stream size. If the stream is seekable, you can read its entire contents by replacing `1000` with `(int)s.Length`.
>
> We describe the `BinaryReader` type further in "Stream Adapters".

The `ReadByte` method is simpler: it reads just a single byte, returning −1 to indicate the end of the stream. `ReadByte` actually returns an `int` rather than a `byte` because the latter cannot return −1.

The `Write` and `WriteByte` methods send data to the stream. If they are unable to send the specified bytes, an exception is thrown.

> **WARNING**
>
> In the `Read` and `Write` methods, the `offset` argument refers to the index in the `buffer` array at which reading or writing begins, not the position within the stream.

## Seeking

A stream is seekable if `CanSeek` returns `true`. With a seekable stream (such as a file stream), you can query or modify its `Length` (by calling `SetLength`) and at any time change the `Position` at which you're reading

or writing. The `Position` property is relative to the beginning of the stream; the `Seek` method, however, allows you to move relative to the current position or the end of the stream.

> **NOTE**
>
> Changing the `Position` on a `FileStream` typically takes a few microseconds. If you're doing this millions of times in a loop, the `MemoryMappedFile` class might be a better choice than a `FileStream` (see "Memory-Mapped Files").

With a nonseekable stream (such as an encryption stream), the only way to determine its length is to read it completely through. Furthermore, if you need to reread a previous section, you must close the stream and start afresh with a new one.

## Closing and Flushing

Streams must be disposed after use to release underlying resources such as file and socket handles. A simple way to guarantee this is by instantiating streams within `using` blocks. In general, streams follow standard disposal semantics:

- `Dispose` and `Close` are identical in function.
- Disposing or closing a stream repeatedly causes no error.

Closing a decorator stream closes both the decorator and its backing store stream. With a chain of decorators, closing the outermost decorator (at the head of the chain) closes the whole lot.

Some streams internally buffer data to and from the backing store to lessen round-tripping and so improve performance (file streams are a good example of this). This means that data you write to a stream might not hit the backing store immediately; it can be delayed as the buffer fills up. The `Flush` method forces any internally buffered data to be written immediately.

`Flush` is called automatically when a stream is closed, so you never need to do the following:

```
s.Flush(); s.Close();
```

## Timeouts

A stream supports read and write timeouts if `CanTimeout` returns `true`. Network streams support timeouts; file and memory streams do not. For streams that support timeouts, the `ReadTimeout` and `WriteTimeout` properties determine the desired timeout in milliseconds, where `0` means no timeout. The `Read` and `Write` methods indicate that a timeout has occurred by throwing an exception.

The asynchronous `ReadAsync`/`WriteAsync` methods do not support timeouts; instead you can pass a cancellation token into these methods.

## Thread Safety

As a rule, streams are not thread-safe, meaning that two threads cannot concurrently read or write to the same stream without possible error. The `Stream` class offers a simple workaround via the static `Synchronized` method. This method accepts a stream of any type and returns a thread-safe wrapper. The wrapper works by obtaining an exclusive lock around each read, write, or seek, ensuring that only one thread can perform such an operation at a time. In practice, this allows multiple threads to simultaneously append data to the same stream—other kinds of activities (such as concurrent reading) require additional locking to ensure that each thread accesses the desired portion of the stream. We discuss thread safety fully in Chapter 21.

> **NOTE**
>
> From .NET 6, you can use the `RandomAccess` class for performant thread-safe file I/O operations. `RandomAccess` also lets you pass in multiple buffers to improve performance.

## Backing Store Streams

Figure 15-2 shows the key backing store streams provided by .NET. A "null stream" is also available via the `Stream`'s static `Null` field. Null streams can be useful when writing unit tests.

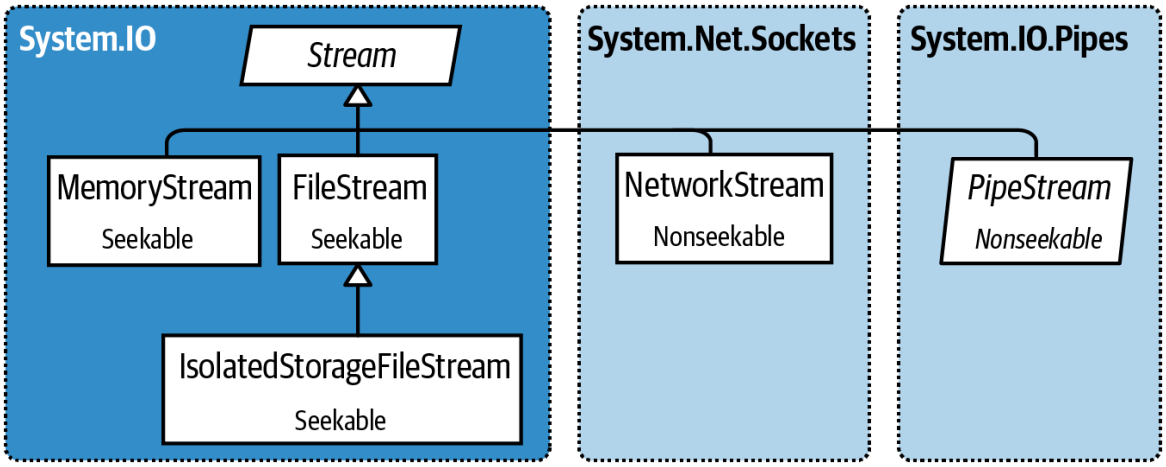


*Figure 15-2. Backing store streams*

In the following sections, we describe `FileStream` and `MemoryStream`; in the final section in this chapter, we describe `IsolatedStorageStream`. In Chapter 16, we cover `NetworkStream`.

## FileStream

Earlier in this section, we demonstrated the basic use of a `FileStream` to read and write bytes of data. Let's now examine the special features of this class.

> **NOTE**
>
> If you're still using Universal Windows Platform [UWP], you can also do file I/O with the types in `Windows.Storage`. We describe this in the online supplement at *http://www.albahari.com/nutshell*.

## Constructing a FileStream

The simplest way to instantiate a `FileStream` is to use one of the following static façade methods on the `File` class:

```
FileStream fs1 = File.OpenRead  ("readme.bin");           // Read-only
FileStream fs2 = File.OpenWrite ("writeme.tmp");          // Write-only
FileStream fs3 = File.Create    ("readwrite.tmp");        // Read/write
```

`OpenWrite` and `Create` differ in behavior if the file already exists. `Create` truncates any existing content; `OpenWrite` leaves existing content intact with the stream positioned at zero. If you write fewer bytes than were previously in the file, `OpenWrite` leaves you with a mixture of old and new content.

You can also directly instantiate a `FileStream`. Its constructors provide access to every feature, allowing you to specify a filename or low-level file handle, file creation and access modes, and options for sharing, buffering, and security. The following opens an existing file for read/write access without overwriting it (the `using` keyword ensures it is disposed when `fs` exits scope):

```
using var fs = new FileStream ("readwrite.tmp", FileMode.Open);
```

We look closer at `FileMode` shortly.

### SHORTCUT METHODS ON THE FILE CLASS

The following static methods read an entire file into memory in one step:

- `File.ReadAllText` (returns a string)
- `File.ReadAllLines` (returns an array of strings)
- `File.ReadAllBytes` (returns a byte array)

The following static methods write an entire file in one step:

- `File.WriteAllText`
- `File.WriteAllLines`
- `File.WriteAllBytes`
- `File.AppendAllText` (great for appending to a log file)

There's also a static method called `File.ReadLines`: this is like `ReadAllLines` except that it returns a lazily evaluated `IEnumerable<string>`. This is more efficient because it doesn't load the entire file into memory at once. LINQ is ideal for consuming the results: the following calculates the number of lines greater than 80 characters in length:

```
int longLines = File.ReadLines ("filePath")
                    .Count (l => l.Length > 80);
```

## Specifying a filename

A filename can be either absolute (e.g., *c:\temp\test.txt*—*or in Unix, /tmp/test.txt*) or relative to the current directory (e.g., *test.txt* or *temp\test.txt*). You can access or change the current directory via the static `Environment.CurrentDirectory` property.

> **WARNING**
>
> When a program starts, the current directory might or might not coincide with that of the program's executable. For this reason, you should never rely on the current directory for locating additional runtime files packaged along with your executable.

`AppDomain.CurrentDomain.BaseDirectory` returns the *application base directory*, which in normal cases is the folder containing the program's executable. To specify a filename relative to this directory, you can call `Path.Combine`:

```
string baseFolder = AppDomain.CurrentDomain.BaseDirectory;
string logoPath = Path.Combine (baseFolder, "logo.jpg");
Console.WriteLine (File.Exists (logoPath));
```

You can read and write across a Windows network via a Universal Naming Convention (UNC) path, such as *\\JoesPC\PicShare\pic.jpg* or *\\10.1.1.2\PicShare\pic.jpg*. (To access a Windows file share from macOS or Unix, mount it to your filesystem following instructions specific to your OS, and then open it using an ordinary path from C#).

### Specifying a FileMode

All of `FileStream`'s constructors that accept a filename also require a `FileMode` enum argument. Figure 15-3 shows how to choose a `FileMode`, and the choices yield results akin to calling a static method on the `File` class.

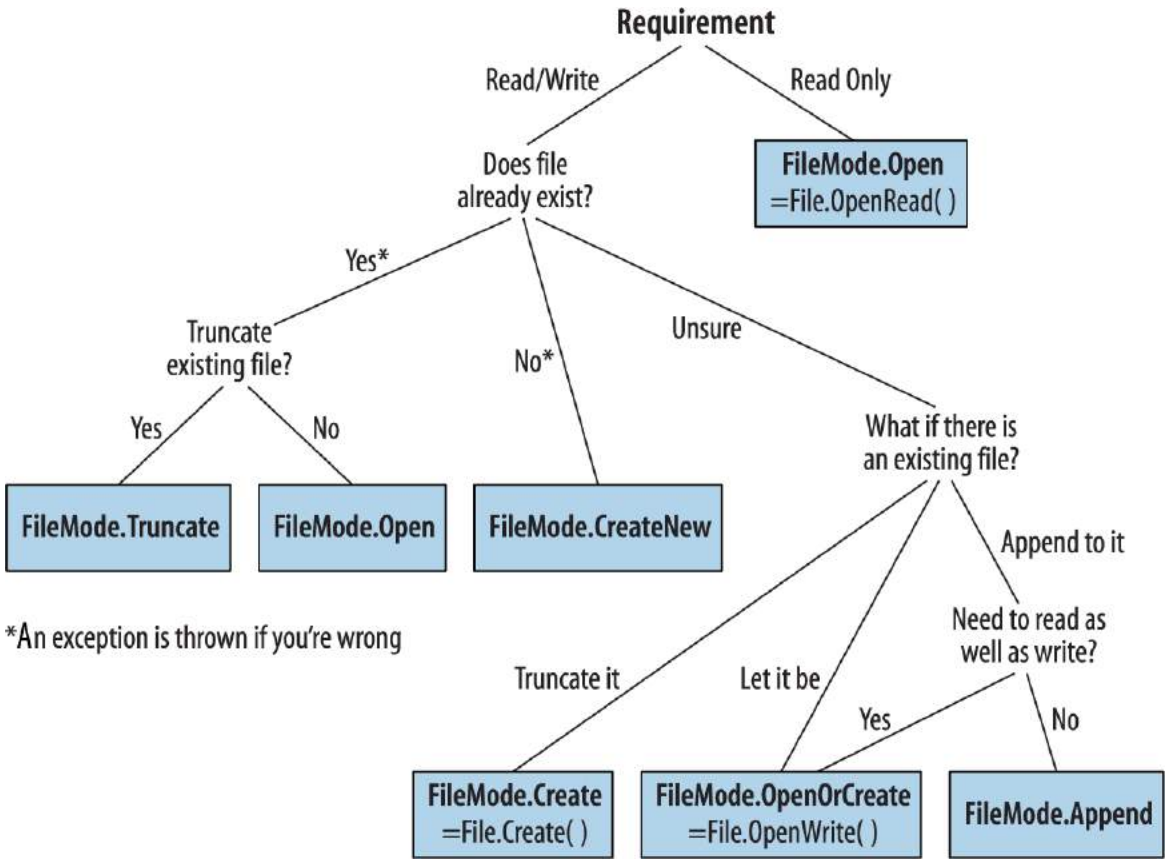


*Figure 15-3. Choosing a FileMode*

**WARNING**

`File.Create` and `FileMode.Create` will throw an exception if used on hidden files. To overwrite a hidden file, you must delete and re-create it:

```
File.Delete ("hidden.txt");
using var file = File.Create ("hidden.txt");
...
```

Constructing a `FileStream` with just a filename and `FileMode` gives you (with just one exception) a readable writable stream. You can request a downgrade if you also supply a `FileAccess` argument:

```
[Flags]
public enum FileAccess { Read = 1, Write = 2, ReadWrite = 3 }
```

The following returns a read-only stream, equivalent to calling `File.OpenRead`:

```
using var fs = new FileStream ("x.bin", FileMode.Open, FileAccess.Read);
...
```

`FileMode.Append` is the odd one out: with this mode, you get a *write-only* stream. To append with read-write support, you must instead use `FileMode.Open` or `FileMode.OpenOrCreate` and then seek the end of the stream:

```
using var fs = new FileStream ("myFile.bin", FileMode.Open);

fs.Seek (0, SeekOrigin.End);
...
```

### Advanced FileStream features

Here are other optional arguments you can include when constructing a `FileStream`:

- A `FileShare` enum describing how much access to grant other processes wanting to dip into the same file before you've finished (`None`, `Read` [default], `ReadWrite`, or `Write`).
- The size, in bytes, of the internal buffer (default is currently 4 KB).
- A flag indicating whether to defer to the operating system for asynchronous I/O.
- A `FileOptions` flags enum for requesting operating system encryption (`Encrypted`), automatic deletion upon closure for temporary files (`DeleteOnClose`), and optimization hints (`RandomAccess` and `SequentialScan`). There is also a `WriteThrough` flag that requests that the OS disable write-behind caching; this is for

transactional files or logs. Flags not supported by the underlying OS are silently ignored.

Opening a file with `FileShare.ReadWrite` allows other processes or users to simultaneously read and write to the same file. To avoid chaos, you can all agree to lock specified portions of the file before reading or writing, using these methods:

```
// Defined on the FileStream class:
public virtual void Lock   (long position, long length);
public virtual void Unlock (long position, long length);
```

`Lock` throws an exception if part or all of the requested file section has already been locked.

## MemoryStream

`MemoryStream` uses an array as a backing store. This partly defeats the purpose of having a stream because the entire backing store must reside in memory at once. `MemoryStream` is still useful when you need random access to a nonseekable stream. If you know the source stream will be of a manageable size, you can copy it into a `MemoryStream` as follows:

```
var ms = new MemoryStream();
sourceStream.CopyTo (ms);
```

You can convert a `MemoryStream` to a byte array by calling `ToArray`. The `GetBuffer` method does the same job more efficiently by returning a direct reference to the underlying storage array; unfortunately, this array is usually longer than the stream's real length.

> **NOTE**
>
> Closing and flushing a `MemoryStream` is optional. If you close a `MemoryStream`, you can no longer read or write to it, but you are still permitted to call `ToArray` to obtain the underlying data. `Flush` does absolutely nothing on a memory stream.

You can find further `MemoryStream` examples in "Compression Streams" and in "Overview".

## PipeStream

`PipeStream` provides a simple means by which one process can communicate with another through the operating system's *pipes* protocol. There are two kinds of pipe:

*Anonymous pipe (faster)*

> Allows one-way communication between a parent and child process on the same computer

*Named pipe (more flexible)*

> Allows two-way communication between arbitrary processes on the same computer or different computers across a network

A pipe is good for interprocess communication (IPC) on a single computer: it doesn't rely on a network transport, which means no network protocol overhead, and it has no issues with firewalls.

> **NOTE**
>
> Pipes are stream-based, so one process waits to receive a series of bytes while another process sends them. An alternative is for processes to communicate via a block of shared memory; we describe how to do this in "Memory-Mapped Files".

`PipeStream` is an abstract class with four concrete subtypes. Two are used for anonymous pipes and the other two for named pipes:

*Anonymous pipes*

`AnonymousPipeServerStream` and `AnonymousPipeClientStream`

*Named pipes*

`NamedPipeServerStream` and `NamedPipeClientStream`

Named pipes are simpler to use, so we describe them first.

### Named pipes

With named pipes, the parties communicate through a pipe of the same name. The protocol defines two distinct roles: the client and server. Communication happens between the client and server as follows:

- The server instantiates a `NamedPipeServerStream` and then calls `WaitForConnection`.
- The client instantiates a `NamedPipeClientStream` and then calls `Connect` (with an optional timeout).

The two parties then read and write the streams to communicate.

The following example demonstrates a server that sends a single byte (100) and then waits to receive a single byte:

```
using var s = new NamedPipeServerStream ("pipedream");
```

```
s.WaitForConnection();
s.WriteByte (100);                // Send the value 100.
Console.WriteLine (s.ReadByte());
```

Here's the corresponding client code:

```
using var s = new NamedPipeClientStream ("pipedream");

s.Connect();
Console.WriteLine (s.ReadByte());
s.WriteByte (200);                // Send the value 200 back.
```

Named pipe streams are bidirectional by default, so either party can read or write their stream. This means that the client and server must agree on some protocol to coordinate their actions, so both parties don't end up sending or receiving at once.

There also needs to be agreement on the length of each transmission. Our example was trivial in this regard, because we bounced just a single byte in each direction. To help with messages longer than one byte, pipes provide a *message* transmission mode (Windows only). If this is enabled, a party calling `Read` can know when a message is complete by checking the `IsMessageComplete` property. To demonstrate, we begin by writing a helper method that reads a whole message from a message-enabled `PipeStream`—in other words, reads until `IsMessageComplete` is true:

```
static byte[] ReadMessage (PipeStream s)
{
  MemoryStream ms = new MemoryStream();
  byte[] buffer = new byte [0x1000];      // Read in 4 KB blocks

  do    { ms.Write (buffer, 0, s.Read (buffer, 0, buffer.Length)); }
  while (!s.IsMessageComplete);

  return ms.ToArray();
}
```

(To make this asynchronous, replace "`s.Read`" with "`await s.ReadAsync`".)

> **WARNING**
>
> You cannot determine whether a `PipeStream` has finished reading a message simply by waiting for `Read` to return 0. This is because, unlike most other stream types, pipe streams and network streams have no definite end. Instead, they temporarily "dry up" between message transmissions.

Now we can activate message transmission mode. On the server, this is done by specifying `PipeTransmissionMode.Message` when constructing the stream:

```
using var s = new NamedPipeServerStream ("pipedream", PipeDirection.InOut,
                                         1, PipeTransmissionMode.Message);

s.WaitForConnection();

byte[] msg = Encoding.UTF8.GetBytes ("Hello");
s.Write (msg, 0, msg.Length);

Console.WriteLine (Encoding.UTF8.GetString (ReadMessage (s)));
```

On the client, we activate message transmission mode by setting `ReadMode` after calling `Connect`:

```
using var s = new NamedPipeClientStream ("pipedream");

s.Connect();
s.ReadMode = PipeTransmissionMode.Message;

Console.WriteLine (Encoding.UTF8.GetString (ReadMessage (s)));

byte[] msg = Encoding.UTF8.GetBytes ("Hello right back!");
s.Write (msg, 0, msg.Length);
```

> **NOTE**
>
> Message mode is supported only on Windows. Other platforms throw `PlatformNotSupportedException`.

## Anonymous pipes

An anonymous pipe provides a one-way communication stream between a parent and child process. Instead of using a system-wide name, anonymous pipes tune in through a private handle.

As with named pipes, there are distinct client and server roles. The system of communication is a little different, however, and proceeds as follows:

1. The server instantiates an `AnonymousPipeServerStream`, committing to a `PipeDirection` of `In` or `Out`.
2. The server calls `GetClientHandleAsString` to obtain an identifier for the pipe, which it then passes to the client (typically as an argument when starting the child process).
3. The child process instantiates an `AnonymousPipeClientStream`, specifying the opposite `PipeDirection`.
4. The server releases the local handle that was generated in Step 2, by calling `DisposeLocalCopyOfClientHandle`.
5. The parent and child processes communicate by reading/writing the stream.

Because anonymous pipes are unidirectional, a server must create two pipes for bidirectional communication. The following Console program creates two pipes (input and output) and then starts up a child process. It then sends a single byte to the child process, and receives a single byte in return:

```
class Program
{
  static void Main (string[] args)
  {
    if (args.Length == 0)
      // No arguments signals server mode
      AnonymousPipeServer();
    else
      // We pass in the pipe handle IDs as arguments to signal client mode
      AnonymousPipeClient (args [0], args [1]);
  }
```

```
  static void AnonymousPipeClient (string rxID, string txID)
  {
    using var rx = new AnonymousPipeClientStream (PipeDirection.In, rxID);
    using var tx = new AnonymousPipeClientStream (PipeDirection.Out, txID);

    Console.WriteLine ("Client received: " + rx.ReadByte ());
    tx.WriteByte (200);
  }

  static void AnonymousPipeServer ()
  {
    using var tx = new AnonymousPipeServerStream (
                     PipeDirection.Out, HandleInheritability.Inheritable);
    using var rx = new AnonymousPipeServerStream (
                     PipeDirection.In, HandleInheritability.Inheritable);

    string txID = tx.GetClientHandleAsString ();
    string rxID = rx.GetClientHandleAsString ();

    // Create and start up a child process.
    // We'll use the same Console executable, but pass in arguments:
    string thisAssembly = Assembly.GetEntryAssembly().Location;
    string thisExe = Path.ChangeExtension (thisAssembly, ".exe");
    var args = $"{txID} {rxID}";
    var startInfo = new ProcessStartInfo (thisExe, args);

    startInfo.UseShellExecute = false;       // Required for child process
    Process p = Process.Start (startInfo);

    tx.DisposeLocalCopyOfClientHandle ();    // Release unmanaged
    rx.DisposeLocalCopyOfClientHandle ();    // handle resources.

    tx.WriteByte (100);    // Send a byte to the child process

    Console.WriteLine ("Server received: " + rx.ReadByte ());

    p.WaitForExit ();
  }
}
```

As with named pipes, the client and server must coordinate their sending and receiving and agree on the length of each transmission. Anonymous pipes don’t, unfortunately, support message mode, so you must implement your own protocol for message length agreement. One solution is to send,

in the first four bytes of each transmission, an integer value defining the length of the message to follow. The `BitConverter` class provides methods for converting between an integer and an array of four bytes.

## BufferedStream

`BufferedStream` decorates, or wraps, another stream with buffering capability, and it is one of a number of decorator stream types in .NET, all of which are illustrated in Figure 15-4.

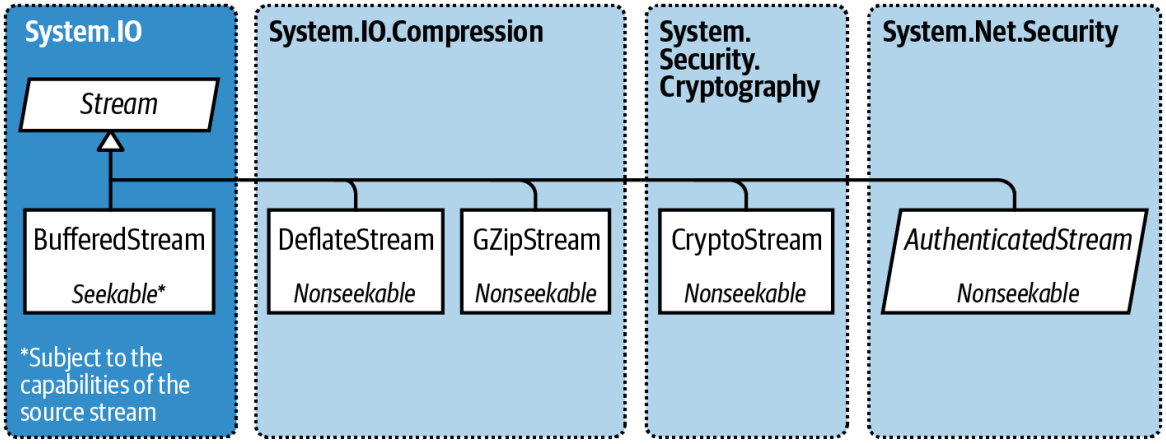


*Figure 15-4. Decorator streams*

Buffering improves performance by reducing round trips to the backing store. Here's how we wrap a `FileStream` in a 20 KB `BufferedStream`:

```
// Write 100K to a file:
File.WriteAllBytes ("myFile.bin", new byte [100000]);

using FileStream fs = File.OpenRead ("myFile.bin");
using BufferedStream bs = new BufferedStream (fs, 20000);  //20K buffer

bs.ReadByte();
Console.WriteLine (fs.Position);         // 20000
```

In this example, the underlying stream advances 20,000 bytes after reading just one byte, thanks to the read-ahead buffering. We could call `ReadByte` another 19,999 times before the `FileStream` would be hit again.

Coupling a `BufferedStream` to a `FileStream`, as in this example, is of limited value because `FileStream` already has built-in buffering. Its only use might be in enlarging the buffer on an already constructed `FileStream`.

Closing a `BufferedStream` automatically closes the underlying backing store stream.

## Stream Adapters

A `Stream` deals only in bytes; to read or write data types such as strings, integers, or XML elements, you must plug in an adapter. Here's what .NET provides:

*Text adapters (for string and character data)*

`TextReader`, `TextWriter`

`StreamReader`, `StreamWriter`

`StringReader`, `StringWriter`

*Binary adapters (for primitive types such as* `int`*,* `bool`*,* `string`*, and* `float`*)*

`BinaryReader`, `BinaryWriter`

*XML adapters (covered in Chapter 11)*

`XmlReader`, `XmlWriter`

Figure 15-5 illustrates the relationships between these types.

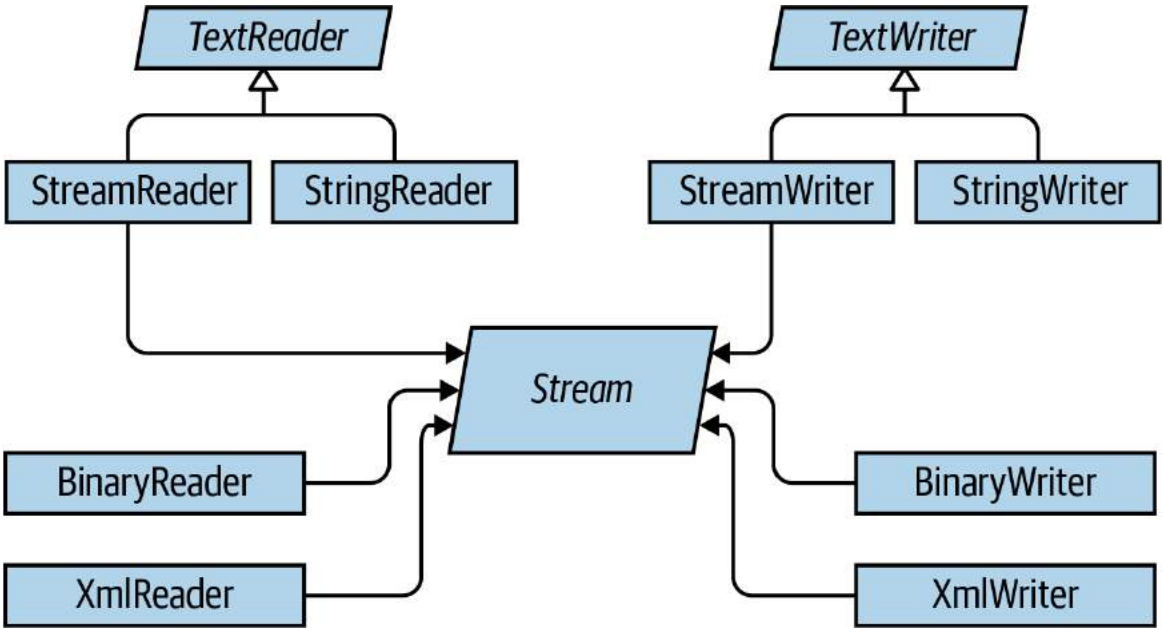


*Figure 15-5. Readers and writers*

## Text Adapters

`TextReader` and `TextWriter` are the abstract base classes for adapters that deal exclusively with characters and strings. Each has two general-purpose implementations in .NET:

`StreamReader`/`StreamWriter`

Uses a `Stream` for its raw data store, translating the stream's bytes into characters or strings

`StringReader`/`StringWriter`

Implements `TextReader`/`TextWriter` using in-memory strings

Table 15-2 lists `TextReader`'s members by category. `Peek` returns the next character in the stream without advancing the position. Both `Peek` and the zero-argument version of `Read` return −1 if at the end of the stream; otherwise, they return an integer that can be cast directly to a `char`. The overload of `Read` that accepts a `char[]` buffer is identical in functionality to

the `ReadBlock` method. `ReadLine` reads until reaching either a CR (character 13) or LF (character 10), or a CR+LF pair in sequence. It then returns a string, discarding the CR/LF characters.

*Table 15-2. TextReader members*

| Category | Members |
|---|---|
| Reading one char | `public virtual int Peek(); // Cast the result to a char` |
| | `public virtual int Read(); // Cast the result to a char` |
| Reading many chars | `public virtual int Read (char[] buffer, int index, int count);` |
| | `public virtual int ReadBlock (char[] buffer, int index, int count);` |
| | `public virtual string ReadLine();` |
| | `public virtual string ReadToEnd();` |
| Closing | `public virtual void Close();` |
| | `public void Dispose(); // Same as Close` |
| Other | `public static readonly TextReader Null;` |
| | `public static TextReader Synchronized (TextReader reader);` |

> **NOTE**
>
> `Environment.NewLine` returns the new-line sequence for the current OS.
>
> On Windows, this is `"\r\n"` (think “ReturN”) and is loosely modeled on a mechanical typewriter: a CR (character 13) followed by an LF (character 10). Reverse the order and you’ll get either two new lines or none!
>
> On Unix and macOS, it’s simply `"\n"`.

`TextWriter` has analogous methods for writing, as shown in Table 15-3. The `Write` and `WriteLine` methods are additionally overloaded to accept every primitive type, plus the `object` type. These methods simply call the `ToString` method on whatever is passed in (optionally through an `IFormatProvider` specified either when calling the method or when constructing the `TextWriter`).

*Table 15-3. TextWriter members*

| Category | Members |
|---|---|
| Writing one char | `public virtual void Write (char value);` |
| Writing many chars | `public virtual void Write (string value);` |
| | `public virtual void Write (char[] buffer, int index, int count);` |
| | `public virtual void Write (string format, params object[] arg);` |
| | `public virtual void WriteLine (string value);` |
| Closing and flushing | `public virtual void Close();` |
| | `public void Dispose(); // Same as Close` |
| | `public virtual void Flush();` |
| Formatting and encoding | `public virtual IFormatProvider FormatProvider { get; }` |
| | `public virtual string NewLine { get; set; }` |
| | `public abstract Encoding Encoding { get; }` |
| Other | `public static readonly TextWriter Null;` |
| | `public static TextWriter Synchronized (TextWriter writer);` |

`WriteLine` simply appends the given text with `Environment.NewLine`. You can change this via the `NewLine` property (this can be useful for

interoperability with Unix file formats).

> **NOTE**
>
> As with `Stream`, `TextReader` and `TextWriter` offer task-based asynchronous versions of their read/write methods.

### StreamReader and StreamWriter

In the following example, a `StreamWriter` writes two lines of text to a file, and then a `StreamReader` reads the file back:

```
using (FileStream fs = File.Create ("test.txt"))
using (TextWriter writer = new StreamWriter (fs))
{
  writer.WriteLine ("Line1");
  writer.WriteLine ("Line2");
}

using (FileStream fs = File.OpenRead ("test.txt"))
using (TextReader reader = new StreamReader (fs))
{
  Console.WriteLine (reader.ReadLine());       // Line1
  Console.WriteLine (reader.ReadLine());       // Line2
}
```

Because text adapters are so often coupled with files, the `File` class provides the static methods `CreateText`, `AppendText`, and `OpenText` to shortcut the process:

```
using (TextWriter writer = File.CreateText ("test.txt"))
{
  writer.WriteLine ("Line1");
  writer.WriteLine ("Line2");
}


using (TextWriter writer = File.AppendText ("test.txt"))
  writer.WriteLine ("Line3");
```

```
using (TextReader reader = File.OpenText ("test.txt"))
  while (reader.Peek() > -1)
    Console.WriteLine (reader.ReadLine());     // Line1
                                               // Line2
                                               // Line3
```

This also illustrates how to test for the end of a file (viz. `reader.Peek()`). Another option is to read until `reader.ReadLine` returns null.

You can also read and write other types such as integers, but because `TextWriter` invokes `ToString` on your type, you must parse a string when reading it back:

```
using (TextWriter w = File.CreateText ("data.txt"))
{
  w.WriteLine (123);          // Writes "123"
  w.WriteLine (true);         // Writes the word "true"
}

using (TextReader r = File.OpenText ("data.txt"))
{
  int myInt = int.Parse (r.ReadLine());     // myInt == 123
  bool yes = bool.Parse (r.ReadLine());     // yes == true
}
```

## Character encodings

`TextReader` and `TextWriter` are by themselves just abstract classes with no connection to a stream or backing store. The `StreamReader` and `StreamWriter` types, however, are connected to an underlying byte-oriented stream, so they must convert between characters and bytes. They do so through an `Encoding` class from the `System.Text` namespace, which you choose when constructing the `StreamReader` or `StreamWriter`. If you choose none, the default UTF-8 encoding is used.

### WARNING

If you explicitly specify an encoding, `StreamWriter` will, by default, write a prefix to the start of the stream to identity the encoding. This is usually undesirable, and you can prevent it by constructing the encoding as follows:

```
var encoding = new UTF8Encoding (
  encoderShouldEmitUTF8Identifier:false,
  throwOnInvalidBytes:true);
```

The second argument tells the `StreamWriter` (or `StreamReader`) to throw an exception if it encounters bytes that do not have a valid string translation for their encoding, which matches its default behavior if you do not specify an encoding.

The simplest of the encodings is ASCII because each character is represented by one byte. The ASCII encoding maps the first 127 characters of the Unicode set into its single byte, covering what you see on a US-style keyboard. Most other characters, including specialized symbols and non-English characters, cannot be represented and are converted to the □ character. The default UTF-8 encoding can map all allocated Unicode characters, but it is more complex. The first 127 characters encode to a single byte, for ASCII compatibility; the remaining characters encode to a variable number of bytes (most commonly two or three). Consider the following:

```
using (TextWriter w = File.CreateText ("but.txt"))    // Use default UTF-8
  w.WriteLine ("but-");                               // encoding.

using (Stream s = File.OpenRead ("but.txt"))
  for (int b; (b = s.ReadByte()) > -1;)
    Console.WriteLine (b);
```

The word “but” is followed not by a stock-standard hyphen but by the longer em dash (—) character, U+2014. This is the one that won’t get you into trouble with your book editor! Let’s examine the output:

```
98     // b
117    // u
116    // t
226    // em dash byte 1       Note that the byte values
128    // em dash byte 2       are >= 128 for each part
148    // em dash byte 3       of the multibyte sequence.
13     // <CR>
10     // <LF>
```

Because the em dash is outside the first 127 characters of the Unicode set, it requires more than a single byte to encode in UTF-8 (in this case, three). UTF-8 is efficient with the Western alphabet as most popular characters consume just one byte. It also downgrades easily to ASCII simply by ignoring all bytes above 127. Its disadvantage is that seeking within a stream is troublesome because a character's position does not correspond to its byte position in the stream. An alternative is UTF-16 (labeled just "Unicode" in the `Encoding` class). Here's how we write the same string with UTF-16:

```
using (Stream s = File.Create ("but.txt"))
using (TextWriter w = new StreamWriter (s, Encoding.Unicode))
  w.WriteLine ("but-");

foreach (byte b in File.ReadAllBytes ("but.txt"))
  Console.WriteLine (b);
```

And here's the output:

```
255    // Byte-order mark 1
254    // Byte-order mark 2
98     // 'b' byte 1
0      // 'b' byte 2
117    // 'u' byte 1
0      // 'u' byte 2
116    // 't' byte 1
0      // 't' byte 2
20     // '--' byte 1
32     // '--' byte 2
13     // <CR> byte 1
0      // <CR> byte 2
10     // <LF> byte 1
0      // <LF> byte 2
```

Technically, UTF-16 uses either two or four bytes per character (there are close to a million Unicode characters allocated or reserved, so two bytes is not always enough). However, because the C# `char` type is itself only 16 bits wide, a UTF-16 encoding will always use exactly two bytes per .NET `char`. This makes it easy to jump to a particular character index within a stream.

UTF-16 uses a two-byte prefix to identify whether the byte pairs are written in a "little-endian" or "big-endian" order (the least significant byte first or the most significant byte first). The default little-endian order is standard for Windows-based systems.

### StringReader and StringWriter

The `StringReader` and `StringWriter` adapters don't wrap a stream at all; instead, they use a string or `StringBuilder` as the underlying data source. This means no byte translation is required—in fact, the classes do nothing you couldn't easily achieve with a string or `StringBuilder` coupled with an index variable. Their advantage, though, is that they share a base class with `StreamReader`/`StreamWriter`. For instance, suppose that we have a string containing XML and want to parse it with an `XmlReader`. The `XmlReader.Create` method accepts one of the following:

- A `URI`
- A `Stream`
- A `TextReader`

So, how do we XML-parse our string? Because `StringReader` is a subclass of `TextReader`, we're in luck. We can instantiate and pass in a `StringReader` as follows:

```
XmlReader r = XmlReader.Create (new StringReader (myString));
```

## Binary Adapters

`BinaryReader` and `BinaryWriter` read and write native data types: `bool`, `byte`, `char`, `decimal`, `float`, `double`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, and `ulong`, as well as `string`s and arrays of the primitive data types.

Unlike `StreamReader` and `StreamWriter`, binary adapters store primitive data types efficiently because they are represented in memory. So, an `int` uses four bytes; a `double` uses eight bytes. Strings are written through a text encoding (as with `StreamReader` and `StreamWriter`) but are length-prefixed in order to make it possible to read back a series of strings without needing special delimiters.

Imagine that we have a simple type, defined as follows:

```
public class Person
{
  public string Name;
  public int    Age;
  public double Height;
}
```

We can add the following methods to `Person` to save/load its data to/from a stream using binary adapters:

```
public void SaveData (Stream s)
{
  var w = new BinaryWriter (s);
  w.Write (Name);
  w.Write (Age);
  w.Write (Height);
  w.Flush();         // Ensure the BinaryWriter buffer is cleared.
                     // We won't dispose/close it, so more data
}                    // can be written to the stream.

public void LoadData (Stream s)
{
  var r = new BinaryReader (s);
  Name   = r.ReadString();
  Age    = r.ReadInt32();
  Height = r.ReadDouble();
}
```

`BinaryReader` can also read into byte arrays. The following reads the entire contents of a seekable stream:

```
byte[] data = new BinaryReader (s).ReadBytes ((int) s.Length);
```

This is more convenient than reading directly from a stream because it doesn't require a loop to ensure that all data has been read.

## Closing and Disposing Stream Adapters

You have four choices in tearing down stream adapters:

1. Close the adapter only
2. Close the adapter and then close the stream
3. (For writers) Flush the adapter and then close the stream
4. (For readers) Close just the stream

> **NOTE**
>
> `Close` and `Dispose` are synonymous with adapters, just as they are with streams.

Options 1 and 2 are semantically identical because closing an adapter automatically closes the underlying stream. Whenever you nest `using` statements, you're implicitly taking option 2:

```
using (FileStream fs = File.Create ("test.txt"))
using (TextWriter writer = new StreamWriter (fs))
  writer.WriteLine ("Line");
```

Because the nest disposes from the inside out, the adapter is first closed, and then the stream. Furthermore, if an exception is thrown within the adapter's constructor, the stream still closes. It's hard to go wrong with nested `using` statements!

> **WARNING**
>
> Never close a stream before closing or flushing its writer—you'll amputate any data that's buffered in the adapter.

Options 3 and 4 work because adapters are in the unusual category of *optionally* disposable objects. An example of when you might choose not to dispose an adapter is when you've finished with the adapter but you want to leave the underlying stream open for subsequent use:

```
using (FileStream fs = new FileStream ("test.txt", FileMode.Create))
{
  StreamWriter writer = new StreamWriter (fs);
  writer.WriteLine ("Hello");
  writer.Flush();

  fs.Position = 0;
  Console.WriteLine (fs.ReadByte());
}
```

Here, we write to a file, reposition the stream, and then read the first byte before closing the stream. If we disposed the `StreamWriter`, it would also close the underlying `FileStream`, causing the subsequent read to fail. The proviso is that we call `Flush` to ensure that the `StreamWriter`'s buffer is written to the underlying stream.

> **NOTE**
>
> Stream adapters—with their optional disposal semantics—do not implement the extended disposal pattern where the finalizer calls `Dispose`. This allows an abandoned adapter to evade automatic disposal when the garbage collector catches up with it.

There's also a constructor on `StreamReader`/`StreamWriter` that instructs it to keep the stream open after disposal. Consequently, we can rewrite the preceding example as follows:

```
using (var fs = new FileStream ("test.txt", FileMode.Create))
{
  using (var writer = new StreamWriter (fs, new UTF8Encoding (false, true),
                                         0x400, true))
    writer.WriteLine ("Hello");

  fs.Position = 0;
  Console.WriteLine (fs.ReadByte());
  Console.WriteLine (fs.Length);
}
```

## Compression Streams

Two general-purpose compression streams are provided in the `System.IO.Compression` namespace: `DeflateStream` and `GZipStream`. Both use a popular compression algorithm similar to that of the ZIP format. They differ in that `GZipStream` writes an additional protocol at the start and end—including a CRC to detect errors. `GZipStream` also conforms to a standard recognized by other software.

.NET also includes `BrotliStream`, which implements the *Brotli* compression algorithm. `BrotliStream` is more than 10 times slower than `DeflateStream` and `GZipStream` but achieves a better compression ratio. (The performance hit applies only to compression—decompression performs very well.)

All three streams allow reading and writing, with the following provisos:

- You always *write* to the stream when compressing.
- You always *read* from the stream when decompressing.

`DeflateStream`, `GZipStream`, and `BrotliStream` are decorators; they compress or decompress data from another stream that you supply in construction. In the following example, we compress and decompress a series of bytes using a `FileStream` as the backing store:

```
using (Stream s = File.Create ("compressed.bin"))
using (Stream ds = new DeflateStream (s, CompressionMode.Compress))
```

```
  for (byte i = 0; i < 100; i++)
    ds.WriteByte (i);

using (Stream s = File.OpenRead ("compressed.bin"))
using (Stream ds = new DeflateStream (s, CompressionMode.Decompress))
  for (byte i = 0; i < 100; i++)
    Console.WriteLine (ds.ReadByte());     // Writes 0 to 99
```

With `DeflateStream`, the compressed file is 102 bytes: slightly larger than the original (`BrotliStream` would compress it to 73 bytes). Compression works poorly with "dense," nonrepetitive binary data (and worst of all with encrypted data, which lacks regularity by design). It works well with most text files; in the next example, we compress and decompress a text stream composed of 1,000 words chosen randomly from a small sentence with the *Brotli* algorithm. This also demonstrates chaining a backing store stream, a decorator stream, an adapter (as depicted at the start of the chapter in Figure 15-1), and the use of asynchronous methods:

```
string[] words = "The quick brown fox jumps over the lazy dog".Split();
Random rand = new Random (0);   // Give it a seed for consistency

using (Stream s = File.Create ("compressed.bin"))
using (Stream ds = new BrotliStream (s, CompressionMode.Compress))
using (TextWriter w = new StreamWriter (ds))
  for (int i = 0; i < 1000; i++)
    await w.WriteAsync (words [rand.Next (words.Length)] + " ");

Console.WriteLine (new FileInfo ("compressed.bin").Length);      // 808

using (Stream s = File.OpenRead ("compressed.bin"))
using (Stream ds = new BrotliStream (s, CompressionMode.Decompress))
using (TextReader r = new StreamReader (ds))
  Console.Write (await r.ReadToEndAsync());  // Output below:

lazy lazy the fox the quick The brown fox jumps over fox over fox The
brown brown brown over brown quick fox brown dog dog lazy fox dog brown
over fox jumps lazy lazy quick The jumps fox jumps The over jumps dog...
```

In this case, `BrotliStream` compresses efficiently to 808 bytes—less than one byte per word. (For comparison, `DeflateStream` compresses the same data to 885 bytes.)

## Compressing in Memory

Sometimes, you need to compress entirely in memory. Here's how to use a `MemoryStream` for this purpose:

```
byte[] data = new byte[1000];          // We can expect a good compression
                                       // ratio from an empty array!
var ms = new MemoryStream();
using (Stream ds = new DeflateStream (ms, CompressionMode.Compress))
  ds.Write (data, 0, data.Length);

byte[] compressed = ms.ToArray();
Console.WriteLine (compressed.Length);       // 11

// Decompress back to the data array:
ms = new MemoryStream (compressed);
using (Stream ds = new DeflateStream (ms, CompressionMode.Decompress))
  for (int i = 0; i < 1000; i += ds.Read (data, i, 1000 - i));
```

The `using` statement around the `DeflateStream` closes it in a textbook fashion, flushing any unwritten buffers in the process. This also closes the `MemoryStream` it wraps—meaning we must then call `ToArray` to extract its data.

Here's an alternative that avoids closing the `MemoryStream` and uses the asynchronous read and write methods:

```
byte[] data = new byte[1000];

MemoryStream ms = new MemoryStream();
using (Stream ds = new DeflateStream (ms, CompressionMode.Compress, true))
  await ds.WriteAsync (data, 0, data.Length);

Console.WriteLine (ms.Length);             // 113
ms.Position = 0;
using (Stream ds = new DeflateStream (ms, CompressionMode.Decompress))
  for (int i = 0; i < 1000; i += await ds.ReadAsync (data, i, 1000 - i));
```

The additional flag sent to `DeflateStream`'s constructor instructs it to not follow the usual protocol of taking the underlying stream with it in disposal.

In other words, the `MemoryStream` is left open, allowing us to position it back to zero and reread it.

## Unix gzip File Compression

`GZipStream`'s compression algorithm is popular on Unix systems as a file compression format. Each source file is compressed into a separate target file with a *.gz* extension.

The following methods do the work of the Unix command-line gzip and gunzip utilities:

```
async Task GZip (string sourcefile, bool deleteSource = true)
{
  var gzip = $"{sourcefile}.gz";
  if (File.Exists (gzip))
    throw new Exception ("Gzip file already exists");

  // Compress
  using (FileStream inStream = File.Open (sourcefile, FileMode.Open))
  using (FileStream outStream = new FileStream (gzip, FileMode.CreateNew))
  using (GZipStream gzipStream =
    new GZipStream (outStream, CompressionMode.Compress))
    await inStream.CopyToAsync (gzipStream);

  if (deleteSource) File.Delete(sourcefile);
}

async Task GUnzip (string gzipfile, bool deleteGzip = true)
{
  if (Path.GetExtension (gzipfile) != ".gz")
    throw new Exception ("Not a gzip file");

  var uncompressedFile = gzipfile.Substring (0, gzipfile.Length - 3);
  if (File.Exists (uncompressedFile))
    throw new Exception ("Destination file already exists");

  // Uncompress
  using (FileStream uncompressToStream =
         File.Open (uncompressedFile, FileMode.Create))
  using (FileStream zipfileStream = File.Open (gzipfile, FileMode.Open))
  using (var unzipStream =
         new GZipStream (zipfileStream, CompressionMode.Decompress))
    await unzipStream.CopyToAsync (uncompressToStream);
```

```
  if (deleteGzip) File.Delete (gzipfile);
}
```

The following compresses a file:

```
await GZip ("/tmp/myfile.txt");      // Creates /tmp/myfile.txt.gz
```

And the following decompresses it:

```
await GUnzip ("/tmp/myfile.txt.gz")  // Creates /tmp/myfile.txt
```

## Working with ZIP Files

The `ZipArchive` and `ZipFile` classes in `System.IO.Compression` support the ZIP compression format. The advantage of the ZIP format over `DeflateStream` and `GZipStream` is that it also acts as a container for multiple files and is compatible with ZIP files created with Windows Explorer.

`ZipArchive` works with streams, whereas `ZipFile` addresses the more common scenario of working with files. (`ZipFile` is a static helper class for `ZipArchive`.)

`ZipFile`'s `CreateFromDirectory` method adds all the files in a specified directory into a ZIP file:

```
ZipFile.CreateFromDirectory (@"d:\MyFolder", @"d:\archive.zip");
```

`ExtractToDirectory` does the opposite and extracts a ZIP file to a directory:

```
ZipFile.ExtractToDirectory (@"d:\archive.zip", @"d:\MyFolder");
```

(From .NET 8, you can also specify a `Stream` instead of a zip file path.)

When compressing, you can specify whether to optimize for file size or speed as well as whether to include the name of the source directory in the archive. Enabling the latter option in our example would create a subdirectory in the archive called *MyFolder* into which the compressed files would go.

`ZipFile` has an `Open` method for reading/writing individual entries. This returns a `ZipArchive` object (which you can also obtain by instantiating `ZipArchive` with a `Stream` object). When calling `Open`, you must specify a filename and indicate whether you want to `Read`, `Create`, or `Update` the archive. You can then enumerate existing entries via the `Entries` property or find a particular file by calling `GetEntry`:

```
using (ZipArchive zip = ZipFile.Open (@"d:\zz.zip", ZipArchiveMode.Read))

  foreach (ZipArchiveEntry entry in zip.Entries)
    Console.WriteLine (entry.FullName + " " + entry.Length);
```

`ZipArchiveEntry` also has a `Delete` method, an `ExtractToFile` method (this is actually an extension method in the `ZipFileExtensions` class), and an `Open` method that returns a readable/writable `Stream`. You can create new entries by calling `CreateEntry` (or the `CreateEntryFromFile` extension method) on the `ZipArchive`. The following creates the archive *d:\zz.zip*, to which it adds *foo.dll*, under a directory structure within the archive called *bin\X86*:

```
byte[] data = File.ReadAllBytes (@"d:\foo.dll");
using (ZipArchive zip = ZipFile.Open (@"d:\zz.zip", ZipArchiveMode.Update))
  zip.CreateEntry (@"bin\X64\foo.dll").Open().Write (data, 0, data.Length);
```

You could do the same thing entirely in memory by constructing `ZipArchive` with a `MemoryStream`.

## Working with Tar Files

The types in the `System.Formats.Tar` namespace (from .NET 7) support the *.tar* archive format, popular on Unix systems for bundling multiple files. To create a *.tar* file (a *tarball*), call `TarFile.CreateFromDirectory`:

```
TarFile.CreateFromDirectory ("/tmp/testfolder", "/tmp/test.tar", false);
```

(The third argument indicates whether to include the base directory name in the archive entries.)

To extract a tarball, call `TarFile.ExtractToDirectory`:

```
TarFile.ExtractToDirectory ("/tmp/test.tar", "/tmp/testfolder", true);
```

(The third argument indicates whether to overwrite existing files.)

Both of these methods let you specify a `Stream` instead of a *.tar* filepath. In the following example, we write the tarball to a memory stream, and then use `GZipStream` to compress that stream to a *.tar.gz* file:

```
var ms = new MemoryStream();
TarFile.CreateFromDirectory ("/tmp/testfolder", ms, false);
ms.Position = 0;    // So that we can re-use the stream for reading.
using (var fs = File.Create ("/tmp/test.tar.gz"))
using (var gz = new GZipStream (fs, CompressionMode.Compress))
  ms.CopyTo (gz);
```

(Compressing a *.tar* into a *.tar.gz* is useful because the *.tar* format does not itself incorporate compression, unlike the *.zip* format.) We can extract the *.tar.gz* file as follows:

```
using (var fs = File.OpenRead ("/tmp/test.tar.gz"))
using (var gz = new GZipStream (fs, CompressionMode.Decompress))
  TarFile.ExtractToDirectory (gz, "/tmp/testfolder", true);
```

You can also access the API at a more granular level with the `TarReader` and `TarWriter` classes. The following illustrates the use of `TarReader`:

```
using (FileStream archiveStream = File.OpenRead ("/tmp/test.tar "))
using (TarReader reader = new (archiveStream))
  while (true)
  {
    TarEntry entry = reader.GetNextEntry();
    if (entry == null) break;   // No more entries
    Console.WriteLine (
      $"Entry {entry.Name} is {entry.DataStream.Length} bytes long");
    entry.ExtractToFile (
      Path.Combine ("/tmp/testfolder", entry.Name), true);
  }
```

# File and Directory Operations

The `System.IO` namespace provides a set of types for performing "utility" file and directory operations, such as copying and moving, creating directories, and setting file attributes and permissions. For most features, you can choose between either of two classes, one offering static methods and the other instance methods:

*Static classes*

`File` and `Directory`

*Instance-method classes (constructed with a file or directory name)*

`FileInfo` and `DirectoryInfo`

Additionally, there's a static class called `Path`. This does nothing to files or directories; instead, it provides string manipulation methods for filenames and directory paths. `Path` also assists with temporary files.

## The File Class

`File` is a static class whose methods all accept a filename. The filename can be either relative to the current directory or fully qualified with a directory. Here are its methods (all `public` and `static`):

```
bool Exists (string path);      // Returns true if the file is present

void Delete  (string path);
void Copy    (string sourceFileName, string destFileName);
void Move    (string sourceFileName, string destFileName);
void Replace (string sourceFileName, string destinationFileName,
                                     string destinationBackupFileName);

FileAttributes GetAttributes (string path);
void SetAttributes           (string path, FileAttributes fileAttributes);

void Decrypt (string path);
void Encrypt (string path);

DateTime GetCreationTime   (string path);      // UTC versions are
DateTime GetLastAccessTime (string path);      // also provided.
DateTime GetLastWriteTime  (string path);

void SetCreationTime   (string path, DateTime creationTime);
void SetLastAccessTime (string path, DateTime lastAccessTime);
void SetLastWriteTime  (string path, DateTime lastWriteTime);

FileSecurity GetAccessControl (string path);
FileSecurity GetAccessControl (string path,
                               AccessControlSections includeSections);
void SetAccessControl (string path, FileSecurity fileSecurity);
```

`Move` throws an exception if the destination file already exists; `Replace` does not. Both methods allow the file to be renamed as well as moved to another directory.

`Delete` throws an `UnauthorizedAccessException` if the file is marked read-only; you can tell this in advance by calling `GetAttributes`. It also throws that exception if the OS denies delete permission for that file to your process. Here are all the members of the `FileAttribute` enum that `GetAttributes` returns:

```
Archive, Compressed, Device, Directory, Encrypted,
Hidden, IntegritySystem, Normal, NoScrubData, NotContentIndexed,
Offline, ReadOnly, ReparsePoint, SparseFile, System, Temporary
```

Members in this enum are combinable. Here's how to toggle a single file attribute without upsetting the rest:

```
string filePath = "test.txt";

FileAttributes fa = File.GetAttributes (filePath);
if ((fa & FileAttributes.ReadOnly) != 0)
{
    // Use the exclusive-or operator (^) to toggle the ReadOnly flag
    fa ^= FileAttributes.ReadOnly;
    File.SetAttributes (filePath, fa);
}

// Now we can delete the file, for instance:
File.Delete (filePath);
```

**NOTE**

`FileInfo` offers an easier way to change a file's read-only flag:

```
new FileInfo ("test.txt").IsReadOnly = false;
```

## Compression and encryption attributes

**NOTE**

This feature is Windows-only and requires the NuGet package `System.Management`.

The `Compressed` and `Encrypted` file attributes correspond to the compression and encryption checkboxes on a file or directory's Properties dialog box in Windows Explorer. This type of compression and encryption is *transparent* in that the OS does all the work behind the scenes, allowing you to read and write plain data.

You cannot use `SetAttributes` to change a file's `Compressed` or `Encrypted` attributes—it fails silently if you try! The workaround is simple in the latter case: you instead call the `Encrypt()` and `Decrypt()` methods in the `File` class. With compression, it's more complicated; one solution is to use the Windows Management Instrumentation (WMI) API in `System.Management`. The following method compresses a directory, returning `0` if successful (or a WMI error code if not):

```
static uint CompressFolder (string folder, bool recursive)
{
  string path = "Win32_Directory.Name='" + folder + "'";
  using (ManagementObject dir = new ManagementObject (path))
  using (ManagementBaseObject p = dir.GetMethodParameters ("CompressEx"))
  {
    p ["Recursive"] = recursive;
    using (ManagementBaseObject result = dir.InvokeMethod ("CompressEx",
                                                           p, null))
      return (uint) result.Properties ["ReturnValue"].Value;
  }
}
```

To uncompress, replace `CompressEx` with `UncompressEx`.

Transparent encryption relies on a key seeded from the logged-in user's password. The system is robust to password changes performed by the authenticated user, but if a password is reset via an administrator, data in encrypted files is unrecoverable.

> **NOTE**
>
> Transparent encryption and compression require special filesystem support. NTFS (used most commonly on hard drives) supports these features; CDFS (on CD-ROMs) and FAT (on removable media cards) do not.

You can determine whether a volume supports compression and encryption with Win32 interop:

```
using System;
using System.IO;
using System.Text;
using System.ComponentModel;
using System.Runtime.InteropServices;

class SupportsCompressionEncryption
{
  const int SupportsCompression = 0x10;
  const int SupportsEncryption = 0x20000;

  [DllImport ("Kernel32.dll", SetLastError = true)]
  extern static bool GetVolumeInformation (string vol, StringBuilder name,
    int nameSize, out uint serialNum, out uint maxNameLen, out uint flags,
    StringBuilder fileSysName, int fileSysNameSize);

  static void Main()
  {
    uint serialNum, maxNameLen, flags;
    bool ok = GetVolumeInformation (@"C:\", null, 0, out serialNum,
                                    out maxNameLen, out flags, null, 0);
    if (!ok)
      throw new Win32Exception();

    bool canCompress = (flags & SupportsCompression) != 0;
    bool canEncrypt = (flags & SupportsEncryption) != 0;
  }
}
```

## Windows file security

> **NOTE**
>
> This feature is Windows-only and requires the NuGet package `System.IO.FileSystem.AccessControl`.

The `FileSecurity` class allow you to query and change the OS permissions assigned to users and roles (namespace `System.Security.AccessControl`).

In this example, we list a file's existing permissions and then assign Write permission to the "Users" group:

```
using System;
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;

void ShowSecurity (FileSecurity sec)
{
  AuthorizationRuleCollection rules = sec.GetAccessRules (true, true,
                                                          typeof (NTAccount));
  foreach (FileSystemAccessRule r in rules.Cast<FileSystemAccessRule>()
    .OrderBy (rule => rule.IdentityReference.Value))
  {
    // e.g., MyDomain/Joe
    Console.WriteLine ($"  {r.IdentityReference.Value}");
    // Allow or Deny: e.g., FullControl
    Console.WriteLine ($"    {r.FileSystemRights}: {r.AccessControlType}");
  }
}

var file = "sectest.txt";
File.WriteAllText (file, "File security test.");

var sid = new SecurityIdentifier (WellKnownSidType.BuiltinUsersSid, null);
string usersAccount = sid.Translate (typeof (NTAccount)).ToString();

Console.WriteLine ($"User: {usersAccount}");

FileSecurity sec = new FileSecurity (file,
                          AccessControlSections.Owner |
                          AccessControlSections.Group |
                          AccessControlSections.Access);

Console.WriteLine ("AFTER CREATE:");
ShowSecurity(sec); // BUILTIN\Users doesn't have Write permission

sec.ModifyAccessRule (AccessControlModification.Add,
    new FileSystemAccessRule (usersAccount, FileSystemRights.Write,
                              AccessControlType.Allow),
    out bool modified);

Console.WriteLine ("AFTER MODIFY:");
ShowSecurity (sec); // BUILTIN\Users has Write permission
```

We give another example, later, in “Special Folders”.

## Unix file security

From .NET 7, the `File` class includes the methods `GetUnixFileMode` and `SetUnixFileMode` to get and set file permissions on Unix systems. The `Directory.CreateDirectory` method is also now overloaded to accept a Unix file mode, and it's possible to specify a file mode when creating a file, as follows:

```
var fs = new FileStream ("test.txt",
  new FileStreamOptions
  {
    Mode = FileMode.Create,
    UnixCreateMode = UnixFileMode.UserRead | UnixFileMode.UserWrite
  });
```

## The Directory Class

The static `Directory` class provides a set of methods analogous to those in the `File` class—for checking whether a directory exists (`Exists`), moving a directory (`Move`), deleting a directory (`Delete`), getting/setting times of creation or last access, and getting/setting security permissions. Furthermore, `Directory` exposes the following static methods:

```
string GetCurrentDirectory ();
void   SetCurrentDirectory (string path);

DirectoryInfo CreateDirectory  (string path);
DirectoryInfo GetParent        (string path);
string        GetDirectoryRoot (string path);

string[] GetLogicalDrives(); // Gets mount points on Unix

// The following methods all return full paths:

string[] GetFiles             (string path);
string[] GetDirectories       (string path);
string[] GetFileSystemEntries (string path);

IEnumerable<string> EnumerateFiles             (string path);
IEnumerable<string> EnumerateDirectories       (string path);
IEnumerable<string> EnumerateFileSystemEntries (string path);
```

> **NOTE**
>
> The last three methods are potentially more efficient than the `Get*` variants because they're lazily evaluated—fetching data from the file system as you enumerate the sequence. They're particularly well suited to LINQ queries.

The `Enumerate*` and `Get*` methods are overloaded to also accept `searchPattern` (string) and `searchOption` (enum) parameters. If you specify `SearchOption.SearchAllSubDirectories`, a recursive subdirectory search is performed. The `*FileSystemEntries` methods combine the results of `*Files` with `*Directories`.

Here's how to create a directory if it doesn't already exist:

```
if (!Directory.Exists (@"d:\test"))
  Directory.CreateDirectory (@"d:\test");
```

## FileInfo and DirectoryInfo

The static methods on `File` and `Directory` are convenient for executing a single file or directory operation. If you need to call a series of methods in a row, the `FileInfo` and `DirectoryInfo` classes provide an object model that makes the job easier.

`FileInfo` offers most of the `File`'s static methods in instance form—with some additional properties such as `Extension`, `Length`, `IsReadOnly`, and `Directory`—for returning a `DirectoryInfo` object. For example:

```
static string TestDirectory =>
  RuntimeInformation.IsOSPlatform (OSPlatform.Windows)
    ? @"C:\Temp"
    : "/tmp";

Directory.CreateDirectory (TestDirectory);

FileInfo fi = new FileInfo (Path.Combine (TestDirectory, "FileInfo.txt"));

Console.WriteLine (fi.Exists);         // false
```

```
using (TextWriter w = fi.CreateText())
  w.Write ("Some text");

Console.WriteLine (fi.Exists);         // false (still)
fi.Refresh();
Console.WriteLine (fi.Exists);         // true

Console.WriteLine (fi.Name);           // FileInfo.txt
Console.WriteLine (fi.FullName);       // c:\temp\FileInfo.txt (Windows)
                                       // /tmp/FileInfo.txt (Unix)
Console.WriteLine (fi.DirectoryName);  // c:\temp (Windows)
                                       // /tmp (Unix)
Console.WriteLine (fi.Directory.Name); // temp
Console.WriteLine (fi.Extension);      // .txt
Console.WriteLine (fi.Length);         // 9

fi.Encrypt();
fi.Attributes ^= FileAttributes.Hidden;   // (Toggle hidden flag)
fi.IsReadOnly = true;

Console.WriteLine (fi.Attributes);    // ReadOnly,Archive,Hidden,Encrypted
Console.WriteLine (fi.CreationTime);  // 3/09/2019 1:24:05 PM

fi.MoveTo (Path.Combine (TestDirectory, "FileInfoX.txt"));

DirectoryInfo di = fi.Directory;
Console.WriteLine (di.Name);             // temp or tmp
Console.WriteLine (di.FullName);         // c:\temp or /tmp
Console.WriteLine (di.Parent.FullName);  // c:\ or /
di.CreateSubdirectory ("SubFolder");
```

Here's how to use `DirectoryInfo` to enumerate files and subdirectories:

```
DirectoryInfo di = new DirectoryInfo (@"e:\photos");

foreach (FileInfo fi in di.GetFiles ("*.jpg"))
  Console.WriteLine (fi.Name);

foreach (DirectoryInfo subDir in di.GetDirectories())
  Console.WriteLine (subDir.FullName);
```

## Path

The static `Path` class defines methods and fields for working with paths and filenames.

Assuming this setup code:

```
string dir  = @"c:\mydir";    // or /mydir
string file = "myfile.txt";
string path = @"c:\mydir\myfile.txt";    // or /mydir/myfile.txt

Directory.SetCurrentDirectory (@"k:\demo");    // or /demo
```

we can demonstrate `Path`'s methods and fields with the following expressions:

| Expression | Result (Windows, then Unix) |
| --- | --- |
| `Directory.GetCurrentDirectory ()` | `k:\demo\` or `/demo` |
| `Path.IsPathRooted (file)` | `False` |
| `Path.IsPathRooted (path)` | `True` |
| `Path.GetPathRoot (path)` | `c:\` or `/` |
| `Path.GetDirectoryName (path)` | `c:\mydir` or `/mydir` |
| `Path.GetFileName (path)` | `myfile.txt` |
| `Path.GetFullPath (file)` | `k:\demo\myfile.txt` or `/demo/myfile.txt` |
| `Path.Combine (dir, file)` | `c:\mydir\myfile.txt` or `/mydir/myfile.txt` |
| **File extensions:** | |
| `Path.HasExtension (file)` | `True` |
| `Path.GetExtension (file)` | `.txt` |
| `Path.GetFileNameWithoutExtension (file)` | `myfile` |
| `Path.ChangeExtension (file, ".log")` | `myfile.log` |
| **Separators and characters:** | |
| `Path.DirectorySeparatorChar` | `\` or `/` |
| `Path.AltDirectorySeparatorChar` | `/` |

| Expression | Result (Windows, then Unix) |
|---|---|
| `Path.PathSeparator` | `;` or `:` |
| `Path.VolumeSeparatorChar` | `:` or `/` |
| `Path.GetInvalidPathChars()` | chars 0 to 31 and `"<>\|`eor 0 |
| `Path.GetInvalidFileNameChars()` | chars 0 to 31 and `"<>\|:*?\/` or 0 and `/` |
| **Temporary files:** | |
| `Path.GetTempPath()` | *<local user folder>*`\Temp` or */tmp/* |
| `Path.GetRandomFileName()` | *`d2dwuzjf.dnp`* |
| `Path.GetTempFileName()` | *<local user folder>*`\Temp\`*`tmp14B.tmp`* or */tmp/* *`tmpubSUYO.tmp`* |

`Combine` is particularly useful: it allows you to combine a directory and filename—or two directories—without first having to check whether a trailing path separator is present, and it automatically uses the correct path separator for the OS. It provides overloads that accept up to four directory and/or filenames.

`GetFullPath` converts a path relative to the current directory to an absolute path. It accepts values such as *..\..\file.txt*.

`GetRandomFileName` returns a genuinely unique 8.3-character filename, without actually creating any file. `GetTempFileName` generates a temporary filename using an autoincrementing counter that repeats every 65,000 files. It then creates a zero-byte file of this name in the local temporary directory.

> **WARNING**
>
> You must delete the file generated by `GetTempFileName` when you're done; otherwise, it will eventually throw an exception (after your 65,000th call to `GetTempFileName`). If this is a problem, you can instead `Combine GetTempPath` with `GetRandomFileName`. Just be careful not to fill up the user's hard drive!

## Special Folders

One thing missing from `Path` and `Directory` is a means to locate folders such as *My Documents*, *Program Files*, *Application Data*, and so on. This is provided instead by the `GetFolderPath` method in the `System.Environment` class:

```
string myDocPath = Environment.GetFolderPath
  (Environment.SpecialFolder.MyDocuments);
```

`Environment.SpecialFolder` is an enum whose values encompass all special directories in Windows, such as `AdminTools`, `ApplicationData`, `Fonts`, `History`, `SendTo`, `StartMenu`, and so on. Everything is covered here except the .NET runtime directory, which you can obtain as follows:

```
System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory()
```

> **NOTE**
>
> Most of the special folders have no path assigned on Unix systems. The following have paths on Ubuntu Linux 18.04 Desktop: `ApplicationData`, `CommonApplicationData`, `Desktop`, `DesktopDirectory`, `LocalApplicationData`, `MyDocuments`, `MyMusic`, `MyPictures`, `MyVideos`, `Templates`, and `UserProfile`.

Of particular value on Windows systems is `ApplicationData`, where you can store settings that travel with a user across a network (if roaming profiles are enabled on the network domain); `LocalApplicationData`,

which is for nonroaming data (specific to the logged-in user); and `CommonApplicationData`, which is shared by every user of the computer. Writing application data to these folders is considered preferable to using the Windows Registry. The standard protocol for storing data in these folders is to create a subdirectory with the name of your application:

```
string localAppDataPath = Path.Combine (
  Environment.GetFolderPath (Environment.SpecialFolder.ApplicationData),
  "MyCoolApplication");

if (!Directory.Exists (localAppDataPath))
  Directory.CreateDirectory (localAppDataPath);
```

There's a horrible trap when using `CommonApplicationData`: if a user starts your program with administrative elevation and your program then creates folders and files in `CommonApplicationData`, that user might lack permissions to replace those files later, when run under a restricted Windows login. (A similar problem exists when switching between restricted-permission accounts.) You can work around it by creating the desired folder (with permissions assigned to everyone) as part of your setup.

Another place to write configuration and log files is to the application's base directory, which you can obtain with `AppDomain.CurrentDomain.BaseDirectory`. This is not recommended, however, because the OS is likely to deny your application permissions to write to this folder after initial installation (without administrative elevation).

## Querying Volume Information

You can query the drives on a computer with the `DriveInfo` class:

```
DriveInfo c = new DriveInfo ("C");       // Query the C: drive.
                                         // On Unix: /

long totalSize = c.TotalSize;            // Size in bytes.
long freeBytes = c.TotalFreeSpace;       // Ignores disk quotas.
```

```
long freeToMe  = c.AvailableFreeSpace;   // Takes quotas into account.

foreach (DriveInfo d in DriveInfo.GetDrives())  // All defined drives.
                                                // On Unix: mount points
{
  Console.WriteLine (d.Name);             // C:\
  Console.WriteLine (d.DriveType);        // Fixed
  Console.WriteLine (d.RootDirectory);    // C:\

  if (d.IsReady)   // If the drive is not ready, the following two
                   // properties will throw exceptions:
  {
    Console.WriteLine (d.VolumeLabel);    // The Sea Drive
    Console.WriteLine (d.DriveFormat);    // NTFS
  }
}
```

The static `GetDrives` method returns all mapped drives, including CD-ROMs, media cards, and network connections. `DriveType` is an enum with the following values:

```
Unknown, NoRootDirectory, Removable, Fixed, Network, CDRom, Ram
```

## Catching Filesystem Events

The `FileSystemWatcher` class lets you monitor a directory (and optionally, subdirectories) for activity. `FileSystemWatcher` has events that fire when files or subdirectories are created, modified, renamed, and deleted, as well as when their attributes change. These events fire regardless of the user or process performing the change. Here's an example:

```
Watch (GetTestDirectory(), "*.txt", true);

void Watch (string path, string filter, bool includeSubDirs)
{
  using (var watcher = new FileSystemWatcher (path, filter))
  {
    watcher.Created += FileCreatedChangedDeleted;
    watcher.Changed += FileCreatedChangedDeleted;
    watcher.Deleted += FileCreatedChangedDeleted;
    watcher.Renamed += FileRenamed;
    watcher.Error   += FileError;
```

```
    watcher.IncludeSubdirectories = includeSubDirs;
    watcher.EnableRaisingEvents = true;

    Console.WriteLine ("Listening for events - press <enter> to end");
    Console.ReadLine();
  }
  // Disposing the FileSystemWatcher stops further events from firing.
}

void FileCreatedChangedDeleted (object o, FileSystemEventArgs e)
  => Console.WriteLine ("File {0} has been {1}", e.FullPath, e.ChangeType);

void FileRenamed (object o, RenamedEventArgs e)
  => Console.WriteLine ("Renamed: {0}->{1}", e.OldFullPath, e.FullPath);

void FileError (object o, ErrorEventArgs e)
  => Console.WriteLine ("Error: " + e.GetException().Message);

string GetTestDirectory() =>
  RuntimeInformation.IsOSPlatform (OSPlatform.Windows)
    ? @"C:\Temp"
    : "/tmp";
```

**WARNING**

Because `FileSystemWatcher` raises events on a separate thread, you must exception-handle the event handling code to prevent an error from taking down the application. For more information, see "Exception Handling".

The `Error` event does not inform you of filesystem errors; instead, it indicates that the `FileSystemWatcher`'s event buffer overflowed because it was overwhelmed by `Changed`, `Created`, `Deleted`, or `Renamed` events. You can change the buffer size via the `InternalBufferSize` property.

`IncludeSubdirectories` applies recursively. So, if you create a `FileSystemWatcher` on *C:* with `IncludeSubdirectories true`, its events will fire when a file or directory changes anywhere on the hard drive.

> **WARNING**
>
> A trap in using `FileSystemWatcher` is to open and read newly created or updated files before the file has been fully populated or updated. If you're working in conjunction with some other software that's creating files, you might need to consider some strategy to mitigate this, such as creating files with an unwatched extension and then renaming them after they're fully written.

# OS Security

All applications are subject to OS restrictions, based on the user's login privileges. These restrictions affect file I/O as well as other capabilities, such as access to the Windows Registry.

In Windows and Unix, there are two types of accounts:

- An administrative/superuser account that imposes no restrictions in accessing the local computer
- A limited permissions account that restricts administrative functions and visibility of other users' data

On Windows, a feature called User Account Control (UAC) means that administrators receive two tokens or "hats" when logging in: an administrative hat and an ordinary user hat. By default, programs run wearing the ordinary user hat—with restricted permissions—unless the program requests *administrative elevation*. The user must then approve the request in the dialog box that's presented.

On Unix, users typically log in with restricted accounts. That is also true for administrators to lessen the probability of inadvertently damaging the system. When a user needs to run a command that requires elevated permissions, they precede the command with `sudo` (short for "super-user do").

*By default*, your application will run with restricted user privileges. This means that you must either:

- Write your application such that it can run without administrative privileges.
- Demand administrative elevation in the application manifest (Windows only), or detect the lack of required privileges and alert the user to restart the application as an administrator/super-user.

The first option is safer and more convenient for the user. Designing your program to run without administrative privileges is easy in most cases.

You can find out whether you're running under an administrative account as follows:

```
[DllImport("libc")]
public static extern uint getuid();

static bool IsRunningAsAdmin()
{
  if (RuntimeInformation.IsOSPlatform (OSPlatform.Windows))
  {
    using var identity = WindowsIdentity.GetCurrent();
    var principal = new WindowsPrincipal (identity);
    return principal.IsInRole (WindowsBuiltInRole.Administrator);
  }
  return getuid() == 0;
}
```

With UAC enabled on Windows, this returns `true` only if the current process has administrative elevation. On Linux, it returns `true` only if the current process is running as super-user (e.g., *sudo myapp*).

## Running in a Standard User Account

Here are the key things that you *cannot* do in a standard user account:

- Write to the following directories:
    - The OS folder (typically *\Windows* or */bin, /sbin, ...*) and subdirectories

    - The program files folder (*\Program Files* or */usr/bin, /opt*) and subdirectories
    - The root of the OS drive (e.g., *C:* or */*)
- Write to the HKEY_LOCAL_MACHINE branch of the Registry (Windows)
- Read performance monitoring (WMI) data (Windows)

Additionally, as an ordinary Windows user (or even as an administrator), you might be refused access to files or resources that belong to other users. Windows uses a system of Access Control Lists (ACLs) to protect such resources—you can query and assert your own rights in the ACLs via types in `System.Security.AccessControl`. ACLs can also be applied to cross-process wait handles, described in Chapter 21.

If you're refused access to anything as a result of OS security, the CLR detects the failure and throws an `UnauthorizedAccessException` (rather than failing silently).

In most cases, you can deal with standard user restrictions as follows:

- Write files to their recommended locations.
- Avoid using the Registry for information that can be stored in files (aside from the HKEY_CURRENT_USER hive, which you will have read/write access to on Windows only).
- Register ActiveX or COM components during setup (Windows only).

The recommended location for user documents is `SpecialFolder.MyDocuments`:

```
string docsFolder = Environment.GetFolderPath
                    (Environment.SpecialFolder.MyDocuments);

string path = Path.Combine (docsFolder, "test.txt");
```

The recommended location for configuration files that a user might need to modify outside of your application is `SpecialFolder.ApplicationData` (current user only) or `SpecialFolder.CommonApplicationData` (all users). You typically create subdirectories within these folders, based on your organization and product name.

## Administrative Elevation and Virtualization

With an *application manifest*, you can request that Windows prompt the user for administrative elevation whenever running your program (Linux ignores this request):

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel level="requireAdministrator" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

(We describe application manifests in more detail in Chapter 17.)

If you replace `requireAdministrator` with `asInvoker`, it instructs Windows that administrative elevation is *not* required. The effect is almost the same as not having an application manifest at all—except that *virtualization* is disabled. Virtualization is a temporary measure introduced with Windows Vista to help old applications run correctly without administrative privileges. The absence of an application manifest with a `requestedExecutionLevel` element activates this backward-compatibility feature.

Virtualization comes into play when an application writes to the *Program Files* or *Windows* directory, or the HKEY_LOCAL_MACHINE area of the Registry. Instead of throwing an exception, changes are redirected to a separate location on the hard disk where they can't affect the original data.

This prevents the application from interfering with the OS—or other well-behaved applications.

# Memory-Mapped Files

*Memory-mapped files* provide two key features:

- Efficient random access to file data
- The ability to share memory between different processes on the same computer

The types for memory-mapped files reside in the `System.IO.MemoryMappedFiles` namespace. Internally, they work by wrapping the operating system's API for memory-mapped files.

## Memory-Mapped Files and Random File I/O

Although an ordinary `FileStream` allows random file I/O (by setting the stream's `Position` property), it's optimized for sequential I/O. As a rough rule of thumb:

- `FileStream`s are approximately 10 times faster than memory-mapped files for sequential I/O.
- Memory-mapped files are approximately 10 times faster than `FileStream`s for random I/O.

Changing a `FileStream`'s `Position` can cost several microseconds—which adds up if done within a loop. A `FileStream` is also unsuitable for multithreaded access—because its position changes as it is read or written.

To create a memory-mapped file:

1. Obtain a `FileStream` as you would ordinarily.
2. Instantiate a `MemoryMappedFile`, passing in the file stream.

3. Call `CreateViewAccessor` on the memory-mapped file object.

The last step gives you a `MemoryMappedViewAccessor` object that provides methods for randomly reading and writing simple types, structures, and arrays (more on this in "Working with View Accessors").

The following creates a one million–byte file and then uses the memory-mapped file API to read and then write a byte at position 500,000:

```
File.WriteAllBytes ("long.bin", new byte [1000000]);

using MemoryMappedFile mmf = MemoryMappedFile.CreateFromFile ("long.bin");
using MemoryMappedViewAccessor accessor = mmf.CreateViewAccessor();

accessor.Write (500000, (byte) 77);
Console.WriteLine (accessor.ReadByte (500000));   // 77
```

You can also specify a map name and capacity when calling `CreateFromFile`. Specifying a non-null map name allows the memory block to be shared with other processes (see the following section); specifying a capacity automatically enlarges the file to that value. The following creates a 1,000-byte file:

```
File.WriteAllBytes ("short.bin", new byte [1]);
using (var mmf = MemoryMappedFile.CreateFromFile
                 ("short.bin", FileMode.Create, null, 1000))
  ...
```

## Memory-Mapped Files and Shared Memory (Windows)

Under Windows, you can also use memory-mapped files as a means of sharing memory between processes on the same computer. One process creates a shared memory block by calling `MemoryMappedFile.CreateNew`, and then other processes subscribe to that same memory block by calling `MemoryMappedFile.OpenExisting` with the same name. Although it's still referred to as a memory-mapped "file," it resides entirely in memory and has no disk presence.

The following code creates a 500-byte shared memory-mapped file and writes the integer 12345 at position 0:

```
using (MemoryMappedFile mmFile = MemoryMappedFile.CreateNew ("Demo", 500))
using (MemoryMappedViewAccessor accessor = mmFile.CreateViewAccessor())
{
  accessor.Write (0, 12345);
  Console.ReadLine();   // Keep shared memory alive until user hits Enter.
}
```

The following code opens that memory-mapped file and reads that integer:

```
// This can run in a separate executable:
using (MemoryMappedFile mmFile = MemoryMappedFile.OpenExisting ("Demo"))
using (MemoryMappedViewAccessor accessor = mmFile.CreateViewAccessor())
  Console.WriteLine (accessor.ReadInt32 (0));   // 12345
```

## Cross-Platform Interprocess Shared Memory

Both Windows and Unix allow multiple processes to memory-map the same file. You must exercise care to ensure appropriate file sharing settings:

```
static void Writer()
{
  var file = Path.Combine (TestDirectory, "interprocess.bin");
  File.WriteAllBytes (file, new byte [100]);

  using FileStream fs =
    new FileStream (file, FileMode.Open, FileAccess.ReadWrite,
                    FileShare.ReadWrite);

  using MemoryMappedFile mmf = MemoryMappedFile
    .CreateFromFile (fs, null, fs.Length, MemoryMappedFileAccess.ReadWrite,
                     HandleInheritability.None, true);
  using MemoryMappedViewAccessor accessor = mmf.CreateViewAccessor();

  accessor.Write (0, 12345);

  Console.ReadLine();   // Keep shared memory alive until user hits Enter.

  File.Delete (file);
}
```

```
static void Reader()
{
  // This can run in a separate executable:
  var file = Path.Combine (TestDirectory, "interprocess.bin");
  using FileStream fs =
    new FileStream (file, FileMode.Open, FileAccess.ReadWrite,
                    FileShare.ReadWrite);
  using MemoryMappedFile mmf = MemoryMappedFile
    .CreateFromFile (fs, null, fs.Length, MemoryMappedFileAccess.ReadWrite,
                     HandleInheritability.None, true);
  using MemoryMappedViewAccessor accessor = mmf.CreateViewAccessor();

  Console.WriteLine (accessor.ReadInt32 (0));   // 12345
}

static string TestDirectory =>
  RuntimeInformation.IsOSPlatform (OSPlatform.Windows)
    ?  @"C:\Test"
    : "/tmp";
```

## Working with View Accessors

Calling `CreateViewAccessor` on a `MemoryMappedFile` gives you a view accessor that lets you read/write values at random positions.

The `Read*`/`Write*` methods accept numeric types, `bool`, and `char`, as well as arrays and structs that contain value-type elements or fields. Reference types—and arrays or structs that contain reference types—are prohibited because they cannot map into unmanaged memory. So, if you want to write a string, you must encode it into an array of bytes:

```
byte[] data = Encoding.UTF8.GetBytes ("This is a test");
accessor.Write (0, data.Length);
accessor.WriteArray (4, data, 0, data.Length);
```

Notice that we wrote the length first. This means we know how many bytes to read back later:

```
byte[] data = new byte [accessor.ReadInt32 (0)];
accessor.ReadArray (4, data, 0, data.Length);
Console.WriteLine (Encoding.UTF8.GetString (data));   // This is a test
```

Here's an example of reading/writing a struct:

```
struct Data { public int X, Y; }
...
var data = new Data { X = 123, Y = 456 };
accessor.Write (0, ref data);
accessor.Read (0, out data);
Console.WriteLine (data.X + " " + data.Y);   // 123 456
```

The `Read` and `Write` methods are surprisingly slow. You can get much better performance by directly accessing the underlying unmanaged memory via a pointer. Following on from the previous example:

```
unsafe
{
  byte* pointer = null;
  try
  {
    accessor.SafeMemoryMappedViewHandle.AcquirePointer (ref pointer);
    int* intPointer = (int*) pointer;
    Console.WriteLine (*intPointer);               // 123
  }
  finally
  {
    if (pointer != null)
      accessor.SafeMemoryMappedViewHandle.ReleasePointer();
  }
}
```

Your project must be configured to allow unsafe code. You can do that by editing your `.csproj` file:

```
<PropertyGroup>
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

The performance advantage of pointers is even more pronounced when working with large structures because they let you work directly with the raw data rather than using `Read`/`Write` to *copy* data between managed and unmanaged memory. We explore this further in Chapter 24.