puter—or different computers across a Windows network.

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

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 later, in the section "Memory-Mapped Files" on page 569.

`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

Named pipes

`NamedPipeServerStream` and `NamedPipeClientStream`

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

A pipe is a low-level construct that allows just the sending and receiving of bytes (or *messages*, which are groups of bytes). The WCF and Remoting APIs offer higher-level messaging frameworks with the option of using an IPC channel for communication.

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);
  Console.WriteLine (s.ReadByte());
}
```

Here's the corresponding client code:

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

```
using (var s = new NamedPipeClientStream ("pipedream"))
{
  s.Connect();
  Console.WriteLine (s.ReadByte());
```

```
  s.WriteByte (200);
}
```

```
// Send the value 200 back.
```

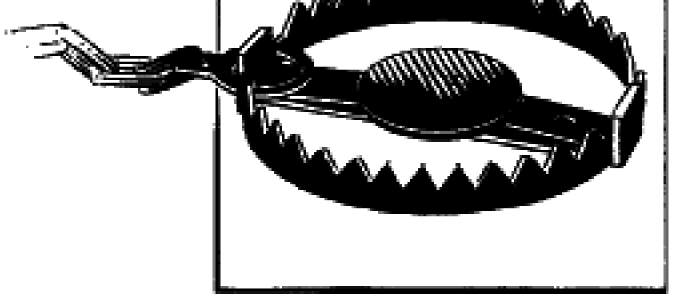
Named pipe streams are bidirectional by default, so either party can read or write their stream. This means the client and server must agree on some protocol to coordinate their actions, so both parties don't end up sending or receiving at once.

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 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. If this is enabled, a party calling Read can know when a message is complete by checking the IsMessageComplete property. To demonstrate, we'll start 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();
}
```



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 kinds of stream, 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"))
```

ing 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);
}
```

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 com-

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 demonstrates a server that sends a single byte to the child process, and then receives a single byte back from that process:

```
    string clientExe = @"d:\PipeDemo\ClientDemo.exe";
```

```
HandleInheritability inherit = HandleInheritability.Inheritable;
HandleInheritability inherit = HandleInheritability.Inheritable;
```

```
using (var tx = new AnonymousPipeServerStream (PipeDirection.Out, inherit))
using (var rx = new AnonymousPipeServerStream (PipeDirection.In, inherit))
{
  string txID = tx.GetClientHandleAsString();
  string rxID = rx.GetClientHandleAsString();
```

```
var startInfo = new ProcessStartInfo (clientExe, txID + " " + rxID);
startInfo.UseShellExecute = false;      // Required for child process
Process p = Process.Start (startInfo);
```

```
tx.DisposeLocalCopyOfClientHandle();
rx.DisposeLocalCopyOfClientHandle();
```

```
// Release unmanaged
// handle resources.
```

```
tx.WriteByte (100);
Console.WriteLine ("Server received: " + rx.ReadByte());

  p.WaitForExit();
}
```

Here's the corresponding client code that would be compiled to *d:\PipeDemo\ClientDemo.exe*:

```
string rxID = args[0];
string txID = args[1];

// Note we're reversing the
// Note we're reversing the
// receive and transmit roles.
```

```
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);
}
```

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

`BufferedStream` decorates, or wraps, another stream with buffering capability, and it is one of a number of decorator stream types in the core .NET Framework, all of which are illustrated in Figure 14-4.

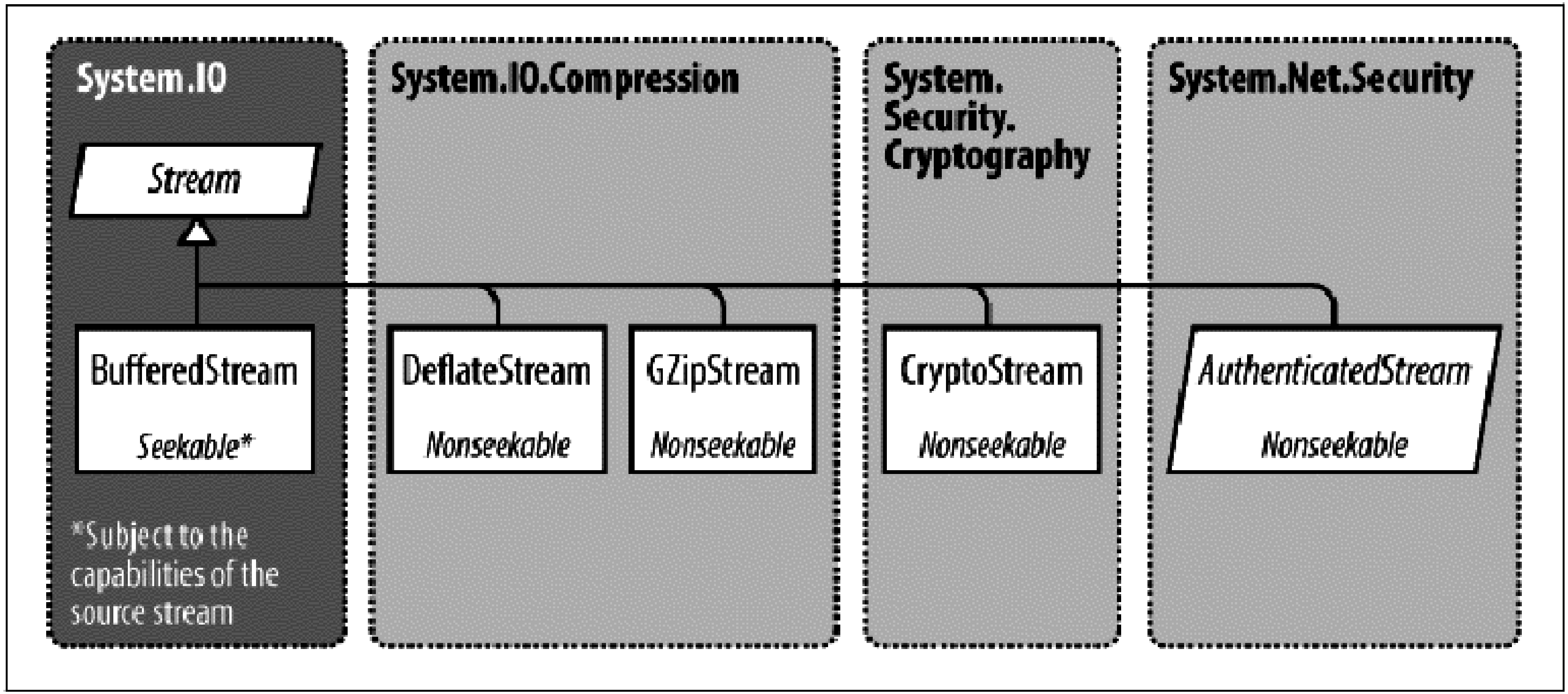


Figure 14-4. Decorator streams

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))
{
  bs.ReadByte();
```

```
{
  bs.ReadByte();
//20K buffer
```

```
  Console.WriteLine (fs.Position);
}
// 20000
```

In this example, the underlying stream advances 20,000 bytes after reading just 1 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 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 the Framework 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

`BinaryReader`, `BinaryWriter`

XML adapters (covered in Chapter 11)

`XmlReader`, `XmlWriter`

The relationships between these types are illustrated in Figure 14-5.

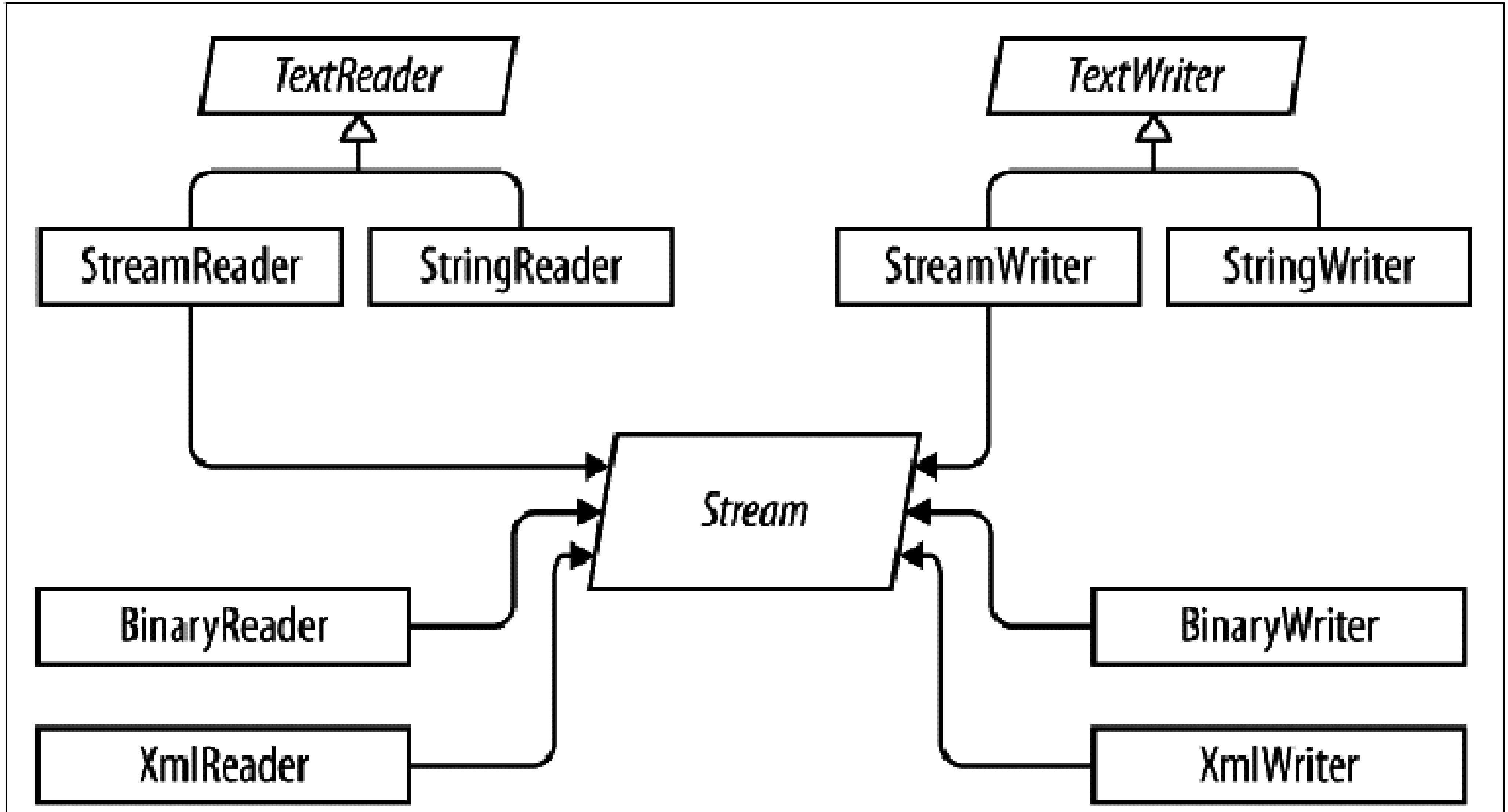


Figure 14-5. Readers and writers

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 the framework:

`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 14-2 lists `TextReader`'s members by category. `Peek` returns the next character in the stream without advancing the position. Both `Peek` and the zero-argument

Table 14-2 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 14-2. TextReader members

Category	Members

Readingonechar	`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 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);`

[redacted]

The new line sequence in Windows is loosely modeled on a mechanical typewriter: a carriage return (character 13) followed by a line feed (character 10). The C# string is "\r\n" (think “ReturN”). Reverse the order and you’ll get either two new lines or none!

TextWriter has analogous methods for writing, as shown in Table 14-3. 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 14-3. TextWriter members

Category	Members
Writing one char	`public virtual void Write (char value);`

Writing one char

```
public virtual void Write (char value);
```

Writing many chars

Closing and flushing

```
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);
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; }
```

encoding	`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 CR+LF. You can change this via the NewLine property (this can be useful for interoperability with Unix file formats).

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");
  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");
```

```
    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 (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.

[redacted]

A StreamReader or StreamWriter will throw an exception if it encounters bytes that do not have a valid string translation for their 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 U.S.-style keyboard. Most other characone byte. The ASCII encoding maps the first 127 characters of the Unicode set into its single byte, covering what you see on a U.S.-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 this:

```
using (TextWriter w = File.CreateText ("but.txt"))
  w.WriteLine ("but-");

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

```
// Use default UTF-8
// encoding.
// encoding.
```

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, since 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:

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);
```

The output is then:

```
255     // Byte-order mark 1
254     // Byte-order mark 2
98      // 'b' byte 1
0       // 'b' byte 2
117     // 'u' byte 1
```

```
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 2 bytes is not always enough).

Technically, UTF-16 uses either two or four bytes per character (there are close to a million Unicode characters allocated or reserved, so 2 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, though, is that they share a base class with StreamReader/StreamWriter. For instance, suppose 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

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 strings and arrays of the primitive data types.

Unlike StreamReader and StreamWriter, binary adapters store primitive data types efficiently, as they are represented in memory. So, an int uses four bytes; a double 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 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();
```

```
    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.

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.

> 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, 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!

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))
```

```
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.

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.

calls Dispose. This allows an abandoned adapter to evade automatic disposal when the garbage collector catches up with it.

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

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 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 meth-

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);
DateTime GetLastAccessTime (string path);
DateTime GetLastWriteTime  (string path);
DateTime GetLastWriteTime  (string path);

// UTC versions are
// also provided.

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. Here are all the members of the can tell this in advance by calling GetAttributes. Here are all the members of the FileAttribute enum that GetAttributes returns:

```
Archive, Compressed, Device, Directory, Encrypted,
Hidden, Normal, 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 = @"c:\temp\test.txt";

FileAttributes fa = File.GetAttributes (filePath);
if ((fa & FileAttributes.ReadOnly) > 0)
{
    fa ^= FileAttributes.ReadOnly;
    File.SetAttributes (filePath, fa);
}
```

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

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

```
new FileInfo (@"c:\temp\test.txt").IsReadOnly = false;
```

Compression and encryption attributes

The Compressed and Encrypted file attributes correspond to the compression and encryption checkboxes on a file or directory's *properties* dialog in Windows Explorer. This type of compression and encryption is *transparent* in that the operating system 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 Instrumentacall 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.

> 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:

Win32 interop:

```
using System;
using System.IO;
using System.Text;
```

```
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,
[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;
  }
}
```

File security

The GetAccessControl and SetAccessControl methods allow you to query

The GetAccessControl and SetAccessControl methods allow you to query and change the operating system permissions assigned to users and roles via a FileSecurity object (namespace System.Security.AccessControl). You can also pass a FileSecurity object to a FileStream's constructor to specify permissions when creating a new file.

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

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

...

FileSecurity sec = File.GetAccessControl (@"c:\temp\test.txt");
AuthorizationRuleCollection rules = sec.GetAccessRules (true, true,
                                                        typeof (NTAccount));
```

```
AuthorizationRuleCollection rules = sec.GetAccessRules (true, true,
                                                         typeof (NTAccount));
foreach (FileSystemAccessRule rule in rules)
{
  Console.WriteLine (rule.AccessControlType);         // Allow or Deny
  Console.WriteLine (rule.FileSystemRights);          // e.g., FullControl
  Console.WriteLine (rule.IdentityReference.Value);   // e.g., MyDomain/Joe
}
```

```
FileSystemAccessRule newRule = new FileSystemAccessRule
  ("Users", FileSystemRights.ExecuteFile, AccessControlType.Allow);
```

```
sec.AddAccessRule (newRule);
File.SetAccessControl (@"c:\temp\test.txt", sec);
```

The Directory Class

The static Directory class provides a set of methods analogous to those in the File

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();

// The following methods all return full paths:

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

```
string[] GetFileSystemEntries (string path);
```

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

The last three methods were added in Framework 4.0. They’re potentially more efficient than the Get* variants, because they’re lazily evaluated—fetching data from the filesystem 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 (@"c:\temp"))
  Directory.CreateDirectory (@"c:\temp");
```

FileInfo and DirectoryInfo

The static methods on File and Directory are convenient for executing a single file

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:

```
FileInfo fi = new FileInfo (@"c:\temp\FileInfo.txt");
Console.WriteLine (fi.Exists);         // false
```

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

```
Console.WriteLine (fi.Exists);
fi.Refresh();
fi.Refresh();
Console.WriteLine (fi.Exists);
```

```
// false (still)
// true
```

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

```
fi.Encrypt();
fi.Attributes ^= FileAttributes.Hidden;
fi.IsReadOnly = true;
fi.IsReadOnly = true;

// (Toggle hidden flag)

Console.WriteLine (fi.Attributes);
Console.WriteLine (fi.CreationTime);

// ReadOnly,Archive,Hidden,Encrypted

fi.MoveTo (@"c:\temp\FileInfoX.txt");
DirectoryInfo di = fi.Directory;
Console.WriteLine (di.Name);
Console.WriteLine (di.FullName);
Console.WriteLine (di.Parent.FullName);
di.CreateSubdirectory ("SubFolder");
```

```
Console.WriteLine (di.Parent.FullName);
di.CreateSubdirectory ("SubFolder");
```

```
// temp
// c:\temp
// c:\
```

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

Path

The static Path class defines methods and fields for working with paths and filenames. Assuming this setup code:

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

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

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

Expression:	Result
Directory.GetCurrentDirectory()	k:\demo\
Path.IsPathRooted (file)	False
Path.IsPathRooted (path)	True
Path.GetPathRoot (path)	c:\

Path.GetPathRoot (path)	c:\
Path.GetDirectoryName (path)	c:\mydir
Path.GetFileName (path)	myfile.txt
Path.GetFullPath (file)	k:\demo\myfile.txt
Path.Combine (dir, file)	c:\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.AltDirectorySeparatorChar	/
Path.PathSeparator	;

Path.VolumeSeparatorChar

Path.GetInvalidPathChars()

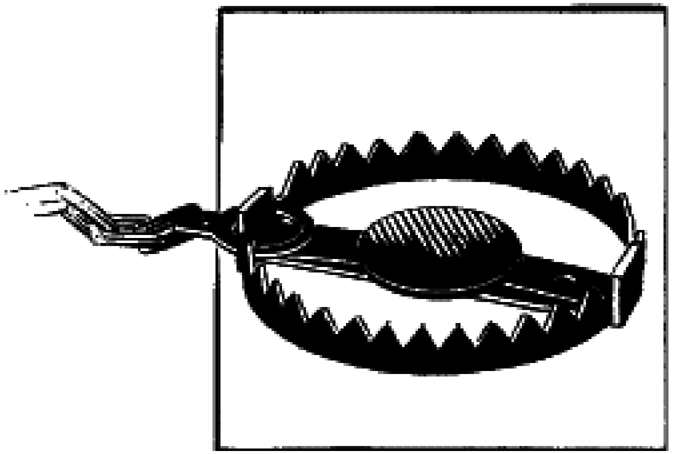
Path.GetInvalidPathChars()	
:	
chars 0 to 31 and "<>|	
Path.GetInvalidFileNameChars()	chars 0 to 31 and "<>|:*?\/
Temporary files:	
Path.GetTempPath()	*<local user folder>*\Temp

`Path.GetRandomFileName()`	*d2dwuzjf.dnp*
`Path.GetTempFileName()`	*<local user folder>*\Temp*tmp14B.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 backslash is present.

`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 auto-incrementing counter that repeats every 65,000 files. It then creates a zero-byte file of this name in the local temporary directory.



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

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:

```
AdminTools
ApplicationData
CDBurning
CommonAdminTools
CommonVideos
Cookies
```

```
Cookies
Desktop
DesktopDirectory
Favorites
Fonts
History
InternetCache
LocalApplicationData
LocalizedResources
MyComputer
MyDocuments
MyMusic
```

```
MyDocuments
MyMusic
MyPictures
MyVideos
NetworkShortcuts
Personal
PrinterShortcuts
ProgramFiles
ProgramFilesX86
Programs
Recent
Resources
```

```
Recent
Resources
SendTo
StartMenu
Startup
System
SystemX86
Templates
UserProfile
Windows

CommonApplicationData
CommonDesktopDirectory
```

```
CommonDesktopDirectory
CommonDocuments
CommonMusic
CommonOemLinks
CommonPictures
CommonProgramFiles
CommonProgramFilesX86
CommonPrograms
CommonStartMenu
CommonStartup
CommonTemplates
```

Of particular value is `ApplicationData`: this is where you can store settings that travel with a user across a network (if roaming profiles are enabled on the network domain). `LocalApplicationData` is for non-roaming data; `CommonApplicationData` is shared by every user of the computer. Writing application data to these folders is considered preferable to using the Windows Registry. A better option still, in most cases, is to use isolated storage (described in the final section of this chapter).

The simplest 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 operating system is likely to deny your application permissions to write to this folder after initial installation (without administrative elevation).

The following method returns the .NET Framework directory:

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

Querying Volume Information

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

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

```
long totalSize = c.TotalSize;
long freeBytes = c.TotalFreeSpace;
long freeToMe  = c.AvailableFreeSpace;
```

```
// Size in bytes.
// Ignores disk quotas.
// Takes quotas into account.
```

```
foreach (DriveInfo d in DriveInfo.GetDrives())    // All defined drives.
{
  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:

change. These events fire regardless of the user or process performing the change. Here's an example:

```
static void Main() { Watch (@"c:\temp", "*.txt", true); }

static 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;
```

```
 watcher.EnableRaisingEvents = true;

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

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

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

static void FileError (object o, ErrorEventArgs e)
{
  Console.WriteLine ("Error: " + e.GetException().Message);
}
```

```
}
```

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. See "Exception Handling" on page 591 for more information on this.

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

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.

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 a strategy to mitigate this, such as creating files with an unwatched extension and then renaming them once fully written.

Memory-Mapped Files

Memory-mapped files are new to Framework 4.0. They 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 Win32 API for memory-mapped files.

Memory-Mapped Files and Random File I/O

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:

●

●

FileStreams are 10 times faster than memory-mapped files for sequential I/O.

Memory-mapped files are 10 times faster than FileStreams 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.

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

The last step gives you a MemoryMappedViewAccessor object, which provides methods for randomly reading and writing simple types, structures, and arrays (more on this in "Working with View Accessors" on page 570).

The following creates a 1 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:

```
    using (var mmf = MemoryMappedFile.CreateFromFile
                     ("long.bin", FileMode.Create, null, 1000))
      ...
```

Memory-Mapped Files and Shared Memory

Memory-Mapped Files and Shared Memory

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`, while 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 lives entirely in memory and has no disk presence.

The following 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.
}
```

while the following opens that same memory-mapped file and reads that integer:

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

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

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)];
```

```
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
```

You can also directly access the underlying unmanaged memory via a pointer. Following on from the previous example:

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

Pointers can be advantageous when working with large structures: 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 25.

Compression

Two general-purpose compression streams are provided in the System.IO.Compression namespace: DeflateStream and GZipStream. Both use a popular compression

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 for errors. `GZipStream` also conforms to a standard recognized by other software.

Both 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` and `GZipStream` 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);
    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
```

Even with the smaller of the two algorithms, the compressed file is 241 bytes long:

Even with the smaller of the two algorithms, the compressed file is 241 bytes long: more than double the original! Compression works poorly with “dense,” nonrepetitive binary files. 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. This also demonstrates chaining a backing store stream, a decorator stream, and an adapter, as depicted at the start of the chapter in Figure 14-1:

```
string[] words = "The quick brown fox jumps over the lazy dog".Split();
Random rand = new Random();
```

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

```
using (TextWriter w = new StreamWriter (ds))
  for (int i = 0; i < 1000; i++)
    w.Write (words [rand.Next (words.Length)] + " ");
```

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

```
// 1073
```

```
  // 1073
```

```
using (Stream s = File.OpenRead ("compressed.bin"))
using (Stream ds = new DeflateStream (s, CompressionMode.Decompress))
using (TextReader r = new StreamReader (ds))
  Console.Write (r.ReadToEnd());                    // 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, DeflateStream compresses efficiently to 1,073 bytes—slightly more than 1 byte per word.

Compressing in Memory

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

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);
```

```
// 113
```

```
// 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.

meaning we must then call ToArray to extract its data.

Here's an alternative that avoids closing the MemoryStream:

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

```
MemoryStream ms = new MemoryStream();
using (Stream ds = new DeflateStream (ms, CompressionMode.Compress, true))
  ds.Write (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 += ds.Read (data, i, 1000 - i));
```

The additional flag sent to DeflateStream's constructor tells it not to 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.

Isolated Storage

Isolated Storage

Each .NET program has access to a special filesystem unique to that program, called *isolated storage*. Isolated storage is useful and important for a number of reasons:

●

●

●

Your application, if subject to code access security, is more likely to be granted permission to isolated storage than any other form of file I/O (by default, even applications running from an Internet URI or Silverlight sandbox have some access to isolated storage).

Data that you create is segregated from other applications.

Isolated storage is UAC-friendly.

In terms of security, isolated storage is a fence designed more to keep you in than to keep other applications out! Data in isolated storage is strongly protected against keep other applications out! Data in isolated storage is strongly protected against intrusion from other .NET applications running under the most restricted permission set (i.e., the “Internet” zone). In other cases, there’s no hard security preventing another application from accessing your isolated storage *if it really wants to*. The benefit of using isolated storage over `CommonApplicationData` is that applications must go out of their way to interfere with each other—it cannot happen through carelessness or by accident.

Applications running in a sandbox can have their quota of isolated storage limited via permissions. The default, for Internet and Silverlight applications, is 1MB in Framework 4.0.

From Framework 4.0, a hosted UI-based application (e.g., Silverlight) can ask the user for permission to increase the isolated storage quota by calling the `IncreaseQuotaTo` method on an `IsolatedStorageFile` object. This must be called from a user-initiated event, such as a button click. If the user agrees, the method returns true.

You can query the current allowance via the `Quota` property.

Isolated storage also has disadvantages:

•

•

The API is somewhat awkward to use—particularly when accessing roaming

The API is somewhat awkward to use—particularly when accessing roaming stores.

You can read/write only with an `IsolatedStorageStream`; you cannot obtain a file or directory path and then use ordinary file I/O.

Isolation Types

Isolated storage can separate by both program and user. This results in three basic types of compartments:

Local user compartments
: One per user, per program, per computer

Roaming user compartments
: One per user, per program

Machine compartments
: One per program, per computer (shared by all users of a program)

Machine compartments

One per program, per computer (shared by all users of a program)

The data in a roaming user compartment follows the user across a network—with appropriate operating system and domain support. If this support is unavailable, it behaves like a local user compartment.

So far, we've talked about how isolated storage separates by "program." Isolated storage considers a program to be one of two things, depending on which mode you choose:

●

●

An assembly

An assembly running within the context of a particular application

The latter is called *domain isolation* and is more commonly used than *assembly isolation*. Domain isolation segregates according to two things: the currently executing assembly and the executable or web application that originally started it. Assembly isolation segregates only according to the currently executing assembly—

Assembly isolation segregates only according to the currently executing assembly—so different applications calling the same assembly will share the same store.

Assemblies and applications are identified by their strong name. If no strong name is present, the assembly's full file path or URI is used instead. This means that if you move or rename a weakly named assembly, its isolated storage is reset.

In total, then, there are six kinds of isolated storage compartments. Table 14-4 compares the isolation provided by each.

Table 14-4. Isolated storage containers

Type	Computer?	Application?	Assembly?	User?	Method to obtain store

Type	Computer?	Application?	Assembly?	User?	Method to obtain store
Domain User (default)	✓	✓	✓	✓	`GetUserStoreForDomain`

Domain

Roaming

Domain

Machine

✓

✓

✓

✓

✓

✓

✓

✓

✓

✓

`GetMachineStoreForDomain`
`GetUserStoreForAssembly`

Assembly User

Type	Computer?	Application?	Assembly?	User?	Method to obtain store

Type	Computer?	Application?	Assembly?	User?	Method to obtain store
Assembly Roaming			✓	✓	
Assembly Machine	✓		✓		`GetMachineStoreForAssembly`

There is no such thing as domain-only isolation. If you want to share an isolated store across all assemblies within an application, there's a simple workaround, however. Just expose a public method in one of the assemblies that instantiates and returns an `IsolatedStorageFileStream` object. Any assembly can access any isolated store if given an `IsolatedStorageFile` object—isolation restrictions are imposed upon construction, not subsequent use.

Similarly, there's no such thing as machine-only isolation. If you want to share an isolated store across a variety of applications, the workaround is to write a common assembly that all applications reference, and then expose a method on the common assembly that creates and returns an assembly-isolated `IsolatedStorageFileStream`. The common assembly must be strongly named for this to work.

Reading and Writing Isolated Storage