El sistema de archivos

Las clases de gestión del sistema de archivos

El Framework .NET proporciona una serie de clases en el espacio de nombres System.IO de la librería **mscorlib.dll** que permiten gestionar el sistema de archivos al completo: las lecturas, los directorios y los archivos.

1. DriveInfo

La clase DriveInfo proporciona los miembros que permiten obtener la información relativa a los lectores de una máquina. Su método estático GetDrives devuelve un array de objetos DriveInfo, correspondiente a los lectores de la máquina sobre la que se ejecuta la instrucción:

DriveInfo[] drives = DriveInfo.GetDrives();

La clase DriveInfo se puede instanciar pasando como argumento al constructor la letra del lector:

DriveInfo driveC = new DriveInfo("C");

Los miembros de la clase DriveInfo exponen la información relativa a los lectores:

- AvailableFreeSpace: indica la cantidad de espacio libre en el lector en bytes.
- DriveFormat: indica el formato del sistema de archivos del lector. Puede ser NTFS, FAT32 o CDFS, según el lector.
- DriveType: indica el tipo de lector, y devuelve uno de los valores de la lista System.IO.DriveType:
- CDRom para los lectores ópticos.
- Fixed para los discos duros.
- Network para los lectores de red.
- NoRootDirectory para un lector que no tenga directorio raíz.

- Ram para un lector RAM.
- Removable para los lectores de disquetes o discos duros externos.
- Unknown cuando el tipo del lector es desconocido.
- IsReady: esta propiedad booleana indica si el lector está listo para ser utilizado.
- Name: indica el nombre del lector, se trata de su letra de acceso.
- RootDirectory: indica la ruta de acceso de la raíz del lector, devolviendo un objeto DirectoryInfo.
- TotalFreeSpace: indica la cantidad total de espacio libre en el lector en bytes.
- TotalSize: indica la cantidad total de espacio del lector.
- VolumeLabel: devuelve el nombre de la unidad de un lector. Por ejemplo: Disco local.

Excepto la propiedad VolumeLabel, todas son en modo de solo lectura.

2. Directory y DirectoryInfo

La clase Directory es estática. Se usa dando la ruta de acceso del directorio durante la llamada de un método estático. Si desea hacer una única acción en un directorio, use esta clase para ahorrar la instanciación de un objeto. La clase DirectoryInfo implementa prácticamente los mismos métodos que la clase Directory. Si se desea ejecutar varias operaciones sobre un mismo directorio, la clase DirectoryInfo será más eficaz, ya que la información del directorio se leerá una única vez durante la instanciación, independientemente del número de operaciones realizadas en el directorio.

La mayor parte de los métodos de la clase DirectoryInfo se implementan en la clase Directory. Es posible crear, eliminar y modificar las propiedades de un directorio gracias a estas clases:

```
// Creación de un directorio mediante la clase estática Directory
Directory.CreateDirectory(@"C:\directorio");

// Creación de un directorio mediante la clase DirectoryInfo
DirectoryInfo directory = new DirectoryInfo(@"C:\directorio");
directory.Create();
```

El uso de la clase estática Directory para realizar una operación sobre un directorio no obliga a la instanciación de una secuencia. Si el directorio que se desea crear ya existe, no se producirá ninguna excepción. El segundo ejemplo, que utiliza la clase DirectoryInfo,

obliga a la instanciación de una secuencia, lo cual resulta algo más largo. La ventaja es que el objeto está listo para realizar múltiples operaciones.

Durante la instanciación de un objeto DirectoryInfo, si la ruta de acceso que se pasa como argumento no existe, no se produce ninguna excepción. Será durante la primera llamada del método (excepto el método Create). Para comprobar la existencia de un directorio, la clase DirectoryInfo expone la propiedad booleana Exists:

```
DirectoryInfo directory = new DirectoryInfo(@"C:\directorio");
if (directory.Exists)
{
   directory.Delete();
}
```

Si intenta realizar una operación sobre un directorio que no existe se producirá una excepción de tipo DirectoryNotFoundException.

La clase estática Directory se comporta de manera similar cuando se realiza una operación sobre un directorio inexistente. Expone un método Exists que recibe como argumento la ruta de acceso del directorio y devuelve un valor booleano que determina la existencia o no del directorio:

```
if (Directory.Exists(@"C:\directorio"))
{
   Directory.Delete(@"C:\directorio");
}
```

Las clases Directory y DirectoryInfo exponen otros métodos que permiten realizar las operaciones más habituales sobre los directorios, como por ejemplo:

- Mover un directorio se realiza mediante el método Move de la clase Directory o MoveTo de la clase DirectoryInfo:

```
if (Directory.Exists(@"C:\directorio"))
{
   Directory.Move(@"C:\directorio", @"C:\NuevoDirectorio");
}

DirectoryInfo directory = new DirectoryInfo(@"C:\directorio");
if (directory.Exists)
{
   directory.MoveTo(@"C:\NuevoDirectorio");
}
```

- Obtener los subdirectorios se realiza mediante los métodos GetDirectories o EnumerateDirectories de las clases Directory y DirectoryInfo:

```
string[] dirs = Directory.GetDirectories(@"C:\directorio");
IEnumerable<string> enumDirs = Directory
                    .EnumerateDirectories(@"C:\directorio");
DirectoryInfo directory = new DirectoryInfo(@"C:\directorio");
DirectoryInfo[] dirs = directory.GetDirectories();
IEnumerable<DirectoryInfo> enumDirs = directory
                                .EnumerateDirectories();
```

Los métodos GetDirectories y EnumerateDirectories de la clase Directory devuelven respectivamente, un array de tipo string o un objeto IEnumerable genérico de tipo string. La clase DirectoryInfo devuelve objetos del tipo DirectoryInfo.

Estos métodos tienen sobrecargas que permiten especificar un modelo de nombres de directorio para filtrar los resultados. Otra sobrecarga también permite especificar si la búsqueda se debe realizar solo en el directorio actual o también en los subdirectorios objeto de una enumeración del tipo System.IO.SearchOption:

```
string[] dirs = Directory.GetDirectories(@"C:\",
                        "m?o",
                        SearchOption.TopDirectoryOnly);
string[] dirs = Directory.GetDirectories(@"C:\",
                        "m*",
                        SearchOption.AllDirectories);
```

El modelo puede contener los caracteres * y ?. El carácter * sustituye a uno o a varios caracteres, mientras que el carácter ? reemplaza a un único carácter.

- La obtención de los archivos contenidos en un directorio se realiza según el mismo modelo que para los subdirectorios con los métodos GetFiles o EnumerateFiles de las clases Directory y DirectoryInfo.

3. File y FileInfo

Las clases File y FileInfo tienen un funcionamiento parecido a las clases Directory y DirectoryInfo en el ámbito de los archivos. La clase File es estática, mientras que la clase FileInfo se debe instanciar. Elegir el uso de una clase u otra depende del número de operaciones que se desea realizar sobre un archivo. Una operación única será más rápida con la clase File, mientras que varias operaciones implicarán el uso de un objeto FileInfo para mejorar el rendimiento.

Estas clases permiten crear, mover y eliminar archivos, así como modificar sus propiedades, además de leer y escribir en los archivos:

```
FileStream fileStream = File.Create(@"C:\archivo.txt");
// Escribir en el archivo
fileStream.Dispose();
FileInfo fileInfo = new FileInfo(@"C:\archivo.txt");
```

```
FileStream fileStream = fileInfo.Create();
// Escribir en el archivo
fileStream.Dispose();
```

La creación de un archivo devuelve un objeto de tipo FileStream asociado. La escritura de datos en este flujo implicará la escritura en el archivo. La llamada al método Create en un archivo ya existente implica el borrado de todos los datos almacenados en él previamente. Para leer o escribir los datos en un archivo existente, hay que utilizar los métodos Open, OpenRead, OpenText u OpenWrite:

```
FileStream fs = File.Open(@"C:\archivo.txt", FileMode.Open);
// Operaciones sobre el archivo
fs.Dispose();

FileInfo fileInfo = new FileInfo(@"C:\archivo.txt");
FileStream fileStream = fileInfo.Open(FileMode.Open);
// Operaciones sobre el archivo
fileStream.Dispose();
```

El argumento de tipo FileMode permite especificar el modo de apertura del archivo. La enumeración contiene los siguientes valores:

- CreateNew: se creará un nuevo archivo. Si el archivo ya existe, se producirá una excepción del tipo IOException.

- Create: se creará un nuevo archivo. Si el archivo ya existe, se sustituirá. Esto equivale a decir que si el archivo no existe, se abre en modo CreateNew y, si existe, se abre en modo Truncate.

- Open: especifica que el archivo solo se debe abrir. Si no existe, se produce una excepción del tipo FileNotFoundException.

- OpenOrCreate: indica que si el archivo existe se debe abrir. En caso contrario, se debe crear.

- Truncate: indica que el archivo se debe abrir y eliminar su contenido. Cuando un archivo se abre con este modo, no se puede leer.

- Append: este modo abre el archivo o lo crea si no existe y el cursor se sitúa al final. El archivo no se podrá leer, ya que se producirá una excepción del tipo NotSupported-Exception.

La lectura y la escritura de datos en los archivos se estudiará en la siguiente sección.

Como ocurre con los directorios, las clases File y FileInfo exponen un miembro Exists en forma de propiedad para la clase FileInfo y en forma de método para la clase File:

```
if (File.Exists(@"C:\archivo.txt"))
```

```
{
    // ...
}

FileInfo fileInfo = new FileInfo(@"C:\archivo.txt");
if (fileInfo.Exists)
{
    // ...
}
```

Las clases File y FileInfo exponen otros métodos que permiten realizar las operaciones más habituales sobre los directorios:

- La eliminación de archivos se realiza mediante el método Delete, presente en las clases File y FileInfo:

```
File.Delete(@"C:\archivo.txt");

FileInfo fileInfo = new FileInfo(@"C:\archivo.txt");
fileInfo.Delete();
```

Si el archivo que se intenta eliminar no existe, no se produce ninguna excepción.

- Para mover o copiar archivos se usan los métodos Copy y Move de la clase File y los métodos CopyTo y MoveTo de la clase FileInfo:

```
File.Copy(@"C:\archivo.txt", @"C:\NuevoArchivo.txt");
File.Move(@"C:\archivo.txt", @"C:\NuevoArchivo.txt");

FileInfo fileInfo = new FileInfo(@"C:\archivo.txt");
fileInfo.CopyTo(@"C:\NuevoArchivo.txt");
fileInfo.MoveTo(@"C:\NuevoArchivo.txt");
```

Cualquier intento de copiar o mover un archivo a una ubicación que contiene un archivo con el mismo nombre producirá una excepción de tipo IOException.

- El cifrado y descifrado de archivos se realiza mediante los métodos Encrypt y Decrypt. El cifrado se hace de tal manera que solo la cuenta que se ha utilizado para cifrar el archivo lo puede descifrar:

```
File.Encrypt(@"C:\archivo.txt");
File.Decrypt(@"C:\archivo.txt");

FileInfo fileInfo = new FileInfo(@"C:\archivo.txt");
fileInfo.Encrypt();
fileInfo.Decrypt();
```

Un archivo cifrado con esos métodos se resalta en el explorador de archivos Windows con su nombre en verde.

4. Path

La clase Path es estática, expone métodos que permiten realizar operaciones sobre la ruta de accesos de los directorios y archivos. Estas operaciones no se realizan físicamente sobre los archivos y directorios:

- ChangeExtensions: este método permite modificar la extensión de un archivo. El primer argumento es la ruta de acceso del archivo y el segundo la nueva extensión:

```
string result = Path.ChangeExtension(@"C:\archivo.txt", "doc");
// result = "C:\archivo.doc"
```

- Combine: este método permite combinar varias rutas de acceso:

```
string result = Path.Combine(@"C:\Directorio1",
                             "Directorio2",
                             "archivo.txt");
// result = "C:\Directorio1\Directorio2\archivo.txt"
```

Es más eficaz usar el método Combine que reunir las diferentes partes de una ruta de acceso especificando el carácter separador, que es diferente de un SO a otro.

- GetDirectoryName: devuelve la ruta de acceso del directorio relativo a la ruta de acceso especificada:

```
string result =
Path.GetDirectoryName(@"C:\Directorio1\archivo.txt");
// result = "C:\Directorio1"
```

- GetExtension: devuelve la extensión del archivo en la ruta de acceso especificada:

```
string result = Path.GetExtension(@"C:\Directorio1\archivo.txt");
// result = ".txt"
```

- GetFileName: devuelve el nombre del archivo con su extensión:

```
string result = Path.GetFileName(@"C:\archivo.txt");
// result = "archivo.txt"
```

- GetFileNameWithoutExtension: devuelve el nombre del archivo sin su extensión:

```
string result =
Path.GetFileNameWithoutExtension(@"C:\archivo.txt");
// result = "archivo"
```

- GetFullPath: devuelve la ruta de acceso absoluta de la ruta de acceso relativa especificada:

```
string result = Path.GetFullPath(@"archivo.txt");
```

La ruta de acceso por defecto es la correspondiente a la ejecución de la aplicación.

- GetInvalidFileNameChars: devuelve un array con los caracteres no permitidos en los nombres de los archivos.

- GetInvalidPathChars: devuelve un array con los caracteres no permitidos en las rutas de acceso.

- GetPathRoot: devuelve el directorio raíz de la ruta de acceso especificada:

```
string result = Path.GetPathRoot(@"C:\Directorio1\archivo.txt");
// result = "C:\"
```

- GetRandomFileName: devuelve un nombre de archivo aleatorio:

```
string result = Path.GetRandomFileName();
// result = "0zyso02u.zgv"
```

- GetTempFileName: crea un archivo temporal vacío con un nombre único y devuelve la ruta de acceso completa del mismo:

```
string result = Path.GetTempFileName();
// result = "C:\Users\Hugon\AppData\Local\Temp\tmp7032.tmp"
```

- GetTempPath: devuelve la ruta de acceso del directorio temporal del sistema:

```
string result = Path.GetTempPath();
// result = "C:\Users\Hugon\AppData\Local\Temp\"
```

- HasExtension: devuelve un valor booleano que indica si la ruta de acceso especificada contiene una extensión de archivo:

```
bool result = Path.HasExtension(@"C:\Directorio1\archivo.txt");
// result = true
bool result = Path.HasExtension(@"C:\Directorio1");
```

```
// result = false
```

- IsPathRooted: devuelve un valor booleano que indica si la ruta de acceso es relativa o absoluta:

```
bool result = Path.IsPathRooted(@"C:\Directorio1");
// result = true
bool result = Path.IsPathRooted(@"Directorio1");
// result = false
```

Trabajar con el sistema de archivos

1. Los objetos Stream

Los objetos Stream se usan para la transferencia de datos, ya sea entre una fuente externa y la aplicación (se trata en este caso de una lectura de datos), o bien entre la aplicación y una fuente externa (se trata entonces de una escritura de datos).

La fuente externa de un flujo puede provenir de un archivo, de una ubicación de memoria e incluso de la red. Dependiendo del origen, se utilizará un objeto u otro. Por ejemplo, para un flujo en memoria, se usará un objeto de tipo System.IO.MemoryStream, mientras que para la transferencia de datos a través de un protocolo de red se usará la clase System.IO.NetworkStream. Cuando trabaje con archivos las clases que usará serán, por un lado, FileStream para escribir y leer datos binarios en particular y para todo tipo de archivo en general. Por otro lado, las clases StreamReader y StreamWriter están diseñadas especialmente para leer y escribir en los archivos de texto.

La ventaja de utilizar un objeto distinto para la transferencia de datos es poder realizar de forma más sencilla el cambio del tipo de la fuente externa. Conservando la separación entre el código de la aplicación y el concepto de fuente de datos particular, el código se puede reutilizar con más facilidad.

2. La clase FileStream

La clase FileStream se utiliza para leer y escribir en archivos binarios. Su constructor y sus sobrecargas pueden tener hasta cuatro argumentos que permiten determinar el archivo, el modo de apertura, el tipo de acceso y el tipo de bloqueo:

```
FileStream stream = new FileStream(@"C:\archivo.dat",
                                   FileMode.OpenOrCreate,
                                   FileAccess.ReadWrite,
                                   FileShare.None);
```

La enumeración FileMode permite especificar la manera en la que se debe abrir el archivo. Los posibles valores se han presentado antes en este capítulo.

La enumeración FileAccess determina los tipos de operación que se podrán realizar sobre el archivo. Los valores son Read para abrir el archivo en modo de solo lectura, Write para abrirlo en modo de solo escritura y ReadWrite para abrirlo en modo de lectura y escritura.

La enumeración FileShare permite definir el tipo de acceso de los demás objetos sobre el archivo. Puede especificar uno o varios valores entre:

- None para rechazar cualquier compartición.
- Read para autorizar la apertura en modo lectura del archivo.
- Write para autorizar la apertura del archivo en modo escritura.
- ReadWrite para autorizar la apertura del archivo en modo lectura y escritura.
- Delete para permitir eliminar el archivo.
- Inheritable para crear un handle del archivo heredado por los procesos hijo.

Los valores de esta enumeración se pueden combinar mediante el operador |:

```
FileStream stream = new FileStream(@"C:\archivo.dat",
                                   FileMode.OpenOrCreate,
                                   FileAccess.ReadWrite,
                                   FileShare.Read |
                                   FileShare.Delete);
```

Los objetos de tipo FileStream también se pueden instanciar a partir de la clase estática File, gracias a los métodos Open, OpenRead y OpenWrite que abren el archivo especificado respectivamente en modo de lectura y escritura, modo de solo lectura y de solo escritura:

```
FileStream stream = File.Open(@"C:\archivo.dat", FileMode.Create);
FileStream stream = File.OpenRead(@"C:\archivo.dat");
FileStream stream = File.OpenWrite(@"C:\archivo.dat");
```

La clase FileInfo contiene métodos idénticos.

Existen dos métodos para leer datos. El primero es el método ReadByte que recibe como primer argumento el byte a partir de la posición actual del archivo y lo transforma en tipo int. Si alcanza el fin del archivo, devuelve el valor -1:

```
int nextByte = stream.ReadByte();
```

El segundo método de lectura de datos es Read. Permite leer un número determinado de bits y almacenarlos en un array de tipo byte. Devuelve el número de bits leídos:

```
byte[] bytes = new byte[10];
int bytesRead = stream.Read(bytes, 0, 10);
```

Para la escritura, la clase FileStream ofrece dos métodos WriteByte y Write que permiten respectivamente escribir un valor binario único o una serie de valores provenientes de un array de tipo byte:

```
stream.WriteByte(50);

bytes = new byte[] { 15, 65, 98, 78, 126 };
stream.Write(bytes, 0, 5);
```

Cuando terminan las operaciones sobre el archivo, hay que cerrarlo siempre. En caso contrario permanece bloqueado y ningún otro proceso puede acceder:

```
stream.Close();
```

La clase FileStream implementa la interfaz IDisposable; también conviene liberar los recursos explícitamente:

```
stream.Dispose();
```

3. Leer un archivo de texto

a. Leer mediante la clase File

La clase estática File expone métodos que permiten leer datos de un archivo en forma binaria o texto, sin tener que instanciar una secuencia.

El método ReadAllText recibe como argumento la ruta de acceso del archivo que se desea leer. Abre este archivo, lee el contenido y lo cierra antes de devolver un objeto string con el contenido del archivo.

```
string content = File.ReadAllText(@"C:\archivo.txt");
```

El método ReadAllText tiene una sobrecarga que permite especificar el tipo de codificación (encode) del contenido de texto, utilizando un argumento de tipo Encoding:

```
string content = File.ReadAllText(@"C:\archivo.txt",
Encoding.ASCII);
```

El método ReadAllLines funciona de la misma manera que el método ReadAllText, excepto que el contenido se devuelve en forma de array de tipo string. Cada elemento se corresponde con una línea del archivo origen:

```
string[] lines = File.ReadAllLines(@"C:\archivo.txt");
```

El método ReadLines devuelve un objeto genérico IEnumerable, donde cada elemento representa una línea del archivo origen:

IEnumerable<string> lines = File.ReadLines(@"C:\archivo.txt");

La clase File expone un método que permite leer datos binarios: el método ReadAllBytes. Recibe como argumento la ruta de acceso del archivo fuente y devuelve un array de tipo byte que representa el contenido del archivo.

byte[] bytes = File.ReadAllBytes(@"C:\archivo.dat");

b. Leer con la clase StreamReader

La clase StreamReader se utiliza para leer archivos de texto. La manera más sencilla de instanciar un nuevo objeto StreamReader es especificar la ruta de acceso del archivo:

StreamReader stream = new StreamReader(@"C:\archivo.txt");

Si no se especifica ningún tipo de codificación (encode), el constructor examina los primeros bits para determinar el tipo de codificación del archivo. Estos bits se conocen como byte code markers. No existen cuando el tipo de codificación del archivo es ASCII por razones de compatibilidad hacia atrás con los sistemas antiguos, que no gestionan Unicode. Cuando el tipo de codificación del archivo es Unicode, UTF7, UTF8 o UTF32 los primeros bits de este archivo se sitúan de manera específica para indicar el formato del archivo.

Existen diversas sobrecargas del constructor de la clase StreamReader que permiten especificar el tipo de codificación mediante un objeto Encoding. En cualquier caso, el tipo de codificación se debe determinar mediante los byte code markers:

```
StreamReader stream = new StreamReader(@"C:\archivo.txt",
                                       Encoding.ASCII);
```

Cuando el archivo se abre, varias posibilidades permiten leer los datos. El objeto StreamReader conserva en memoria la posición del cursor que permite identificar las secciones ya leídas del archivo.

El método ReadLine devuelve un objeto de tipo string correspondiente a la línea del cursor y posiciona en ella la siguiente línea:

string line = stream.ReadLine();

El método Read devuelve el número entero del siguiente carácter después del cursor o -1 si se ha alcanzado el final del archivo:

int nextChar = stream.Read();

La propiedad booleana EndOfStream es otra manera de comprobar si se ha alcanzado el final del archivo:

```
while (!stream.EndOfStream)
{
   string s = stream.ReadLine();
}
```

Una sobrecarga del método Read permite especificar el número de caracteres que se deben leer y los almacena en un array de tipo char:

```
char[] chars = new char[3];
int buffChar = stream.Read(chars, 0, 3);
```

El método ReadToEnd devuelve un objeto de tipo string que contiene todo el archivo entre la posición del cursor y el final del archivo:

```
string all = stream.ReadToEnd();
```

Cuando terminan las operaciones de lectura del archivo, siempre hay que cerrar el archivo. En caso contrario queda bloqueado y ningún otro proceso puede acceder a él:

```
stream.Close();
```

La clase StreamReader implementa la interfaz IDisposable. También es conveniente liberar los recursos explícitamente:

```
stream.Dispose();
```

Tomando como ejemplo un archivo con el siguiente contenido:

```
Contenido
del
archivo
```

Y las siguientes instrucciones:

```
StreamReader stream = new StreamReader(@"C:\archivo.txt",
                                       Encoding.ASCII);
string line = stream.ReadLine();
int nextChar = stream.Read();
char[] chars = new char[3];
int buffChar = stream.Read(chars, 0, 3);
string all = stream.ReadToEnd();
stream.Close();
stream.Dispose();
```

Los valores de las variables después de la ejecución de las instrucciones anteriores son los siguientes:

```
line = "Contenido"
nextChar = 100 es el carácter 'd'
chars[0] = 'e'
chars[1] = 'l'
chars[2] = '\r'
all = "archivo"
```

4. Escribir en un archivo de texto

a. Escribir mediante la clase File

La clase estática File expone métodos que permiten escribir datos en un archivo en forma binaria o de texto, sin tener que instanciar una secuencia.

El método WriteAllText recibe como argumentos la ruta de acceso al archivo y el contenido que se desea escribir. Crea un nuevo archivo, escribe el contenido en dicho archivo, lo guarda y después lo cierra. Si el archivo ya existe, se elimina.

```
File.WriteAllText(@"C:\archivo.txt", "Contenido");
```

El método WriteAllText tiene una sobrecarga que permite especificar el tipo de codificación del contenido de texto, utilizando un argumento de tipo Encoding del espacio de nombres System.Text:

```
File.WriteAllText(@"C:\archivo.txt", "Contenido", Encoding.ASCII);
```

El método WriteAllLines funciona de la misma manera que el método WriteAllText, excepto que el contenido se pasa en un array de tipo string o en una colección genérica IEnumerable. Cada elemento se escribe en una línea del archivo de destino:

```
string[] lines = { "Contenido", "del", "archivo" };
File.WriteAllLines(@"C:\archivo.txt", lines, Encoding.ASCII);

List<string> lines = new List<string>()
                    { "Contenido", "del", "archivo" };
File.WriteAllLines(@"C:\archivo.txt", lines, Encoding.ASCII);
```

La clase File expone un método que permite escribir datos binarios: el método WriteAllBytes. Recibe como argumentos la ruta de acceso del archivo de destino, así como un array de tipo byte que representa el contenido del archivo:

```
byte[] B = { 1, 78, 96, 135 };
File.WriteAllBytes(@"C:\archivo.dat", B);
```

b. Escribir con la clase StreamWriter

La clase StreamWriter se usa para escribir en archivos de texto. La manera más sencilla de instanciar un nuevo objeto StreamWriter es especificar la ruta de acceso del archivo:

```
StreamWriter stream = new StreamWriter(@"C:\archivo.txt");
```

El constructor de la clase StreamWriter tiene sobrecargas que permiten especificar el tipo de codificación mediante un objeto Encoding e indicar si se debe o no añadir el contenido a los datos existentes:

```
StreamWriter stream = new StreamWriter(@"C:\archivo.txt",
```

```
                    false,
                    Encoding.ASCII);
```

Los métodos Write y WriteLine permiten escribir datos en el archivo. Contienen muchas sobrecargas para cada uno de los tipos básicos, como los tipos string, char, char[], bool, int, long u object:

```
stream.WriteLine("Contenido");
stream.Write('d');
stream.Write(new char[] { 'l', '\r', '\n' });
stream.Write("archivo");
```

Cuando terminan las operaciones de escritura en el archivo, siempre hay que cerrar el archivo. En caso contrario queda bloqueado y ningún otro proceso puede acceder a él:

```
stream.Close();
```

La clase StreamWriter implementa la interfaz IDisposable. Conviene liberar los recursos explícitamente:

```
stream.Dispose();
```