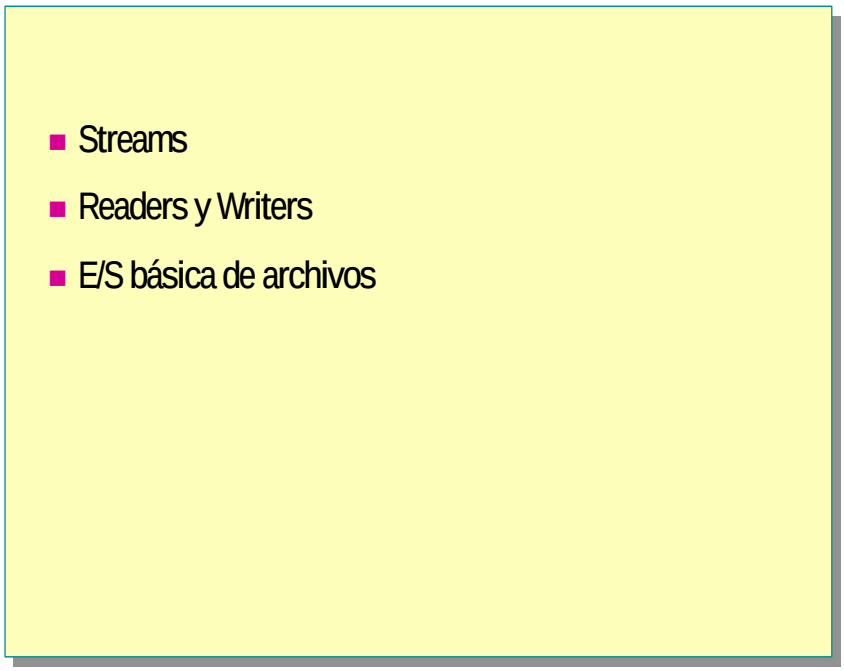
Streams de datos y archivos

Índice

Introducción

Objetivo
Proporcionar una descripción de los temas y objetivos del módulo.

Presentación
En este módulo, estudiaremos cómo utilizar tipos que permiten leer y escribir de/a streams de datos y archivos.



El espacio de nombres **System.IO** contiene tipos que permiten la lectura y la escritura síncrona y asíncrona desde/a streams de datos y archivos. Este módulo trata únicamente sobre operaciones síncronas, ya que las operaciones asíncronas están fuera del alcance de este curso.

En este módulo, aprenderemos a:

Sugerencia
Cuando comente una clase concreta, puede mostrar la información de clase de System.IO desde la sección .NET Framework Reference del SDK .NET Framework.

- Utilizar objetos **Stream** para leer y escribir bytes a repositorios de seguridad, como cadenas y archivos.
- Utilizar objetos **BinaryReader** y **BinaryWriter** para leer y escribir tipos primitivos como valores binarios.
- Utilizar objetos **StreamReader** y **StreamWriter** para leer y escribir caracteres en un stream.
- Utilizar objetos **StringReader** y **StringWriter** para leer y escribir caracteres en cadenas.
- Utilizar objetos **Directory** y **DirectoryInfo** para crear, mover y enumerar a través de directorios y subdirectorios.
- Utilizar objetos **FileSystemWatcher** para monitorizar y reaccionar frente a cambios del sistema de archivos.
- Explicar las principales características del mecanismo de almacenamiento aislado del .NET Framework.

Streams

Objetivo
Hacer una introducción a las funciones de la clase Stream y sus subclases.

Presentación
Los streams proporcionan una forma de leer y escribir bytes desde y hacia un repositorio de seguridad. Un repositorio de seguridad es un medio de almacenamiento, como un disquete o una memoria.

- Stream classes provide a mechanism to read and write bytes from and to a backing store
  - Stream classes inherit from System.IO.Stream
- Fundamental stream operations include Read, Write, and Seek
  - CanRead, CanWrite, and CanSeek properties
- Some streams support buffering for performance
  - Flush method outputs and clears internal buffers
- Close method frees resources
  - Close method performs an implicit Flush for buffered streams
- Stream classes provided by the .NET Framework
  - NetworkStream, BufferedStream, MemoryStream, FileStream
- Null Stream instance has no backing store

Los streams proporcionan una forma de leer y escribir bytes desde y hacia un repositorio de seguridad. Un *repositorio de seguridad* es un medio de almacenamiento, como un disquete o una memoria.

Todas las clases que representan streams heredan de la clase **Stream**. La clase **Stream** y sus subclases proporcionan una vista genérica de fuentes de datos y repositorios, y protegen al programador de los detalles específicos del sistema operativo y de los dispositivos subyacentes.

Principales operaciones de stream

Los streams permiten realizar tres operaciones principales:

1. Podemos leer de streams.

   La lectura es la transferencia de datos desde un stream a una estructura de datos (como una matriz de bytes).

2. Podemos escribir a streams.

   La escritura es la transferencia de datos desde una estructura de datos a un stream.

3. Los streams pueden soportar búsqueda.

   La búsqueda es la consulta y modificación de la posición actual en un stream. La capacidad de búsqueda depende del tipo de repositorio de seguridad que tenga un stream. Por ejemplo: el concepto de posición actual no se aplica a streams de red y, por tanto, típicamente los streams de red no soportan búsqueda.

Dependiendo de la fuente de datos subyacente o repositorio, los streams pueden soportar únicamente algunas de estas capacidades. Una aplicación puede realizar una consulta a un stream sobre sus capacidades utilizando las propiedades **CanRead**, **CanWrite** y **CanSeek**.

Los métodos **Read** y **Write** leen y escriben datos en forma de bytes. Para los streams que soportan búsqueda, los métodos **Seek** y **SetLength** y las propiedades **Position** y **Length** pueden utilizarse para consultar y modificar la posición y longitud actuales de un stream.

Soporte de *buffering*

Algunas implementaciones de streams realizan un proceso de *buffering* local de los datos subyacentes para mejorar el rendimiento. Para estos streams, podemos utilizar el método **Flush** tanto para eliminar buffers internos como para asegurar que todos los datos se han escrito en la fuente de datos subyacente o el repositorio.

Invocar el método **Close** en un stream realiza un *flush* de los datos almacenados en buffer. El método **Close** también libera recursos del sistema operativo, como descriptores de archivos, conexiones a redes, o memoria utilizada para algún proceso de *buffering* interno.

Las clases Stream proporcionadas por el .NET Framework

El .NET Framework contiene varias clases stream que derivan de la clase **System.IO.Stream**. El espacio de nombres **System.Net.Sockets** contiene la clase **NetworkStream**. **NetworkStream** proporciona el stream subyacente de datos para el acceso a redes.

El espacio de nombres **System.IO** contiene las clases **BufferedStream**, **MemoryStream** y **FileStream**, derivadas de la clase **System.IO.Stream**.

Clase BufferedStream

La clase **BufferedStream** se utiliza para invocar el proceso de lectura de buffer desde otro stream, y escritura de buffer a otro stream. Un buffer es un bloque de bytes en memoria que se utiliza para almacenar datos en caché, reduciendo así el número de llamadas al sistema operativo. Los buffers pueden utilizarse para mejorar el rendimiento de la lectura y escritura. Ninguna otra clase puede heredar de la clase **BufferedStream**.

Clase MemoryStream

La clase **MemoryStream** proporciona un método de creación de streams que utiliza la memoria (en lugar de un disquete o una conexión a red) como repositorio de seguridad. La clase **MemoryStream** crea un stream desde una matriz de bytes.

Clase FileStream

La clase **FileStream** se utiliza tanto para leer de archivos como para escribir a ellos. De forma predeterminada, la clase **FileStream** abre archivos síncronamente, pero también proporciona un constructor para abrir archivos asíncronamente.

Instancia de Stream Null

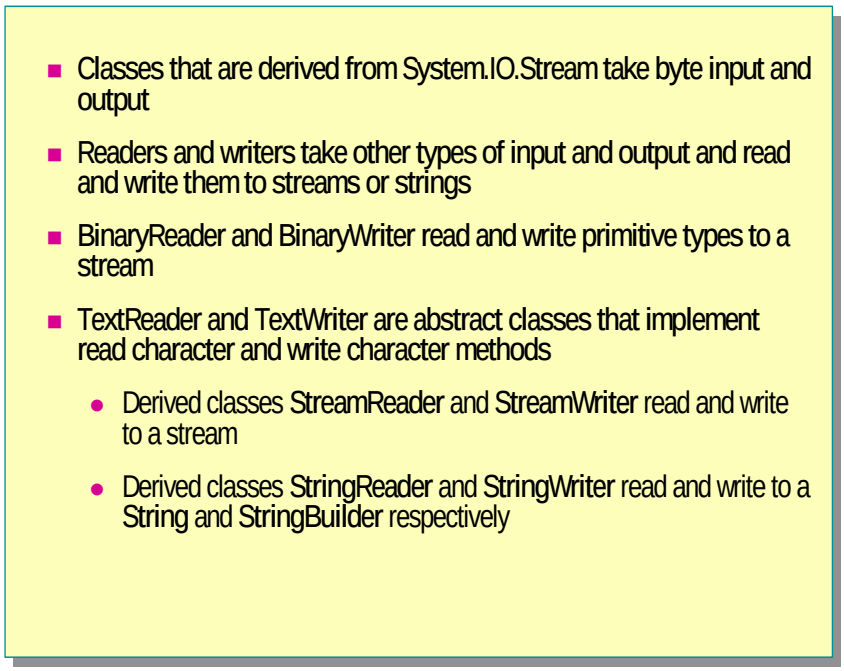
En ocasiones, una aplicación simplemente necesita que un stream elimine su salida y no devuelva ninguna entrada. Podemos obtener este tipo de stream que no tiene repositorio de seguridad y que no consumirá ningún recurso operativo, del campo estático público de la clase **Stream** denominado **Null**.

Por ejemplo, podemos codificar una aplicación para que siempre escriba su salida al **FileStream** especificado por el usuario. Cuando el usuario no desea un archivo de salida, la aplicación dirige su salida al stream **Null**. Cuando se invocan los métodos **Write** de **Stream** en este stream **Null**, la invocación simplemente regresa y no se escribe ningún dato. Cuando se invocan los métodos **Read**, el stream **Null** devuelve 0 sin leer datos.

Readers y Writers

Objetivo
Mostrar cómo las clases reader y writer se utilizan para realizar la entrada y salida de datos a streams y cadenas.

Presentación
Como hemos mencionado antes, la clase Stream está diseñada para extraer e introducir bytes. Podemos utilizar las clases reader y writer tanto para realizar la entrada y salida de datos en streams y cadenas que usan otros tipos.



Como se ha comentado en la sección Streams de este módulo, la clase **Stream** está diseñada para la entrada/salida de bytes. Podemos utilizar las clases reader y writer para extraer e introducir a streams y cadenas que usan otros tipos.

La siguiente tabla describe algunas clases reader y writer utilizadas habitualmente.

Clase	Descripción
BinaryReader y **BinaryWriter**	Estas clases leen y escriben tipos como valores binarios en una codificación específica a y desde un stream.
TextReader y **TextWriter**	Las implementaciones de estas clases están diseñadas para introducir y extraer caracteres.
StreamReader y **StreamWriter**	Estas clases se derivan de las clases **TextReader** y **TextWriter**, que leen sus caracteres desde un stream y escriben sus caracteres a un stream respectivamente.
StringReader y **StringWriter**	Estas clases también derivan de las clases **TextReader** y **TextWriter**, pero leen sus caracteres desde una cadena y escriben sus caracteres a una clase **StringBuilder** respectivamente.

Un reader o writer se adjunta a un stream, de modo que los tipos deseados pueden leerse o escribirse fácilmente.

El siguiente ejemplo muestra cómo escribir datos de tipo **Integer** y leer datos desde un nuevo stream de archivos vacío denominado Test.data. Una vez creado el archivo de datos en el directorio actual, la clase **BinaryWriter** se utiliza para escribir enteros de 0 a 10 en Test.data. A continuación, la clase **BinaryReader** lee el archivo y muestra el contenido del archivo en la consola.

```
Imports System
Imports System.IO

Class MyStream
 Private Const FILE_NAME As String = "Test.data"

 Shared Sub Main()
     ' Create the new, empty data file.
     If (File.Exists(FILE_NAME)) Then
         Console.WriteLine("{0} already exists!", FILE_NAME)
         Exit Sub
     End If

     Dim fs As New FileStream(FILE_NAME, FileMode.CreateNew)
     ' Create the writer for data.
     Dim w As New BinaryWriter(fs)
     ' Write data to Test.data.
     Dim i As Integer
     For i = 0 To 10
         w.Write(i)
     Next
     w.Close()
     fs.Close()

     ' Create the reader for data.
     fs = New FileStream(FILE_NAME, FileMode.Open, _
                         FileAccess.Read)
     Dim r As New BinaryReader(fs)
     ' Read data from Test.data.
     For i = 0 To 10
         Console.WriteLine(r.ReadInt32())
     Next
     w.Close()
 End Sub
End Class
```

En el siguiente ejemplo, el código define una cadena y la convierte en una matriz de caracteres. Esta matriz puede leerse utilizando el método **StringReader.Read** adecuado:

```
Imports System
Imports System.IO

Class CharsFromStr
    Shared Sub Main()
        ' Create a string to read characters from.
        Dim str As String = "Some number of characters"
        ' Size the array to hold all the characters of the
        ' string, so that they are all accessible.
        Dim b(24) As Char
        ' Create a StringReader and attach it to the string.
        Dim sr As New StringReader(str)
        ' Read 13 characters from the array that holds
        ' the string, starting from the first array member.
        sr.Read(b, 0, 13)
        ' Display the output.
        Console.WriteLine(b)
        ' Close the StringReader.
        sr.Close()
    End Sub
End Class
```

El ejemplo anterior produce la siguiente salida:

```
Some number o
```

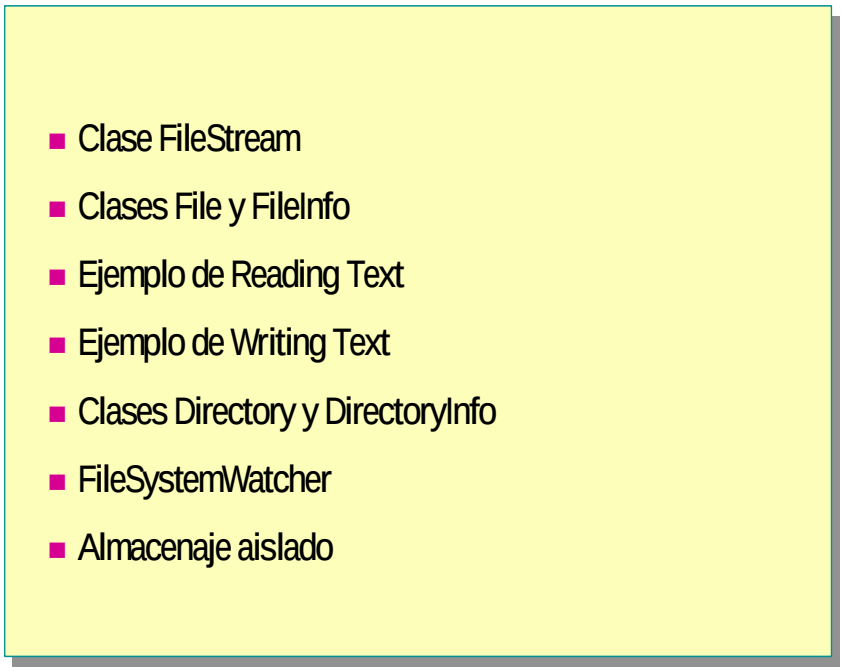
System.Text.Encoding

Internamente, el entorno de ejecución representa todos los caracteres en formato Unicode. Sin embargo, Unicode puede resultar ineficaz en la transferencia de datos a través de una red, o cuando persiste en un archivo. Para mejorar la eficacia, la biblioteca de clases del .NET Framework proporciona varios tipos derivados de la clase base abstracta **System.Text.Encoding**. Estas clases pueden codificar y decodificar caracteres Unicode a ASCII, UTF-7, UTF-8, Unicode y otras páginas de código arbitrarias. Cuando construimos un **BinaryReader**, **BinaryWriter**, **StreamReader** o **StreamWriter**, podemos escoger cualquiera de estas codificaciones. La codificación predeterminada es UTF-8.

E/S básica de archivos

Objetivo
Hacer una introducción a las clases del espacio de nombres System.IO, que se explica en esta sección.

Presentación
El espacio de nombres System.IO del .NET Framework proporciona varias clases útiles para manipular archivos y directorios.



El espacio de nombres **System.IO** del .NET Framework proporciona varias clases útiles para manipular archivos y directorios.

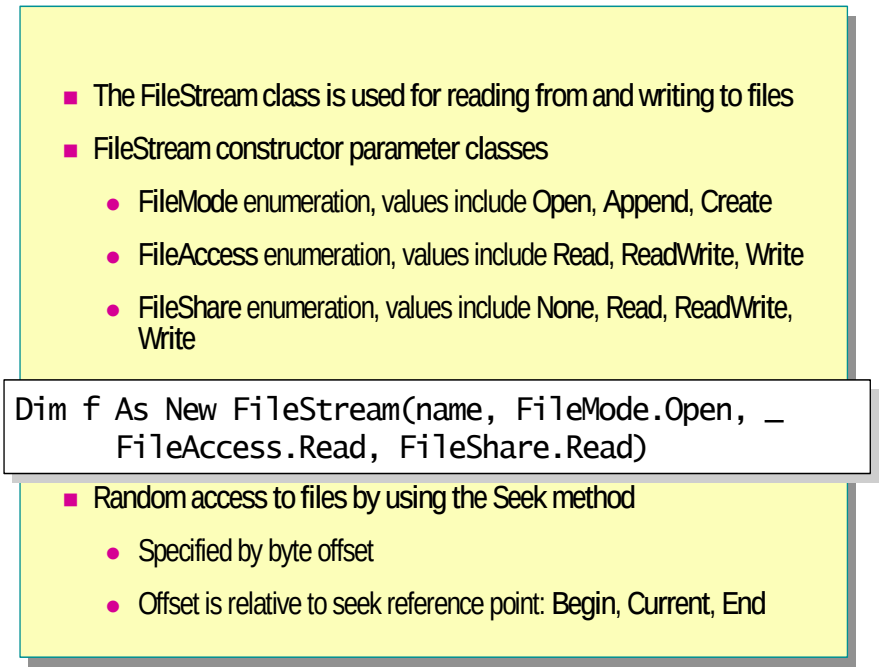
Importante La política de seguridad predeterminada para Internet y las intranets no permite acceder a archivos. Por ello, si escribe código que se descargará desde Internet, no utilice las clases E/S normales de almacenamiento no aislado. Utilice el **almacenamiento aislado**.

Precaución Cuando se abre un stream de archivos o de redes, sólo se realiza una comprobación de seguridad si el stream se construye. Por ello, debe tener cuidado cuando expone estos streams con código o dominios de aplicaciones de menor confianza.

La clase FileStream

Objetivo
Definir la clase FileStream y los tipos que se utilizan como parámetros en algunos constructores FileStream.

Presentación
La clase FileStream se utiliza para leer y escribir de/a archivos. Los tipos FileMode, FileAccess y FileShare se utilizan como parámetros en algunos constructores FileStream.



La clase **FileStream** se utiliza para leer y escribir de/a archivos. Los tipos **FileMode**, **FileAccess** y **FileShare** se utilizan como parámetros en algunos constructores **FileStream**.

Los parámetros FileMode

Los parámetros **FileMode** controlan si un archivo se ha sobrescrito, creado o abierto, o sometido a cualquier combinación de estas operaciones. La siguiente tabla describe constantes que se utilizan con la clase de parámetros **FileMode**.

Constante	Descripción
Open	Esta constante se utiliza para abrir un archivo existente.
Append	Esta constante se utiliza para añadir un archivo existente.
Create	Esta constante se utiliza para crear un archivo si el archivo no existe todavía.

La enumeración FileAccess

La enumeración **FileAccess** define constantes para en acceso en modo lectura, escritura o lectura/escritura a un archivo. Esta enumeración tiene un atributo **FlagsAttribute** que permite una combinación de bits de sus valores miembro. Se especifica un parámetro **FileAccess** en muchos de los constructores para **File**, **FileInfo** y **FileStream**, y en otros constructores de clases en los que es importante controlar el tipo de acceso de los usuarios a un determinado archivo.

La enumeración FileShare

La enumeración **FileShare** contiene constantes para controlar el tipo de acceso que otros objetos **FileStream** pueden tener al mismo archivo. Esta enumeración tiene un atributo **FlagsAttribute** que permite una combinación de bits de sus valores miembro.

La enumeración **FileShare** se utiliza típicamente para definir si varios procesos pueden leer simultáneamente desde el mismo archivo. Por ejemplo, si se abre un archivo y está especificado **FileShare.Read**, otros usuarios podrán abrir el archivo para leerlo pero no para escribir. **FileShare.Write** especifica que otros usuarios pueden escribir simultáneamente en el mismo archivo. **FileShare.None** declina toda compartición del archivo.

En el siguiente ejemplo, un constructor **FileStream** abre un archivo existente para acceder en modo lectura y permite a otros usuarios leer el archivo simultáneamente:

```
Dim f As New FileStream(name, FileMode.Open, _
  FileAccess.Read, FileShare.Read)
```

Uso del método Seek para el acceso aleatorio a archivos

Los objetos **FileStream** soportan el acceso aleatorio a archivos utilizando el método **Seek**. El método **Seek** permite mover la posición de lectura/escritura del stream de archivos a cualquier posición del archivo. La posición de lectura/escritura puede moverse utilizando los parámetros del punto de referencia del *offset* de bytes.

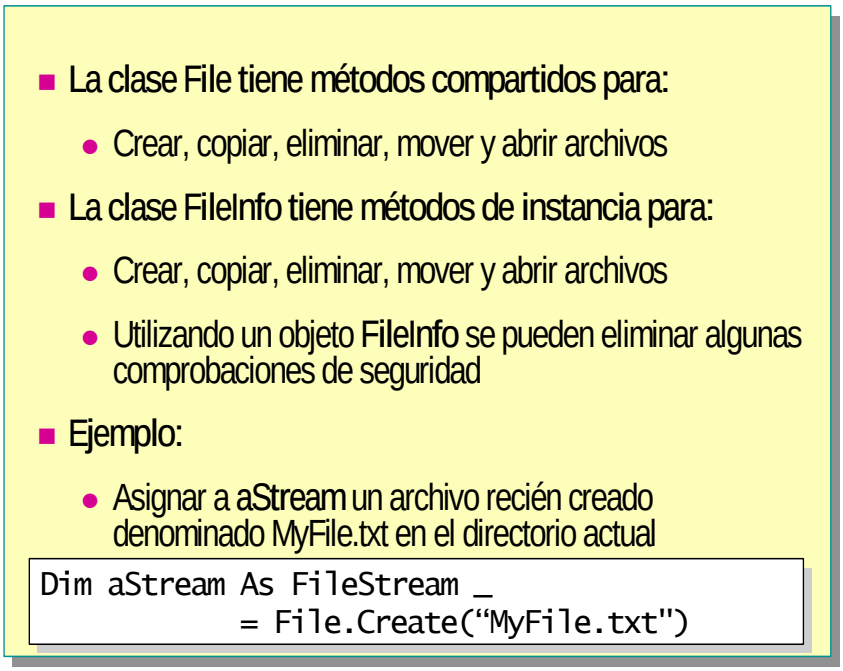
El *offset* de bytes es relativo al punto de referencia de búsqueda, como se representa con las tres propiedades de la clase **SeekOrigin**, descritas en la siguiente tabla:

Nombre de la propiedad	Descripción
Begin	Posición de referencia de búsqueda del principio de un stream
Current	Posición de referencia de búsqueda de la posición actual en un stream
End	Posición de referencia de búsqueda del final de un stream

Las clases File y FileInfo

Objetivo
Presentar las clases File y FileInfo, y mostrar cómo se utilizan para crear un nuevo objeto.

Presentación
Las clases File y FileInfo son clases útiles con métodos que se utilizan principalmente para crear, copiar, eliminar, mover y abrir archivos.



Las clases **File** y **FileInfo** son clases útiles que contienen métodos que se utilizan principalmente para crear, copiar, eliminar, mover y abrir archivos.

Todos los métodos de la clase **File** son compartidos y, por tanto, pueden invocarse sin crear una instancia de la clase. La clase **FileInfo** contiene únicamente métodos de instancia. Los métodos compartidos de la clase **File** realizan comprobaciones de seguridad en todos los métodos. Si vamos a reutilizar un objeto varias veces, pensemos en utilizar en su lugar el método de instancia de **FileInfo** correspondiente. De este modo, se minimizarán el número de comprobaciones de seguridad.

Por ejemplo, para crear un archivo denominado MyFile.txt y devolver un objeto **FileStream**, utilice el siguiente código:

```
Dim aStream As FileStream = File.Create("MyFile.txt")
```

Para crear un archivo denominado MyFile.txt y devolver un objeto **StreamWriter**, utilice el siguiente código:

```
Dim sw As StreamWriter = File.CreateText("MyFile.txt")
```

Para abrir un archivo denominado MyFile.txt y devolver un objeto **StreamReader**, utilice el siguiente código:

```
Dim sr As StreamReader = File.OpenText("MyFile.txt")
```

Ejemplo de lectura de texto

Objetivo
Ofrecer un ejemplo de lectura de texto.

Presentación
En el siguiente ejemplo, leeremos un archivo completo y se nos notificará cuando se detecte el final del archivo.

- Leer texto de un archivo y escribirlo en la consola

```
'...
Dim sr As StreamReader = File.OpenText(FILE_NAME)
Dim Line As String = sr.ReadLine()
While Not(Line Is Nothing)
   Console.WriteLine(Line)
   Line = sr.ReadLine()
End While
Console.WriteLine ( _
   "The end of the stream has been reached.")
sr.Close()
'...
```

En el siguiente ejemplo de lectura de texto, leeremos un archivo completo y se nos notificará cuando se detecte el final del archivo.

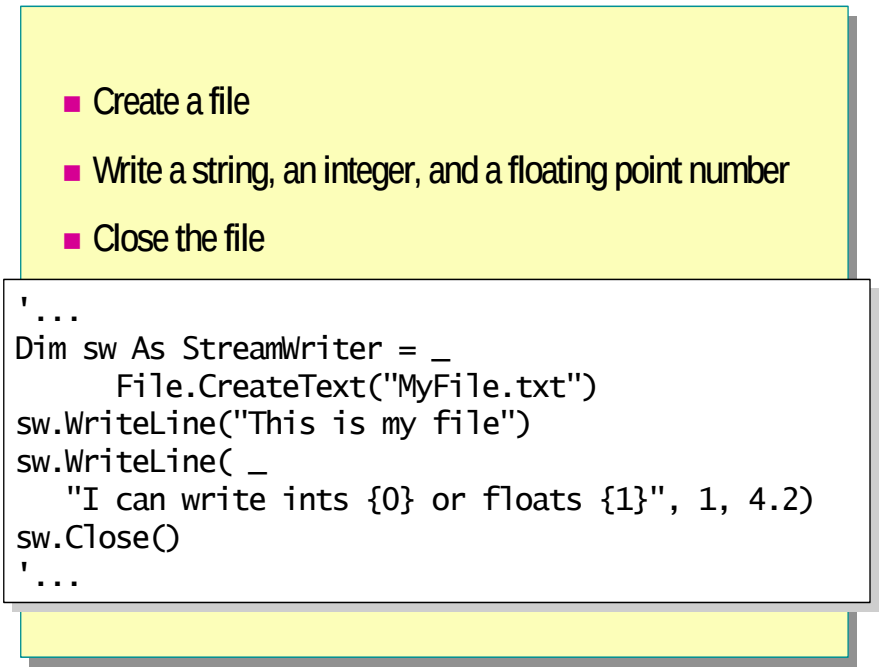
```
Imports System
Imports System.IO

Class TextFromFile
    Private Const FILE_NAME As String = "MyFile.txt"

    Shared Sub Main()
        ' Create the new, empty data file.
        If (File.Exists(FILE_NAME) = False) Then
           Console.WriteLine("{0} does not exist!", FILE_NAME)
           Exit Sub
        End If

        Dim sr As StreamReader = File.OpenText(FILE_NAME)
        Dim Line As String = sr.ReadLine()
        While Not(Line Is Nothing)
            Console.WriteLine(Line)
            Line = sr.ReadLine()
        End While

        Console.WriteLine( _
             "The end of the stream has been reached.")
        sr.Close()
    End Sub
End Class
```

Este código crea un objeto **StreamReader** que apunta a un archivo denominado MyFile.txt utilizando una llamada a **File.OpenText**. **StreamReader.ReadLine** devuelve cada línea como una cadena. Cuando no hay más caracteres para leer, se muestra un mensaje a tal efecto, y el stream se cierra.

Ejemplo de escritura de texto

Objetivo
Ofrecer un ejemplo de escritura de texto.

Presentación
Este ejemplo crea un nuevo archivo de texto denominado MyFile.txt, escribe una cadena, un entero y un número de coma flotante en él y, finalmente, cierra el archivo.



El siguiente ejemplo crea un nuevo archivo de texto denominado MyFile.txt, escribe una cadena, entero y un número de coma flotante en él y, finalmente, cierra el archivo.

```
Imports System
Imports System.IO

Class TextToFile
  Private Const FILE_NAME As String = "MyFile.txt"

  Shared Sub Main()
    If (File.Exists(FILE_NAME)) Then
        Console.WriteLine("{0} already exists!", FILE_NAME)
        Exit Sub
    End If
    Dim sw As StreamWriter = File.CreateText("MyFile.txt")
    sw.WriteLine("This is my file")
    sw.WriteLine( _
     "I can write ints {0} or floats {1}, and so on.", 1, 4.2)
    sw.Close()
  End Sub
End Class
```

Las clases Directory y DirectoryInfo

Objetivo
Explicar cómo las clases Directory y DirectoryInfo se utilizan para crear listados de directorios.

Presentación
Las clases Directory y DirectoryInfo exponen rutinas para crear, mover y enumerar a través de directorios y subdirectorios.

- Directory tiene métodos compartidos para:
  - Crear, mover y enumerar a través de directorios y subdirectorios
- DirectoryInfo tiene métodos de instancia para:
  - Crear, navegar y enumerar a través de directorios y subdirectorios
  - Poder eliminar algunas comprobaciones de seguridad cuando se reutiliza un objeto
- Ejemplo:
  - Enumerar a través del dirctorio actual

```
Dim dir As New DirectoryInfo(".")
Dim f As FileInfo, name As String
For Each f in dir.GetFiles("*.vb")
    name = f.FullName
Next
```

- Utilizar objetos de la clase Path para procesar cadenas de directorio

Las clases **Directory** y **DirectoryInfo** contienen rutinas para crear, mover y enumerar a través de directorios y subdirectorios. Todos los métodos de la clase **Directory** son compartidos y, por tanto, pueden invocarse sin crear una instancia de un directorio. La clase **DirectoryInfo** contiene todos los métodos de instancia. Los métodos compartidos de la clase **Directory** realizan una comprobación de seguridad en todos los métodos. Si vamos a volver a utilizar un objeto varias veces, pensemos en utilizar en su lugar el método de instancia de **DirectoryInfo** correspondiente. De este modo, se minimizará el número de comprobaciones de seguridad.

El siguiente ejemplo muestra el uso de la clase **DirectoryInfo** para crear un listado de un directorio:

```
Imports System
Imports System.IO

Class DirectoryLister
  Shared Sub Main()
    Dim dir As New DirectoryInfo(".")
    Dim f As FileInfo
    Dim name As String, size As Long, creationTime As DateTime
    For Each f In dir.GetFiles("*.vb")
        name = f.FullName
        size = f.Length
        creationTime = f.CreationTime
        Console.WriteLine("{0,-12:N0} {1,-20:g} {2}", _
                size, creationTime, name)
    Next
  End Sub
End Class
```

En el ejemplo anterior, el objeto **DirectoryInfo** es el directorio actual, indicado por ("."). El código lista los nombres de todos los archivos del directorio actual con extensión .vb, junto con su tamaño de archivo y fecha de creación.

Si hay archivos .vb en el subdirectorio \Bin de la unidad C, la salida de este código es la siguiente:

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

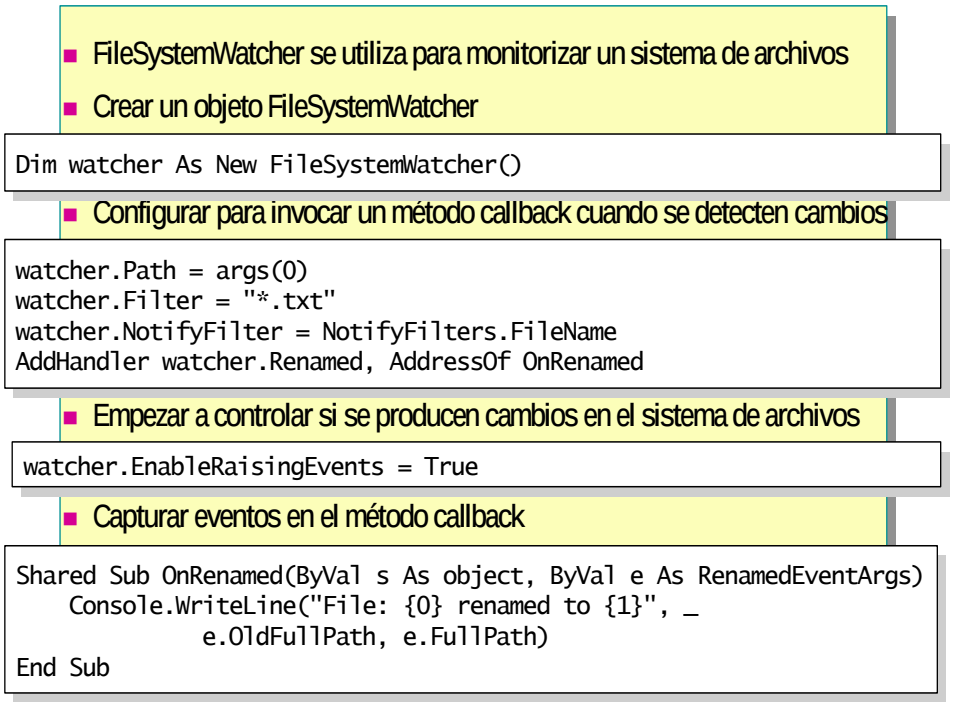
Rutas

Utilice la clase **Path** para procesar cadenas de directorios de modo multiplataforma. Los miembros de la clase **Path** permiten realizar rápida y fácilmente las operaciones más habituales, como averiguar si una extensión de archivo forma parte de una ruta, y combinar dos cadenas en un nombre de ruta.

FileSystemWatcher

Objetivo
Explicar cómo puede utilizarse el componente FileSystemWatcher para monitorizar y responder a cambios en un sistema de archivos.

Presentación
Utilizamos el componente FileSystemWatcher para monitorizar un sistema de archivos y para reaccionar cuando se producen cambios en ese sistema de archivos.



Utilizamos el componente **FileSystemWatcher** para monitorizar un sistema de archivos y para reaccionar cuando se producen cambios en dicho sistema. Utilizando el componente **FileSystemWatcher**, podemos lanzar rápida y fácilmente procesos de negocio cuando se creen, modifiquen o eliminen directorios o archivos específicos.

Por ejemplo, si un grupo de usuarios está colaborando en el mismo documento y ese documento está almacenado en el directorio compartido de un servidor, podemos utilizar fácilmente el componente **FileSystemWatcher** para programar su aplicación para monitorizar cambios. Cuando se detecte un cambio, el componente puede ejecutar un proceso que avise a cada usuario a través del correo electrónico.

Podemos configurar el componente para monitorizar un directorio completo y su contenido o para buscar únicamente un archivo específico o un conjunto de archivos de un determinado directorio. Para monitorizar cambios en todos los archivos, estableceremos la propiedad **Filter** en una cadena vacía (""). Para monitorizar cambios en un archivo específico, estableceremos la propiedad **Filter** en el nombre del archivo. Por ejemplo, para monitorizar cambios en el archivo MyDoc.txt, estableceremos la propiedad **Filter** en "MyDoc.txt". También podemos monitorizar cambios en todos los archivos de un determinado tipo. Por ejemplo, para monitorizar cambios en todos los archivos de texto, estableceremos la propiedad **Filter** en "*.txt".

Nota Los archivos ocultos no se ignoran.

Existen varios tipos de cambios que podemos monitorizar en un directorio o en un archivo. Por ejemplo, podemos buscar cambios en atributos (**Attributes)**, en

la fecha y hora (**LastWrite)**, o en el tamaño (**Size**) de archivos o directorios. Para ello, estableceremos la propiedad **FileSystemWatcher.NotifyFilter** en uno de los valores **NotifyFilters**. Si deseamos obtener más información sobre los tipos de cambios que podemos monitorizar, consultar **NotifyFilters** en el kit de desarrollo de software (SDK) del .NET Framework.

Podemos monitorizar el cambio de nombre o la eliminación y creación de archivos o directorios. Por ejemplo, para monitorizar el cambio de nombre de archivos de texto, estableceremos la propiedad **Filter** en "*.txt" e invocaremos uno de los métodos **WaitForChanged** con el valor **WatcherChangeTypes** en **Renamed**.

Creación de un componente FileSystemWatcher

El siguiente ejemplo crea un componente **FileSystemWatcher** para monitorizar el directorio que se especifica en tiempo de ejecución. El componente se establece para monitorizar cambios en los instantes **LastWrite** y **LastAccess**, y la creación, borrado o renombrado de los archivos de texto del directorio. Si se cambia, crea o borra un archivo, se imprime la ruta al archivo en la consola. Cuando se cambia el nombre de un archivo, se imprimen las rutas antigua y nueva en la consola.

```
Imports System
Imports System.IO
Imports Microsoft.VisualBasic

Class Watcher

    Shared Sub Main()
        ' If a directory is not specified, exit program.
        Dim args() As String = Split(Command())
        If Command().Length = 0 Or args.Length<>1 Then
           ' Display the proper way to call the program.
           Console.WriteLine("Usage: Watcher.exe (directory)")
           Exit Sub
        End If

        ' Create a new FileSystemWatcher
        ' and set its properties.
        Dim watcher As New FileSystemWatcher()
        watcher.Path = args(0)

        ' Watch for changes in LastAccess and LastWrite
        ' times, and the renaming of files or directories
        watcher.NotifyFilter = NotifyFilters.LastAccess + _
         NotifyFilters.LastWrite + NotifyFilters.FileName + _
         NotifyFilters.DirectoryName

        ' Only watch text files.
        watcher.Filter = "*.txt"

        ' Add event handlers.

        ' The Changed event occurs when changes are made to
        ' the size, system attributes, last write time, last
        ' access time, or security permissions in a file or
        ' directory in the specified Path of a
        ' FileSystemWatcher.
        AddHandler watcher.Changed, AddressOf OnChanged
```

(continuación del código en la página siguiente)

```
            ' The Created event occurs when a file or directory
            ' in the specified Path of a FileSystemWatcher is
            ' created.
            AddHandler watcher.Created, AddressOf OnChanged

            ' The Deleted event occurs when a file or directory
            ' in the specified Path of a FileSystemWatcher is
            ' deleted.
            AddHandler watcher.Deleted, AddressOf OnChanged

            ' The Renamed event occurs when a file or directory
            ' in the specified Path of a FileSystemWatcher is
            ' renamed.
            AddHandler watcher.Renamed, AddressOf OnRenamed

            ' Begin watching.
            watcher.EnableRaisingEvents = True

            ' Wait for the user to quit the program.
            Console.WriteLine("Press 'q' to quit the sample.")
            While (Console.ReadLine() <> "q")
                'do nothing
            End While
        End Sub

        ' Define the event handlers.
        Public Shared Sub OnChanged(ByVal sender As Object, _
                                    ByVal e As FileSystemEventArgs)
            ' Specify what is done when a file is changed,
            '  created, or deleted.
            Console.WriteLine("File: " & e.FullPath & " " _
                              & e.ChangeType)
        End Sub

        Public Shared Sub OnRenamed(ByVal sender As Object, _
                                    ByVal e As RenamedEventArgs)
            ' Specify what is done when a file is renamed.
            Console.WriteLine("File: {0} renamed to {1}", _
                              e.OldFullPath, e.FullPath)
        End Sub
    End Class
```

Almacenamiento aislado

Objetivo
Presentar el almacenamiento aislado y sus posibles usos.

Presentación
Para algunas aplicaciones, como aplicaciones descargadas de la Web y código que puede provenir de fuentes sin confianza, el sistema de archivos básico no ofrece el aislamiento y la seguridad necesarios.

- El almacenamiento aislado ofrece formas estandarizadas de asociar aplicaciones a datos guardados
- Las aplicaciones Web parcialmente confiadas requieren:
  - Aislamiento de sus datos de los datos de otras aplicaciones
  - Acceso seguro al sistema de archivos de un equipo
- El espacio de nombres System.IO.IsolatedStorage contiene las siguientes clases

```
NotInheritable Public Class IsolatedStorageFile
    Inherits IsolatedStorage
    Implements IDisposable
```

```
Public Class IsolatedStorageFileStream
    Inherits FileStream
```

La funcionalidad E/S básica de archivos, que se encuentra en la raíz **System.IO**, ofrece la capacidad de acceder, almacenar y manipular datos almacenados en sistemas de archivos jerárquicos cuyos archivos están referenciados mediante rutas exclusivas. Para algunas aplicaciones, como las aplicaciones descargadas de la Web y código que puede provenir de fuentes sin confianza, el sistema básico de archivos no ofrece el aislamiento y seguridad necesarios. El *almacenamiento aislado* es un mecanismo de almacenamiento de datos que proporciona aislamiento y seguridad definiendo formas estandarizadas de asociar código a los datos almacenados.

Aislamiento

Cuando una aplicación almacena datos en un archivo, debe elegirse cuidadosamente tanto el nombre del archivo como la ubicación del almacenamiento para minimizar la posibilidad de que la ubicación del almacenamiento sea conocida por otra aplicación y, por tanto, ser vulnerable a la corrupción. El almacenamiento aislado ofrece los medios para gestionar archivos de aplicaciones descargadas de la Web y minimizar posibles conflictos de almacenamiento.

Riesgos de seguridad de código de semi-confianza

Es importante limitar el acceso del código de semi-confianza al sistema de archivos de un equipo de un modo no restringido. Permitir que el código descargado y ejecutado desde Internet pueda acceder a funciones E/S hace que el sistema sea vulnerable a virus y a daños involuntarios.

En ocasiones, los riesgos de seguridad asociados al acceso a archivos se abordan utilizando listas de control de acceso (*access control lists,* ACLs), las

cuales restringen el acceso de los usuarios a los archivos. Sin embargo, este planteamiento a menudo no es factible con aplicaciones Web, ya que requiere que los administradores configuren ACLs en todos los sistemas en que se ejecutará cada aplicación.

Seguridad mediante el almacenamiento aislado

Cuando los administradores configuran el espacio para el almacenamiento de archivos, establecen políticas de seguridad y eliminan datos no utilizados, pueden utilizar herramientas diseñadas para manipular el almacenamiento aislado. Con el almacenamiento aislado, el código ya no necesita inventar rutas exclusivas para especificar ubicaciones seguras en el sistema de archivos, y los datos se protegen del acceso no autorizado. No es necesario codificar explícitamente información que indique dónde se encuentra el área de almacenamiento de una aplicación. Con el almacenamiento aislado, las aplicaciones con confianza parcial pueden almacenar datos de un modo controlado por la política de seguridad del equipo. Las políticas de seguridad raramente conceden permisos para utilizar mecanismos de E/S estándares y acceder al sistema de archivos. Sin embargo, de forma predeterminada, el código que se ejecuta desde un equipo local, una red local o Internet tiene el derecho de utilizar almacenamiento aislado. Las aplicaciones Web también pueden utilizar el almacenamiento aislado con perfiles de usuarios móviles, permitiendo así que los almacenes aislados de un usuario sigan al perfil de ese usuario.

El espacio de nombres **System.IO.IsolatedStorage** contiene las clases **IsolatedStorageFile** y **IsolatedStorageFileStream**, que las aplicaciones pueden utilizar para acceder a archivos y directorios en su área de almacenamiento aislado.