Capítulo 5
Manipulando o sistema de arquivos

Introdução

A maioria das aplicações que existem atualmente sempre necessitam de, alguma forma, manipular arquivos que estão em algum local do disco. Sejam arquivos de configurações, texto, XML, etc.. Geralmente as aplicações também utilizam o sistema de arquivos para persistir alguns objetos para mais tarde por recuperá-los e, para isso, podemos utilizar o sistema de arquivos.

Além disso, há uma série de instituições, mais precisamente os bancos, utilizam arquivos para a comunicação entre o banco e os clientes. Com isso, o cliente precisa tanto gerar o arquivo para quando quiser enviar algo ao banco como ter a possibilidade de ler e interpretar o arquivo quando o banco disponibilizar o retorno.

A finalidade deste capítulo é mostrar como utilizar as classes contidas dentro do .NET Framework para a manipulação de arquivos. Inicialmente falaremos sobre como recuperar as informações a nível de drive, arquivo e diretórios. Vamos verificar como proceder manipular um caminho até um diretório ou arquivo. Para finalizar esta primeira parte, iremos por em funcionamento o objeto que a Microsoft disponibilizou para monitorar um determinado diretório.

Já na segunda parte, vamos aprender como ler e salvar os dados a partir de *streams*. Além disso, veremos as alternativas, agora embutidas, que permitem comprimir e descomprimir um arquivo. Finalmente, vamos abordar o uso de um objeto que existe desde a primeira versão do .NET Framework: a **StringBuilder**. Essa classe é responsável por manipular uma *string* de forma bem eficiente, principalmente quando se diz respeito a concatenação de *strings*.

Extraindo informações do sistema de arquivos a partir de classes fornecidas pelo .NET Framework

O .NET Framework disponibiliza um *namespace* chamado **System.IO** que contém todas as classes necessárias para a manipulação de arquivos e diretórios. Algumas classes também fornecem suporte assíncrono, o que permite ler dados de um arquivo sem a necessidade de "congelar" a tela que o usuário está utilizando. Veremos a partir de agora três classes interessantes que são utilizadas para trabalhar com todas as espécies de objetos que temos a nível de sistema de arquivos: Drive, Diretório e Arquivo.

Drivers

A partir do .NET 2.0 temos uma classe chamada **DriveInfo**. Essa classe basicamente acessa informações de um determinado drive. Essa classe possui vários métodos e propriedades para manipular um determinado drive do computador. Além desses

membros de instância, temos um método estático chamado *GetDrives* que retorna um *array* de objetos do tipo **DriveInfo** onde, cada um, representa um drive do computador onde a aplicação está sendo executada. Entre as propriedades da classe **DriveInfo**, temos:

Propriedade	**Descrição**
AvaliableFreeSpace	Indica a quantidade de espaço disponível no disco.
DriveFormat	Especifica o sistema de arquivo, como NTFS ou FAT32
DriveType	Indica o tipo de drive, pode ser: • CDRom • Fixed • Unknow • Network • NoRootDirectory • Ram • Removable
IsReady	Retorna um valor booleano indicando se o drive está pronto para ser acessado/lido.
Name	Nome do drive.
RootDirectory	Retorna o diretório raiz do drive.
TotalFreeSpace	Indica o total de espaço disponível no drive.
TotalSize	Retorna o total de espaço que o drive possui.
VolumeLabel	Recupera o rótulo do volume do drive.

Para exemplificar o uso desta classe, podemos fazer como é mostrado através do código abaixo:

VB.NET
```
Imports System.IO

For Each d As DriveInfo In DriveInfo.GetDrivers()
    If d.IsReady Then
        Console.WriteLine(d.DriveType.ToString())
        Console.WriteLine(d.Name)
    End If
Next
```

C#
```
using System.IO;

foreach (DriveInfo d in DriveInfo.GetDrives())
{
    if (d.IsReady)
    {
        Console.WriteLine(d.DriveType.ToString());
        Console.WriteLine(d.Name);
```

```
    }
}
```

Diretórios

Assim como a classe **DriveInfo**, também temos uma classe estática chamada **Directory** que permite manipularmos os diretórios que existem na máquina em que a aplicação está sendo executada ou de algum outro local, dependendo da autorização. Esta classe expõe métodos estáticos para criar, mover, renomear e excluir diretórios. Além disso, permite recuperar todos os seus subdiretórios e arquivos para manipulá-los ou exibí-los.

Ainda há a classe **DirectoryInfo** que também permite executar as mesmas operações que a classe **Directory** fornece, mas agora para um diretório específico. Se você precisar invocar várias propriedades e/ou métodos relacionados a um diretório, considere o uso desta classe em relação a classe **Directory** porque a checagem de segurança não será sempre necessária. Caso o teu cenário não seja esse, então opte por invocar os métodos estáticos da classe **Directory**. Entre as propriedades disponíveis pela classe **DirectoryInfo**, podemos citar as principais:

Propriedade	Descrição
CreationTime	Retorna a data de criação do diretório.
FullName	Retorna o caminho completo do diretório, desde a sua raiz.
Name	Retorna o nome do diretório.
Parent	Retorna o diretório que é "pai" do diretório corrente.
Root	Retorna o nível mais alto onde, na maioria dos casos, é o drive.

Para citar alguns exemplos do uso desta classes, veremos abaixo como criar, ler o conteúdo e apagar um determinado diretório:

```
VB.NET
Imports System.IO

Dim path As String = "c:\Temp\ArquivosProcessados"
Dim dir As DirectoryInfo = Nothing

If Not Directory.Exists(path) Then
    dir = Directory.CreateDirectory(path)
End If

If Not IsNothing(dir) Then
    'Sub-diretórios
    For Each subDir As DirectoryInfo in dir.GetDirectories()
        Console.WriteLine(subDir.Name)
    Next
```

```
    'Arquivos
    For Each fileName As FileInfo in dir.GetFiles()
        Console.WriteLine(filename.Name)
    Next

    dir.Delete(True)
End If
```

C#

```
using System.IO;

string path = @"c:\Temp\ArquivosProcessados";
DirectoryInfo dir = null;

if (!Directory.Exists(path))
{
    dir = Directory.CreateDirectory(path);
}

if (dir != null)
{
    //Sub-diretórios
    foreach (DirectoryInfo subDir in dir.GetDirectories())
        Console.WriteLine(subDir.Name);

    //Arquivos
    foreach (FileInfo fileName in dir.GetFiles())
        Console.WriteLine(fileName.Name);

    dir.Delete(true);
}
```

Como podemos ver, criamos o diretório a partir do método estático **CreateDirectory** da classe **Directory**. Depois de criado, recuperamos a instância do tipo **DirectoryInfo** que traz as informações a respeito do diretório recém criado. De posse desta instância, podemos ter acesso a diversas propriedades e métodos para manipular o diretório. Se analisarmos o código acima, utilizarmos dois métodos: *GetDirectories* e *GetFiles*. O primeiro deles retorna um *array* de *DirectoryInfo* contendo todos os sub-diretórios de um diretório qualquer. Já o segundo também retorna um *array* de *FileInfo* contendo todos os arquivos de um determinado diretório. Esse método ainda possui um *overload* que permite informar qual tipo de arquivos que desejamos recuperar., como por exemplo: se desejarmos apenas recuperar os arquivos texto, então poderíamos fazer: *dir.GetFiles("*.txt")*. Como esses métodos são invocados a partir de uma instância do tipo **DirectoryInfo**, todas as informações são pertinentes ao diretório nele informado. Finalmente chamamos o método *Delete* para excluir o diretório. O parâmetro *True* que passamos para ele, indica que será uma exclusão recursiva, ou seja, tudo que estiver dentro dele, também será excluído.

Nota: Quando invocamos os métodos estáticos *GetDirectories* e *GetFiles* da classe **Directory**, é retornado um *array* de string contendo os nomes dos diretórios e arquivos, respectivamente.

Arquivos

De forma parecida aos anteriores, temos também duas classes para manipulação de arquivos: **File** e **FileInfo**. A classe **File** fornece métodos estáticos para as operações mais típicas com arquivos, como por exemplo, criação, cópia, exclusão e a movimentação de arquivos.

Já a classe **FileInfo** traz informações completas de um determinado arquivo, além também, de fornecer as operações típicas e, como já era de se esperar, operando apenas com o arquivo que ela representa. Assim como a classe **DirectoryInfo**, utilize-a somente se precisar efetuar várias operações em cima deste arquivo. Do contrário, opte pela uso da classe **File** que, para uma única operação, é mais performática. Entre as propriedades disponíveis pela classe **FileInfo**, podemos citar as principais:

Propriedade	**Descrição**
CreationTime	Retorna a data de criação do arquivo.
Directory	Retorna uma instância da classe **DirectoryInfo** que representa o diretório em que o arquivo está contido.
DirectoryName	Retorna o caminho completo até o diretório onde o arquifo está contido.
Extension	Retorna a extensão do arquivo. Se o arquivo chamar "Arquivo.txt" essa propriedade retornará ".txt".
FullName	Resgata o caminho completo do arquivo.
IsReadOnly	Retorna um valor booleano indicando se o arquivo é ou não de somente leitura.
Length	Retorna o tamanho do arquivo.
Name	Recupera o nome do arquivo.

Para manipular os arquivos do disco utilizando essas classes, podemos analisar o código abaixo que explifica o que falamos acima com um trecho de código:

```
VB.NET
Imports System.IO

Dim fileName As String = "Arquivo.txt"
If Not File.Exists(fileName) Then
    File.Create(fileName)
    Dim info As New FileInfo(fileName)
    Console.WriteLine(info.CreationTime)
```

```
    File.Delete(fileName)
End If
```

C#

```
using System.IO;

string fileName = "Arquivo.txt";
if (!File.Exists(fileName))
{
    File.Create(fileName);
    FileInfo info = new FileInfo(fileName);
    Console.WriteLine(info.CreationTime);

    File.Delete(fileName);
}
```

Manipulando caminhos (paths)

Muitas vezes precisamos manipular os caminhos físicos que temos para atender uma determinada necessidade. Para isso, sempre manipulamos a *string* que contém o caminho completo, até chegarmos no valor que desejamos.

Como isso é muito trabalhoso e pode gerar alguns erros, a Microsoft decidiu criar uma classe estática chamada **Path**. Essa classe encapsula todo o trabalho para manipular caminhos de arquivos e diretórios. Além disso, as operações são executadas para suportar multi-plataforma, ou seja, como isso pode variar de sistema operacional para sistema operacional, a classe **Path** se encarrega de retornar o valor correto baseando-se na plataforma que a aplicação está sendo executada.

Para citar alguns métodos, vamos nos restringir aos mais populares e, para uma melhor descrição e exemplos, consulte a documentação do Visual Studio .NET ou, se desejar, pode consultar online: http://www.msdn.com/library.

Método	Descrição
ChangeExtension	Dado um arquivo com sua extensão e uma nova extensão, esse método retorna uma nova string contendo o mesmo arquivo (com seu caminho), mas agora com a nova extensão informada.
Combine	Permite combinar dois caminhos, fazendo todo o trabalho para manipulação dos separadores.
GetDirectoryName	Dado um caminho, ele retorna uma string contendo apenas o caminho até o último diretório.
GetExtension	Dado um nome de arquivo junto com a sua extensão, ele retorna apenas a extensão do arquivo. Exemplo: "Arquivo.txt" retorna ".txt".

GetFileName	Dado um caminho completo até um determinado arquivo, retorna apenas o nome do arquivo junto com a sua extensão.
GetFileNameWithoutExtension	Dado um caminho completo até um determinado arquivo, retorna apenas o nome do arquivo, sem a extensão.
GetRandomFileName	Retorna um nome de arquivo aleatório, podendo inclusive ser utilizado como nome para diretórios. Este método não cria o arquivo/pasta fisicamente.
GetTempFileName	Cria fisicamente um arquivo vazio com um nome único e de extensão .TMP, que é retornado pelo método.
GetTempPath	Retorna o caminho até o diretório de sistema que é a pasta temporária. Geralmente o caminho é: C:\Documents and Settings\<Usuario>\Local Settings\Temp\
HasExtension	Retorna um valor booleano indicando se o caminho possui ou não uma extensão de arquivo.

Monitoramento de diretórios

O *namespace* **System.IO** fornece uma classe um tanto quanto interessante. Ela chama-se **FileSystemWatcher** que tem a finalidade de monitorar um determinado diretório e, qualquer mudança que nele aconteça, a classe é capaz de notificar a aplicação e, conseqüentemente, você pode tratar isso da forma que desejar. Essa classe recebe em seu construtor o caminho que o *watcher* irá monitorar.

Entre os eventos que esta classe fornece temos: *Created, Changed*, *Deleted* e *Renamed*. O primeiro deles é invocado pelo runtime quando um novo arquivo ou diretório é criado. O segundo é disparado quando um arquivo ou diretório é alterado; já o evento *Deleted* é disparado quando algum diretório ou arquivo é excluído e, finalmente, o evento *Renamed*, é disparado quando algum diretório ou arquivo é renomeado. Uma propriedade também muito importante é a propriedade *EnableRaisingEvents*. Essas propriedade recebe um valor booleano indicando se o *watcher* está ou não habilitado. Enquanto essa propriedade estiver definida como *False*, os eventos não serão disparados.

Para exemplificar o uso desta classe, criaremos um *wrapper* para o mesmo que encapsulará todo o trabalho que o **FileSystemWatcher** irá desempenhar dentro da aplicação. Esse criação deste *wrapper* é mostrado abaixo:

```
VB.NET
Imports System.IO

Public Class FileWatcher
```

```
Implements IDisposable

Private _watcher As FileSystemWatcher

Public Sub New(ByVal path As String)
    Me._watcher = New FileSystemWatcher(path)

    Me.InitializeEvents()
    Me._watcher.EnableRaisingEvents = True
End Sub

Private Sub InitializeEvents()
    AddHandler Me._watcher.Created, _
        AddressOf Me._watcher_Created
    AddHandler Me._watcher.Changed, _
        AddressOf Me._watcher_Changed
    AddHandler Me._watcher.Deleted, _
        AddressOf Me._watcher_Deleted
    AddHandler Me._watcher.Renamed, _
        AddressOf Me._watcher_Renamed
End Sub

Private Sub _watcher_Created(ByVal sender As Object, _
    ByVal e As FileSystemEventArgs)
    Me.Show(e)
End Sub

Private Sub _watcher_Renamed(ByVal sender As Object, _
    ByVal e As RenamedEventArgs)
    Me.Show(e)
End Sub

Private Sub _watcher_Deleted(ByVal sender As Object, _
ByVal e As FileSystemEventArgs)
    Me.Show(e)
End Sub

Private Sub _watcher_Changed(ByVal sender As Object, _
    ByVal e As FileSystemEventArgs)
    Me.Show(e)
End Sub

Private Sub Show(ByVal e As FileSystemEventArgs)
    Console.WriteLine(e.ChangeType.ToString())
    Console.WriteLine(e.FullPath)
    Console.WriteLine("---------------------")
End Sub

Public Sub Dispose() Implements IDisposable.Dispose
```

```
        If Not IsNothing(Me._watcher) Then
            Me._watcher.Dispose()
        End If
    End Sub

End Class
```

C#

```
using System.IO;

public class FileWatcher : IDisposable
{
    private FileSystemWatcher _watcher;

    public FileWatcher(string path)
    {
        this._watcher = new FileSystemWatcher(path);

        this.InitializeEvents();
        this._watcher.EnableRaisingEvents = true;
    }

    private void InitializeEvents()
    {
        this._watcher.Created +=
            new FileSystemEventHandler(_watcher_Created);
        this._watcher.Changed +=
            new FileSystemEventHandler(_watcher_Changed);
        this._watcher.Deleted +=
            new FileSystemEventHandler(_watcher_Deleted);
        this._watcher.Renamed +=
            new RenamedEventHandler(_watcher_Renamed);
    }

    void _watcher_Created(object sender, FileSystemEventArgs e)
    {
        this.Show(e);
    }

    void _watcher_Renamed(object sender, RenamedEventArgs e)
    {
        this.Show(e);
    }

    void _watcher_Deleted(object sender, FileSystemEventArgs e)
    {
        this.Show(e);
    }

    void _watcher_Changed(object sender, FileSystemEventArgs e)
```

```
    {
        this.Show(e);
    }

    private void Show(FileSystemEventArgs e)
    {
        Console.WriteLine(e.ChangeType.ToString());
        Console.WriteLine(e.FullPath);
        Console.WriteLine("---------------------");
    }

    public void Dispose()
    {
        if (this._watcher != null)
            this._watcher.Dispose();
    }
}
```

Como podemos analisar, assinamos todos os eventos e, quando uma determinada ação acontecer, os respectivos eventos serão disparados. Esse *wrapper* serve apenas para encapsular todo o trabalho que o **FileSystemWatcher** irá desenvolver na aplicação e que não seria obrigatoriamente necessário criá-lo para o **FileSystemWatcher** funcionar. Finalmente, para utilizar este wrapper, devemos fazer:

VB.NET

```
Using watcher As New FileWatcher("c:\Temp\")
    While (True)
    End While
End Using
```

C#

```
using (FileWatcher w = new FileWatcher(@"c:\Temp\"))
{
    while (true);
}
```

A classe é criada dentro de um bloco *Using* para garantir que o método *Dispose* seja executado mesmo que um erro ocorra. Para o exemplo, vamos criar, renomear, alterar e excluir um arquivo dentro do diretório *c:\Temp*. A imagem abaixo mostra os logs que foram efetuados a partir do *wrapper* que criamos acima:

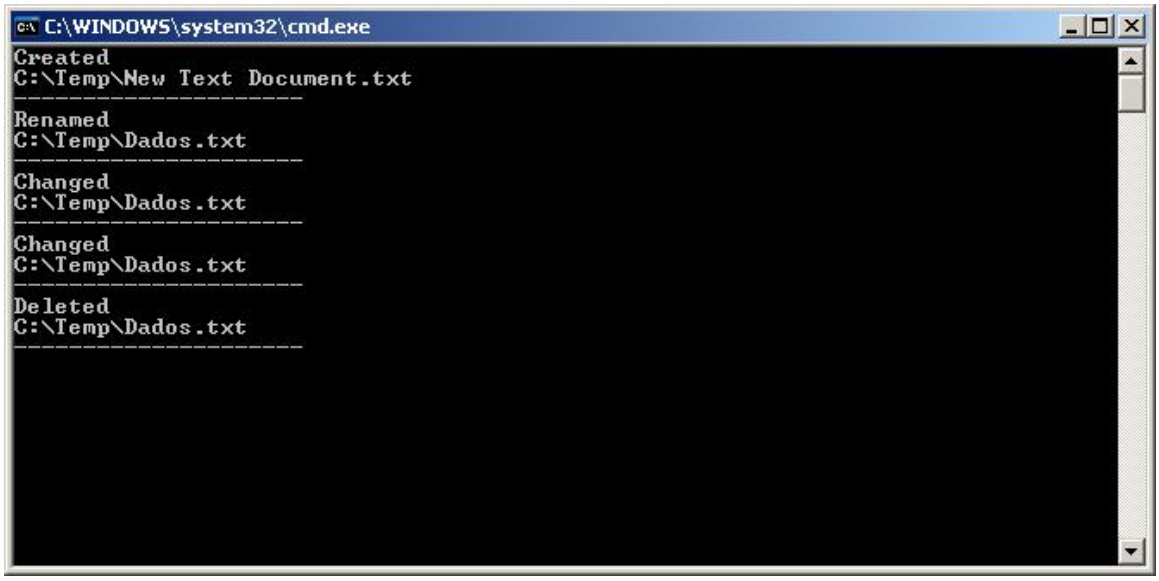

Imagem 5.1 – *Output* do wrapper *FileWatcher*

Lendo e gravando dados em arquivos

Quando estamos desenvolvendo uma aplicação que permite ler os dados de um repositório de dados, como por exemplo, o banco de dados, arquivos do disco ou qualquer outro recurso que seja capaz de persistir os dados, precisamos de alguma forma, extrair e interpretar essas informações. Um *stream* serve como condutor entre o código da aplicação e a fonte (arquivo, banco de dados, memória, etc.). O processo de mover os dados (*bytes*) entre a fonte de dados e a aplicação é chamada de *streaming*.

O .NET Framework possui várias classes que permitem manipular os mais diversos tipos de *streams*. Essas classes estão contidas dentro do *namespace* **System.IO**. Dentro deste *namespace* temos a classe **Stream**. Essa é a classe base (e também abstrata) para todos os tipos de *streams*. Através de streams podemos realizar dois tipos de operações:

1. Leitura de dados trata-se de transferir os valores que estão na fonte para um tipo de dado estruturado, dentro da aplicação.
2. Escrita de dados em *streams*, que permite você enviar valores para serem gravados no *stream*.

A classe abstrata Stream fornece vários métodos interessantes, também abstratos, que devem ser obrigatoriamente implementados nas classes derivadas e, entre eles estão:

Membro	Descrição
CanReader	Indica se o *stream* suporta ou não leitura de dados.
CanWriter	Indica se o *stream* suporta ou não a escrita de dados.
CanSeek	Indica se o stream permite ou não manter e manipular um ponteiro da posição dentro do *stream*.

Length	Retorna o comprimento do *stream* em *bytes*.
Position	Indica a posição do ponteiro dentro do *stream*.
Read	Efetua a leitura de uma séria de *bytes* do *stream*.
Write	Grava uma séria de bytes dentro do *stream*.
Seek	Move o ponteiro do *stream* para um determinada posição.
Flush	Quando invocado, esse método salva todo o conteúdo do *stream* em sua fonte.
Close	Fecha o *stream*. Quando esse método é chamado, todo oseu conteúdo é salvo na fonte.

Essa classe abstrata **Stream** é implementada em diversos outros tipos de streams e, entre eles, temos: **FileStream**, **MemoryStream** e **BufferedStream**.

A primeira delas, **FileStream**, é um *wrapper* para um *stream* que manipula um determinado arquivo, já dando suporte a operações síncronas e assíncronas. Essa classe é utilizada para ler, salvar, abrir e fechar arquivos do disco, lembrando que ele armazena o conteúdo em *buffer* para ter um ganho de performance. A classe **File** (que vimos um pouco mais acima), fornece alguns métodos que, ao serem executados, retornam um *stream* do tipo **FileStream**. Entre esses métodos temos: *Create*, *Open*, *OpenRead* e *OpenWrite*. Para utilizar a classe **FileStream**, podemos fazer:

VB.NET
```
Imports System.IO
Imports System.Text

Dim conteudo As String = "Utilizando o FileStream via VB."
Dim bytes() As Byte = Encoding.Default.GetBytes(conteudo)

Using stream As FileStream = File.Create("Teste.txt")
    stream.Write(bytes, 0, bytes.Length)
End Using
```

C#
```
using System.IO;
using System.Text;

string conteudo = "Utilizando o FileStream via C#.";
byte[] bytes = Encoding.Default.GetBytes(conteudo);

using (FileStream stream = File.Create("Teste.txt"))
{
    stream.Write(bytes, 0, bytes.Length);
}
```

Via método estático *Create* da classe **File**, criamos um arquivo chamado "Teste.txt" e atribuimos o retorno deste método, que será um objeto *stream* do tipo **FileStream**. Notem que isso está sendo feito dentro de um bloco *using* que, mesmo que algum erro aconteça entre esse bloco, o *stream* será fechado de forma segura, invocando o método *Dispose* automaticamente. Depois disso, utilizamos o método *Write* que escreve um conteúdo sincronamente no arquivo.

Nota: A técnica do bloco *using* será utilizada daqui para frente.

Dando seqüencia, vamos falar um pouco sobre a classe **MemoryStream**. Como o próprio nome diz, essa classe cria um *stream* que tem como repositório, a memória e, sendo assim, quando a aplicação é finalizada, esses valores serão perdidos, não tendo uma forma de persitência física. Apesar de "deficiência", ela tem um ótimo benefício, que é justamente o armazenamento na memória, que possibilita um acesso bem mais rápido que o sistema de arquivo, **FileStream**.

Como essa classe é "volátil", o interessante é utilizá-la para manipular dados temporários, que não exigem serem persistidos. Ela também estende a classe **Stream**, fornecendo suporte a chamadas síncronas e assíncronas. A utilização dessa classe é bem semelhante ao que vimos um pouco acima com o **FileStream**. O trecho de código abaixo exemplifica o uso dela:

```
VB.NET
Imports System.IO
Imports System.Text

Dim conteudo As String = "Utilizando o FileStream via VB."
Dim bytes() As Byte = Encoding.Default.GetBytes(conteudo)
Dim length As Integer = bytes.Length

Using ms As New MemoryStream(bytes)
    ms.Write(bytes, 0, length)
    ms.Seek(0, SeekOrigin.Begin)

    Dim output() As Byte = New Byte(length) {}
    ms.Read(output, 0, length)
    Console.WriteLine(Encoding.Default.GetString(output))
End Using

C#
using System.IO;
using System.Text;

string conteudo = "Utilizando o FileStream via C#.";
byte[] bytes = Encoding.Default.GetBytes(conteudo);
int length = bytes.Length;
```

```
using (MemoryStream ms = new MemoryStream(length))
{
    ms.Write(bytes, 0, length);
    ms.Seek(0, SeekOrigin.Begin);

    byte[] output = new byte[length];
    ms.Read(output, 0, length);
    Console.WriteLine(Encoding.Default.GetString(output));
}
```

Apesar de ser bem semelhante ao **FileStream**, há um detalhe bem interessante que precisamos analisar. Esse detalhe é com relação ao construtor da classe **MemoryStream**. Há um *overload* que permite passar um *array* de *bytes*. Isso evitaria de chamar o método *Write* e *Seek* para escrever o conteúdo e retornar a posição do ponteiro para o ínicio, pois o código interno ao construtor se encarregaria de realizar todas essas tarefas.

Ainda descendendo da classe **Stream**, temos a classe **BufferedStream**. *Buffer* é um bloco de *bytes* que residem na memória, usado para efetuar o *cache* de dados para evitar, a todo momento, fazer chamadas ao sistema operacional e, conseqüentemente, melhorar a performance. O *buffer* pode ser usado para leitura ou para gravação, mas nunca simultâneamente. A classe **BufferedStream** possui uma *layer* de *buffer* e é também responsável por gerenciá-la. O exemplo abaixo exibe como criar um arquivo através do método *Create* da classe **File** e atribuir o seu retorno, um objeto do tipo **FileStream**, a um **BufferedStream**:

```
VB.NET
Imports System.IO

Using stream As New BufferedStream(File.Create("Teste.txt"))
    ' realização das operações
End Using

C#
using System.IO;

using (BufferedStream stream = new
BufferedStream(File.Create("Teste.txt")))
{
    // realização das operações
}
```

Além desses *streams* que vimos até o momento, ainda temos ainda dois *streams* que são exclusivos para trabalhar com caracteres de entrada em um *encoding* específico. O primeiro deles é o **StreamReader**. Ele geralmente é utilizado para ler linhas de informações de um arquivo texto qualquer. A utilização desta classe é bem simples, ou

seja, precisamos somente no construtor dela, passar o caminho do arquivo que desejamos ler e, em seguida, invocar o método *ReadLine* ou o método *ReadToEnd*. A diferença entre os dois é que o primeiro lê apenas uma linha por vez, o que necessitaria você colocá-lo dentro de um *loop* para ler todas as linhas do arquivo; já o segundo, em um único método, retorna todo o conteúdo do arquivo. O código abaixo ilustra o uso desta classe:

```
VB.NET
Imports System.IO

Using stream As New StreamReader("c:\Temp\Teste.txt")
    Console.WriteLine(stream.ReadToEnd())
End Using

C#
using System.IO;

using (StreamReader stream = new StreamReader
("c:\Temp\Teste.txt"))
{
    Console.WriteLine(stream.ReadToEnd());
}
```

Já a classe corresponde ao **StreamReader** para gravação é a classe **StreamWriter**. Trabalhamos basicamente da mesma forma: passamos para o construtor o caminho e nome do arquivo que desejamos salvar o conteúdo e, via método *Write* ou *WrilteLine*, vamos adicionando o conteúdo ao arquivo, conforme o exemplo abaixo:

```
VB.NET
Imports System.IO

Using stream As New StreamWriter("c:\Temp\Teste.txt")
    stream.WriteLine("Gravando via StreamWriter no VB.")
End Using

C#
using System.IO;

using (StreamWriter stream = new StreamWriter
("c:\Temp\Teste.txt"))
{
    stream.WriteLine("Gravando via StreamWriter no C#.");
}
```

Finalmente, temos as classes **BinaryReader** e **BinaryWriter**. Essas duas classes servem para ler e gravar tipos de dados primitivos como valores binários em um *encoding* específico. A gravação de dados via **BinaryWriter** é tranquila, pois informamos o *stream* (ou o arquivo) em que os dados serão gravados e, em seguida, invocamos o método *Write* que é sobrecarregado, possuindo sobrecargas para os mais diversos tipos de dados que o .NET Framework suporta.

A classe **BinaryReader**, que lê os dados de um determinado *stream*, precisa de alguns cuidados a parte. Primeiramente precisamos invocar o método *PeekChar* para que o **BinaryReader**, que retorna o próximo caracter disponível mas não avança a posição do ponteiro. A partir de agora, você chama os métodos respectivos a cada tipo que você salvo, exemplo: suponhamos que você criou um inteiro e uma *string*, logo, deve invocar os método *ReadInt32* e *ReadString*, sucessivamente. Todos os métodos *ReadXXX*, além de retornar o valor correspondente, também é responsável por avançar a posição do ponteiro interno do *stream*. Para exemplificar o uso do que vimos aqui sobre o **BinaryWriter** e **BinaryReader**, veja o código abaixo:

VB.NET
```
Imports System.IO

Using bw As New BinaryWriter(File.Create("Teste.bin"))
    bw.Write(123)
    bw.Write("VB.NET e C#")
    bw.Write(true)
    bw.Write(false)
    bw.Write(1.290)
End Using

Using    br    As    New    BinaryReader(File.Open("Teste.bin",
FileMode.Open))
    If Not br.PeekChar() = -1 Then
        Console.WriteLine(br.ReadInt32())
        Console.WriteLine(br.ReadString())
        Console.WriteLine(br.ReadBoolean())
        Console.WriteLine(br.ReadBoolean())
        Console.WriteLine(br.ReadDouble())
    End If
End Using
```

C#
```
using System.IO;

using          (BinaryWriter          bw          =          new
BinaryWriter(File.Create("Teste.bin")))
{
    bw.Write(123);
    bw.Write("VB.NET e C#");
```

```
    bw.Write(true);
    bw.Write(false);
    bw.Write(1.290);
}

using (BinaryReader br = new BinaryReader(File.Open("Teste.bin",
FileMode.Open)))
{
    if (br.PeekChar() != -1)
    {
        Console.WriteLine(br.ReadInt32());
        Console.WriteLine(br.ReadString());
        Console.WriteLine(br.ReadBoolean());
        Console.WriteLine(br.ReadBoolean());
        Console.WriteLine(br.ReadDouble());
    }
}
```

A imagem abaixo trata-se do arquivo "Teste.txt" com os valores salvos em formato binário:

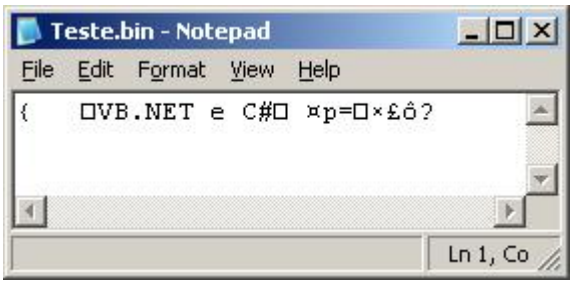

Imagem 5.2 – Arquivo "Teste.bin" com seus valores binários

Compressão de dados

Quando precisávamos comprimir ou descomprimir arquivos (*streams*) nas versões 1.x do .NET Framework tínhamos que recorrer a bibliotecas de terceiros para isso. A Microsoft se preocupou com essa necessidade e decidiu incorporar essa funcionalidade dentro do .NET Framework 2.0. É com isso que surge um novo *namespace* chamado **System.IO.Compression**, fornecendo classes que efetuam compressão e descompressão de *streams*.

Atualmente temos duas classes contidas dentro deste *namespace* (que herdam diretamente da classe **Stream**): **DeflateStream** e **GZipStream**. A classe **DeflateStream** fornece métodos e propriedades para compressão e descompressão de *streams* utilizando o algoritmo *Deflate*. Essa classe não permite a compressão de arquivos com um tamanho maior que 4GB.

Já a classe **GZipStream** também utiliza o mesmo algoritmo da classe **DeflateStream**, mas pode ser estendido para utilizar outros formatos de compressão. Basicamente, a classe **GZipStream** é um *wrapper* para um **DeflateStream**, incluindo *header*, *body* e *footer* onde, a seção *header* contém informações de metadados a respeito do conteúdo comprimido, o *body* é a seção onde o conteúdo comprimido é armazenado e a seção *footer* permite incluir valores de verificações de redundância cíclica para detectar o corrompimento dos dados. Além disso, essa classe ainda possibilita detectar erros durante a compressão e, se desejar compartilhar os dados comprimidos com outros programas, utilize o **GZipStream** ao invés da classe **DeflateStream**. Essa classe não permite a descompressão de arquivos com um tamanho maior que 4GB.

Exemplos: Como o código para exemplificar o uso destas classes é um pouco extenso, ele não será colocado aqui, mas você pode analisar a utilização deles na aplicação de demonstração.

Manipulação de Strings

Utilização da classe StringBuilder

Quando manipulamos *strings* dentro de uma aplicação .NET elas são mapeadas para a classe **System.String**. Essas *strings* são imutáveis, ou seja, quando fazemos uma operação onde alteramos o seu valor ou até mesmo chamando um método dela, como por exemplo, o método *Substring*, uma nova *string* é gerada.

Outro ponto importante é quando estamos falando de concatenação de *strings*. Geralmente utilizamos isso quando precisamos montar um valor proveniente de uma base de dados, ou até mesmo de uma construção mais complexa. Popularmos essas concatenações são realizadas dentro de *loops* que, se não tomarmos um pouco de cuidado, a performance pode ser comprometida. A questão da performance também é bastante importante quando precisamos manipular a *string*, como por exemplo, inserindo um valor em um local especificado, remover uma *string* ou até mesmo trocar uma *string* por outra dentro de uma *string* maior.

Tudo isso que as vezes pode ser muito custoso, a Microsoft se antecipou e criou uma classe para manipular de forma mais ágil as *strings*. Trata-se da classe **StringBuilder**, que está contida dentro do namespace **System.Text**. Esse classe representa agora um *string* mutável, ou seja, ele pode sofrer inserções de novas *strings*, removê-las e trocar caracteres. Essa classe fornece uma porção de métodos que permitem manipular a *string* que está contida dentro da mesma. Entre os principais métodos e propriedades temos:

Membro	Descrição
Capacity	Define a quantidade máxima de memória alocada pela instância corrente. A capacidade pode ser diminuida, mas não pode ser menor que o valor especificado pela propriedade *Length* e também não pode ser maior que o valor especificado pela propriedade *MaxCapacity*.

	O valor padrão para essa propriedade é 16. Ao acrescentar caracteres dentro dele, o **StringBuilder** verifica se o *array* interno que armazena os caracteres está tentando aumentar além da sua capacidade. Se sim, automaticamente dobrará o tamanho dessa capacidade, alocando um novo *array* e copiando os caracteres anteriores para este novo e, conseqüentemente, o *array* original será descartado. Esse aumento dinâmico prejudica a performance e, sendo assim, tente definir uma capacidade para evitar isso.
Chars	Baseando-se em um índice, retorna um tipo *Char* que indica o caracter que está dentro da *string*.
Length	Retorna um número inteiro que indica o tamanho da *string*.
MaxCapacity	Define a quantidade máxima de caracteres que a instância pode armazenar.
Append	Esse é um método que possui vários *overloads*, basicamente suportando em cada *overload* um tipo de dado diferente. Quando esse método é invocado, o valor passado para o mesmo será adicionado ao fim da *string*.
AppendFormat	Permite adicionar um valor no final da *string*, mas a diferença em relação ao método *Append* é que permite formatar o valor antes dele ser efetivamente adicionado.
Insert	Permite adicionar um valor em uma posição específica. Assim como o método *Append*, ele possui vários overloads que permitem passamos os mais diversos tipos para ser adicionado.
Remove	Este método aceita dois números inteiros como parâmetro. O primeiro deles é a posição inicial dentro da *string* e, o segundo, a quantidade de caracteres que deseja remover da *string*.
Replace	Modifica todas as ocorrências que forem encontradas dentro da *string* por um outro valor.
ToString	Esse método permite converter a instância corrente da classe **StringBuilder** em uma *string*.

Para exemplificar o uso desta classe e de seus membros, o código abaixo manipula uma *string* que está contida dentro da classe **StringBuilder**:

```
VB.NET
Imports System.Text

Dim sb As New StringBuilder(16, 150)
Dim versoesNET As Object() = {"1.0", "1.1", "2.0"}

sb.Append("Utilizando a Classe String Builder.")
sb.AppendFormat(" Ela é disponilizada nas seguintes versões: {0},
{1} e {2}.", versoesNET)
sb.Insert(34, " do namespace System.Text")
```

```
sb.Remove(0, 13)

Console.WriteLine(sb.ToString())
```

C#

```
using System.Text;

StringBuilder sb = new StringBuilder(16, 150);
object[] versoesNET = { "1.0", "1.1", "2.0" };

sb.Append("Utilizando a Classe String Builder.");
sb.AppendFormat(" Ela é disponilizada nas seguintes versões: {0},
{1} e {2}.", versoesNET);
sb.Insert(34, " do namespace System.Text");
sb.Remove(0, 13);

Console.WriteLine(sb.ToString());
```

Inicialmente criamos um objeto do tipo **StringBuilder**, definindo a sua capacidade de memória alocada como 16 e a capacidade máxima de caracteres que ele pode suportar como 150. Em seguida, utilizamos os métodos que vimos detalhadamente na tabela, um pouco mais acima, suas respectivas funcionalidades. O resultado para ambos os códigos é:

```
Classe String Builder do namespace System.Text. Ela é
disponilizada nas seguintes versões: 1.0, 1.1 e 2.0.
```

Considerações de Performance

Assim como há o método *Append* para a classe **StringBuilder**, existe o método *Concat* para a classe *String*. A finalidade de ambos é adicionar um novo conteúdo ao fim de uma determinada *string*. A principal diferença entre eles é que o concatenação via classe **String** sempre retorna uma nova *string* pois, como vimos acima, as *strings* são imutáveis. Já a classe **StringBuilder** alterar o seu conteúdo interno de forma mais performática mas, temos um *overhead* a mais que é justamente a criação do objeto **StringBuilder**.

Sendo assim, quantas vezes precisamos chamar o método *Append* para compensar a construção e o uso do objeto **StringBuilder**? Pois bem, existem uma porção de informações a respeito e abaixo tentaremos elencar os mais típicos, lembrando que isso pode variar de um cenário ao outro.

- Se você não tiver idéia do tamanho final da *string*, então utilize o **StringBuilder** se tiver ao menos sete concatenações

- Se você puder antecipar o tamanho da *string* (em pelo menos 30%), então utilize o **StringBuilder** se tiver ao menos cinco concatenações
- Se você puder antecipar precisamente o resultado da *string*, então utilize o **StringBuilder** se você tiver ao menos três concatenações
- Sem qualquer uma das condições atendidas, **StringBuilder** é mais rápido se você tiver ao menos três concatenações

Regular Expressions

Expressões regulares forencem uma forma potente, eficiente e flexível para prcessar textos. Trata-se uma linguagem que foi criada e otimizada para manipular textos e, através delas, você pode facilmente extrair, ler, editar, trocar, excluir *substrings* de um determinado texto. Elas são ferramentas indispensáveis quando utilizamos arquivos de logs, processamento de arquivos HTML, validações de informações que o usuário fornece para a aplicação, entre outras inúmeras necessidades.

As expressões regulares consistem em dois tipos de caracteres: literal (que são os caracteres normais) e os *metacaracteres* (símbolos). O conjunto desses caracteres dão as expressões regulares todo o poder de precessamento do texto. Para mostrar um exemplo de expressão regular, podemos citar: "\s2000". Quando ela é aplicada em um texto, ela irá procurar por todas as ocorrências com o valor 2000 que estão precedidos de qualquer caracter de espaço, podendo ser um espaço simples ou um *TAB*.

Como manipular expressões regulares não é uma tarefa fácil, o .NET Framework fornece uma gama de classes que permitem manipular as expressões regulares de forma fácil. Todas as classes estão contidas dentro de um *namespace* chamado **System.Text.RegularExpressions** e, entre as classes disponíveis temos: **RegEx**, **Match** , **MatchCollection, Group**, **GroupCollection, Capture** e **CaptureCollection**. Ainda temos um delegate chamado **MatchEvaluator** que representa um método que será invocado quando um determinado valor é encontrado dentro de um texto durante o processo de *replace*.

Mas antes de analisar as classes que o .NET Framework fornece, precisamos entender um pouco mais sobre as expressões regulares e como construí-las. Vale lembrar que não iremos nos aprofundar muito porque esse assunto é muito mais extenso do que veremos aqui. Através da tabela abaixo, vamos analisar, inicialmente, os *metacaracteres* e o que eles representam dentro de uma expressão regular:

Metacaracter	Descrição	Grupo
.	Estipula qualquer caracter. Exemplo: 1. **"n.o":** A expressão irá procurar por essa palavra, não importando que caracter esteja substituindo o ponto.	Representantes

[ ]	Defina uma lista de caracteres permitidos. Permite também definir um intervalo de caracteres. Exemplo: 2. **“n[ãa]o”:** Se executando a expressão com o ponto para procurarmos, por exemplo, pela palavra “não”, se existir palavras no texto que atendam aos caracteres n e o, eles também serão retornados, exemplo: “tecla**ndo**”, “expandi**ndo**”. Como a idéia é recuperar apenas a palavra “não”, independente de ter colocado o til ou não, pode optar por essa expressão regular. 3. **“[0-9]”:** Permite somente números que estiverem entre 0 e 9.	Representantes
[^]	Defina uma lista de caracteres proibidos. Exemplo: 4. **“[^0-9]”:** A partir desta expressão regular, você está querendo procurar por todos os caracteres, com exceção dos números de 0 a 9. 5. **“[^a-f]”:** Neste, você está procurando por todos os caracteres, com exceção do intervalo de “a” até “f” (minúsculo).	Representantes
?	Zero ou um (opcional). Exemplo: 6. **“bater?”:** Essa expressão indica que o “r” será opcional e, sendo assim as palavras “bate” e “bater” são perfeitamente válidas.	Quantificadores
*	Zero, um ou mais. Exemplo: 7. **“10*”:** O asterísco não se importa com o valor, ou seja, pode ter, não ter ou pode ter inúmeras ocorrências. No caso mostrado no exemplo, todos os valores que contemplem ou não o zero será retornado, tipo: 1, 100, 1000, 10000, etc. Mas o 2 não se enquadrara.	Quantificadores
+	Um ou mais. Exemplo: 8. **“10+”:** Neste caso, o zero precisa ocorrer pelo menos uma vez. Sendo assim, 10, 100,	Quantificadores

	1000, etc. seria retornado mas, 1 não se enquadra.	
{n,m}	De x até m. Exemplo: 9. **"[a-g]{2,4}":** Permite encontrar dentro de uma *string* os caracteres de "a" até "g" que contenham no mínimo dois e no máxima quatro ocorrências.	Quantificadores
^	Início da linha. Exemplo: 10. **"^[a-z]":** Neste cenário, a finalidade do acento circunflexo é apenas de indicar o ínicio da linha e, nesta expressão regular, estamos procurando por linhas que começam com caracteres de "a" até "z".	Âncoras
$	Fim da linha. Exemplo: 11. **"^$":** Essa expressão regular contempla uma linha vazia.	Âncoras
\b	Ínicio ou fim de uma palavra. Exemplo: 12. **"\ba":** Irá retornar todas as palavras que começa com a letra "a". 13. **"a\b":** Irá retornar todas as palavras que acaba com a letra "a".	Âncoras
\	Define um caracter como literal. Exemplo: 14. **"[0-9]\.[0-9]":** Neste caso, o metacaracter não exercerá a sua funcionalidade dentro do mundo das expressões regulares, sendo agora, um caracter normal.	-
|	Ou um ou outro. Exemplo: 15. **"item1|item2":** Permite encontrar um ou outro item especificado na criação.	-
( )	Delimita um grupo. Exemplo: 16. **"([0-9]){2}":** Todo o conteúdo definido	-

	dentro de um grupo são encarados como sendo um único bloco na expressão e, no nosso exemplo, ele trará todos os números que estiverem separados de dois em dois, ou seja, dado uma *string* do tipo "*1234 567 89023*", os valores retornados são: 12, 34, 56, 89 e 02.	
\1...\9	Retrovisores. Exemplo: 17. **"([a-z]+)\1":** Os retrovisores permitem recuperarmos informações repetidas. Eles são utilizados em conjunto com os grupos e repetem o resultado (não a expressão) do grupo, ou seja, para o exemplo acima, o grupo procura por todas as letras entre "a" e "z" e, em conjunto com o "\1", retornará o apenas os casos onde a letra seguinte é a mesma definida pelo grupo. Para o exemplo "*ahjkko ppoijj yytrwwul ppmnbbvf*" ele retornará: kk, pp, jj, yy, ww, pp, bb.	-

Ainda existem vários outros caracteres que utilizamos em expressões regulares, mas não vamos abordar todos aqui justamente por não ser o foco do capítulo. A seguir, veremos as classes que o *namespace* **System.Text.RegularExpressions** fornece para manipular *strings* e *regular expressions*. Assim como os caracteres, não analisaremos as classes fornecidas pelo *namespace*, pois como já sabemos, não é foco do capítulo e, vamos analisar somente as principais classes necessárias para tratarmos expressões regulares.

Para iniciarmos, vamos analisar a classe **RegEx**. Essa classe representa uma expressão regular imutável, ou seja, de somente leitura, fornecendo métodos estáticos que permitem a análise de expressões regulares sem a necessidade de criar a instância da classe. O exemplo abaixo mostra como devemos proceder para utilizá-la. A expressão regular a seguir, verifica se o valor informado é ou não um número:

```
VB.NET
Imports System.Text.RegularExpressions

Dim numero1 As String = "123a"
Dim numero2 As String = "456"
Console.WriteLine(Regex.IsMatch(numero1, "^[0-9]+$"))
Console.WriteLine(Regex.IsMatch(numero2, "^[0-9]+$"))

'Output
False
```

```
True

C#
using System.Text.RegularExpressions;

string numero1 = "123a";
string numero2 = "456";
Console.WriteLine(Regex.IsMatch(numero1, "^[0-9]+$"));
Console.WriteLine(Regex.IsMatch(numero2, "^[0-9]+$"));

//Output
False
True
```

Neste momento apresentaremos uma nova classe chamada **Match**. Essa representa o resultado de uma análise de expressão regular que é retornado através do método *Match* da classe **RegEx**. O exemplo abaixo ilustra o uso desta classe:

```
VB.NET
Imports System.Text.RegularExpressions

Dim numero1 As String = "123"
Dim r As New Regex("^[0-9]+$")
Dim match As Match = r.Match(numero1)

Console.WriteLine(match.Success)
Console.WriteLine(match.Value)
Console.WriteLine(match.Length)

'Output
True
123
3

C#
using System.Text.RegularExpressions;

string numero1 = "123";
Regex r = new Regex("^[0-9]+$");
Match match = r.Match(numero1);

Console.WriteLine(match.Success);
Console.WriteLine(match.Value);
Console.WriteLine(match.Length);

//Output
True
```

```
123
3
```

Mudamos ligeiramente o código para análise da expressão regular. Ao invés de chamarmos o método estático *Match*, optamos por criar a instância da classe **RegEx**, passando em seu construtor a expressão que será aplicada a *string*. Criamos um objeto do tipo **Match** e atribuimos a ele o retorno do método *Match* da classe **RegEx**. A classe **Match** possui algumas informações importantes a respeito do resultado da análise. Abaixo estão listados as principais propriedades:

Propriedade	Descrição
Index	Retorna um número inteiro indicando a posição onde o primeiro caracter foi encontrado.
Length	Retorna um número inteiro indicando a posição inicial da *substring*.
Success	Através de um valor booleano, indica se foi a expressão foi ou não satisfeita.
Value	Retorna a *substring* que foi encontrado a partir da expressão regular.

Para encerrar a seção sobre expressões regulares e as classes do .NET Framework que as manipulam, vamos analisar a classe **MatchCollection** que, como o próprio nome diz, trata-se de uma coleção de objetos do tipo **Match**. Nos exemplos que vimos na tabela de *metacaracteres* há a possibilidade de retornar vários resultados. Sendo assim, utilizaremos um novo método da classe **RegEx** que é chamado de *Matches*. Lembre-se de que também existe esse mesmo método em sua forma estática, que não necessita da instância da classe e que voltaremos a utilizá-lo no exemplo a seguir:

VB.NET
```
Imports System.Text.RegularExpressions

Dim coll As MatchCollection = _
    Regex.Matches("1234 567 89023 2", "([0-9]){3}")

For Each m As Match In coll
    Console.WriteLine(m.Value)
Next
```

C#
```
using System.Text.RegularExpressions;

MatchCollection coll =
    Regex.Matches("1234 567 89023 2", "([0-9]){3}");

foreach (Match m in coll)
    Console.WriteLine(m.Value);
```

Para ambos os códigos, o resultado é o seguinte:

```
123
567
890
```