Capítulo 12
Interoperabilidade com componentes COM

Introdução

Antes mesmo da plataforma .NET surgir, as empresas já estavam desenvolvendo aplicações e componentes nas mais diversas linguagens, como por exemplo C++, Visual Basic 6, etc.. Não há dúvidas que as empresas investiram tempo e dinheiro na criação destes componentes que são, em algumas vezes, largamente utilizados e, dificilmente, serão sobrescritos instantaneamento para .NET, utlizando o código gerenciado.

Não somente esse cenário, mas temos diversos outros componentes que são expostos via COM que poderíamos estar utilizando dentro da plataforma .NET. Felizmente o .NET tem a capacidade de comunicar com o mundo COM, permitindo criarmos componentes que sejam acessíveis a linguagens não .NET e, também, permite utilizarmos os componentes legados dentro das aplicações .NET.

Na primeira parte deste capítulo, vamos abordar como devemos proceder para a criação de componentes que serão utilizados por aplicações/componentes COM e também como consumir componentes COM. A segunda, e última parte, destina-se a analisarmos um serviço chamado **PInvoke** que a plataforma .NET nos disponibiliza. Ela nos permite fazer chamadas a DLLs não gerenciadas e, além disso, efetuarmos chamadas as APIs que o Windows disponibiliza para termos acesso a informações do sistema operacional.

Acessando componentes COM

Criação de Assembly para Interoperabilidade

Antes de mais nada, vamos entender o que é algo "interoperável":

> *"Interoperabilidade é a capacidade de um sistema (informatizado ou não) de se comunicar de forma transparente (ou o mais próximo disso) com outro sistema (semelhante ou não). Para um sistema ser considerado interoperável, é muito importante que ele trabalhe com padrões abertos. Seja um sistema de portal, seja um sistema educacional ou ainda um sistema de comércio eletrônico, ou e-commerce, hoje em dia se caminha cada vez mais para a criação de padrões para sistemas."*
>
> *Wikipédia*

Como o .NET já é uma realidade, muitas empresas já estão trabalhando em cima dessa plataforma, mas mantendo alguns softwares e componentes ainda desenvolvidos em uma tecnologia mais antiga. Hoje encontramos muitas aplicações rodando em Visual Basic 6 e, como já discutimos acima, foi injetado muitos recursos para o desenvolvimento dessas aplicações, que não podem ser descartados.

Quando começamos a falar em migração, se o software foi "politicamente escrito", então a parte da *interface* com o usuário, geralmente, é onde temos menos códigos, o que conseqüentemente será a primeira parte a ser migrada para a nova plataforma. Sendo assim, mesmo em .NET, queremos continuar utilizando os componentes feitos em Visual Basic 6.0 que, neste primeiro momento, não serão migrados.

Para tornar a comunicação entre o mundo COM e o mundo .NET possível, entra em cena um *Assembly* chamado ***Interop Assembly***. Esse *Assembly*, fornece uma *layer* intermediária entre os dois mundos, que habilita a comunicação entre o *runtime* .NET Framework e o componente (ou aplicação) COM. O ***Interop Assembly*** nada mais é que um *wrapper* para os membros (metadados) que o mesmo expõe. Para gerar esse *Assembly* de interoperabilidade, basta você adicionar uma referência a um componente COM em uma aplicação .NET. Ao compilar a aplicação, dentro do diretório **\bin** da mesma você notará que, além do *Assembly* que é gerado pela aplicação, um outro *Assembly*, este de interoperabilidade, também é gerado com a seguinte nomenclatura: **Interop.[Nome Componente COM].dll**. E esses ***Interop Assemblies*** são criados um para cada referência COM que tiver em seu projeto.

Se você não tiver o Visual Studio .NET no momento, então você pode recorrer a um utilitário de linha de comando, qual também tem a finalidade de criar ***Interop Assemblies***. Esse utilitário é chamado **TlbImp.exe** e é encontrado dentro do diretório onde está contido o .NET Framework. Uma dica aqui é abrir o utilitário a partir do *prompt* do Visual Studio .NET para evitar procurar o *path* completo do mesmo. Uma possível sintaxe para utilizar este utilitário é:

```
C:\ tlbimp Componente.dll /out:Interop.Componente.dll
```

Além dessas duas forma, ainda há a possibilidade de criar esse componente dinamicamente, ou seja, o .NET Framework fornece classes que permitem, via código, criarmos um ***Interop Assembly***. Para isso, existe um *namespace* chamado **System.Runtime.InteropServices** que fornece todos os tipos necessários para implementarmos essa solução.

A principal classe que temos dentro deste namespace que fornece o conjunto de serviços necessários para converter um *Assembly* gerenciado acessível via COM e extrair um **Interop Assembly** de um componente COM é a classe **TypeLibConverter** que, inclui as mesmas opções do utilitário **Tlbimp.exe**. Essa classe possui três métodos que podemos utilizar, dependendo da necessidade. Esses métodos são:

Método	Descrição
ConvertAssemblyToTypeLib	Dado um *Assembly* .NET, extrai a biblioteca de tipos e cria um *Assembly* que pode ser utilizado através do mundo COM.

ConvertTypeLibToAssembly	Dado um *Assembly* COM, extrai a biblioteca de tipos e cria um *Assembly* de interoperabilidade que pode ser utilizado por aplicações .NET.
GetPrimaryInteropAssembly	Através de uma **GUID**, que identifica o objeto dentro (*type library*) do *Registry*, recupera o nome e o *code base* de um *Assembly* de interoperabilidade. Esse método retorna um valor booleano indicando se *Assembly* foi ou não encontrado.

Expondo componentes .NET para o mundo COM

Uma vez que criamos classes em .NET aplicações COM podem utilizá-las, se assim desejar. Entretanto, para assegurar que isso seja possível, é necessário nos atentarmos em alguns detalhes que precisam ser analisados para que o mesmo consiga ser visível ao mundo COM.

Para disponibilizar o componente para interoperabilidade com o mundo COM, você primeiramente deve desenhar o seu componente visando facilitar esse processo e, para isso, você deve explicitamente implementar *Interfaces* em seus componentes que serão expostos. Isso é necessário porque componentes COM "não podem conter" os membros diretamente e, sendo assim, devem ter as *Interfaces* implementadas. Apesar do COM poder gerar a Interface automaticamente, é melhor você criar isso manualmente, o que permitirá um maior controle sob o componente e suas formas de atualização.

Quando os componentes COM geram automaticamente a sua *Interface*, você não pode fazer nenhum mudança em sua estrutura pública. Isso acontece porque componentes COM são imutáveis. Se você romper essa regra, o componente deixará de ser invocado. Justamente por essa questão que permitir que a *Interface* seja criada automaticamente não é uma boa idéia e, o ideal é estar criando as suas próprias Interfaces, pois você terá uma maior segurança, já que conseguirá garantir que a *Interface* não mudará. Para manipularmos a forma com a qual a Interface é criada e manipulada, temos um atributo chamado **ClassInterfaceAttribute** que devemos decorar o componente. Esse atributo recebe em seu construtor, uma das opções do enumerador **ClassInterfaceType**, que estão descritas abaixo:

Opção	**Descrição**
AutoDispatch	Opção padrão que indica que a classe suporta *late-binding* quando chamada através de um cliente COM e omite a descrição da *Interface*. Utilize essa opção quando a classe pode sofrer mudanças futuras.
AutoDual	Esta opção cria uma *Interface* que permite o *early-binding*. Somente utilize essa opção se essa classe não sofrerá mudanças futuras.
None	O COM *Interop* não criará a *Interface* padrão para a classe. Neste caso, você deverá, explicitamente, fornecer uma *Interface* para ser implementada na classe que será exposta para o mundo COM.

O segundo detalhe que temos que nos atentar é com relação aos membros que desejamos expor ao mundo COM. Por padrão, construtores parametrizados, métodos estáticos e constantes não são expostos. Com exceção disso, todos os tipos e membros públicos são acessíveis via COM. Mas e se existir algum tipo ou membro público que você quer ocultar dos clientes COM? É neste caso que entra em cena o atributo **ComVisibleAttribute**. Esse atributo pode ser utilizado nos mais diversos tipos e membros e, em seu construtor, recebe um parâmetro booleano indicando se o membro será ou não visível para os clientes COM.

O código abaixo exemplifica a implementação de uma Interface em uma classe. Esse processo é idêntico ao que já conhecemos dentro da plataforma .NET. Em seguida, aplicamos o atributo **ClassInterfaceAttribute** no componente, definindo em seu construtor a opção *None* do enumerador **ClassInterfaceType**. Finalmente, uma propriedade é criada mas negamos o acesso a mesma para clientes COM através do atributo **ComVisibleAttribute**, definindo-o como *False*:

VB.NET
```
Imports System.Runtime.InteropServices

Public Interface IMensagens
    Function BoasVindas(ByVal nome As String) As String
End Interface

<ClassInterface(ClassInterfaceType.None)> _
Public Class Usuarios
    Implements IMensagens

    Public Function BoasVindas(ByVal nome As String) As String _
        Implements IMensagens.BoasVindas

        Return "Olá " & nome
    End Function

    <ComVisible(False)> _
    Public ReadOnly Property SenhaTemporaria() As String
        Get
            Return "P@$$w0rd"
        End Get
    End Property
End Class
```

C#
```
using System.Runtime.InteropServices;

public interface IMensagens
{
    string BoasVindas(string nome);
```

```
}

[ClassInterface(ClassInterfaceType.None)]
public class Usuarios : IMensagens
{
    public string BoasVindas(string nome)
    {
        return "Olá " + nome;
    }

    [ComVisible(false)]
    public string SenhaTemporaria
    {
        get
        {
            return "P@$$w0rd";
        }
    }
}
```

Um detalhe importante é com relação ao atributo **ComVisibleAttribute**. Por padrão, esse atributo é definido como *True* e, sendo assim, todos os tipos são gerenciados estão disponíveis para o mundo COM. Esse atributo, além de decorar individualmente cada tipo, pode também ser aplicado a nível de *Assembly*, através do arquivo *Assembly.vb* ou *Assembly.cs*.

Depois do *Assembly* .NET devidamente gerado, chega o momento de utilizá-lo em uma aplicação COM, como por exemplo, o Visual Basic 6. Mas somente com o *Assembly* .NET "puro" ainda não é possível, pois o mesmo precisa ser primeiramente registrado para, somente assim, ser acessado através do aplicações COM. Para registrá-lo, existe um utilitário de linha de comando chamado **regasm.exe**, que está contido dentro da seguinte pasta: *%windir%\Microsoft.NET\Framework\v2.0.50727*. Esse utilitário lê os metadados do *Assembly*, os extrai e adiciona as entradas necessárias dentro do *Registry*, permitindo assim, o acesso via clientes COM. Abaixo um exemplo de como registrar um componente .NET com esse utilitário:

```
C:\ regasm Componente.dll /tlb:Componente.tlb
```

Uma vez feito isso, basta você adicionar a referência ao mesmo em seu projeto (COM) e utilitá-lo.

Definindo a Interface padrão

Quando desejamos expor um componente .NET para o mundo COM, ele deve ter algumas configurações extras em relação aos componentes que são utilizados somente por aplicações .NET.

Quando o definimos com o valor **ClassInterfaceType.None** no componente, indicará que a criação da *Interface* será fornecida por nós. Neste caso, ao registrar o componente com o ***Type Library Exporter (Tlbexp.exe)***, o mesmo será exposto com a primeira *interface* pública visível encontrada pelo utilitário, definindo assim, a *interface* padrão do componente para o mundo COM.

Mas e quando existerem mais que uma *Interface* implementada no componente e, por algum motivo, queremos expor não a primeira *Interface* pública visível, mas a segunda ou a terceira. A versão 2.0 do .NET Framework introduz um novo atributo chamado **ComDefaultInterfaceAttribute** que, podemos expecificá-lo no componente para determinar qual será a *interface* padrão utilizada independentemente da ordem de implementação. Em seu construtor, devemos especificar a *Interface* (através de um objeto do tipo **Type**) padrão para o mundo COM. O exemplo abaixo exemplifica o componente decorado com este atributo:

```
VB.NET
Imports System.Runtime.InteropServices

<ClassInterface(ClassInterfaceType.None)> _
<ComDefaultInterface(GetType(ILogin))> _
Public Class AuthenticationServices
    Implements IData
    Implements ILogin

    'Implementação...
End Class

C#
using System.Runtime.InteropServices;

[ClassInterface(ClassInterfaceType.None)]
[ComDefaultInterface(typeof(ILogin))]
public class AuthenticationServices : IData, ILogin
{
    //Implementação...
}
```

Para consumir o componente no mundo COM, podemos fazer (exemplo em VB6):


```
[ Sem o atributo ComDefaultInterface ]
Dim authService As New AuthenticationServices
```

```
Dim login As ILogin
Set login = authService

authService.Update()
MsgBox login.Validate("IA", "Pa$$w0rd")

[ Com o atributo ComDefaultInterface ]
Dim authService As New AuthenticationServices
Dim data As IData
Set data = authService

MsgBox authService.Validate("IA", "Pa$$w0rd")
data.Update()
```

PInvoke

O .NET Framework fornece uma serviço chamado **PInvoke** – **Platform Invoke**. Esse serviço possibilita a chamada a código não gerenciado, que estão implementados em DLL (*dynamic link library*), como é o caso das APIs do Windows. Esse serviço localiza e invoca uma função e se encarrega de efetuar o *marshaling* dos argumentos automaticamente quando é necessário.

Marshaling é a técnica utilizada para converter valores e trafegá-los entre processos e *threads* diferentes. As técnicas de *marshaling* são utilizadas para compatibilidade de tipos. Quando invocamos um método não gerenciado a partir do código gerenciado, os tipos de dados devem ser convertidos em um tipo correspondente do mundo não gerenciado e, quando não é possível mapear tipos apenas com a declaração de tipos correspondentes, temos que utilizar técnicas diferentes como conversão de formato e o comportamento dos mesmos. Veremos mais a respeito do *mashalling* no decorrer deste capítulo.

A imagem abaixo ilustra o processo de chamada a um código gerenciado a partir do **PInvoke**:

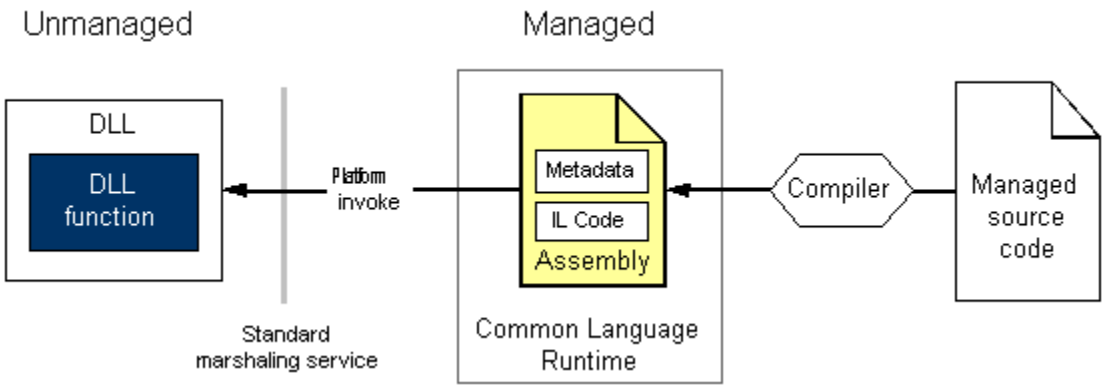

Imagem 13.1 – Processo de chamada do **PInvoke**.

O processo ocorre em quatro passos:

1. Localiza a DLL contendo a função.
2. Carrega a DLL na memória.
3. Localiza a função dentro da memória, passando os parâmetros e efetuando o *marshaling* quando necessário.
4. Transfere o controle para a função do código não gerenciado.
5. Se exceções acontecerem elas são atiradas pela função de código não gerenciado para o chamador do código gerenciado.

Observação: Localizar, carregar a DLL e localizar a função na memória somente ocorre na primeira chamada para a função.

Para chamar uma função de uma DLL que está escrita em código não gerenciado a partir do código gerenciado, primeiramente devemos criar uma definição desta função em código gerenciado, dentro de uma classe qualquer. Essa função não tem nenhuma implementação de código e obrigatoriamente deverá conter a mesma assinatura da função não gerenciada que deseja invocar. Depois disso, utilizamos um atributo chamado **DllImportAttribute** que é utilizado para especificar a localização da DLL onde está contido o método externo.

Observação 1: Existe um site chamada *PInvoke.net*: http://www.pinvoke.net que fornece gratuitamente uma listagem, já com exemplos de códigos, para as APIs do Win32 e outros componentes não gerenciados, com exemplos em Visual Basic .NET e Visual C#.

Observação 2: Antes de utilizar uma API do Windows, verifique se não existe uma classe gerenciada (dentro do .NET Framework) que tenha a mesma finalidade.

O atributo **DllImporteAttribute** possue em seu construtor alguns *named parameters* que podemos configurá-los para customizar o comportamento do *PInvoke*. Esses parâmetros são descritos na tabela abaixo:

Parâmetro	**Descrição**
BestFitMapping	Indica se o ajuste durante a conversão de *ANSI* para *Unicode*. Isso permitirá uma análise mais precisa com relação a conversão que, as vezes, pode não ser o ideal, pois podemos perder ou ser substituída alguma informação. Caracteres que não conseguem ser mapeados são representados pelo ponto de interrogação "?".
CallingConvention	Este campo indica qual será a convensão adotada para a chamada do método não gerenciado. Esse campo aceita umas das opções contidas no enumerador **CallingConvention**. Abaixo estão as descrições das opções

	que ele fornece. • **Cdecl –** O chamador limpa a *stack*. Utilizado para casos em que o método um número variado de parâmetros. • **FastCall –** Não suportado. • **StdCall –** A função limpa a *stack*. Esta é a convensão padrão para quando desejar invocar funções não gerenciadas através do *PInvoke*. • **Winapi –** Indica que usará a conversão adotada pelo sistema operacional. Esta também é a opção padrão.
CharSet	Indica qual será o conjuto de caracteres utilizados durante o *marshaling* de *strings*. Esse valor é definido a partir do enumerador **CharSet**, que contém as seguintes opções: • **Ansi –** Efetua o *marshaling* de *strings* em formato *ANSI*. • **Auto –** O *PInvoke* decide em tempo de execução entre *Ansi* e *Unicode*, baseando-se no sistema operacional. • **Unicode –** Efetua o *marshaling* de *strings* em formato *Unicode* de 2 *bytes*.
EntryPoint	Indica o nome da função da DLL não gerenciada que será invocada. Esse valor somente é exigido quando o nome da função não gerenciada difere da definição (nome) da função gerenciada.
ExactSpelling	Este campo controla como o *runtime* irá efetuar a busca pela função a ser executada. Se este campo estiver definido como *False*, e o **CharSet** estiver definido como *Ansi*, a letra "A" será adicionada no final do nome da função; agora, se o **CharSet** estiver definido como *Unicode*, a letra "W" é adicionada no final do nome da função. Isso porque quando estamos trabalhando com as funções das APIs do Windows, que trabalham com *strings*, normalmente temos duas versões da mesma função disponíveis, que é a versão *Ansi* e a versão *Unicode*, que são sufixadas respectivamente por "A" e "W". A tabela abaixo exibe um relacionamento entre os campos **CharSet** e **ExactSpelling**, exibindo o valor da propriedade **ExactSpelling** baseando-se nos valores padrões definidos por cada linguagem:

	<table><tr><th>Linguagem</th><th>Ansi</th><th>Unicode</th><th>Auto</th></tr><tr><td>VB.NET</td><td>True</td><td>True</td><td>False</td></tr><tr><td>C#</td><td>False</td><td>False</td><td>False</td></tr></table>
PreserveSig	Indica se os métodos não gerenciados que possuem **HRESULT** ou ***retval*** como valores de retornos são diretamente "traduzidos" ou convertidos em exceções. • **True –** O retorno do método será um número inteiro que conterá o **HRESULT**. • **False –** O método automaticamente converterá os valores de **HRESULT** ou **retval** em exceções. É importante se atentar aqui, pois exige que a assinatura do método seja mudada para retornar um número inteiro.
SetLastError	Indica se **SetLastError** é ou não chamado. Se o valor for definido como *True*, o marshaler invoca *GetLastError* and faz o *cache* do valor retornado para previnir que ele seja sobrescrito por alguma outra API. Assim, você poderá recuperar o código de erro através do método estático *GetLastWin32Error* da classe **Marshal**.
ThrowOnUnmappableChar	Valor booleano que indica se uma exceção será atirada quando um caracter não conseguir ser mapeado.

O código abaixo exemplifica como devemos criar o método gerenciado para a chamada da função não gerenciada e, neste exemplo, iremos invocar a função *LogonUser* da API **advapi32.dll**. Esse método retornará uma valor booleano indicando se as credenciais informadas através dos parâmetros *userName* e *password* são ou não válidas.

```
VB.NET
Imports System
Imports System.Runtime.InteropServices

Public Class Program
    <DllImport("advapi32.dll")> _
    Public Shared Function LogonUser( _
        ByVal lpszUsername As String, _
        ByVal lpszDomain As String, _
        ByVal lpszPassword As String, _
        ByVal dwLogonType As Integer, _
        ByVal dwLogonProvider As Integer, _
        ByRef phToken As IntPtr) As Boolean
    End Function

    Public Shared Sub Main()
```

```
        Dim token As IntPtr
        If LogonUser("Usuario", String.Empty, "P@$$w0rd", 2, 0,
token) Then
            Console.WriteLine("Usuário válido.")
        Else
            Console.WriteLine("Usuário inválido.")
        End If
    End Sub
End Class
```

C#

```
using System;
using System.Runtime.InteropServices;

class Program
{
    [DllImport("advapi32.dll")]
    private static extern bool LogonUser(
        String lpszUsername,
        String lpszDomain,
        String lpszPassword,
        int dwLogonType,
        int dwLogonProvider,
        out IntPtr phToken);

    static void Main(string[] args)
    {
        IntPtr token;
        if (LogonUser("login", string.Empty, "P@$$w0rd", 2, 0,
out token))
            Console.WriteLine("Usuário válido.");
        else
            Console.WriteLine("Usuário inválido.");
    }
}
```

Como podemos notar no código acima, apenas decoramos o método gerenciado *LogonUser* com o atributo **DllImportAttribute**, sem definir os demais campos que vimos na tabela acima, o que fará com que ele assuma os valores padrões. É importante dizer que você poderá customizar esses valores de acordo com a sua necessidade, olhando sempre a finalidade para qual cada um deles se destina.

Nota

Quando estamos utilizando o Visual Basic .NET, ele fornece uma alternativa, mas restrita, para invocar métodos não gerenciados. Para isso, utilizamos uma *keyword* chamada **Declare** em conjunto com outra *keyword* chamada **Alias** que, por sua vez,

permite definirmos qual a função do código gerenciado queremos invocar. Abaixo está um exemplo da utilização da função *LogonUser* da API **advapi32.dll** com esta forma exclusiva do Visual Basic .NET:

```
Imports System
Imports System.Runtime.InteropServices

Public Class Program
    Public Declare Auto Function LogonUser Lib "advapi32.dll"
Alias "LogonUser" ( _
        ByVal lpszUsername As String, _
        ByVal lpszDomain As String, _
        ByVal lpszPassword As String, _
        ByVal dwLogonType As Integer, _
        ByVal dwLogonProvider As Integer, _
        ByRef phToken As IntPtr) As Boolean

    Public Shared Sub Main()
        Dim token As IntPtr
        If LogonUser("Israel Aece", String.Empty, "P@$$w0rd",
2, 0, token) Then
            Console.WriteLine("Usuário válido.")
        Else
            Console.WriteLine("Usuário inválido.")
        End If
    End Sub
End Class
```

Como podemos notar, a função deve receber os mesmos parâmetros da função não gerenciada e, a forma de acesso a mesma dentro do método *Main* é exatamente a mesma. Para finalizar, essa forma de acesso é interessante, porém possui apenas um subconjunto de configurações para a chamada da função não gerenciada. Se desejar ter uma maior controle nas configurações, utilize o atribute **DllImportAttribute**.

Passagem de Parâmetros

Como comentado acima, quando invocamos uma função de código não gerenciado, os parâmetros e seus valores de retorno (quando existem) devem ser convertidos em parâmetros correspondentes ao mundo não gerenciado. Esse processo, como já sabemos, chama-se *mashalling*.

A maioria dos tipos de dados da plataforma .NET são convertidos sem nenhum problema para o mundo gerenciado, pois há sempre um tipo correspondente. Se desejar visualizar uma tabela completa com o mapeamente de tipos entre código gerenciado e não

gerenciado, consulte este link: http://msdn2.microsoft.com/en-us/library/ac7ay120.aspx (*MSDN Library*).

As exceções são para tipos genéricos, estruturas e objetos. Tipos genéricos não são suportados e estruturas ou objetos exigem informações adicionais para essa conversão. Geralmente o CLR controla o layout físico dos campos de uma estrutura ou objeto dentro da memória gerenciada. Se essa estrutura ou classe precisa ser ajustado de alguma forma, você deve decorar a estrutura ou objeto com o atributo **StructLayoutAttribute**. Esse atributo permite você controlar explicitamente o *layout* físico dos campos de uma estrutura ou objeto que serão passados para o código não gerenciado que, por sua vez, espera em um layout específico. Esse atributo fornece um construtor sobrecarregado que aceita como parâmetro uma das opções fornecidas pelo enumerador **LayoutKind**. As opções fornecidas por esse enumerador estão descritas logo abaixo:

Opção	Descrição
Auto	O *runtime* automaticamente escolhe um *layout* apropriado para os membros de um determinado objeto dentro da memória não gerenciada.
Explicit	Especifica que cada de cada membro dentro da memória não gerenciada é explicitamente controlada. Para isso, cada membro deve ser marcada com o atributo **FieldOffsetAttribute** para indicar a posição do campo dentro do tipo.
Sequential	Indica que os membros de um determinado objeto serão expostos de forma seqüencial, na ordem em que eles aparecem quando eles são exportados para a memória não gerenciada.

Abaixo é exibido um exemplo de como utilizar este atributo:

```
VB.NET
Imports System
Imports System.Runtime.InteropServices

<StructLayout(LayoutKind.Sequential)> _
Public Structure Point
   Public X As Integer
   Public Y As Integer
End Structure

C#
using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
public struct Point
{
   public int X;
   public int Y;
```

```
}
```

Apesar do *marshaling* ser um processo que acontece de forma transparente para os demais tipos de dados, há a possibilidade de customizarmos essa conversão para que problemas com incompatibilidade de tipos não aconteça.

Para essa customização, temos a disposição o atributo **MarshalAsAttribute** que permite você mudar o comportamento padrão da conversão de tipos (campos, parâmetros ou valores de retornos) e especificar um tipo mais específico. No entanto esse atributo somente é necessário quando um tipo pode ser convertido em vários outros tipos.

Um exemplo disso é a conversão de uma *string* em código não gerenciado. Ela pode ser convertida em vários tipos: **LPStr**, **LPWStr**, **LPTStr** ou **Bstr**. Por padrão, ela é convertida em **Bstr** para métodos COM.

O atributo **MarshalAsAttribute** possui um construtor sobrecarregado que aceita uma das opções fornecidas pelo enumerador **UnmanagedType**. Esse enumerador determinada como os tipos serão convertido para o código não gerenciado. Suas opções são os tipos de dados não gerenciados disponíveis que podemos escolher para mapear um tipo gerenciado em um tipo não gerenciado, mudando assim o comportamento padrão. O código abaixo mostra um exemplo do uso deste atributo em um parâmetro de método e em um campo público:

VB.NET

```
Imports System
Imports System.Runtime.InteropServices

<DllImport("Componente.dll")> _
Private                        Shared                          Function
FuncaoTeste(<MarshalAs(UnmanagedType.LPStr)>  ByVal  s  As  String)
As  Boolean
End Function

'----

<MarshalAs(UnmanagedType.LPStr)> _
Public Silly As Nome
```

C#

```
using System;
using System.Runtime.InteropServices;

[DllImport("Componente.dll"]
private                static                extern                bool
FuncaoTeste([MarshalAs(UnmanagedType.LPStr)] string s)
```

```
//----

[MarshalAs(UnmanagedType.LPStr)]
public string Nome;
```

A classe Marshal

A classe **Marshal** fornece uma coleção de estáticos métodos para interagir e manipular o código não gerenciado. Esses metódos estáticos fornecidos pela classe **Marshal** são essenciais para trabalhar com código não gerenciado.

Muitos desses métodos que estão definidos nesta classe são tipicamente utilizados por desenvolvedores que precisam fornecer/criar uma ponte entre os mundos gerenciados e não gerenciados.