Capítulo 7
Globalização de Aplicações

Introdução

Atualmente é muito comum as empresas estarem expandindo seus negócios entre vários países do mundo. Além disso, ainda há casos onde pessoas estrangeiras são comumente contratados por essas empresas. Com isso, o software ou até mesmo o website da empresa, deverá contemplar vários idiomas.

Para possibilitar isso, as aplicações devem ser capazes de configurar o padrão e formatação dos números (isso inclui o sistema monetário) e datas e, principalmente, serem capazes de ajustar a *interface* com usuário, pois em determinados idiomas, podem exigir mais espaço para uma determinada informação e, se a aplicação não previnir isso, poderá deixar a tela inutilizável.

Neste capítulo veremos detalhadamente como devemos proceder para configurarmos uma aplicação para que a mesma suporte as funcionalidades de globalização, analisando as classes e tipos fornecidos pelo *namespace* **System.Globalization**.

Globalização e Localização

Quando estamos falando de aplicações para múltiplos idiomas e países, temos que entender dois aspectos muito importantes que são a globalização e localização:

- **Globalização:** O processo de globalização é a habilidade de construir aplicações/websites que são suportadas e adaptáveis para as mais diferentes culturas.
- **Localização:** É a habilidade de localizar a aplicação para uma cultura e região específica, criando traduções para os recursos que a aplicação utiliza em seu interior. Um exemplo típico é a localização de uma aplicação/website para o português para várias regiões, como o Brasil (*pt-BR*) e Portugal (*pt-PT*).

Como podemos notas na descrição acima, as culturas são identificadas por um padrão universal, contendo duas partes, sendo a primeira delas dois caracteres minúsculos que identificam o idioma. Já na segunda parte, existem mais dois caracteres maiúsculos que representam o país.

Existem uma enorme lista com todos as culturas suportadas pelo .NET Framework, mas que não será exibida aqui por questões de espaço. Mas se desejar consultar, basta localizar a classe **CultureInfo** no *MSDN Library* local ou no site e lá terá a lista completa.

Quando trabalhamos com culturas, temos dois conceitos importantes que devemos levar em consideração. Trata-se da cultura corrente e da cultura de interface (*uiculture*). A

primeira delas, é utilizada para operarmos com formatação de datas e números e, além disso, utilizada durante a escrita do código; já a segunda, é utilizada pelo *Resource Manager* para analisar uma cultura específica e recuperar os recursos em tempo de execução. Para definirmos cada uma dessas culturas, utilizamos as propriedades *CurrentCulture* e *CurrentUICulture*, respectivamente. Essas propriedades são estáticas e estão contidas dentro da classe **Thread** que, por sua vez, está dentro do *namespace* **System.Threading**.

Utilizando Culturas

Uma das classes essencias na criação de aplicações globalizadas, cada instância da classe **CultureInfo** representa uma cultura específica, contendo informações específicas da cultura que ela representa e, entre essas informações, temos o nome da cultura, sistema de escrita, calendários, como formatar datas, etc.. A tabela abaixo exibe os principais métodos e propriedades desta classe.

Membro	**Descrição**
Calendar	Propriedade de somente leitura que retorna um objeto do tipo **Calendar**. O objeto **Calendar** representa as divisões do tempo, como semanas, meses e anos.
ComparerInfo	Propriedade de somente leitura que retorna um objeto do tipo **CompararInfo** que define como comparar as *strings* para a cultura corrente.
CultureTypes	Propriedade de somente leitura que retorna uma combinação do enumerador **CultureTypes**, indicando à que tipo a cultura corrente pertence. Entre as opções fornecidas pelo enumerador **CultureTypes**, temos: • **AllCultures –** Todas as culturas, incluindo as culturas que fazem parte do .NET Framework (neutras e específicas), culturas instaladas no Windows e as culturas customizadas, criadas pelo usuário. • **FrameworkCultures –** Culturas específicas e neutras que fazem parte do .NET Framework. • **InstalledWin32Cultures –** Todas as culturas instaladas dentro do Windows. • **NeutralCultures –** Culturas que estão associadas com um idioma mas não com uma região/país específico. • **ReplacementCultures –** Culturas customizadas pelo usuário que substituem as culturas disponibilizadas pelo .NET Framework. • **SpecificCultures –** Culturas que não são específicas para uma região/país. • **UserCustomCulture –** Culturas customizadas, criadas pelo usuário.

	• **WindowsOnlyCultures** – Somente as culturas instaladas dentro do Windows e não fazem parte do .NET Framework.
CurrentCulture	Propriedade estática de somente leitura que retorna um objeto do tipo **CultureInfo** que está sendo utilizada pela *thread* corrente. Essa propriedade nada mais é que um wrapper para a propriedade estática *CurrentCulture* da classe **Thread**.
CurrentUICulture	Propriedade estática de somente leitura que retorna um objeto do tipo **CultureInfo** que está sendo utilizada pelo Resource Manager para extrair os recursos em tempo de execução. Essa propriedade nada mais é que um wrapper para a propriedade estática *CurrentUICulture* da classe **Thread**.
DateTimeFormat	Propriedade de somente leitura que retorna um objeto do tipo **DateTimeFormatInfo** que define as formas apropriadas para exibir e formatar datas e horas para a cultura corrente.
DisplayName	Propriedade de somente leitura que retorna uma *string* com o nome da cultura no formato *full*, exemplo: *en-US*.
EnglishName	Propriedade de somente leitura que retorna uma *string* com o nome da cultura em inglês.
InstalledUICulture	Propriedade estática de somente leitura que retorna um objeto do tipo **CultureInfo** que representa a cultura instalada com o sistema operacional.
IsNeutralCulture	Propriedade de somente leitura que retorna um valor booleano indicando se o objeto **CultureInfo** corrente representa uma cultura neutra.
IsReadOnly	Propriedade de somente leitura que retorna um valor booleano indicando se o objeto **CultureInfo** corrente é ou não somente leitura.
Name	Propriedade de somente leitura que retorna uma *string* contendo o nome da cultura corrente no seguinte formato: *English (United States)*.
NativeName	Propriedade de somente leitura que retorna uma *string* contendo o nome da cultura corrente em seu idioma atual: *English (United States)*.
NumberFormat	Propriedade de somente leitura que retorna um objeto do tipo **NumberFormatInfo** que define as formas apropriadas para exibir e formatar números (inclusive o sistema monetário, porcentagens) para a cultura corrente.
Parent	Propriedade de somente leitura que retorna um objeto do tipo **CultureInfo** que representa a cultura "pai" da cultura corrente.
TextInfo	Propriedade de somente leitura que retorna um objeto do tipo **TextInfo** que define a forma de escrita associada com a cultura corrente.
UseUserOverride	Propriedade de somente leitura que retorna um valor booleano indicando se o objeto **CultureInfo** corrente utiliza as opções de

	culturas definidas pelo usuário através das Configurações Regionais do Painel de Controle do Windows.
CreateSpecificCulture	Método estático que, dado uma cultura específica, cria e retorna um objeto do tipo **CultureInfo** associado com a cultura informada.
GetCultureInfo	Método estático que, dado uma cultura específica, retorna uma instância do objeto **CultureInfo** (*read-only*) associado com a cultura informada.
GetCultures	Método estático que retorna um *array* de culturas, onde cada um dos elementos é representado por um objeto do tipo **CultureInfo**.
GetFormat	Este método, através de um objeto do tipo **Type**, retorna uma instância de um formatador associadao com a cultura corrente. Esse método somente aceita como parâmetro um objeto **Type** que representa a classe **NumberFormatInfo** ou a classe **DateTimeFormatInfo**. Do contrário, esse método retornará nulo.

O código abaixo exibe a forma de criação e a exibição de algumas das propriedades do objeto CultureInfo:

VB.NET
```
Imports System.Globalization

Sub Main()
    Dim pt As New CultureInfo("pt-BR")
    Dim           en           As           CultureInfo           =
CultureInfo.CreateSpecificCulture("en-US")
    Show(New CultureInfo() {pt, en})
End Sub

Private Sub Show(ByVal cultures() As CultureInfo)
    For Each ci As CultureInfo In cultures
        Console.WriteLine("------------------------")
        Console.WriteLine(ci.DisplayName)
        Console.WriteLine(ci.DateTimeFormat.DateSeparator)

Console.WriteLine(ci.DateTimeFormat.FirstDayOfWeek.ToString())

Console.WriteLine(ci.NumberFormat.CurrencyDecimalSeparator)
    Next
End Sub
```

C#
```
using System.Globalization;
```

```
static void Main(string[] args)
{
    CultureInfo pt = new CultureInfo("pt-BR");
    CultureInfo en = CultureInfo.CreateSpecificCulture("en-US");
    Show(new CultureInfo[] { pt, en });
}

static void Show(CultureInfo[] cultures)
{
    foreach (CultureInfo ci in cultures)
    {
        Console.WriteLine("------------------------");
        Console.WriteLine(ci.DisplayName);
        Console.WriteLine(ci.DateTimeFormat.DateSeparator);

Console.WriteLine(ci.DateTimeFormat.FirstDayOfWeek.ToString());

Console.WriteLine(ci.NumberFormat.CurrencyDecimalSeparator);
    }
}
```

A única diferença entre a criação do objeto *pt* e *en* é que no primeiro, *pt*, foi criado a partir da instância da classe **CultureInfo**; já com o *en*, optamos por criar a partir do método estático *CreateSpecificCulture* da classe **CultureInfo**. O resultado para ambos os códigos são idênticos e é exibido abaixo:

```
------------------------
Portuguese (Brazil)
/
Sunday
,
------------------------
English (United States)
/
Sunday
.
```

Recuperando informações de uma região (país)

Existe uma classe dentro do *namespace* **System.Globalization** chamada **RegionInfo**. Ao contrário da classe **CultureInfo**, ela não representa as preferências do usuário e não depende do idioma ou cultura do mesmo. A classe **RegionInfo** fornece informações referente a uma região/país especifíco.

Essa classe contém um *overload* que recebe uma *string*. Essa *string* deve conter o nome da região que você deseja recuperar as informações e, esse nome, deve ser dois caracteres maiúsculos de acordo com o padrão estabelecido pela ISO. A tabela completa pode ser consultada quando você abre a documentação da classe **RegionInfo**. Entre as principais propriedades desta classe tempos (com os exemplos baseados em uma instância da classe **RegionInfo** que representa o Brasil):

Propriedade	Descrição
CurrencyEnglishName	Propriedade de somente leitura que retorna uma *string* contendo o nome, em inglês, da moeda utilizada pela região corrente. Exemplo: *Real*
CurrencyNativeName	Propriedade de somente leitura que retorna uma *string* contendo o nome no idioma nativo da região corrente. Exemplo: *Real*
CurrencySymbol	Propriedade de somente leitura que retorna uma *string* contendo o símbolo monetário utilizado pela região corrente. Exemplo: *R$*
CurrentRegion	Propriedade estática de somente leitura que retorna uma instância da classe **RegionInfo** representando a região da *thread* corrente.
DisplayName	Propriedade de somente leitura que retorna uma *string* contendo o nome da região corrente no idioma localizado do .NET Framework. Exemplo: *Brazil*
EnglishName	Propriedade de somente leitura que retorna uma *string* contendo o nome, em inglês, da região corrente. Exemplo: *Brazi*
Name	Propriedade de somente leitura que retorna uma *string* com o nome da região corrente. Esse nome é representado por dois caracteres maiúsculos. Exemplo: *BR*
NativeName	Propriedade de somente leitura que retorna uma *string* contendo o nome da região corrente em seu idioma nativo. Exemplo: *Brasil*.

Logo abaixo temos um exemplo da utilização desta classe:

```
VB.NET
Imports System.Globalization

Dim r As New RegionInfo("BR")
Console.WriteLine(r.CurrencyNativeName)
Console.WriteLine(r.CurrencySymbol)
Console.WriteLine(r.NativeName)

C#
using System.Globalization;

RegionInfo r = new RegionInfo("BR");
```

```
Console.WriteLine(r.CurrencyNativeName);
Console.WriteLine(r.CurrencySymbol);
Console.WriteLine(r.NativeName);
```

Formatação de Datas e Números

Em alguns momentos acima vimos mencionados as classes **DateTimeFormatInfo** e **NumberFormatInfo**. Essas classes fornecem uma grande funcionalidade, que é a formatação e padronização de datas, horas e números dentro da plataforma .NET. Essas classes fornecem informações específicas para a manipulação de datas, horas e números em diferentes regiões e, como vimos, a classe **CultureInfo** disponibiliza propriedades que retornam a instância dessas respectivas classes já com a cultura especificada mas, nada impede que você crie a sua própria instância e customize as suas propriedades necessárias.

A classe DateTimeFormatInfo

Como falamos acima, a classe **DateTimeFormatInfo** irá auxiliar na formatação de data e hora de acordo com a cultura selecionada. Todas as propriedades que essa classe possui já refletem as informações da cultura específica quando você extrai a instância dessa classe a partir do método *GetFormat* ou da propriedade *DateTimeFormat* da classe **CultureInfo**.

Essa classe também suporta um padrão pré-determinado, onde podemos especificar alguns caracteres que determinam a formatação da data/hora que será exibido. Alguns desses especificadores são mostrados através da tabela abaixo:

Especificador	Descrição
d	Especifica um padrão para a exibição de uma data de uma forma reduzida. É um atalho para a propriedade *ShortDatePattern* da classe **DateTimeFormatInfo**.
D	Especifica um padrão para a exibição de uma data de uma forma mais completa. É um atalho para a propriedade *LongDatePattern* da classe **DateTimeFormatInfo**.
f	Especifica um padrão para a exibição de uma hora de uma forma reduzida. É um atalho para a propriedade *ShortTimePattern* da classe **DateTimeFormatInfo**.
F	Especifica um padrão para a exibição de uma hora de uma forma mais completa. É um atalho para a propriedade *LongTimePattern* da classe **DateTimeFormatInfo**.
t	Exibe uma combinação da data em seu formato completo e a hora em sua forma reduzida.
T	Exibe uma combinação da data em seu formato completo e a hora em sua forma completa. É um atalho para a propriedade

	FullDateTimePattern da classe **DateTimeFormatInfo**.

Além dos padrões que vimos na tabela acima, ainda há a possibilidade de, com esses mesmos caracteres, combiná-los para customizarmos como a data/hora será exibida.

A estrutura **DateTime** fornece alguns métodos que serve como *wrapper* para as propriedades que citamos acima e, além disso, o método *ToString* desta estrutura possui alguns *overloads* que permitem customizarmos a formatação. Para testarmos as formatações acima, vamos utilizá-la no exemplo a seguir:

```
VB.NET
Dim hoje As DateTime = DateTime.Now
Console.WriteLine(hoje.ToString("d"))
Console.WriteLine(hoje.ToString("D"))
Console.WriteLine(hoje.ToString("f"))
Console.WriteLine(hoje.ToString("F"))
Console.WriteLine(hoje.ToString("t"))
Console.WriteLine(hoje.ToString("T"))
Console.WriteLine(hoje.ToString("dd/MM/yyyy"))

C#
DateTime hoje = DateTime.Now;
Console.WriteLine(hoje.ToString("d"));
Console.WriteLine(hoje.ToString("D"));
Console.WriteLine(hoje.ToString("f"));
Console.WriteLine(hoje.ToString("F"));
Console.WriteLine(hoje.ToString("t"));
Console.WriteLine(hoje.ToString("T"));
Console.WriteLine(hoje.ToString("dd/MM/yyyy"));
```

Como resultado obtemos:

```
31/3/2007
sábado, 31 de março de 2007
sábado, 31 de março de 2007 18:23
sábado, 31 de março de 2007 18:23:13
18:23
18:23:13
31/03/2007
```

Na última linha do exemplo, utilizamos um *overload* do método *ToString* que permite passarmos uma combinação de alguns caracteres que permite-nos customizar a formatação da data/hora. No exemplo MM significa que se trata de mês com duas casas. Atente-se, pois mm (minúsculos) trata-se de minutos.

Quando fazemos a formatação da forma acima, por padrão, o **DateTimeFormatInfo** que é utilizado é extraído através da propriedade estática *CurrentInfo* da mesma. Vale lembrar que essa propriedade nada mais é que um *wrapper* para a propriedade *CurrentCulture* da *thread* corrente. Mas podemos criar instâncias da nossa própria classe **DateTimeFormatInfo** para customizarmos como a data/hora será tratada. Antes de analisarmos como devemos proceder para utilizá-la, vamos entender um pouco melhor algumas das propriedades que ela nos fornece através da tabela abaixo:

Membro	**Descrição**
AbbreviateDayNames	Propriedade que retorna um *array* de *strings*, onde cada elemento do mesmo corresponde a um dia em sua forma abreviada.
CurrentInfo	Propriedade estática de somente leitura que retorna um objeto do tipo **DateTimeFormatInfo** da *thread* atual.
DateSeparator	Uma *string* que determina qual será o separador das datas.
DayNames	Propriedade que retorna um *array* de *strings*, onde cada elemento do mesmo corresponde a um dia com o seu nome completo.
FirstDayOfWeek	Propriedade que podemos definir qual será o primeiro dia da semana. Essa propriedade definimos com alguma opção do enumerador **DayOfWeek**. As opções fornecidas por esse enumerador são: • **Friday –** Indica sexta-feira. • **Saturday –** Indica sábado. • **Sunday –** Indica domingo. • **Monday –** Indica segunda-feira. • **Thursday –** Indica terça-feira. • **Tuesday –** Indica quinta-feira. • **Wednesday –** Indica quarta-feira.
FullDateTimePattern	Propriedade onde definimos o formato da data e hora em seu formato longo. Este padrão está associado ao caracter “F” que vimos acima.
MonthDayPattern	Propriedade onde definimos o formato do dia e mês, que estão associados com os caracteres “d” e “M”.
MonthNames	Propriedade que retorna um *array* de *strings*, onde cada elemento do mesmo corresponde a um mês com o seu nome completo.
TimeSeparator	Uma *string* que determina qual será o separador de horas.
YearMonthPattern	Propriedade onde definimos o formato do ano e mês, que estão associados com os caracteres “y” e “Y”.

Como dissemos acima, podemos criar uma instância dessa classe customizarmos como desejarmos. Vemos mais utilidade nisso quando estamos criando uma cultura customizada, que analisaremos mais adiante. Abaixo está a forma que devemos proceder para a criação deste objeto:

```
VB.NET
Dim dtfi As New DateTimeFormatInfo()
dtfi.DateSeparator = "|"
dtfi.TimeSeparator = "."
Console.WriteLine(DateTime.Now.ToString(dtfi))

C#
DateTimeFormatInfo dtfi = new DateTimeFormatInfo();
dtfi.DateSeparator = "|";
dtfi.TimeSeparator = ".";
Console.WriteLine(DateTime.Now.ToString(dtfi));
```

É importante notar que um dos *overloads* do método *ToString* da estrutura **DateTime** recebe um tipo **IFormatProvider** e, como a classe **DateTimeFormatInfo** implementa essa *Interface*, ela é pode ser passada para o mesmo. O resultado do código é exibido abaixo:

```
04|02|2007 10.19.49
```

A classe NumberFormatInfo

Assim como a classe **DateTimeFormatInfo**, a classe **NumberFormatInfo** é responsável por tratar da formatação de números dentro da plataforma .NET, ainda estendendo para o sistema monetário da região corrente.

Essa classe também fornece alguns caracteres, com padrões pré-determinados que podemos utilizar para customizar a formatação de um determinado valor. Além disso, ela fornece também propriedades que podemos definir os símbolos, separadores decimais, etc. Através da tabela abaixo, vamos analisar os caracteres que temos disponíveis para a formatação de valores numéricos:

Caracter	Descrição
c, C	Formato monetário.
d, D	Formato decimal.
e, E	Formato científico (exponencial).
f, F	Formato fixo.
g, G	Formato padrão.

n, N	Formato numérico.
r, R	Formato *roundtrip*. Esse formato assegura que os números convertidos em *strings* terão o mesmo valor quando eles forem convertidos de volta para números.
x, X	Formato hexadecimal.

Com o conhecimento desses caracteres, já é possível utilizá-los no *overload* do método *ToString* da estrutura **Double**, onde podemos determinar qual a forma que o valor contido dentro dela será exibido. Além do caracter, ainda é possível determinar a quantidade de casas decimais que será exibido, usando em conjunto com o caracter de formatação um número que irá informar quantas casas decimais deverá ter:

```
VB.NET
Dim valor As Double = 123.2723
Console.WriteLine(valor.ToString("C2"))
Console.WriteLine(valor.ToString("N3"))

C#
Double valor = 123.2723;
Console.WriteLine(valor.ToString("C2"));
Console.WriteLine(valor.ToString("N3"));
```

O resultado desse código é mostrado abaixo:

```
R$ 123,27
123,272
```

Mais uma vez, é importante dizer que quando utilizamos as formatações dessa forma, apesar de explicitamente não estarmos utilizando a classe **NumberFormatInfo**, ela é extraída automaticamente da *thread* corrente e, conseqüentemente, formatando os valores baseando-se nessas informações.

Para conhecermos um pouco mais sobre a classe **NumberFormatInfo**, vamos analisar algumas das propriedades que elas nos fornece através da tabela abaixo:

Propriedade	**Descrição**
CurrencyDecimalDigits	Propriedade que recebe um número inteiro indicando quantas casas decimais é utilizada em valores monetários.
CurrencyDecimalSeparator	Propriedade que recebe uma *string* contendo o caracter que será utilizado como separador de casas decimais.
CurrencyGroupSeparator	Propriedade que recebe uma *string* contendo o caracter que será utilizado como separador dos grupos de números. É

	utilizado entre os grupos de números.
CurrencyGroupSizes	Propriedade que recebe um *array* de números inteiros que representam qual será o comprimento de cada grupo.
CurrentInfo	Propriedade estática de somente leitura que retorna um objeto do tipo **NumberFormatInfo** da *thread* atual.
NegativeSign	Propriedade que recebe uma *string* contendo o caracter que será associado ao número quando ele for negativo.
NumberDecimalDigits	Propriedade que recebe um número inteiro indicando quantas casas decimais é utilizada em números em geral.
NumberDecimalSeparator	Propriedade que recebe uma *string* contendo o caracter que será utilizado como separador de casas decimais para números em geral.
NumberGroupSeparator	Propriedade que recebe uma *string* contendo o caracter que será utilizado como separador dos grupos de números. É utilizado entre os grupos de números.
NumberGroupSizes	Propriedade que recebe um *array* de números inteiros que representam qual será o comprimento de cada grupo.
PositiveSign	Propriedade que recebe uma *string* contendo o caracter que será associado ao número quando ele for positivo.

Como dissemos acima, podemos criar uma instância dessa classe customizarmos como desejarmos. Vemos mais utilidade nisso quando estamos criando uma cultura customizada, que analisaremos mais adiante. Abaixo está a forma que devemos proceder para a criação do objeto **NumberFormatInfo**:

```
VB.NET
Dim nfi As New NumberFormatInfo()
nfi.CurrencyDecimalDigits = 3
nfi.CurrencyDecimalSeparator = "*"
nfi.CurrencyGroupSeparator = "_"
nfi.CurrencyGroupSizes = New Integer(1) { 2 }
nfi.CurrencySymbol = "Dinheiro do Brasil "

Dim d As Double = 8789282212.9384738747
Console.WriteLine(d.ToString("C", nfi))

C#
NumberFormatInfo nfi = new NumberFormatInfo();
nfi.CurrencyDecimalDigits = 3;
nfi.CurrencyDecimalSeparator = "*";
nfi.CurrencyGroupSeparator = "_";
nfi.CurrencyGroupSizes = new int[1] { 2 };
nfi.CurrencySymbol = "Dinheiro do Brasil ";

Double d = 8789282212.9384738747;
Console.WriteLine(d.ToString("C", nfi));
```

O resultado desse código é mostrado abaixo:

```
Dinheiro do Brasil 87_89_28_22_12*938
```

Criando uma cultura customizada

Como vimos até o momento, utilizamos as culturas definidas pelo próprio .NET Framework, como é o caso de *pt-BR*, *en-US* ou *pt-PT*. Essas culturas satisfazem a maior parte das aplicações que necessitam serem globalizadas. Só que pode haver cenário onde é necessário criarmos uma cultura própria, customizando-a para que atenda ao problema pelo qual ela foi criada.

Felizmente, como o .NET Framework é estensível, ele fornece uma classe chamada **CultureAndRegionInfoBuilder** que permite a criação de uma cultura customizada de forma bem simples e rápida, sem a necessidade de aplicar o conceito de herança. Ela, por sua vez, contém várias propriedades e métodos que nos auxiliam na criação dessa cultura customizada e que é importante analisá-los para saber qual a sua finalidade:

Membro	**Descrição**
CompareInfo	Define um objeto do tipo **CompareInfo** que define como as *strings* serão comparadas com essa cultura.
CultureName	Propriedade de somente leitura que retorna o nome da cultura.
GregorianDateTimeFormat	Define um objeto do tipo **DateTimeFormatInfo** que define como as datas e horas são tratadas com essa cultura.
NumberFormat	Define um objeto do tipo **NumberFormatInfo** que define como os números (e valores monetários) são tratados com essa cultura.
LoadDataFromCultureInfo	Método que recebe como parâmetro um objeto do tipo **CultureInfo** que carrega as propriedades do objeto corrente com as propriedades correspondentes da cultura informada.
LoadDataFromRegionInfo	Método que recebe como parâmetro um objeto do tipo **RegionInfo** que carrega as propriedades do objeto corrente com as propriedades correspondentes da região informada
Register	Persiste a cultura criada como uma cultura customizada no computador local e a disponibiliza para as aplicações.
Save	Permite persistir a cultura criada em um arquivo físico, em formato XML para uso futuro.
Unregister	Método estático que, dado uma *string* contendo o nome da cultura customizada, ele exclui a mesma do computador local.

Para exemplificar a criação desa cultura customizada, vamos analisar o código abaixo:

```
VB.NET
Dim cultureKey As String = "pt-BRCustom"
Dim cst As Newew CultureAndRegionInfoBuilder( _
        cultureKey, _
        CultureAndRegionModifiers.None)

cst.LoadDataFromCultureInfo(New CultureInfo("pt-BR"))
cst.LoadDataFromRegionInfo(New RegionInfo("pt-BR"))
cst.NumberFormat.CurrencyDecimalDigits = 4
cst.NumberFormat.CurrencyDecimalSeparator = "*"
cst.Register()

Dim pt As New CultureInfo(cultureKey)
Dim valor As Double = 8789282212.9384738747
Console.WriteLine(valor.ToString("C", pt.NumberFormat))
CultureAndRegionInfoBuilder.Unregister(cultureKey)

C#
string cultureKey = "pt-BRCustom";
CultureAndRegionInfoBuilder cst =
    new CultureAndRegionInfoBuilder(
        cultureKey,
        CultureAndRegionModifiers.None);

cst.LoadDataFromCultureInfo(new CultureInfo("pt-BR"));
cst.LoadDataFromRegionInfo(new RegionInfo("pt-BR"));
cst.NumberFormat.CurrencyDecimalDigits = 4;
cst.NumberFormat.CurrencyDecimalSeparator = "*";
cst.Register();

CultureInfo pt = new CultureInfo(cultureKey);
double valor = 8789282212.9384738747;
Console.WriteLine(valor.ToString("C", pt.NumberFormat));
CultureAndRegionInfoBuilder.Unregister(cultureKey);
```

O resultado desse código é mostrado abaixo:

```
R$ 8.789.282.212*9385
```


Nota: Apesar da classe **CultureAndRegionInfoBuilder** estar contida dentro do *namespace* **System.Globalization**, é necessário adicionar uma referência ao *Assembly* **sysglobl.dll** na aplicação que desejar utilizá-la.

Encoding

A codificação de caracteres é a forma que temos de representar caracteres em uma seqüencia de *bits*. Cada caracter, depois de codificado, é representado por um código único (também chamado de *code point*) dentro de uma *code page*. Uma *code page* é uma lista de *code points* em uma certa ordem e é utilizada para suportar idiomas específicos ou grupos de idiomas que compartilham o mesmo sistema de escrita. As *code pages* do Windows contém 256 *code points* (baseado em zero: 0 – 255). Na maioria das *code pages*, os primeiros 127 *code points* são sempre os mesmos caracteres para permitir a compatibilidade com o legado. Os 128 *code points* restantes diferem entre as *code pages*.

Existem vários padrões de codificação disponíveis para representar os caracteres nos mais diversos idiomas. Inicialmente o padrão **ASCII**, que é baseado no alfabeto inglês foi desenvolvido pela *IBM*. Esse padrão utiliza 7 *bits* mas como há idiomas que possuem vários outros caracteres e, conseqüentemente, esse padrão não suportaria todos os idiomas. Neste momento é introduzido o padrão **Unicode**, que possibilita 8 bits para os caracteres. Além de suportar os caracteres definidos pelo **ASCII**, ele também suporta todos os caracteres conhecidos usados nos mais diversos idiomas. Atualmente, temos 3 "versões" do padrão **Unicode**: **UTF-8**, **UTF-16** e **UTF-32**. O que diferem nestes padrões, é a quantidade de *bytes* utilizados para armazenar os caracteres: o primeiro utiliza 1 *byte*, o segundo 2 *bytes* e o último 4 *bytes*.

O .NET Framework suporta esses padrões que podemos, através de classes, utilizá-los em nossas aplicações. As classes para isso estão contidas dentro do *namespace* **System.Text** e, uma das princpais delas é a classe **Encoding**. O .NET Framework fornece as seguintes implementações da classe **Encoding** para suportar alguns dos padrões existentes atualmente:

Classe	**Descrição**
ASCIIEncoding	Codifica os caracteres baseando-se no padrão **ASCII**. Essa classe corresponde ao *code page* 20127. Para criá-la basta criar uma instância da mesma ou chamar a propriedade estática *ASCII* da classe **Encoding** que já retornará uma instância dessa classe.
UTF7Encoding	Codifica os caracteres baseando-se no padrão **UTF-7**. Essa classe corresponde ao *code page* 65000. Para criá-la basta criar uma instância da mesma ou chamar a propriedade estática *UTF7* da classe **Encoding** que já retornará uma instância dessa classe.
UTF8Encoding	Codifica os caracteres baseando-se no padrão **UTF-8**. Essa classe corresponde ao *code page* 65001.

	Para criá-la basta criar uma instância da mesma ou chamar a propriedade estática *UTF8* da classe **Encoding** que já retornará uma instância dessa classe.
UnicodeEncoding	Codifica os caracteres baseando-se no padrão **UTF-16**. Esse padrão podem utilizar uma espécie de ordenação (*little-endian* e *big-endian*), onde cada uma delas representam um *code page* diferente. A primeira deles equivale ao *code page* 1200 e a segunda ao *code page* 1201. Para criá-la basta criar uma instância da mesma ou chamar a propriedade estática *Unicode* da classe **Encoding** que já retornará uma instância dessa classe.
UTF32Encoding	Codifica os caracteres baseando-se no padrão **UTF-32**. Esse padrão podem utilizar uma espécie de ordenação (*little-endian* e *big-endian*), onde cada uma delas representam um *code page* diferente. A primeira deles equivale ao *code page* 65005 e a segunda ao *code page* 65006. Para criá-la basta criar uma instância da mesma ou chamar a propriedade estática *UTF32* da classe **Encoding** que já retornará uma instância dessa classe.

Nota: Para uma referência completa de todas as *code pages* e seus respectivos códigos, sugiro consultar a documentação da classe **Encoding** do *MSDN Library*.

Além das propriedade estáticas que vimos acima que a classe Encoding fornece, ainda existem algumas outras propriedades e métodos importantes e que merecem serem citados. A tabela abaixo descreve alguns desses membros:

Membro	**Descrição**
BodyName	Propriedade de somente leitura que retorna uma *string* contendo o nome do codificador corrente.
CodePage	Propriedade de somente leitura que retorna um número inteiro contendo o identificar do codificar corrente.
Default	Propriedade estática de somente leitura que retorna um objeto do tipo **Encoding** representando o codificador corrente do sistema.
EncodingName	Propriedade de somente leitura que retorna uma *string* contendo a descrição (em forma legível) do codificador corrente.
Convert	Método estático que converte um *array* de *bytes* de um codificador para outro.
GetBytes	Método que retorna um *array* de *bytes* contendo o resultado da codificação dos caracteres passado para o método.
GetDecoder	Método que retorna um objeto do tipo **Decoder**, que é responsável por converter uma determinada seqüencia de *bytes* em uma seqüencia de caracteres.

GetEncoder	Método que retorna um objeto do tipo **Encoder**, que é responsável por converter uma determinada seqüencia de caracteres em uma seqüencia de *bytes*.
GetPreamble	Método que retorna um array de bytes que irá identificar se o codificador suporta ordenação dos *bytes* em um formato *big-endian* ou *little-endian*.
GetString	Método que, dado um *array* de *bytes*, ele decodifica e retorna uma *string* contendo o seu valor legível.

Para demonstrar a utilização de duas dessas classes de codificação fornecidas pelo .NET Framework (**ASCIIEncoding** e **UnicodeEncoding**), vamos analisar o código abaixo que, dado uma mensagem, ele recupera os bytes do mesma e, em seguida, traz em seu formato original.

VB.NET

```
Imports System.Text

Sub Main()
    Dim msg As String = "Codgificação - .NET Framework"

    Dim ascii As New ASCIIEncoding
    Dim unicode As New UnicodeEncoding

    Dim asciiBytes() As Byte = ascii.GetBytes(msg)
    Dim unicodeBytes() As Byte = unicode.GetBytes(msg)
    ShowBytes(asciiBytes)
    ShowBytes(unicodeBytes)

    Dim asciiMsg As String = ascii.GetString(asciiBytes)
    Dim unicodeMsg As String = unicode.GetString(unicodeBytes)

    Console.WriteLine(Environment.NewLine)
    Console.WriteLine(asciiMsg)
    Console.WriteLine(unicodeMsg)
End Sub

Sub ShowBytes(ByVal msg() As Byte)
    Console.WriteLine(Environment.NewLine)
    For Each b As Byte In msg
        Console.Write("[{0}]", b)
    Next
End Sub
```

C#

```
using System.Text;

static void Main(string[] args)
```

```
{
    string msg = "Codificação - .NET Framework";

    ASCIIEncoding ascii = new ASCIIEncoding();
    UnicodeEncoding unicode = new UnicodeEncoding();

    byte[] asciiBytes = ascii.GetBytes(msg);
    byte[] unicodeBytes = unicode.GetBytes(msg);
    ShowBytes(asciiBytes);
    ShowBytes(unicodeBytes);

    string asciiMsg = ascii.GetString(asciiBytes);
    string unicodeMsg = unicode.GetString(unicodeBytes);

    Console.WriteLine(Environment.NewLine);
    Console.WriteLine(asciiMsg);
    Console.WriteLine(unicodeMsg);
}

static void ShowBytes(byte[] msg)
{
    Console.WriteLine(Environment.NewLine);
    foreach (byte b in msg)
    {
        Console.Write("[{0}]", b);
    }
}
```

O resultado para ambos os código é:

```
[67][111][100][103][105][102][105][99][97][63][63][111][32][45][3
2][46][78][69][
84][32][70][114][97][109][101][119][111][114][107]

[67][0][111][0][100][0][103][0][105][0][102][0][105][0][99][0][97
][0][231][0][22
7][0][111][0][32][0][45][0][32][0][46][0][78][0][69][0][84][0][32
][0][70][0][114
][0][97][0][109][0][101][0][119][0][111][0][114][0][107][0]

Codgifica??o - .NET Framework
Codgificaçao - .NET Framework
```


Como falamos mais acima, o padrão **ASCII** somente somente 256 caracteres e, somente caracteres do alfabeto inglês. Como o idioma inglês não possui caracteres acentuadas,

algumas coisas são perdidas e, quando não são encontradas um correspondente, um ponto de interrogação "?" é colocado no lugar do caracter desconhecido por aquele padrão.

Tratando as falhas na codificação/decodificação

Quando utilizamos algum codificador que não consegue codificar ou decodificar algum caracter, ele coloca um ponto de interrogação "?" para indicar que o codificador corrente não é capaz de "traduzí-lo".

A classe **Encoding** fornece dois métodos chamados *GetEncoder* e *GetDecoder*, que retornam os objetos responsáveis por codificar e decodificar as *strings*, respectivamente. Ambas as classes possuem uma propriedade chamada *FallBack*. Essa propriedade recebe um objeto do tipo **EncoderFallback** ou **DecoderFallback** que representam uma ação que será executada quando um caracter ou um *byte* não puder ser convertido.

As classes **EncoderFallback** e **DecoderFallback** são utilizadas na codificação e decodificação, respectivamente. A primeira delas é a classe base para todos os *fallbacks* de codificação e, a segunda, a classe base para todos os *fallbacks* de decodificação. Conseguimos, através da imagem abaixo, entender a hierarquia dessas classes.

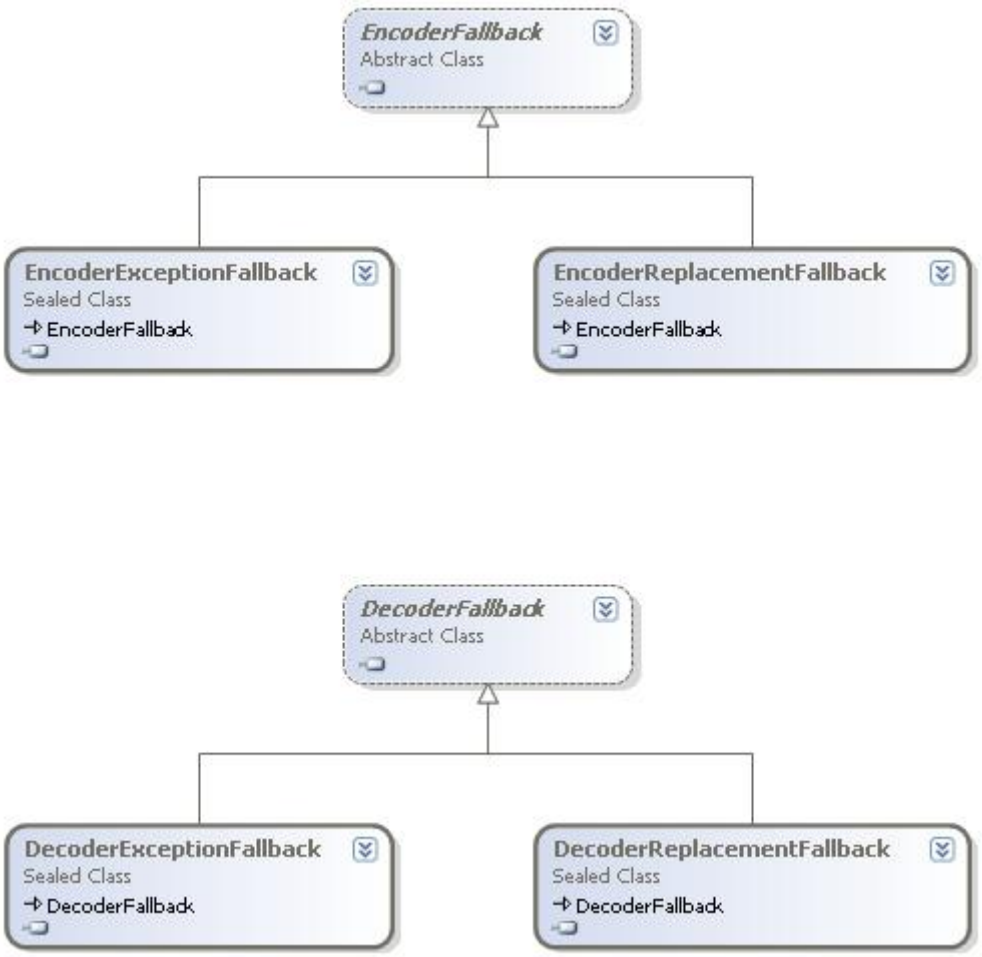

Imagem 7.1 – Hierarquia dos *fallbacks* de codificação e decodificação.

Basicamente, para ambos os casos, temos dois tipos de *fallbacks*:

- ***Fallbacks* de Substituição:** Quando um caracter ou *byte* não consegue ser "traduzido", ele substitui o mesmo por um caracter que podemos determinar. Por padrão, o ponto de interrogação "?" é utilizado.
  - **EncoderReplacementFallback:** *Fallback* que é invocado quando um caracter que puder ser convertido em uma seqüencia de *bytes*, convertendo o caracter para um valor pré-definido.
  - **DecoderReplacementFallback:** *Fallback* que é invocado quando uma seqüencia de *bytes* não puder ser convertida em um caracter, convertendo o *byte* para um valor pré-definido.
- ***Fallbacks* de Excessões:** Quando um caracter ou *byte* não consegue ser "traduzido", ele atira uma excessão do tipo **EncoderFallbackException** ou **DecoderFallbackException**, dependendo da operação que estamos tentando realizar.

- **EncoderExceptionFallback:** *Fallback* que é invocado quando um caracter que puder ser convertido em uma seqüencia de *bytes*, atirando uma excessão do tipo **EncoderFallbackException**.
- **DecoderExceptionFallback:** *Fallback* que é invocado quando uma seqüencia de *bytes* não puder ser convertida em um caracter, atirando uma excessão do tipo **DecoderFallbackException**.

O código abaixo exemplifica a utilização dos *fallbacks*:

```
VB.NET
Imports System.Text

Dim encoder As Encoder = Encoding.ASCII.GetEncoder()
encoder.Fallback = New EncoderReplacementFallback("*")
Dim chars() As Char = "ãBC".ToCharArray()
Dim buffer(chars.Length) As Byte
encoder.GetBytes(chars, 0, chars.Length, buffer, 0, True)
Console.WriteLine(Encoding.ASCII.GetString(buffer))

C#
using System.Text;

Encoder encoder = Encoding.ASCII.GetEncoder();
encoder.Fallback = new EncoderReplacementFallback("*");
char[] chars = "ãBC".ToCharArray();
Byte[] buffer = new byte[chars.Length];
encoder.GetBytes(chars, 0, chars.Length, buffer, 0, true);
Console.WriteLine(Encoding.ASCII.GetString(buffer));
```

O resultado para ambos os código é:

```
*BC
```

O "ã" é substituído pelo "*" porque esse caracter não está contemplado no padrão **ASCII** e, conseqüentemente, não consegue ser "traduzido". A utilização dos *fallbacks* são muito úteis quando você quer customizar a leitura ou gravação de *streams*. A utilização de *fallbacks* de excessões são as vezes mais utilizadas quando a leitura dos caracteres devem ser muito mais precisa.