Capítulo 6
Serialização

Introdução

Para enviar objetos de um local para outro, é preciso que você converta o objeto em um determinado formato. Quando estamos desenvolvendo uma aplicação, é muito comum precisarmos transferir dados desta aplicação para uma outra qualquer. Neste momento precisamos fazer uso de um processo, chamado de serialização onde o objeto que deverá ser transferido será persistido em um determinado formato e, em seguida, será enviado para o destino. Quando o destino recebe a informação, antes de a processar, ele efetua o processo de deserialização onde, converteremos esse "pacote" em um objeto conhecido. Esse processo chama-se deserialização.

O que é Serialização?

Serialização (*serialization*) é o processo onde você converte um objeto em um *stream* de *bytes* para persistí-lo na memória, em um banco de dados ou em arquivo. A idéia é salvar o estado do objeto para que a aplicação seja capaz de restaurar o estado atual do objeto quando necessário. O processo inverso é chamado de deserialização (*deserialization*).

O .NET Framework 2.0 já fornece classes e *Interfaces* para persistir os objetos em formato binário e SOAP e, além disso, traz também classes e *Interfaces* para que possa persistir o objeto em formato XML, para facilitar a comunicação entre diferentes plataformas e ainda, fornece mecanismo para podermos customizarmos a serialização do objeto, mudando o seu comportamente padrão para um mais específico.

O formato da serialização vai depender para onde deseja enviar o objeto. Por exemplo, se desejarmos passar o objeto entre aplicações .NET, ou melhor, entre *AppDomains* diferentes, podemos serializar o objeto em formato binário. Se desejarmos enviar o objeto para um Web Service, então é necessário serializar este objeto em formato XML. A imagem abaixo ilustra como funciona o processo de serialização:

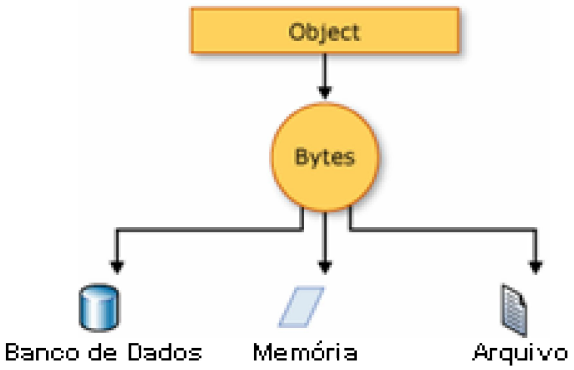

Imagem 6.1 – Processo de serialização

Quando você desenvolve uma aplicação que internamente faz o uso de objetos, quando a aplicação é fechada, todos os objetos são descartados juntamente com a mesma, ou seja, se você fez várias configurações em um determinado objeto, definiu valores em propriedades, invocou métodos que efetuaramos operações internas ao objeto, tudo isso será perdido quando a aplicação for finalizada. Se desejar manter esses valores, para mais tarde quando retornar a aplicação, é necessário serializar esse objeto fisicamente no disco.

Para efetuar essa serialização podemos, em primeiro momento, apenas utilizar as classes padrões fornecidas pelo .NET Framework 2.0. Inicialmente, precisamos de um *formatter*. Como já vimos acima, a serialização pode ser realizada em um *stream* usando SOAP, XML ou Binary. Os *formatters* são utilizados para determinar em qual desses formatos o objeto será serializado. O .NET Framework fornece dois *formatters*: **BinaryFormatter** e **SoapFormatter**, onde ambas fornecem os métodos de serialização e deserialização. A classe **BinaryFormatter** está disponível dentro do *namespace* **System.Runtime.Serialization.Formatters.Binary**. Já a classe SoapFormatter encontra-se dentro do *namespace* **System.Runtime.Serialization.Formatters.Soap** só que é necessário adicionar referência à *System.Runtime.Serialization.Formatters.Soap.dll*.

A classe **BinaryFormatter**, como o próprio nome diz, persite os dados em formato binário, serializando todos os membros da classe, inclusive os membros privados. Já a classe **SoapFormatter** persiste os dados em formato SOAP, que é uma XML especializado para facilitar o transporte de dados via web e permite apenas a serialização de membros definidos com o atributo **Serializable**. Ambas as classes fornecem também um método para serialização chamado *Serialize* e outro para deserialização, chamado de *Deserialize*, métodos provenientes da *Interface* **IFormatter**.

O método *Serialize* recebe um stream que é onde o objeto, que é passado como segundo parâmetro, será persistido. Já o método *Deserialize*, dado um stream contendo o valor persitido, retorna um **Object**, resultando do processo de deserialização. Para exemplificar a utilização dos dois tipos de *formatters* que vimos até agora, vamos criar uma classe chamada *Funcionario* que terá duas propriedades, a saber: *Nome* e *Salario*, sendo a primeira do tipo *string* e a segunda do tipo *double*. A única diferença aqui é que temos que dizer ao runtime que a classe *Funcionario* é uma classe serializável. Como vimos anteriormente, precisamos aplicar a classe o atributo **Serializable**. Por questões de espaço, somente colocaremos aqui a declaração da classe para exibir como aplicar o atributo. A parte importante resume-se na criação de dois métodos auxiliares, sendo uma para serializar e outro para deserializar o objeto *Funcionario*. Vejamos o código a seguir:

```
VB.NET
Imports System.Runtime.Serialization.Formatters.Soap

<Serializable()> Public Class Funcionario
    '...
End Class
```

```
Private fileName As String = "Cliente.bin"

Sub Main()
    Dim f As New Funcionario("João Mendes", 1300.0)
    Serialize(f)

    Dim f1 As Funcionario = Deserialize()
    Console.WriteLine(f1.Nome & " - " & f1.Salario)
End Sub

Private Sub Serialize(ByVal f As Funcionario)
    Using stream As Stream = File.Open(fileName, FileMode.Create)
        Dim formatter As New BinaryFormatter()
        formatter.Serialize(stream, f)
    End Using
End Sub

Private Function Deserialize() As Funcionario
    Dim f As Funcionario = Nothing
    Using stream As Stream = File.Open(fileName, FileMode.Open)
        Dim formatter As New BinaryFormatter()
        f = TryCast(formatter.Deserialize(stream), Funcionario)
    End Using

    Return f
End Function
```

C#

```
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]public class Funcionario
{
    //...
}

private static string fileName = "Cliente.bin";

static void Main(string[] args)
{
    Funcionario f = new Funcionario("João Mendes", 1300.0);
    Serialize(f);

    Funcionario f1 = Deserialize();
    Console.WriteLine(f1.Nome + " - " + f1.Salario);
}

private static void Serialize(Funcionario f)
{
    using (Stream stream = File.Open(fileName, FileMode.Create))
    {
```

```
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Serialize(stream, f);
    }
}

private static Funcionario Deserialize()
{
    Funcionario f = null;
    using (Stream stream = File.Open(fileName, FileMode.Open))
    {
        BinaryFormatter formatter = new BinaryFormatter();
        f = formatter.Deserialize(stream) as Funcionario;
    }
    return f;
}
```

O exemplo acima faz a serialização e deserialização utilizando o *formatter* **BinaryFormatter**. Criamos dois métodos auxilares para facilitar o trabalho. O primeiro deles, *Serialize*, recebe a instância de um objeto *Funcionario* que o persiste. Já o segundo método auxiliar, *Deserialize*, retorna um objeto *Funcionario* que é extraído do stream, que foi anteriormente salvo. A imagem abaixo exibe o conteúdo do arquivo que serializamos:

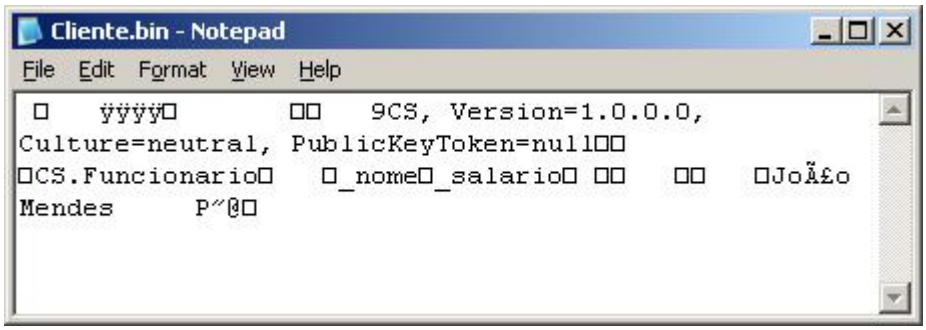

Imagem 6.2 – Arquivo com o objeto serializado

Agora, se desejarmos persistir o mesmo objeto em formato SOAP, para facilitar o transporte dos dados via web, o código que vimos acima, muda ligeiramente. As únicas alterações é com relação ao *formatter* e o *namespace* onde ele reside. Temos então que trocar de **BinaryFormatter** para **SoapFormatter**.

```
VB.NET
Imports System.Runtime.Serialization.Formatters.Soap

'...
Dim formatter As New SoapFormatter()
'...
```

```
C#
using System.Runtime.Serialization.Formatters.Soap;

//...
SoapFormatter formatter = new SoapFormatter();
//...
```

Finalmente temos o conteúdo salva em um formato XML:

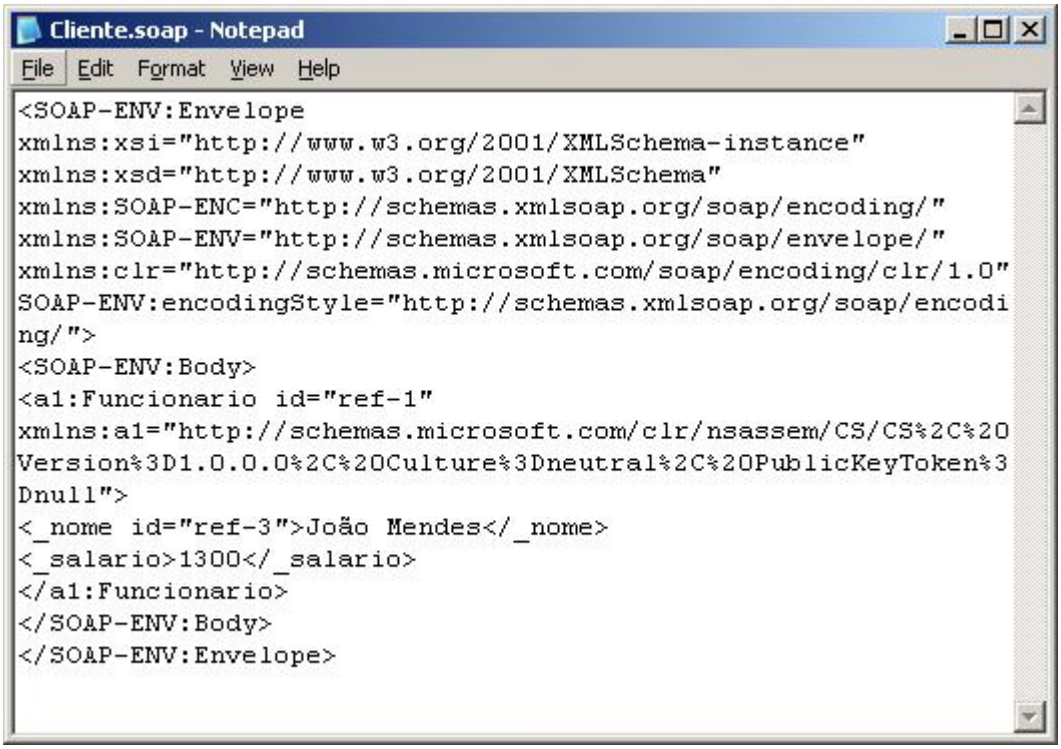

Imagem 6.3 – Objeto persistido através da classe **SoapFormatter**

Nota: Assim como existe o atributo **Serializable** para indicarmos o que pode e o que não pode ser serializado no objeto, temos também o atributo **NonSerializable** que indica se um determinado membro pode ou não ser serializado.

Serialização em formato XML

Haverá alguns momentos onde será necessário persistir um determinado objeto em um formato para que ele seja independente de plataforma, ou seja, nem sempre você pode garantir que o cliente que irá consumir o objeto serializado utilize também .NET.

Felizmente a Microsoft pensou nessa possibilidade e disponibilizou uma classe chamada **XmlSerializer** disponível dentro do *namespace* **System.Xml.Serialization**. Essa classe permite serializar propriedades públicas e membros do objeto em um formato XML para armazenamento ou mesmo para transporte.

Mais uma vez, se utilizarmos o mesmo exemplo que usamos para exemplificar a utilização das classes **BinaryFormatter** e **SoapFormatter**, basta alterarmos o formatter para **XmlSerializer** e, em seu construtor, especificar o tipo qual ele irá trabalhar. O trecho de código abaixo exibe como devemos configurar o **XmlSerializer** para serializarmos o objeto Funcionario que utilizamos um pouco mais acima em outros exemplos:

```
VB.NET
Imports System.Xml.Serialization

'...
Dim formatter As New XmlSerializer(GetType(Funcionario))
'...

C#
using System.Xml.Serialization;

//...
XmlSerializer formatter =
    new XmlSerializer(typeof(Funcionario));
//...
```

A imagem abaixo exibe o output gerado pela serialização através da classe **XmlSerializer**:

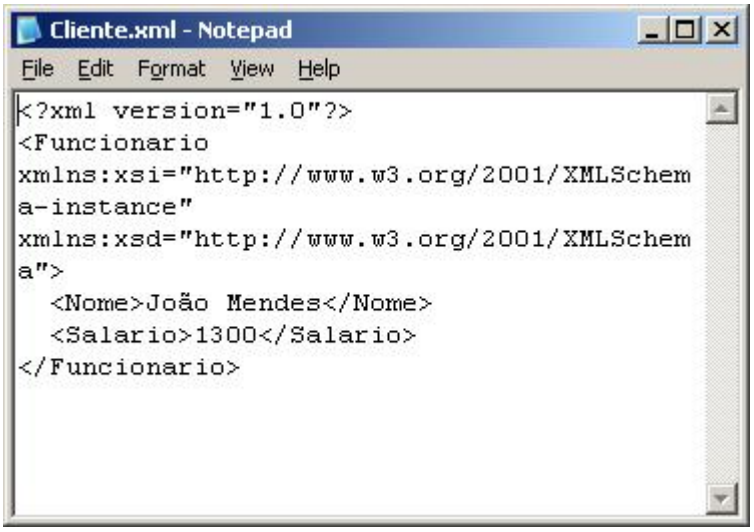

Imagem 6.4 – Output da classe **XmlSerializer**

Como vimos anteriormente, a classe **SoapFormatter** serializa e deserializa objetos em formato SOAP, de acordo com as especificações estipuladas pelo **W3C**, incluindo

informações extras, exclusivas para o contexto, como por exemplo, o *header* de uma mensagem SOAP e é bem limitado em relação a customização do formato a ser gerado.

Já a classe **XmlSerializer** faz o trabalho de persistência em formato XML, permitindo controlar como será o formato do documento XML que será gerado, ao contrário da classe **SoapFormatter**. Essa customização é possível graças aos atributos que estão disponíveis dentro do *namespace* **System.Xml.Serialization**, que podem ser aplicados nas propriedades e classes que desejamos persistir. Por padrão, a serialização XML não inclui informações sobre os tipos do objeto, assim como podemos reparar através da imagem 6.4. Isso acontece porque, durante o processo de deserialização, o tipo é extraído do próprio objeto. Por padrão, a classe **XmlSerializer** mapeia cada campo e propriedade do objeto para um elemento dentro do XML com o mesmo nome.

Esses atributos estão divididos entre duas categorias: atributos para XML e atributos para SOAP e todos eles estão contidos dentro do *namespace* **System.Xml.Serialization**. Esses atributos controlam a forma que o XML e o SOAP é gerado, permitindo uma formatação mais customizada ao nosso cenário. As duas tabelas a seguir detalham cada um desses atributos:

Atributo - SOAP	**Aplica-se**	**Descrição**
SoapAttribute	Campo público, propriedade, parâmetro ou valor de retorno	O membro da classe será serializado como um atributo XML.
SoapElement	Campo público, propriedade, parâmetro ou valor de retorno	A classe será serializada como um elemento XML.
SoapEnum	Campo público que é um enumerador	Aplicado a enumeradores para customizar como seus valores serão serializados.
SoapIgnore	Campos e propriedades públicos	A propriedade ou o campo será ignorado quando a classe for serializada.
SoapInclude	Classes derivadas e métodos públicos para documentos WSDL	Um determinado tipo deve ser incluído quando for gerar *schemas* e, conseqüentemente, ser reconhecido quando for serializado.
SoapType	Classes públicas	A classe deve ser serializada como um tipo XML.
Atributo – XML	**Aplica-se**	**Descrição**
XmlAnyAttribute	Campo público, propriedade, parâmetro ou valor de retorno que retorna um *array* de objetos do tipo **XmlAttribute**	Quando deserializado, o *array* será carregado com objetos do tipo **XmlAttribute** que representam todos os atributos XML desconhecidos do *schema*.

XmlAnyEment	Campo público, propriedade, parâmetro ou valor de retorno que retorna um *array* de objetos do tipo **XmlElemet**	Quando deserializado, o *array* será carregado com objetos do tipo **XmlElemet** que representam todos os elementos XML desconhecidos do *schema*.
XmlArray	Campo público, propriedade, parâmetro ou valor de retorno que retorna um *array* de objetos complexos	Os membros do *array* serão gerados como membros do *array* XML.
XmlArrayItem	Campo público, propriedade, parâmetro ou valor de retorno que retorna um *array* de objetos complexos	Os tipos derivados que podem ser inseridos dentro *array*. Usualmente é aplicado junto com um **XmlArray**.
XmlAttribute	Campo público, propriedade, parâmetro ou valor de retorno	O membro será serializado como um atributo XML.
XmlChoiceIdentifier	Campo público, propriedade, parâmetro ou valor de retorno	Especifica que membro pode ser futuramente detectado utilizando um enumerador.
XmlElement	Campo público, propriedade, parâmetro ou valor de retorno	O campo ou a propriedade será serializado como um elemento XML.
XmlEnum	Campo público que é um enumerador	Aplicado a enumeradores para customizar como seus valores serão serializados.
XmlIgnore	Campos e propriedades públicos	A propriedade ou o campo será ignorado quando a classe for serializada.
XmlInclude	Classes derivadas e retorno de métodos públicos para documentos WSDL	Um determinado tipo deve ser incluído quando for gerar *schemas* e, conseqüentemente, ser reconhecido quando for serializado.
XmlRoot	Classes públicas	Controla a serialização XML do atributo, definindo ele como sendo o *root*.
XmlText	Campos e propriedades públicos	O campo ou a propriedade deve ser serializada como um texto XML.
XmlType	Classes públicas	O nome e *namespace* do tipo XML.

Observação: O sufixo *Attribute* foi removido da coluna **Atributo** por questões de espaço. Apesar de existir, o compilador consegue entender e torna o sufixo não obrigatório quando é utilizado.

Para exemplificar a utilização de algum desses atributos vamos analisar trechos do código que estará disponível na íntegra na demonstração do capítulo. Através do código abaixo veremos a utilização dos atributos **XmlRoot**, **XmlAttribute** e **XmlIgnore**. O **XmlRoot** vai determinar qual será o elemento raiz do documento XML. O construtor deste atributo pode receber uma string contendo o nome mas, se nenhum for informado, o mesmo nome do membro será definido como raiz. Já o segundo atributo, **XmlAttribute**, é utilizado para marcarmos uma propriedade ou campo como sendo um atributo ao invés de um elemento. Finalmente, o **XmlIgnore** que irá evitar de serializar um determinado campo do objeto.

VB.NET

```
Imports System.Xml
Imports System.Xml.Serialization

<XmlRoot("funcionariosAtivos"), Serializable> _
Public Class Empresa
    Private _sala As String

    '...

    <XmlAttribute("salaDoEscritorio")> _
    Public Property Sala As String
        '...
    End Property
End Class

<Serializable> _
Public Class Funcionario
    Private _salario As Double

    '...

    <XmlIgnore> _
    Public Property Salario As Double
        '...
    End Property
End Class
```

C#

```
using System.Xml;
using System.Xml.Serialization;

[Serializable]
[XmlRoot("funcionariosAtivos")]
public class Empresa
{
    private string _sala;
```

```
    //...

    [XmlAttribute("salaDoEscritorio")]
    public string Sala
    {
        //...
    }
}

[Serializable]
public class Funcionario
{
    private double _salario;

    //...

    [XmlIgnore]
    public double Salario
    {
        //...
    }
}
```

O documento XML gerado é exibido abaixo. Se analisarmos, o elemento raiz chama-se "funcionariosAtivos", assim como definimos através do elemento **XmlRoot**. Repare também que a propriedade *Sala* foi serializada com o nome "salaDoEscritorio" e sendo atributo do elemento "funcionariosAtivos". Finalmente, a propriedade *Salario* não é persitido, justamente porque o atributo **XmlIgnore** foi especificado nele.

XML

```
<?xml version="1.0"?>
<funcionariosAtivos
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  salaDoEscritorio="Oitavo andar">
  <Funcionarios>
    <Funcionario>
      <Nome>João Mendes</Nome>
      <Superior>
        <Nome>Mateus Golias</Nome>
      </Superior>
    </Funcionario>
  </Funcionarios>
</funcionariosAtivos>
```

Customizando a serialização XML

Se desejarmos customizar a serialização e deserialização realizada pela classe **XmlSerializer**, podemos implementar a *Interface* **IXmlSerializable** na classe que será persistida. Essa *Interface* fornece três métodos: *GetSchema*, *ReadXml* e *WriteXml*. O primeiro deles, retorna um objeto do tipo **XmlSchema** que descreve o documento XML que é gerado pelo método *WriteXml* e lido pelo método *ReadXml*. Já os métodos *ReadXml* e *WriteXml* lê e gera o documento XML, respectivamente. Para o método *ReadXml* é passado como parâmetro um objeto do tipo **XmlReader**, que é o stream que representa o objeto serializado. Esse método deve ser capaz de extrair as informações do objeto e reconstituí-lo. Já para o método *WriteXml*, um objeto do tipo **XmlWriter** é passado como parâmetro. Esse parâmetro fornecerá métodos para que seja possível persistir o objeto em formato XML.

A classe *CPF* abaixo implementa a *Interface* **IXmlSerializable** e customizada cada um dos métodos de uma forma simples para fins de exemplo.

```
VB.NET
Imports System.Xml
Imports System.Xml.Serialization
Imports System.Xml.Schema

Public Class CPF
    Implements IXmlSerializable

    Private _numero As String

    Public Property Numero() As String
        Get
            Return Me._numero
        End Get
        Set(ByVal value As String)
            Me._numero = value
        End Set
    End Property

    Public Function GetSchema() As XmlSchema _
        Implements IXmlSerializable.GetSchema

        Return Nothing
    End Function

    Public Sub ReadXml(ByVal reader As XmlReader) _
        Implements IXmlSerializable.ReadXml

        Me._numero = reader.ReadString()
    End Sub
```

```
    Public Sub WriteXml(ByVal writer As XmlWriter) _
        Implements IXmlSerializable.WriteXml

        writer.WriteString(Me._numero)
    End Sub

End Class
```

C#

```
using System.Xml;
using System.Xml.Serialization;
using System.Xml.Schema;

public class CPF : IXmlSerializable
{
    private string _numero;

    public string Numero
    {
        get
        {
            return this._numero;
        }
        set
        {
            this._numero = value;
        }
    }

    public XmlSchema GetSchema()
    {
        return null;
    }

    public void ReadXml(XmlReader reader)
    {
        this._numero = reader.ReadString();
    }

    public void WriteXml(XmlWriter writer)
    {
        writer.WriteString(this._numero);
    }
}
```


Se repararmos, quando a *Interface* **IXmlSerializable** é implementada, a classe dispensa o uso do atributo **Serializable**. É importante dizer também que a forma como se faz a serialização através do objeto **XmlSerializer** não muda absolutamente nada.

A classe **XmlSerializer** ainda fornece quatro eventos interessantes que são invocados durante a deserialização do objeto. Quando serializamos um determinado objeto, persistimos as informações necessárias que desejamos que sejam armazenadas. Suponhamos que mais tarde, quando queremos recuperar esse objeto através do processo de deserialização, o arquivo em que o persistimos tiver atributos ou elementos desconhecidos, ou seja, membros que não fazem parte do *schema* atual do objeto, eles serão ignorados pelo processo de deserialização. Apesar de ignorados pelo processo, podemos ser notificados quando isso acontecer e isso é possível através desses quatro eventos que falamos. A tabela abaixo explica detalhadamente cada um deles:

Evento	Descrição
UnknownAttribute	Ocorre quando o objeto **XmlSerializer** encontrar um atributo desconhecido durante o processo de deserialização.
UnknownElement	Ocorre quando o objeto **XmlSerializer** encontrar um elemento desconhecido durante o processo de deserialização.
UnknownNode	Ocorre quando o objeto **XmlSerializer** encontrar um nó desconhecido durante o processo de deserialização.
UnreferencedObject	Ocorre quando durante o processo de deserialização de um *stream* baseado em SOAP, quando o objeto **XmlSerializer** encontrar um tipo conhecido que não é usado ou referenciado.

Para que seja possível interceptar o lançamento desses eventos, é necessário anexarmos os procedimentos que serão disparados quando o mesmo for disparado. Temos abaixo um exemplo simples de como proceder neste caso:

```
VB.NET
Imports System.Xml.Serialization

Using stream As Stream = File.Open(fileName, FileMode.Open)
    Dim f As New XmlSerializer(GetType(CPF))
    AddHandler f.UnknownAttribute, AddressOf UnknownAttribute
    AddHandler f.UnknownElement, AddressOf UnknownElement
    AddHandler f.UnknownNode, AddressOf UnknownNode
    Dim c As CPF = TryCast(f.Deserialize(stream), CPF)
End Using

Private Sub UnknownAttribute(ByVal sender As Object, _
    ByVal e As XmlAttributeEventArgs)

    Console.WriteLine(e.Attr.Name)
    Console.WriteLine(e.LineNumber)
End Sub

Private Sub UnknownNode(ByVal sender As Object, _
    ByVal e As XmlNodeEventArgs)
```

```
    Console.WriteLine(e.Name)
    Console.WriteLine(e.LineNumber)
End Sub

Private Sub UnknownElement(ByVal sender As Object, _
    ByVal e As XmlElementEventArgs)

    Console.WriteLine(e.Element.Name)
    Console.WriteLine(e.LineNumber)
End Sub
```

C#

```
using System.Xml.Serialization;

using (Stream stream = File.Open(fileName, FileMode.Open))
{
    XmlSerializer f = new XmlSerializer(typeof(T));

    f.UnknownElement +=
        new XmlElementEventHandler(UnknownElement);

    f.UnknownNode +=
        new XmlNodeEventHandler(UnknownNode);

    f.UnknownAttribute +=
        new XmlAttributeEventHandler(UnknownAttribute);

    CPF c = f.Deserialize(stream) as CPF;
}

static void UnknownAttribute(object sender, XmlAttributeEventArgs
e)
{
    Console.WriteLine(e.Attr.Name);
    Console.WriteLine(e.LineNumber);
}

static void UnknownNode(object sender, XmlNodeEventArgs e)
{
    Console.WriteLine(e.Name);
    Console.WriteLine(e.LineNumber);
}

static void UnknownElement(object sender, XmlElementEventArgs e)
{
    Console.WriteLine(e.Element.Name);
    Console.WriteLine(e.LineNumber);
}
```

Serialização customizada

Quanto mais você trabalha com serialização e deserialização de objetos para enviá-los para os mais diversos locais, mais aumenta a necessidade de customizar esses processos para atender uma determinada funcionalidade.

Como vimos até o momento, podemos utilizar as classes fornecidas pelo .NET Framework para efetuarmos a persistência, mas se estivermos falando de um cenário muito específico, felizmente como grande parte do .NET Framework, ele também fornece classes e *Interfaces* para customizarmos os processos de serialização e deserialização, como por exemplo, podemos customizar um *formatter* específico para que ele atenda ao nosso cenário.

Já notamos que durante o processo de serialização, precisaremos especificar quais valores serão salvos e, quando fazer o inverso, o processo de deserialização, devemos extrair os valores do *stream* para o nosso objeto corrente. O .NET Framework fornece duas estruturas **SerializationEntry**, **StreamingContext** e a classe **SerializationInfo**. Esses tipos são utilizados para especificar os valores que são requeridos para representar o objeto na sua forma serializada e recuperá-los durante o processo de deserialização. Além disso, esses tipos fornecem informações a respeito do tipo que foi serializado, *Assembly*, etc..

Para um detalhamento melhor, a estrutura **SerializationEntry** contém as propriedades *Name*, *ObjectType* e *Value*, quais disponibilizam o nome, tipo e valor, respectivamente, do objeto serializado. Já a estrutura **StreamingContext** fornece informações a respeito da fonte e do destino do *stream*, ou seja, durante o processo de serialização, define o destino dos dados e, quando for o processo de deserialização, especifica a fonte do *stream*. Finalmente a classe **SerializationInfo**, que está contida dentro do *namespace* **System.Runtime.Serialization**, armazena todo o conteúdo que é necessário tanto para serializar quanto para deserializar um determinado objeto em uma coleção do tipo chave-valor.

Interfaces

Como dissemos há pouco tempo atrás, o .NET Framework disponibiliza várias *Interfaces* que podemos utilizar para customizar os processos de serialização e deserialização. Essas *Interfaces* estão contidas dentro do *namespace* **System.Runtime.Serialization**. Essas *Interfaces*, quando implementadas, permitem um melhor controle sob como os objetos são serializados e deseriliazados.

As *Interfaces* que temos a nossa disposição são: **ISerializable**, **IFormatter**, **IFormatterConverter** e **IDeserializationCallback**. Abaixo analisaremos com mais detalhes cada uma delas e qual a sua utilidade.

ISerializable

Qualquer classe pode ser seralizada desde que esteja marcada com o atributo **Serializable**. Se a classe precisa controlar o processo de serialização, você pode implementar a *Interface* **ISerializable**. Por padrão, o *formatter* que está sendo utilizado para o processo de serialização invoca o método *GetObjectData* (fornecido pela *Interface* em questão) e fornece um objeto do tipo **SerializationInfo** com todos os dados que são requeridos para representar o objeto.

A implementação desta *Interface* implica em a classe possuir um construtor que recebe como parâmetro um objeto do tipo **SerializationInfo** e outro do tipo **StreamingContext**. Durante o processo de deserialização, esse construtor é invocado somente depois que os dados foram deserializados pelo *formatter*. Geralmente esse construtor deve ser *protected* se a classe permitir derivações. O techo de código abaixo ilustra a utilização da implementação desta *Interface*:

```
VB.NET
Imports System.Runtime.Serialization

Public Class CPF
    Implements ISerializable

    Private _numero As Integer
    Private _nome As String

    Public Sub New()
    End Sub

    Protected Sub New(ByVal info As SerializationInfo, _
        ByVal context As StreamingContext)

        Me._numero = info.GetInt32("numero")
        Me._nome = info.GetString("nome")
    End Sub

    ' Propriedades que expõe os membros internos
    ' _numero e _nome

    Public Sub GetObjectData(ByVal info As SerializationInfo, _
        ByVal context As StreamingContext) _
        Implements ISerializable.GetObjectData

        info.AddValue("numero", Me._numero)
        info.AddValue("nome", Me._nome)
    End Sub
End Class
```

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

public class CPF : ISerializable
{
    private int _numero;
    private string _nome;

    public CPF() { }

    protected CPF(SerializationInfo info, StreamingContext
context)
    {
        this._numero = info.GetInt32("numero");
        this._nome = info.GetString("nome");
    }

    // Propriedades que expõe os membros internos
    // _numero e nome

    public void GetObjectData(SerializationInfo info,
StreamingContext context)
    {
        info.AddValue("numero", this._numero);
        info.AddValue("nome", this._nome);
    }
}
```

IFormatter

A *Interface* **IFormatter** por qualquer classe que irá ser um *formatter*, fornecendo funcionalidades para formatar objetos serializados e também possibilitando o controle do *output* dos processos de serialização e deserialização. Essa *Interface* fornece os dois principais métodos que são utilizados por um formatter: *Serialize* e *Deserialize*.

Além de disponibilizar todos os membros necessários para a criação de um *formatter* customizado, essa *Interface* também é implementada em uma classe abstrata chamada **Formatter**. Essa classe fornece funcionalidades básicas e pode ser utilizada (herdada) para todos os *formatters* customizados que forem construídos ao invés de implementar diretamente a *Interface* **IFormatter**. Neste caso, todos os membros que são herdados da *Interface* **IFormatter** são mantidos como abstratos; ela somente adiciona outros membros que auxiliam durante os processos.

IFormatterConverter

Essa *Interface* fornece uma conexão entre a instância da classe **SerializationInfo** e o *formatter* para que seja possível efetuar as conversões de tipos necessárias durante os

processos. Assim como a *Interface* **IFormatter** é implementada na classe abstrata **Formatter** mas, *Interface* **IFormatter** também é implementada em uma classe chamada **FormatterConverter**, contendo toda a implementação básica para o código necessário para efetuar as conversões. Internamente, essa classe faz o uso da classe **Convert**.

IDeserializationCallback

Vamos imaginar o seguinte cenário: temos uma classe chamada *Calculo* que, dados dois números ele calcula a soma entre eles. Naturalmente quando optar por serializar esse objeto, você não irá armazenar o valor do resultado, ou seja, vai apenas se preocupar em salvar os valores que efetivamente geram o resultado.

Até o momento, nada diferente. Como propriedades públicas são, por padrão, persistidas automaticamente, não teremos problemas com relação a perda dos dados. Mas, e se no momento da deserialização quisermos notificar o objeto recém "revitalizado" para que ele faça alguma operação? É neste momento que a *Interface* **IDeserializationCallback** entra em ação. Quando o método *Deserialize* do formatter é chamado, internamente ele verifica se o tipo do objeto que está sendo recuperado implementa ou não essa *Interface*. Se estiver implementada, o método *OnDeserialization*, fornecido por ela, é executado. Trazendo isso para o nosso exemplo, poderemos nesse momento, efetuarmos o cálculo (soma) dos números para quando o usuário recuperar o valor, o mesmo já estar processado. Para exemplificar a utilização desta *Interface*, o código abaixo exibe apenas os pontos relevantes do código necessários para essa implementação:

VB.NET
```
Imports System.Runtime.Serialization

<Serializable()> Public Class Soma
    Implements IDeserializationCallback

    Private _numero1 As Integer
    Private _numero2 As Integer
    Private _total As Integer

    ' propriedades ocultadas por
    ' questões de espaço

    Public Sub OnDeserialization(ByVal sender As Object) _
      Implements IDeserializationCallback.OnDeserialization

        Me._total = Me._numero1 + Me._numero2
    End Sub
End Class
```

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

```
[Serializable]
public class Soma : IDeserializationCallback
{
    private int _numero1;
    private int _numero2;
    private int _total;

    // propriedades ocultadas por
    // questões de espaço

    public void OnDeserialization(object sender)
    {
        this._total = this._numero1 + this._numero2;
    }
}
```

Quando o usuário requisitar pelo membro *_total* ele já conterá o resultado da soma entre os dois membros internos.